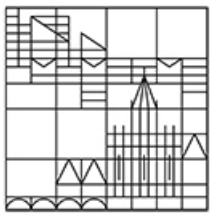
Fachbereich Informatik und
Informationswissenschaft

Universität
Konstanz



Skriptum

zur Vorlesung

Theoretische Grundlagen der Informatik

gehalten in Sommersemester 2013

von

Sven Kosub

19. Juli 2013

Version v0.28

Inhaltsverzeichnis

Mathematische Grundlagen 1

1.1 Wörter und Zahlen

Es sei Σ eine endliche, nichtleere Menge. Dann heißt Σ *Alphabet*. Die Elemente von Σ heißen *Symbole* (auch *Buchstaben* oder *Zeichen*). Ein *Wort* x über Σ ist eine endliche Aneinanderreihung von Symbolen, d.h., $x = a_1 a_2 \dots a_n$ mit $a_i \in \Sigma$ für alle $i \in \{1, \dots, n\}$, $n \geq 0$. Mit $|x|$ wird die *Länge* eines Wortes x bezeichnet, d.h., die Anzahl der Symbole, aus denen x besteht; formal $|a_1 a_2 \dots a_n| = n$. Das *leere Wort* wird mit ε bezeichnet, d.h., es gilt $|\varepsilon| = 0$. Σ^* ist die Menge aller Wörter über Σ. Σ^+ ist die Menge der nichtleeren Wörter über Σ. Es gilt also $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

Wir vereinbaren Operationen auf Wörtern ($x = a_1 a_2 \dots a_n, y = b_1 b_2 \dots b_m \in \Sigma^*$):

- *Konkatenation* von x und y: $xy =_{\text{def}} a_1 a_2 \dots a_n b_1 b_2 \dots b_m$
- *Spiegelung* von x: $x^R =_{\text{def}} a_n \dots a_2 a_1$
- *k-te Potenz* von x: $x^0 =_{\text{def}} \varepsilon$ und $x^k =_{\text{def}} x^{k-1} x$ für $k \geq 1$

Eine Menge $L \subseteq \Sigma^*$ heißt *(formale) Sprache* über Σ.

Wir übertragen Operationen auf Wörtern auf Operationen auf Sprachen ($L, L' \subseteq \Sigma^*$):

- *Konkatenation* von L und L': $L \cdot L' =_{\text{def}} \{\, xy \mid x \in L \text{ und } y \in L' \,\}$
- *k-te Potenz* von L: $L^0 =_{\text{def}} \{\varepsilon\}$ und $L^k =_{\text{def}} L^{k-1} \cdot L$ für $k \geq 1$
- *Iteration* von L: $L^* =_{\text{def}} \bigcup_{k=0}^{\infty} L^k$ und $L^+ =_{\text{def}} \bigcup_{k=1}^{\infty} L^k$

Insbesondere gilt $\{x\}^k = \{x^k\}$.

$\mathbb{N} = \{0, 1, 2, \dots\}$ ist die Menge der natürlichen Zahlen; $\mathbb{N}_+ = \mathbb{N} \setminus \{0\}$.

Neben Addition $+$ und Multiplikation $\cdot$ betrachten wir weitere arithmetische Operationen auf Zahlen ($x, y \in \mathbb{N}$):

- *modifizierte Subtraktion* von x und y:

$$x \dot{-} y =_{\text{def}} \begin{cases} x - y & \text{falls } x \geq y \\ 0 & \text{sonst} \end{cases}$$

- *ganzzahlige Division* von x mit y:

$$\left\lfloor \frac{x}{y} \right\rfloor =_{\text{def}} \begin{cases} \text{größtes } z \in \mathbb{N} \text{ mit } z \cdot y \leq x & \text{falls } y > 0 \\ x & \text{sonst} \end{cases}$$

Wir verwenden auch die folgenden funktionalen Darstellungen:

- $\text{sum} : \mathbb{N}^2 \to \mathbb{N}$ mit $\text{sum}(x, y) =_{\text{def}} x + y$
- $\text{md} : \mathbb{N}^2 \to \mathbb{N}$ mit $\text{md}(x, y) =_{\text{def}} x \dot{-} y$
- $\text{prod} : \mathbb{N}^2 \to \mathbb{N}$ mit $\text{prod}(x, y) =_{\text{def}} x \cdot y$
- $\text{div} : \mathbb{N}^2 \to \mathbb{N}$ mit $\text{div}(x, y) =_{\text{def}} \left\lfloor \frac{x}{y} \right\rfloor$
- $\exp : \mathbb{N}^2 \to \mathbb{N}$ mit $\exp(x, y) =_{\text{def}} x^y$

Für die verschiedenen Berechnungsmodelle kodieren wir Zahlen als Wörter und umgekehrt:

- Die *unäre* Kodierung stellt Zahlen über einem einelementigen Alphabet dar und ist definiert als $\text{un} : \mathbb{N} \to \{a\}^* : n \mapsto a^n$. Die Kodierung ist bijektiv.
- Die *dezimale* Kodierung stellt Zahlen über dem Alphabet $\{0, 1, \ldots, 9\}$ dar und ist definiert als $\text{dec}(0) =_{\text{def}} 0$ und für $n > 0$ als $\text{dec}(n) =_{\text{def}} a_m a_{m-1} \ldots a_1 a_0$, wobei $n = \sum_{i=0}^{m} a_i \cdot 10^i$ mit $a_0, \ldots, a_{m-1} \in \{0, 1, 2, \ldots, 9\}$ und $a_m \in \{1, \ldots, 9\}$ gilt. Die Kodierung ist nicht surjektiv, also auch nicht bijektiv.
- Die *binäre* Kodierung stellt Zahlen über dem Alphabet $\{0, 1\}$ dar und ist definiert als $\text{bin}(0) =_{\text{def}} 0$ und für $n > 0$ als $\text{bin}(n) =_{\text{def}} a_m a_{m-1} \ldots a_1 a_0$, wobei $n = \sum_{i=0}^{m} a_i \cdot 2^i$ mit $a_0, \ldots, a_{m-1} \in \{0, 1\}$ und $a_m = 1$ gilt. Die Kodierung ist nicht surjektiv, also auch nicht bijektiv.
- Die *k-adische* Kodierung stellt Zahlen über dem Alphabet $\{1, \ldots, k\}$ für $k \geq 1$ dar und ist definiert als $\text{ad}_k(0) =_{\text{def}} \varepsilon$ und für $n > 0$ als $\text{ad}_k(n) =_{\text{def}} a_m a_{m-1} \ldots a_1 a_0$, wobei $n = \sum_{i=0}^{m} a_i \cdot 10^i$ mit $a_0, \ldots, a_m \in \{1, \ldots, k\}$ gilt. Die Kodierung ist für alle $k \geq 1$ bijektiv. Insbesondere gilt $\text{ad}_1 = \text{un}$.
- Die *dyadische* Kodierung $\text{dya} : \mathbb{N} \to \{1, 2\}^*$ ist definiert als $\text{dya} =_{\text{def}} \text{ad}_2$.

1.2 Algebraische Erzeugung

1.2.1 Hüllenoperatoren

Es sei A eine Menge. Eine totale Funktion $\Gamma : \mathcal{P}(A) \to \mathcal{P}(A)$ heißt *Hüllenoperator* über A, falls folgende Eigenschaften erfüllt sind:

1. Für alle $B \subseteq A$ gilt $B \subseteq \Gamma(B)$ (*Einbettung*)

2. Für alle $B, C \subseteq A$ gilt $B \subseteq C \rightarrow \Gamma(B) \subseteq \Gamma(C)$ (*Monotonie*)
3. Für alle $B \subseteq A$ gilt $\Gamma(\Gamma(B)) = \Gamma(B)$ (*Abgeschlossenheit*)

Beispiel: Für $\mathbb{N}$ und die Teilbarkeitsrelation $|$ ist der durch

$$\Gamma^{|}(B) =_{\text{def}} \{\ n \in \mathbb{N} \mid \text{es gibt ein } m \in B \text{ mit } n|m\ \}$$

für alle $B \subseteq N$ definierte Operator $\Gamma^{|} : \mathcal{P}(\mathbb{N}) \rightarrow \mathcal{P}(\mathbb{N})$ ein Hüllenoperator. Insbesondere gilt:

$$\begin{aligned}
\Gamma^{|}(\{0\}) &= \mathbb{N} \\
\Gamma^{|}(\{24\}) &= \{1,2,3,4,6,8,12,24\} \\
\Gamma^{|}(\{113\}) &= \{1,113\} \\
\Gamma^{|}(\{3,25\}) &= \{1,3,5,25\} \\
\Gamma^{|}(\{\ 2n \mid n \in \mathbb{N}\ \}) &= \mathbb{N} \\
\Gamma^{|}(\{\ 2n+1 \mid n \in \mathbb{N}\ \}) &= \{\ 2n+1 \mid n \in \mathbb{N}\ \}
\end{aligned}$$

1.2.2 Algebraische Hüllenoperatoren

Im Folgenden interessieren wir uns für algebraische Hüllenoperatoren. Dazu erweitern wir unser Vokabular.

Es sei $O : A^s \rightarrow A$ eine s-stellige Operation auf A. Dann heißt eine Menge $B \subseteq A$ *abgeschlossen unter* O, falls aus $a_1, \ldots, a_s \in B$ stets $O(a_1, \ldots, a_s) \in B$ folgt.

Sind $O_i : A^{s_i} \rightarrow A$ für $i \in \{1, \ldots, k\}$ Operationen auf A, so definieren wir für alle $B \subseteq A$ die Menge $\Gamma_{O_1,\ldots,O_k}(B)$ wie folgt:

1. Ist $a \in B$, so ist $a \in \Gamma_{O_1,\ldots,O_k}(B)$.
2. Sind $a_1, \ldots, a_{s_i} \in \Gamma_{O_1,\ldots,O_k}(B)$ und ist $O_i(a_1, \ldots, a_{s_i})$ definiert, so ist $O_i(a_1, \ldots, a_{s_i}) \in \Gamma_{O_1,\ldots,O_k}(B)$.
3. Nichts sonst gehört zu $\Gamma_{O_1,\ldots,O_k}(B)$.

Proposition 1.1 *1. $\Gamma_{O_1,\ldots,O_k}(B)$ die bezüglich der Mengeninklusion kleinste Menge, die B enthält und abgeschlossen ist unter $O_1, \ldots, O_k$.*

2. $\Gamma_{O_1,\ldots,O_k}$ ist ein Hüllenoperator über A.

$\Gamma_{O_1,\ldots,O_k}$ heißt *durch $O_1, \ldots, O_k$ definierter algebraischer Hüllenoperator.*

Beispiele:

- $\Gamma_{\cap,\cup}, \Gamma_{\cap,\cup,\text{-}}, \Gamma_{\cup,\text{-}}$ und $\Gamma_{\cap,\text{-}}$ sind algebraische Hüllenoperatoren über einer beliebigen Menge A. Es gilt $\Gamma_{\cap,\cup,\text{-}} = \Gamma_{\cup,\text{-}} = \Gamma_{\cap,\text{-}}$

- Γ_{sum}, Γ_{prod} und $\Gamma_{\text{sum,prod}}$ sind algebraische Hüllenoperatoren über $\mathbb{N}$. Gilt $\Gamma_{\text{sum}} = \Gamma_{\text{sum,prod}}$?
- Für Halbordnungen $(A, \leq)$ ist der durch

  $$\Gamma^{\leq}(B) =_{\text{def}} \{\ x \in A \mid \text{es gibt ein } y \in B \text{ mit } x \leq y\ \}$$

  für alle $B \subseteq A$ definierte Hüllenoperator $\Gamma^{\leq}$ im Allgemeinen kein algebraischer Hüllenoperator.

1.2.3 Algebraische Erzeugung von Funktionen

Für eine beliebige Menge A definieren die Menge

$$\mathbf{FUNK}(A) =_{\text{def}} \{\ \varphi \mid \text{es gibt ein } n \in \mathbb{N} \text{ mit } \varphi : A^n \to A\ \}$$

von Funktionen beliebiger Stelligkeit.

Wir betrachten spezielle Operationen auf $\mathbf{FUNK}(A)$ (für $\varphi : A^n \to A, \psi : A^m \to A$):

- zyklische Vertauschung der Variablen ZV (für $n \geq 2$):

  $$\text{ZV}(\varphi)(x_1, \ldots, x_n) =_{\text{def}} \varphi(x_2, x_3, \ldots, x_n, x_1)$$

- Vertauschung der beiden letzten Variablen LV (für $n \geq 2$):

  $$\text{LV}(\varphi)(x_1, \ldots, x_n) =_{\text{def}} \varphi(x_1, \ldots, x_{n-2}, x_n, x_{n-1})$$

- Identifizierung der beiden letzten Variablen ID (für $n \geq 2$):

  $$\text{ID}(\varphi)(x_1, \ldots, x_{n-1}) =_{\text{def}} \varphi(x_1, \ldots, x_{n-1}, x_{n-1})$$

- Substitution von Funktionen an der letzten Stelle SUB (für $n \geq 1$):

  $$\text{SUB}(\varphi, \psi)(x_1, \ldots, x_{n-1}, y_1, \ldots, y_m) =_{\text{def}} \varphi(x_1, \ldots, x_{n-1}, \psi(y_1, \ldots, y_m)),$$

  wobei $\{x_1, \ldots, x_{n-1}\} \cap \{y_1, \ldots, y_m\} = \emptyset$ (d.h. die in φ und ψ vorkommenden Variablennamen sind disjunkt) gilt.

  Beispiele: Es sei $A = \mathbb{N}$.

  - Wir betrachten die Wirkungsweise von ZV und LV. Es gilt

    $$\text{ZV}(\text{sum})(x, y) = \text{LV}(\text{sum})(x, y) = \text{sum}(y, x) = \text{sum}(x, y)$$

    und somit $\text{ZV}(\text{sum}) = \text{LV}(\text{sum}) = \text{sum}$. Andererseits gilt

    $$\text{ZV}(\text{md})(x, y) = \text{LV}(\text{md})(x, y) = \text{md}(y, x) = y \dot{-} x$$

    und somit $\text{ZV}(\text{md}) = \text{LV}(\text{md}) \neq \text{md}$.

- Für $n, m \in \mathbb{N}$ definieren wir die *n-stellige m-Konstante*

$$C_m^n : \mathbb{N}^n \to \mathbb{N} : (x_1, \ldots, x_n) \mapsto m.$$

  Dann gilt

$$\begin{aligned} \mathrm{ID}(\mathrm{sum})(x) &= \mathrm{sum}(x,x) = x + x = 2x \\ \mathrm{ID}(\mathrm{md})(x) &= \mathrm{md}(x,x) = x \dot{-} x = 0 = C_0^1(x) \end{aligned}$$

  Also gilt $\mathrm{ID}(\mathrm{md}) = C_0^1$.

- Für die Substitution ergibt sich beispielsweise:

$$\begin{aligned} \mathrm{SUB}(\mathrm{sum},\mathrm{sum})(x,y,z) &= \mathrm{sum}(x,\mathrm{sum}(y,z)) = x + (y + z) \\ \mathrm{ID}(\mathrm{SUB}(\mathrm{sum},\mathrm{sum}))(x,y) &= \mathrm{SUB}(\mathrm{sum},\mathrm{sum})(x,y,y) = x + 2y \\ \mathrm{ID}(\mathrm{ID}(\mathrm{SUB}(\mathrm{sum},\mathrm{sum})))(x) &= \mathrm{ID}(\mathrm{SUB}(\mathrm{sum},\mathrm{sum}))(x,x) = 3x \end{aligned}$$

- Eine komplexere Wirkung ergibt sich beispielsweise mit nachfolgender Rechnung:

$$\begin{aligned} \mathrm{ID}(\mathrm{ZV}(\mathrm{ZV}(\mathrm{SUB}(\mathrm{sum},\mathrm{md}))))(x,y) &= \mathrm{ZV}(\mathrm{ZV}(\mathrm{SUB}(\mathrm{sum},\mathrm{md})))(x,y,y) \\ &= \mathrm{ZV}(\mathrm{SUB}(\mathrm{sum},\mathrm{md})))(y,y,x) \\ &= \mathrm{SUB}(\mathrm{sum},\mathrm{md})(y,x,y) \\ &= \mathrm{sum}(y,\mathrm{md}(x,y)) \\ &= y + (x \dot{-} y) \\ &= \max\{x,y\} \end{aligned}$$

  Es gilt also $\mathrm{ID}(\mathrm{ZV}(\mathrm{ZV}(\mathrm{SUB}(\mathrm{sum},\mathrm{md})))) = \max$. Mit anderen Worten gilt $\max \in \Gamma_{\mathrm{ZV},\mathrm{ID},\mathrm{SUB}}(\{\mathrm{sum},\mathrm{md}\})$.

Bequemer als die oben eingeführten Operationen sind in ihrer konstruktiven Anwendung die folgenden allgemeinen Operationen auf $\mathbf{FUNK}(A)$:

- Eine Funktion $\psi : A^n \to A$ ist *durch Permutation der Variablen* aus einer Funktion $\varphi : A^n \to A$ entstanden, falls eine Permutation $\pi : \{1,2,\ldots,n\} \to \{1,2,\ldots,n\}$ existiert mit

$$\psi(x_1, \ldots, x_n) = \varphi(x_{\pi(1)}, \ldots, x_{\pi(n)}).$$

- Eine Funktion $\psi : A^{n-1} \to A$ ist *durch Identifizierung von Variablen* aus einer Funktion $\varphi : A^n \to A$ entstanden, falls eine $1 \leq i < j \leq n$ gibt mit

$$\psi(x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n) = \varphi(x_1, \ldots, x_{j-1}, x_i, x_{j+1}, \ldots, x_n).$$

- Eine Funktion $\eta : A^m \to A$ ist *durch simultane Substitution* von $\psi_1, \ldots, \psi_n : A^m \to A$ in $\varphi : A^n \to A$ entstanden, falls

$$\eta(x_1, \ldots, x_m) = \varphi(\psi_1(x_1, \ldots, x_m), \ldots, \psi_n(x_1, \ldots, x_m))$$

  gilt. Mittels $\mathrm{SIM}_n(\varphi, \psi_1, \ldots, \psi_n) =_{\mathrm{def}}$ ist die $(n+1)$-stellige Operation SIM_n auf $\mathbf{FUNK}(A)$ definiert.

Lemma 1.2

1. *Ist die Funktion ψ durch Permutation der Variablen aus der Funktion φ entstanden, so gilt $\psi \in \Gamma_{\mathrm{ZV,LV}}(\{\varphi\})$.*
2. *Ist die Funktion ψ durch Identifizierung von Variablen aus der Funktion φ entstanden, so gilt $\psi \in \Gamma_{\mathrm{ZV,LV,ID}}(\{\varphi\})$.*
3. *Ist die Funktion η durch simultane Substitution von $\psi_1, \ldots, \psi_n$ in φ entstanden, so gilt $\psi \in \Gamma_{\mathrm{ZV,LV,ID,SUB}}(\{\varphi, \psi_1, \ldots, \psi_n\})$.*

Beweis: Wir zeigen die Aussagen einzeln.

1. Jede Permutation kann durch Vertauschung benachbarter Elemente (Transpositionen) erzeugt werden. Für eine Transposition der Variablen x_i und x_{i+1} wenden wir erst $(i+1)$-mal ZV, dann LV und schließlich wieder $(n-i+1)$-mal ZV an.
2. Für die Identifizierung der Variablen x_i und x_j mit $i < j$ wähle zunächst eine Permutation mit $\pi(n-1) = i$ und $\pi(n) = j$ und permutiere geeignet (gemäß der ersten Aussage), wende ID an und tausche Variablen wieder zurück mit einer geeigneten Permutation.
3. Wir bringen in $\varphi(z_1, \ldots, z_n)$ zunächst z_i in die letzte Position durch eine geeignete Permutation und ersetzen z_i mittels SUB durch $\psi_i(x_{i1}, \ldots, x_{im})$, dann identifizieren wir x_{1j} mit x_{ij} für alle $i \in \{2, \ldots, n\}$ und alle $j \in \{1, \ldots, m\}$ und ordnen schließlich die Variablen wieder geeignet um.

Damit ist das Lemma bewiesen. ■

1.3 Induktionsprinzip

Für mit Hilfe algebraischer Hüllenoperatoren erzeugte Mengen lässt sich das Induktionsprinzip formulieren. Ziel ist dabei immer, für eine Menge D nachzuweisen, dass alle Elemente $a \in D$ einer gewissen Eigenschaft E genügen, d.h., $E(a)$ gilt für alle $a \in D$.

1. Form:

Es sei die Menge D durch die Operationen $O_i : A^{s_i} \to A$ aus $B \subseteq A$ erzeugt. Es gilt also $D = \Gamma_{O_1,\ldots,O_k}(B)$. Dann lautet der Induktionsschluss über den Aufbau der Elemente von D wie folgt:

> Gilt
>
> - *Induktionsanfang* (IA): $E(a)$ für alle $a \in B$ und
> - *Induktionsschritt* (IS): aus $E(a_1), \ldots, E(a_{s_i})$ folgt $E(O_i(a_1, \ldots, a_{s_i}))$ für alle $i \in \{1, 2, \ldots, k\}$ und $a_1, \ldots, a_{s_i} \in A$,
>
> so gilt $E(a)$ für alle $a \in D$.

Dieses Prinzip kann auch zur Definition verwendet werden: $D =_{\text{def}} \Gamma_{O_1,\ldots,O_k}(B)$, wobei in (IA) die Menge B und in (IS) die Operationen $O_1 \ldots, O_k$ festgelegt werden.

2. Form:

Es sei $\beta : D \to \mathbb{N}$ eine totale Funktion. Dann lautet der Induktionsschluss über den Parameter β wie folgt:

> Gibt es ein $n_0 \in \mathbb{N}$ mit
>
> - *Induktionsanfang* (IA): für alle $a \in D$ mit $\beta(a) \leq n_0$ gilt $E(a)$ und
> - *Induktionsschritt* (IS): für alle $a \in D$ mit $\beta(a) > n_0$ folgt $E(a)$ aus der Tatsache, dass $E(b)$ für alle $b \in D$ mit $\beta(b) < \beta(a)$ gilt,
>
> so gilt $E(a)$ für alle $a \in D$.

Als Spezialfall der 2. Form erhalten wir die vollständige Induktion mit $D = \mathbb{N}$ und $\beta = \text{id}$.

Algorithmentheorie 2

2.1 Random-Access-Maschinen

Random-Access-Maschinen (RAMs) sind einfache Modelle realer Computer.

Aufgebaut sind RAMs aus folgenden Komponenten:

- Die *Steuereinheit* enthält und führt das *Programm* aus. Ein Programm besteht aus einer Liste von nummerierten *Befehlen*.
- Das *Befehlsregister* enthält die Nummer des aktuellen Befehls im Programm.
- Jedes der unendlich vielen (Daten-)Register $\mathtt{R}0, \mathtt{R}1, \mathtt{R}2, \ldots$ enthält eine natürliche Zahl (gespeichert in dyadischer Darstellung). Die Nummer i von Register $\mathtt{R}i$ heißt *Adresse*.

Verschiedene RAMs unterscheiden sich durch das Programm.

Die Arbeitsweise einer RAM ist wie folgt:

- Sie arbeitet in Takten.
- Sie führt pro Takt genau einen Befehl, und zwar denjenigen, dessen Nummer im Befehlsregister steht.
- Sie stoppt, wenn die Befehlsnummer nicht vorhanden oder der Stoppbefehl erreicht wird.

Bevor wir den zur Verfügung stehenden Befehlssatz angeben, führen wir Bezeichnungen ein, um die Wirkung der Befehle präzise festzulegen:

- $\langle \mathtt{R}i \rangle$ bezeichnet den Inhalt von $\mathtt{R}i$, somit die in $\mathtt{R}i$ gespeicherte Zahl.
- $\langle \mathtt{BR} \rangle$ bezeichnet den Inhalt von $\mathtt{BR}$.
- $\langle \mathtt{R} \rangle := x$ bedeutet, dass das Register $\mathtt{R}$ den neuen Inhalt x bekommt.

Uns steht folgender Befehlssatz zur Verfügung:

Befehl	Wirkung auf Datenregister	Wirkung auf Befehlsregister
Transportbefehle		
$\mathtt{R}i \leftarrow \mathtt{R}j$	$\langle \mathtt{R}i \rangle := \langle \mathtt{R}j \rangle$	$\langle \mathtt{BR} \rangle := \langle \mathtt{BR} \rangle + 1$
$\mathtt{R}i \leftarrow \mathtt{RR}j$	$\langle \mathtt{R}i \rangle := \langle \mathtt{R}\langle \mathtt{R}j \rangle \rangle$	$\langle \mathtt{BR} \rangle := \langle \mathtt{BR} \rangle + 1$
$\mathtt{RR}i \leftarrow \mathtt{R}j$	$\langle \mathtt{R}\langle \mathtt{R}i \rangle \rangle := \langle \mathtt{R}j \rangle$	$\langle \mathtt{BR} \rangle := \langle \mathtt{BR} \rangle + 1$
Arithmetische Befehle		
$\mathtt{R}i \leftarrow k$	$\langle \mathtt{R}i \rangle := k$	$\langle \mathtt{BR} \rangle := \langle \mathtt{BR} \rangle + 1$
$\mathtt{R}i \leftarrow \mathtt{R}j + \mathtt{R}k$	$\langle \mathtt{R}i \rangle := \langle \mathtt{R}j \rangle + \langle \mathtt{R}k \rangle$	$\langle \mathtt{BR} \rangle := \langle \mathtt{BR} \rangle + 1$
$\mathtt{R}i \leftarrow \mathtt{R}j - \mathtt{R}k$	$\langle \mathtt{R}i \rangle := \langle \mathtt{R}j \rangle \dot{-} \langle \mathtt{R}k \rangle$	$\langle \mathtt{BR} \rangle := \langle \mathtt{BR} \rangle + 1$
Sprungbefehle		
$\mathtt{GOTO}\ m$	–	$\langle \mathtt{BR} \rangle := m$
$\mathtt{IF}\ \mathtt{R}i = 0\ \mathtt{GOTO}\ m$	–	$\langle \mathtt{BR} \rangle := \begin{cases} m & \text{falls } \langle \mathtt{R}i \rangle = 0 \\ \langle \mathtt{BR} \rangle + 1 & \text{sonst} \end{cases}$
$\mathtt{IF}\ \mathtt{R}i > 0\ \mathtt{GOTO}\ m$	–	$\langle \mathtt{BR} \rangle := \begin{cases} m & \text{falls } \langle \mathtt{R}i \rangle > 0 \\ \langle \mathtt{BR} \rangle + 1 & \text{sonst} \end{cases}$

Die Befehle $\mathtt{R}i \leftarrow \mathtt{RR}j$ und $\mathtt{RR}i \leftarrow \mathtt{R}j$ verwenden die *indirekte Adressierung*. Dabei wird auf das Register zugegriffen, dessen Nummer Inhalt von Register $\mathtt{R}j$ im ersten Fall bzw. von Register $\mathtt{R}i$ im zweiten Fall ist.

Beispiele: Wir wollen für wichtige Grundoperationen geeignete RAMs konstruieren und diskutieren.

- *Multiplikation*: Wir nehmen an, dass beim Start stets eine Konfiguration mit $\langle \mathtt{BR} \rangle = 0$, $\langle \mathtt{R0} \rangle = x$, $\langle \mathtt{R1} \rangle = y$ sowie $\langle \mathtt{R}i \rangle = 0$ für $i \geq 2$ vorliegt. Die Register R0 und R1 enthalten also die Eingabe. Gewährleisten soll die RAM, dass sie nach endlich vielen Schritten stoppt und $\langle \mathtt{R0} \rangle = x \cdot y$ gilt. Die Inhalte der anderen Register dürfen beliebig sein.
  Die folgende RAM realisiert dies, indem sie die Summe aus y vielen Summanden x bildet:

  0 $\mathtt{R3} \leftarrow 1$
  1 $\mathtt{IF}\ \mathtt{R1} = 0\ \mathtt{GOTO}\ 5$
  2 $\mathtt{R2} \leftarrow \mathtt{R2} + \mathtt{R0}$
  3 $\mathtt{R1} \leftarrow \mathtt{R1} - \mathtt{R3}$
  4 $\mathtt{GOTO}\ 1$
  5 $\mathtt{R0} \leftarrow \mathtt{R2}$
  6 $\mathtt{STOP}$

  Wenden wir beispielsweise diese RAM auf $x = 5$ und $y = 3$ so ergeben sich folgende Registerinhalte während der 16 Takte, die die RAM benötigt, um den Stoppbefehl zu erreichen. In der Tabelle sind dabei die Registerinhalte nach Vollendung des jeweiligen Arbeitstaktes angegeben:

Takt	⟨BR⟩	⟨R0⟩	⟨R1⟩	⟨R2⟩	⟨R3⟩
–	0	5	3	0	0
1	1	5	3	0	1
2	2	5	3	0	1
3	3	5	3	5	1
4	4	5	2	5	1
5	1	5	2	5	1
6	2	5	2	5	1
7	3	5	2	10	1
8	4	5	1	10	1
9	1	5	1	10	1
10	2	5	1	10	1
11	3	5	1	15	1
12	4	5	0	15	1
13	1	5	0	15	1
14	5	5	0	15	1
15	6	15	0	15	1
16	STOP				

- *Exponentiation*: Wir gehen wieder von einer Startkonfiguration $\langle \texttt{BR} \rangle = 0$, $\langle \texttt{R0} \rangle = x$, $\langle \texttt{R1} \rangle = y$ sowie $\langle \texttt{R1} \rangle = 0$ für $i \geq 2$ aus. Für die Zielkonfiguration soll diesmal jedoch $\langle \texttt{R0} \rangle = x^y$ gelten.

  Dies kann realisiert werden, indem wir das Produkt aus y vielen Faktoren x bilden. Das Vorgehen ist also ganz ähnlich wie im ersten Beispiel. Wenn wir für einen Moment annehmen, wir könnten Registerinhalte multiplizieren, so könnte eine RAM wie folgt aussehen:

  0 R3 ← 1
  1 R2 ← 1
  2 IF R1 = 0 GOTO 6
  3 R2 ← R2 · R0
  4 R1 ← R1 − R3
  5 GOTO 2
  6 R0 ← R2
  7 STOP

  Denn Befehl R2 ← R2 · R0 können wir jedoch auf die RAM aus dem ersten Beispiel zurückführen. Durch Einsetzen der entsprechend angepassten RAM erhalten wir folgende RAM zur Berechnung der Exponentialfunktion $\exp(x, y) = x^y$:

```
 0  R3 ← 1
 1  R2 ← 1
 2  IF R1 = 0 GOTO 13
 3  R4 ← R2
 4  R5 ← R0
 5  R6 ← 0
 6  IF R5 = 0 GOTO 10
 7  R6 ← R6 + R4
 8  R5 ← R5 − R3
 9  GOTO 6
10  R2 ← R6
11  R1 ← R1 − R3
12  GOTO 2
13  R0 ← R2
14  STOP
```

- *Maximum*: Wir wollen das Maximum von n Zahlen $x_1, \ldots, x_n$ bestimmen. Die Startkonfiguration sei $\langle \mathtt{BR} \rangle = 0$, $\langle \mathtt{R10} \rangle = x_1$, $\langle \mathtt{R11} \rangle = x_2, \ldots, \langle \mathtt{R}\langle \mathtt{R1} \rangle \rangle = x_n$ sowie $\langle \mathtt{R1} \rangle = n + 9$. Die Inhalte aller anderen Register seien 0. Die Zielkonfiguration soll $\langle \mathtt{R0} \rangle = \max\{x_1, \ldots, x_n\}$ erfüllen.

  Folgende RAM realisiert die Maximum bestimmung, indem der Reihe nach die in den Registern $\langle \mathtt{R10}, \mathtt{R11}, \ldots, \mathtt{R}\langle \mathtt{R1} \rangle$ befindlichen Zahlen mit dem in Register $\langle \mathtt{R0} \rangle$ gespeicherte aktuellen Maximalwert verglichen werden. Dabei fangen wir mit der größten Adresse $\langle \mathtt{R1} \rangle$ an.

```
0  R0 ← 0
1  IF R1 ≤ 9 GOTO 6
2  IF RR1 ≤ R0 GOTO 4
3  R0 ← RR1
4  R1 ← R1 − 1
5  GOTO 1
6  STOP
```

  Wir ersetzen nunmehr die unerlaubte Befehle durch Programmteile, wobei wir für einen Vergleich die Äquivalenz $a \leq b \Leftrightarrow a \dot{-} b = 0$.

0	R0 ← 0	Befehl 0
1	R2 ← 9	Befehl 1
2	R3 ← R1 − R2	
3	IF R3 = 0 GOTO 11	
4	R3 ← RR1	Befehl 2
5	R3 ← R3 − R0	
6	IF R3 = 0 GOTO 8	
7	R0 ← RR1	Befehl 3
8	R2 ← 1	Befehl 4
9	R1 ← R1 − R2	
10	GOTO 1	Befehl 5
11	STOP	Befehl 6

Definition 2.1

1. *Eine Funktion* $\varphi : \mathbb{N}^n \to \mathbb{N}$ *heißt* von einer RAM M berechnet*, falls für alle* $x_1, \ldots, x_n \in \mathbb{N}$ *gilt:*

$$\varphi(x_1, \ldots, x_n) = \begin{cases} \textit{letzter Inhalt von } \langle \mathtt{R0} \rangle & \textit{falls } M \textit{ beim Start mit } \langle \mathtt{BR} \rangle = 0, \\ & \langle \mathtt{R}i \rangle = x_{i+1} \textit{ für } 0 \leq i \leq n-1 \textit{ und} \\ & \langle \mathtt{R}i \rangle = 0 \textit{ für } i \geq n \textit{ nach endlich} \\ & \textit{vielen Schritten hält} \\ \textit{nicht definiert} & \textit{sonst} \end{cases}$$

2. *Eine Funktion* $\varphi : \mathbb{N}^n \to \mathbb{N}$ *heißt* RAM-berechenbar*, falls es eine RAM* M *gibt, die* φ *berechnet.*

3. $\mathbf{RAM} =_{\text{def}} \{\ \varphi \mid \varphi : \mathbb{N}^n \to \mathbb{N}$ *ist RAM-berechenbar* $\}$.

Gemäß obiger Definition berechnet jede RAM für jedes $n \geq 0$ genau eine n-stellige Funktion.

Beispiele: Die erste RAM aus obigem Beispiel berechnet die 2-stellige Funktion prod, die 1-stellige Funktion C_0^1 sowie die 3-stellige Funktion $f(x, y, z) = x \cdot y + z$. Die RAM aus dem zweiten Beispiel berechnet die 2-stellige Funktion exp. Somit gilt $\text{prod}, C_0^1, \text{exp} \in \mathbf{RAM}$.

2.2 Höhere Programmiersprachen

2.2.1 Die Programmiersprache RIES

Syntax von RIES. Im Folgenden legen wir schrittweise die Sprachelemente von RIES fest. Im Einzelnen sind dies der Reihe nach Konstanten, Variablen, Wertausdrücke, Bedingungen, Anweisungen, Funktionen und schließlich Programme.

Konstanten

Eine *Wertkonstante* ist eine nichtleere, endliche Folge von Ziffern `0`, `1`, ..., `9`.

Variablen

Wir unterscheiden zwei Arten von Variablen.

1. Eine *Wertvariable* ist eine endliche Folge von Kleinbuchstaben `a`, `b`, `c`, ..., `z` und von Ziffern `0`, `1`, ..., `9`, die mit einem Buchstaben beginnt. Reserviert und somit nicht zu verwenden sind `begin`, `end`, `function`, `if`, `then`, `else`, `for`, `to`, `do`, `while`, `not`, `and` und `or`.

2. Eine *Feldvariable* ist eine endliche Folge von Kleinbuchstaben `a`, `b`, `c`, ..., `z` und von Ziffern `0`, `1`, ..., `9`, die mit einem Buchstaben beginnt, wobei die oben angegebenen Ausnahmen auch hier gelten, und die von [] gefolgt wird. Ist $i \in \mathbb{N}$ und ist $a[]$ eine Feldvariable, so ist $a[i]$ ein *Feldelement*.

Wertausdrücke

Die Menge der Wertausdrücke wird wie folgt induktiv festgelegt:

- (IA) Wir definieren zunächst die *elementaren Wertausdrücke*.
  1. Ist a eine Wertkonstante, so ist a ein Wertausdruck.
  2. Ist a eine Wertvariable, so ist a ein Wertausdruck.
- (IS) Wir definieren nun die *zusammengesetzten Wertausdrücke*.
  1. Ist $a[]$ eine Feldvariable und ist b ein Wertausdruck, so ist ist $a[b]$ ein Wertausdruck.
  2. Sind a und b Wertausdrücke, so sind auch $(a + b)$, $(a - b)$, $(a * b)$ und $(a : b)$ Wertausdrücke.
  3. Ist f eine Wertvariable und sind $b_0, \ldots, b_n$ Wertausdrücke, so ist $f(b_0, \ldots, b_n)$ ein Wertausdruck. Ein solcher Wertausdruck heißt auch *Funktionsaufruf*.

Bedingungen

Die Menge der *Bedingungen* wird wie folgt induktiv festgelegt:

- (IA) Sind a und b Wertausdrücke, so sind $(a \leq b)$, $(a < b)$, $(a \geq b)$, $(a > b)$, $(a = b)$ und $(a \neq b)$ Bedingungen.
- (IS) Sind a und b Bedingungen, so sind `not`(a), $(a$ `and` $b)$ und $(a$ `or` $b)$ Bedingungen.

Anweisungen

Die Menge der Anweisungen wird wie folgt induktiv festgelegt:

- (IA) Wir definieren zunächst *Wertzuweisungen*.
  1. Ist a eine Wertvariable und ist b ein Wertausdruck, so ist
  $$a := b$$
  eine Wertzuweisung.

2. Ist $a[]$ eine Feldvariable und sind b, c ein Wertausdrücke, so ist

$$a[b] := c$$

eine Wertzuweisung.

- (IS) Wir definieren nun *zusammengesetzte Anweisungen.*

1. Sind $s_1, s_2, \ldots, s_n$ Anweisungen, so ist auch

$$\texttt{begin}\ s_1; s_2; \ldots; s_n\ \texttt{end}$$

eine Anweisung. Eine solche Anweisung heißt auch *Hintereinanderausführung.*

2. Ist b eine Bedingung und sind s_1, s_2 Anweisungen, so ist auch

$$\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2$$

sowie

$$\texttt{if}\ b\ \texttt{then}\ s_1$$

Anweisungen. Solche Anweisungen heißen auch *bedingte Anweisungen.*

3. Ist i eine Wertvariable, sind a_1 und a_2 Wertausdrücke und ist s eine Anweisung, in der Wertzuweisung der Form $i := b$ vorkommt, so ist

$$\texttt{for}\ i := a_1\ \texttt{to}\ a_2\ \texttt{do}\ s$$

eine Anweisung. Eine solche Anweisung heißt `for`-Schleife.

4. Ist b eine Bedingung und ist s eine Anweisung, so ist auch

$$\texttt{while}\ b\ \texttt{do}\ s$$

eine Anweisung. Eine solche Anweisung heißt `while`-Schleife.

Funktionsdeklarationen

Es seien $f, a_1, a_2, \ldots, a_n$ Wertvariablen und s eine Anweisung, in der f nur in der Form $f := \ldots$ (Wertzuweisung) oder $f(\ldots)$ (Funktionsaufruf) vorkommt. Dann ist

$$\texttt{function}\ f(a_1, \ldots, a_n); s$$

die Deklaration der Funktion f. Ein Aufruf einer Funktion innerhalb ihrer eigenen Deklaration heißt *Selbstaufruf.*

Programme

Ein RIES-*Programm* ist eine endliche Folge von Funktionsdeklarationen, wobei jede in einer der Funktionsdeklarationen aufgerufene Funktion genau einmal im Programm deklariert wird und deklarierte Funktionen verschiedener Stellenzahl nicht den gleichen Namen haben dürfen.

Semantik von RIES. Durch die Semantik legen wir fest, welche Funktion von einem RIES-Programm berechnet wird. Dazu muss definiert werden, wie eine Berechnung abläuft. Eine Berechnung ist eine Folge von Veränderungen der in Wertvariablen und Feldelementen gespeicherten natürlichen Zahlen. Formal beschrieben werden diese Inhalte durch eine Abbildung I, die jeder Wertvariablen und jedem Feldelement eine natürliche Zahl zuordnet. Die Abbildung I nennen wir *Interpretation*.

Im Folgenden beschreiben wir, wie sich eine Interpretation entsprechend dem Aufbau eines RIES-Programmes auf komplexere Sprachelemente fortsetzt. Dazu werden wir Interpretationen I auf Wertausdrücken, Bedingungen, Anweisungen und schließlich Programmen definieren.

Zunächst vereinbaren wir eine Notation für einen speziellen Typ von Interpretationen: Für Wertvariablen $a_1, \ldots, a_n$ und natürliche Zahlen $x_1, \ldots, x_n$ bezeichnen wir mit $I^{x_1,\ldots,x_n}_{a_1,\ldots,a_n}$ diejenige Interpretation, die a_i den Wert x_i für $i \in \{1, \ldots, n\}$ und allen anderen Wertvariablen und Feldelementen den Wert 0 zuordnet.

Induktive Fortsetzung von I auf Wertausdrücke:

- (IA) Wir setzen I zunächst auf elementare Wertausdrücke fort.
  1. Ist a eine Wertkonstante, so ist $I(a)$ diejenige natürliche Zahl, deren Dezimaldarstellung durch a beschrieben wird (unter Zulassung von führenden Nullen).
  2. Ist a eine Wertvariable, so ist $I(a)$ bereits festgelegt.
- (IS) Wir setzen I nun auf zusammengesetzte Wertausdrücke fort.
  1. Ist $a[]$ eine Feldvariable und ist b ein Wertausdruck, so ist:
  $$I(a[b]) =_{\text{def}} I(a[I(b)])$$
  2. Sind a und b Wertausdrücke, so ist:
  $$\begin{aligned} I((a+b)) &=_{\text{def}} I(a) + I(b) \\ I((a-b)) &=_{\text{def}} I(a) \dot{-} I(b) \\ I((a*b)) &=_{\text{def}} I(a) \cdot I(b) \\ I((a:b)) &=_{\text{def}} \left\lfloor \frac{I(a)}{I(b)} \right\rfloor \end{aligned}$$
  3. Ist f eine Wertvariable, sind $b_1, \ldots b_n$ Wertausdrücke, und ist die Funktion f deklariert durch `function` $f(a_1, \ldots, a_n); s$, so ist:
  $$I(f(b_1, \ldots, b_n)) =_{\text{def}} \left(I^{I(b_1),\ldots,I(b_n)}_{a_1,\ldots,a_n}\right)_s (f),$$
  wobei die Bedeutung des unteren Index s weiter unten bei der induktiven Fortsetzung auf Anweisungen definiert wird.

Induktive Fortsetzung von I auf Bedingungen:

- (IA) Sind a und b Wertausdrücke und ist $R \in \{\leq, <, \geq, >, =, \neq\}$, so ist:

$$I(a\ R\ b) =_{\text{def}} \begin{cases} 1 & \text{falls } I(a)\ R\ I(b) \\ 0 & \text{sonst} \end{cases}$$

- (IS) Sind a und b Bedingungen, so ist:

$$\begin{aligned} I(\texttt{not}(a)) &=_{\text{def}} 1 \dot{-} I(a) \\ I((a\ \texttt{and}\ b)) &=_{\text{def}} \min\{I(a), I(b)\} \\ I((a\ \texttt{or}\ b)) &=_{\text{def}} \max\{I(a), I(b)\} \end{aligned}$$

Induktive Fortsetzung von I auf Anweisungen:

Wir legen induktiv die Interpretation I_s fest, die durch Ausführung der Anweisung s aus der Interpretation I entsteht. Im Folgenden sei d stets eine Wertvariable oder ein Feldelement.

- (IA) Wir beginnen mit dem Fall, dass s eine Wertzuweisung ist.
  1. Ist a eine Wertvariable und ist b ein Wertausdruck, so ist:

$$I_{a:=b}(d) =_{\text{def}} \begin{cases} I(b) & \text{falls } d = a \\ I(d) & \text{falls } d \neq a \end{cases}$$

     Mit anderen Worten: Die Anweisung $a := b$ bewirkt, dass der bisherige Wert von a durch den Wert $I(b)$ von b ersetzt wird.
  2. Ist $a[]$ eine Feldvariable und sind b und c Wertausdrücke, so ist:

$$I_{a[b]:=c}(d) =_{\text{def}} \begin{cases} I(c) & \text{falls } d = a[I(b)] \\ I(d) & \text{falls } d \neq a[I(b)] \end{cases}$$

     Mit anderen Worten: Die Anweisung $a[b] := c$ bewirkt, dass der bisherige Wert von $a[I(b)]$ durch den Wert $I(c)$ von c ersetzt wird.

- (IS) Wir betrachten den Fall, dass s eine zusammengesetzte Anweisung ist.
  1. Sind $s_1, \ldots, s_n$ Anweisungen, so ist:

$$\begin{aligned} I_{\texttt{begin}\ s_1\ \texttt{end}} &=_{\text{def}} I_{s_1} \\ I_{\texttt{begin}\ s_1;\ldots;s_n\ \texttt{end}} &=_{\text{def}} \left(I_{\texttt{begin}\ s_1;\ldots;s_{n-1}\ \texttt{end}}\right)_{s_n} \end{aligned}$$

     Mit anderen Worten: Die Anweisung $\texttt{begin}\ s_1; \ldots; s_n\ \texttt{end}$ bewirkt die Hintereinanderausführung der Anweisungen $s_1, \ldots, s_n$ in genau dieser Reihenfolge.

2. Ist b eine Bedingung und sind s_1, s_2 Anweisungen, so ist:

$$I_{\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2} =_{\text{def}} \begin{cases} I_{s_1} & \text{falls } I(b) = 1 \\ I_{s_2} & \text{falls } I(b) = 0 \end{cases}$$

$$I_{\texttt{if } b \texttt{ then } s_1} =_{\text{def}} \begin{cases} I_{s_1} & \text{falls } I(b) = 1 \\ I & \text{falls } I(b) = 0 \end{cases}$$

   Mit anderen Worten: Die Anweisung `if` b `then` s_1 `else` s_2 bewirkt die Ausführung von s_1, falls b den Wert 1 hat, und die Ausführung von s_2, falls b den Wert 0 hat. Die Anweisung `if` b `then` s_1 bewirkt die Ausführung von s_1, falls b den Wert 1 hat.

   Ist die Anweisung s gleichzeitig von der Form $s =$ `if` b `then` s_1 `else` s_2 und $s =$ `if` b `then` s_3 (beispielsweise für $s =$ `if` b `then if` b' `then` s_4 `else` s_2), so wird s gemäß der zweiten Form interpretiert. Anders ausgedrückt: Im Zweifelsfall wird `else` immer dem innersten `if` zugeordnet.

3. Ist i eine Wertvariable, sind a_1, a_2 Wertausdrücke und ist s eine Anweisung, in der es keine Wertzuweisungen der Form $i := b$ gibt, so wird festgelegt, dass die Anweisung

   `for` $i := a_1$ `to` a_2 `do` s

   die Wirkung der Anweisung

   ```
   begin
      i := a1; j := a2;
      while (i ≤ j) do begin s; i := (i + 1) end
   end
   ```

   besitzt, wobei j eine im sonstigen Programm nicht vorkommende, neue Wertvariable ist.

4. Ist b eine Bedingung und ist s eine Anweisung, so ist

$$I_{\texttt{while } b \texttt{ do } s} =_{\text{def}} \begin{cases} (I_s)_{\texttt{while } b \texttt{ do } s} & \text{falls } I(b) = 1 \\ I & \text{falls } I(b) = 0 \end{cases}$$

   Mit anderen Worten: Die Anweisung `while` b `do` s bewirkt, dass die Anweisung s so lange ausgeführt wird, bis b den Wert 0 annimmt. Zu beachten ist, dass der Wert $I(b)$ stets vor der Ausführung von s getestet wird. Wird b niemals 0, so bricht der Prozess nicht ab.

   Zur Berechnung des Wertes $I_{\texttt{while } b \texttt{ do } s}(d)$ gehen wir also wie folgt vor: Es sei $I_s^0 =_{\text{def}} I$ und $I_s^k =_{\text{def}} \left(I_s^{k-1}\right)_s$ für $k \geq 1$. Zunächst werden $I_s^0(b), I_s^1(b), I_s^2(b), \ldots$ berechnet, bis ein k mit $I_s^k(b) = 0$ gefunden wird. Dann wird $I_s^k(d)$ berechnet.

Fortsetzung von I auf Programme

Es sei P ein RIES-Programm, dessen erste Funktionsdeklaration `function` $f(a_1, \ldots, a_n); s$ ist. Dann ist die von P berechnete Funktion $f_P : \mathbb{N}^n \to \mathbb{N}$ definiert für alle $x_1, \ldots, x_n \in \mathbb{N}$ durch:

$$f_P(x_1, \ldots, x_n) =_{\text{def}} \begin{cases} (I^{x_1,\ldots,x_n}_{a_1,\ldots,a_n})_s\,(f) & \text{falls dieser Wert definiert ist} \\ \text{nicht definiert} & \text{sonst} \end{cases}$$

Damit ist die Semantik von RIES vollständig festgelegt und wir können den Begriff der RIES-Berechenbarkeit einführen.

Definition 2.2

1. *Eine Funktion* $\varphi : \mathbb{N}^n \to \mathbb{N}$ *heißt* von einem RIES-Programm P berechnet*, falls für alle* $x_1, \ldots, x_n \in \mathbb{N}$ *gilt:*

$$\varphi(x_1, \ldots, x_n) = f_P(x_1, \ldots, x_n)$$

2. *Eine Funktion* $\varphi : \mathbb{N}^n \to \mathbb{N}$ *heißt* RIES-*berechenbar, falls es ein* RIES-*Programm gibt, das* φ *berechnet.*

3. **RIES** $=_{\text{def}} \{\ \varphi \mid \varphi : \mathbb{N}^n \to \mathbb{N}$ *ist* RIES-*berechenbar* $\}$

Beispiele:

1. Die wie folgt definierte, partielle Funktion `di` $: \mathbb{N}^2 \to \mathbb{N}$

$$\texttt{di}(x, y) =_{\text{def}} \begin{cases} \left\lfloor \frac{x}{y} \right\rfloor & \text{falls } y \neq 0 \\ \text{nicht definiert} & \text{sonst} \end{cases}$$

   wird durch foldendes RIES-Programm berechnet:

```
function di(a,b);
begin
   while ((c*b) ≤ a) do c:=(c+1);
   di:=(c-1)
end
```

2. Ein Programm, das aus mehreren Funktionsdeklarationen besteht, dient der Berechnung der n-ten Primzahl $p_n \geq 2$. Dazu werden die folgenden Funktionen benötigt:

$$\begin{aligned} \text{prim}(n) &=_{\text{def}} n\text{-te Primzahl (wobei 2 die 0-te Primzahl ist)} \\ \text{teil}(m) &=_{\text{def}} \text{Anzahl der Teiler von } m \\ \text{mod}(m, i) &=_{\text{def}} \text{Rest der Division von } m \text{ durch } i \end{aligned}$$

   Das Programm berechnet die zuerst deklarierte Funktion, also die Funktion prim:

```
function prim(n);
begin
  m:=3;
  while (n>0) do begin
    if (teil(m)=2) then n:=(n-1);
    m:=(m+1)
  end;
  prim:=(m-1)
end

function teil(m);
begin
  for i:=1 to m do if (mod(m,i)=0) then k:=(k+1);
  teil:=k
end

function mod(m,i);
begin
  while (m≥i) do m:=m-i;
  mod:=m
end
```

3. Wir wollen die FIBONACCI-Zahlen von RIES-Programmen berechnen lassen. Zur Erinnerung: Die Folge der FIBONACCI-Zahlen ist definiert als

$$F_0 =_{\text{def}} 0, F_1 =_{\text{def}} 1 \text{ und } F_n =_{\text{def}} F_{n-1} + F_{n-2} \text{ für } n \geq 2$$

Folgendes Programm berechnet die Funktion `fib` : $\mathbb{N} \to \mathbb{N} : n \mapsto F_n$ mit Hilfe von Selbstaufrufen, also rekursiv:

```
function fib(n);
if (n ≤ 1) then fib:=n else fib:=(fib(n-1)+fib(n-2))
```

Rekursionsbaum für $n = 5$ einfügen ...

Folgendes Programm verwendet keine Selbstaufrufe und vermeidet die wiederholte Berechnung von Funktionswerten durch Speicherung aller Zwischenergebnisse.

```
function fib(n);
begin
  fi[0]:=0;
  fi[1]:=1;
  for i:=2 to n do fi[i]:=(fi[i-1]+fi[i-2]);
  fib:=fi[n]
end
```

Theorem 2.3 RAM $\subseteq$ RIES.

Beweis: Es sei $\varphi : \mathbb{N}^n \to \mathbb{N}$ eine Funktion, die von einer RAM M mit den Befehlen $b_0, b_1, \ldots, b_k$ berechnet wird. Wir dürfen ohne Beeinträchtigung der Allgemeinheit annehmen, dass $b_k =$ `STOP` und $b_i \neq$ `STOP` für alle $i \in \{0, 1, \ldots, k-1\}$ gilt (sonst hängen wir eine neuen Befehl `STOP` als k-ten Befehl an das Programm an und ersetzen alle `STOP`-Befehle durch den Befehl `GOTO` k).

Wir müssen zeigen, dass φ von einem RIES-Programm berechnet wird. Dazu simulieren die RAM M durch folgendes RIES-Programm, das die Wertvariable `br` für den Inhalt des Befehlsregisters `BR` und die Feldvariable `r[]` verwendet, um in Feldelement `r[`a`]` den Inhalt von Register `R`a zu speichern:

```
function phi(x1,x2,...,xn);
begin
   r[0]:=x1;
   r[1]:=x2;
   ⋮
   r[n-1]:=xn;
   while (br < k) do begin
      if (br = 0) then s_0;
      if (br = 1) then s_1;
      ⋮
      if (br = (k-1)) then s_{k-1};
   end;
   phi:=r[0]
end;
```

Hierbei sind $s_0, s_1, \ldots, s_{k-1}$ Anweisungen, die den Befehlen $b_0, b_1, \ldots, b_{k-1}$ entsprechen und folgender Tabelle entnommen werden können:

Befehl b_i	Anweisung s_i
`Ra ← Rb`	`begin r[a]:=r[b]; br:=(br+1) end`
`Ra ← RRb`	`begin r[a]:=r[r[b]]; br:=(br+1) end`
`RRa ← Rb`	`begin r[r[a]]:=r[b]; br:=(br+1) end`
`Ra ← b`	`begin r[a]:=b; br:=(br+1) end`
`Ra ← Rb+Rc`	`begin r[a]:=(r[b]+r[c]); br:=(br+1) end`
`Ra ← Rb-Rc`	`begin r[a]:=(r[b]-r[c]); br:=(br+1) end`
`GOTO a`	`br:=a`
`IF Ra=0 GOTO b`	`if (r[a] = 0) then br:=b else br:=(br+1)`
`IF Ra>0 GOTO b`	`if (r[a] > 0) then br:=b else br:=(br+1)`

Damit ist das Theorem bewiesen. ■

2.2.2 Das Programmiersprachenfragment MINI-RIES

Es ist unser Ziel, auch die umgekehrte Richtung von Theorem **??**, also **RIES** $\subseteq$ **RAM**, zu zeigen. Dazu zeigen wir zunächst die Inklusion für eine eingeschränkte Teilsprache von RIES.

Syntax von MINI-RIES. *Konstanten* und *Variablen* sind uneingeschränkt wie in RIES erlaubt. *Wertausdrücke* und *Bedingungen* sind hingegen nur ohne Funktionsaufrufe sowie nur im Rahmen der nachfolgend induktiv beschriebenen Anweisungen zulässigen Wertzuweisungen erlaubt.

- (IA) Wir legen zunächst die Wertzuweisungen unter den Anweisungen fest:
  1. Ist a eine Wertvariable und ist b eine Konstante, so ist $a := b$ eine Anweisung.
  2. Sind a, b Wertvariablen und ist $c[]$ eine Feldvariable, so sind $c[a] := b$ und $b := c[a]$ Anweisungen.
  3. Sind a, b und c Wertvariablen, so sind $a := (b+c)$ und $a := (b-c)$ Anweisungen.
- (IS) Zusammengesetzte Anweisungen sind wie folgt definiert:
  1. Sind $s_1, \ldots, s_n$ Anweisungen, so ist auch

     $$\texttt{begin}\ s_1; s_2; \ldots; s_n\ \texttt{end}$$

     eine Anweisung.
  2. Ist s eine Anweisung und ist a eine Wertvariable, so ist auch

     $$\texttt{while}\ (a \neq 0)\ \texttt{do}\ s$$

     eine Anweisung.

Funktionsdeklarationen sind wieder uneingeschränkt wie in RIES erlaubt. Eine *Programm* besteht aus genau einer Funktionsdeklaration. (Da in MINI-RIES keine Funktionsaufrufe erlaubt sind, hätte eine weitere Funktionsdeklaration in einem Programm auch keinen Effekt.)

Semantik von MINI-RIES. Jedes MINI-RIES-Programm ist auch ein syntaktisch korrektes RIES-Programm. Folglich überträgt sich die Semantik von RIES auf MINI-RIES. Mithin ist auch festgelegt, wann ein MINI-RIES-Programm eine Funktion berechnet.

Wir können somit die MINI-RIES-Berechenbarkeit definieren.

Definition 2.4 *1. Eine Funktion* $\varphi : \mathbb{N}^n \to \mathbb{N}$ *heißt* MINI-RIES-berechenbar*, falls es ein* MINI-RIES-*Programm* P *gibt, das* φ *berechnet.*

2. **MINI-RIES** $=_{\text{def}} \{\ \varphi \mid \varphi : \mathbb{N}^n \to \mathbb{N}$ *ist* MINI-RIES-*berechenbar* $\}$

Theorem 2.5 $\textbf{MINI-RIES} \subseteq \textbf{RAM}$.

Beweis: Es sei $\varphi : \mathbb{N}^n \to \mathbb{N}$ eine Funktion, die von einem MINI-RIES-Programm P

$$\texttt{function}\ b_n(b_0, \ldots, b_{n-1}); s$$

berechnet wird. In der Anweisung s werden die Wertvariablen $b_0, \ldots, b_{m-1}$ mit $m - 1 \geq n$ sowie die Feldvariablen $a_1[], \ldots, a_k[]$ verwendet.

Wir müssen zeigen, dass φ von einer RAM M berechnet wird. Dazu speichern wir den Wert von b_i im Register $\texttt{R}i$ für $i \in \{0, 1, \ldots, m-1\}$ sowie den Wert von $a_j[\ell]$ im Register $\texttt{R}m + j + k \cdot \ell$ für $j \in \{1, \ldots, k\}$ und $\ell \in \mathbb{N}$. Feldelemente werden also in den Registern $\texttt{R}m + 1$, $\texttt{R}m + 2$, ... gespeichert. Das noch freie Register $\texttt{R}m$ dient als Hilfsregister.

Die Konstruktion der RAM M erfolgt induktiv über den Aufbau von P. Für jede in s vorkommende Anweisung s' konstruieren wir eine RAM $M(s')$, die s' simuliert:

Anweisung s'	RAM $M(s')$
$b_i := r$	$\mathtt{R}i \leftarrow r$
$b_{i_1} := a_j[b_{i_2}]$	$\mathtt{R}m \leftarrow m + j$ $\left.\begin{array}{l}\mathtt{R}m \leftarrow \mathtt{R}m + \mathtt{R}i_2\\ \vdots\\ \mathtt{R}m \leftarrow \mathtt{R}m + \mathtt{R}i_2\end{array}\right\}$ k-mal $\mathtt{R}i_1 \leftarrow \mathtt{RR}m$
$a_j[b_{i_1}] := b_{i_2}$	$\mathtt{R}m \leftarrow m + j$ $\left.\begin{array}{l}\mathtt{R}m \leftarrow \mathtt{R}m + \mathtt{R}i_1\\ \vdots\\ \mathtt{R}m \leftarrow \mathtt{R}m + \mathtt{R}i_1\end{array}\right\}$ k-mal $\mathtt{RR}m \leftarrow \mathtt{R}i_2$
$b_{i_1} := b_{i_2} + b_{i_3}$	$\mathtt{R}i_1 \leftarrow \mathtt{R}i_2 + \mathtt{R}i_3$
$b_{i_1} := b_{i_2} - b_{i_3}$	$\mathtt{R}i_1 \leftarrow \mathtt{R}i_2 - \mathtt{R}i_3$
`begin` $s_1; \ldots; s_\ell$ `end`	$M(s_1)$ $M'(s_2)$ $\vdots$ $M'(s_\ell)$
`while` $(b_i \neq 0)$ `do` s''	`IF` $\mathtt{R}i = 0$ `GOTO` $r + 2$ $M'(s'')$ `GOTO` 0 $\mathtt{R}m \leftarrow 0$

Die RAMs M' und M'' stimmen mit den jeweiligen konstruierten RAMs M überein bis auf die neuen Befehlsnummern, die sich entsprechend der Aneinanderreihung der RAMs ergeben. Außerdem nehmen wir an, dass das Programm der RAM $M(s'')$ aus genau r Befehlen besteht.

Damit ist das Theorem bewiesen. ■

2.2.3 Der RIES-Compiler

Theorem 2.6 RIES $\subseteq$ MINI-RIES.

Beweis: Wir zeigen, wie aus RIES-Programmen alle in MINI-RIES nicht zugelassenen Teile eliminiert werden können. Dies führen wir in den folgenden Einzelschritten aus, wobei die Reihenfolge der Schritte wesentlich ist:

1. Elimination von `for`-Schleifen
2. Elimination von `if-then-else`-Anweisungen
3. Elimination von `if-then`-Anweisungen
4. Elimination von Bedingungen, die nicht von der Form $a \neq 0$ mit einer Wertvariablen a sind
5. Elimination von Funktionsaufrufen, die nicht von der Form $a := f(c_0, \ldots, c_n)$ mit einer Wertvariablen a sind
6. Elimination von Funktionsaufrufen (unter Einführung zweidimensionaler Feldvariablen $a[][]$ und Feldelemente $a[i][j]$)
7. Elimination zweidimensionaler Feldvariablen
8. Elimination von Wertausdrücken $a[b]$ mit einer Feldvariablen $a[]$, wobei b keine Wertvariable ist
9. Elimination von Wertausdrücken $a[b]$, die nicht von der Form $a[b] := c$ oder $c := a[b]$ mit einer Feldvariablen $a[]$ und mit Wertvariablen b, c sind
10. Elimination von Wertzuweisungen $a := (c \circ d)$ mit $\circ \in \{+, -, *, :\}$, in denen c oder d keine Wertvariable ist
11. Elimination von Wertzuweisungen $a := (c : d)$ mit Wertvariablen c, d
12. Elimination von Wertzuweisungen $a := (c * d)$ mit Wertvariablen c, d
13. Elimination von Wertzuweisungen $a := b$ mit einer Wertvariablen b

Wir behandeln die einzelnen Schritte wie folgt:

1. Ersetze alle Vorkommen von

   `for` $i := a_1$ `to` a_2 `do` s

   äquivalent durch

   ```
   begin
      i := a1;
      j := a2;
      while (i ≤ j) do begin
         s;
   ```

```
      i := (i + 1)
    end
end
```

2. Ersetze alle Vorkommen von

   `if` b `then` s_1 `else` s_2

   äquivalent durch

```
begin
  d := 0;                                  /* d ist eine neue Wertvariable */
  if b then begin
    d := 1;
    s_1
  end;
  if (d = 0) then s_2
end
```

3. Ersetze alle Vorkommen von

   `if` b `then` s

   äquivalent durch

```
begin
  d := 0;                                  /* d ist eine neue Wertvariable */
  while (b and (d = 0)) do begin
    d := 1;
    s
  end
end
```

4. Nach den bisherigen Eliminationsschritte treten Bedingungen nur noch in `while`-Schleifen auf. Ersetze alle Vorkommen von b im Kontext von `while`-Schleifen

   `while` b `do` s

   äquivalent durch

```
begin
  d := D(b);                               /* d ist eine neue Wertvariable */
  while (d ≠ 0) do begin
    s;
```

```
    d := D(b)
  end
end
```

mit dem Wertausdruck $D(b)$, der wie folgt induktiv definiert ist:

- *Induktionsanfang*: Wir setzen für elementare Bedingungen

$$\begin{aligned} D((a > c)) &=_{\text{def}} (a - c) \\ D((a \leq c)) &=_{\text{def}} (1 - (a - c)) \\ D((a < c)) &=_{\text{def}} (c - a) \\ D((a \geq c)) &=_{\text{def}} (1 - (c - a)) \\ D((a \neq c)) &=_{\text{def}} ((a - c) + (c - a)) \\ D((a = c)) &=_{\text{def}} (1 - ((a - c) + (c - a))) \end{aligned}$$

- *Induktionsschritt*: Für zusammengesetzte Bedingungen setzen wir

$$\begin{aligned} D(\texttt{not}(b)) &=_{\text{def}} (1 - D(b)) \\ D((b_1 \texttt{ and } b_2)) &=_{\text{def}} (D(b_1) * D(b_2)) \\ D((b_1 \texttt{ or } b_2)) &=_{\text{def}} (D(b_1) + D(b_2)) \end{aligned}$$

Wie leicht entlang des induktiven Aufbaus einer Bedingung zu überprüfen ist, gilt für alle Bedingungen b stets $I(D(b)) > 0 \Longleftrightarrow I(b) = 1$.

5. Nach den bisherigen Eliminationsschritten kommen Funktionsaufrufe nicht mehr in Bedingungen vor, sondern nur noch in Wertzuweisungen $a := b$. Kommt der Funktionsaufruf $f(c_0, \ldots, c_n)$ in b vor, so ersetze ihn äquivalent durch

```
begin
  d := f(c_0, ..., c_n);          /* d ist eine neue Wertvariable */
  a := e
end
```

wobei e ein neuer Wertausdruck ist, bei dem d an Stelle von $f(c_0, \ldots, c_n)$ steht. Gegebenenfalls muss die Ersetzung iteriert werden.

6. Ohne Beeinträchtigung der Allgemeinheit dürfen wir annehmen, dass die Funktion $f_1 : \mathbb{N}^{n_1} \to \mathbb{N}$ durch ein RIES-Programm

```
function f_1(b_1, ..., b_{n_1}); s_1
function f_2(b_1, ..., b_{n_2}); s_2
⋮
function f_k(b_1, ..., b_{n_k}); s_k
```

berechnet werde. Wir definieren $n =_{\text{def}} \max\{n_1, \ldots, n_k\}$ und deklarieren eine $(n+1)$-stellige Funktion f

```
function f(b_0, b_1, ..., b_n); s
```

mit der Anweisung s

```
begin
   if (b_0 = 1) then A(s_1);
   if (b_0 = 2) then A(s_2);
   ⋮
   if (b_0 = k) then A(s_k)
end
```

Dabei entstehen die Anweisungen $A(s_i)$ aus s_i durch Ersetzung von $f_j(c_1, \ldots, c_j)$ durch $f(j, c_1, \ldots, c_{n_j}, 0, \ldots, 0)$ sowie durch Ersetzung von f_i auf der linken Seite einer Wertzuweisung durch f.

Wir müssen nunmehr nur noch zeigen, wie wir für eine Funktionsdeklaration

```
function f(b_0, b_1, ..., b_n); s
```

bei der nur noch Selbstaufrufe von f vorkommen, alle Selbstaufrufe komplett eliminieren können. Wir müssen also s durch eine Anweisung s' ersetzen, in der kein Selbstaufruf mehr vorkommt. Gelingt uns dies, so erhalten wir eine äquivalente Deklaration für die Funktion f_1 vermöge

```
function f_1(b_1, ..., b_{n_1});
begin
   b_0 := 1;
   s';
   f_1 := f;
end
```

Dieses Programm enthält dann keine Funktionsaufrufe mehr.

Es seien $b_0, b_1, \ldots, b_n, b_{n+1}, \ldots, b_m$ und f alle in der Anweisung s vorkommenden Wertvariablen sowie $a_1[], \ldots, a_k[]$ alle Feldvariablen. Die Idee zur Eliminierung der Selbstaufrufe besteht darin, bei jedem Aufruf von f die aktuelle Berechnung zu unterbrechen, um den Rückgabewert des Funktionsaufruf zu berechnen. Ist der Rückgabewert bekannt, so wird in der aktuellen Berechnung fortgefahren. Dieses Vorgehen führt im Allgemeinen zu einer Reihe ineinander geschachtelter Berechnunge, die wir durchnummerieren und die alle ihren eigene Menge an Variablen besitzen. Beispielhaft kann die Abarbeitung einer rekursiven Berechnung mit Selbstaufrufen wie folgt vereinfacht dargestellt werden:

Rekursionsbaum hier einfügen ...

Zur Umsetzung dieser Idee versehen wir zunächst alle Variablen mit einer weiteren Dimension, um die Werte in den einzelnen Berechnungen zu halten. Wir haben somit die eindimensionalen Feldvariablen $b_0[], \ldots, b_m[]$ und $f[]$ sowie die eigentlich nicht zulässigen zweidimensionalen Feldvariablen $a_1[][], \ldots, a_k[][]$. Weiterhin erhalten alle in s vorkommenden Wertzuweisungen eine eindeutige Nummer $1, \ldots, r$ in der Reihenfolge ihres Aufretens. Mit $\alpha(s')$ bzw. $\beta(s')$ bezeichnen wir die kleinste bzw. größte Nummer einer in s' vorkommenden Wertzuweisung.

Wir verwenden Variablen und Feldelemente der folgenden drei Kategorien, um die Berechnungen konsistent miteinander zu verknüpfen:

Datenvariablen

- $b_j[i]$ ist der akutelle Wert von b_j in der i-ten Berechnung
- $f[i]$ ist der akutelle Wert von f in der i-ten Berechnung
- $a_j[\ell, i]$ ist der akutelle Wert von $a_j[\ell]$ in der i-ten Berechnung

Navigationsvariablen

- $rs[i]$ ist die Rücksprungstelle (Nummer der Wertzuweisung), bei der die i-te Berechnung nach Unterbrechung durch einen Funktionsaufruf fortgesetzt werden muss; ohne Unterbrechung gilt $rs[i] = 0$
- $\ell[i]$ ist die letzte aufgerufene Berechnung während der i-ten Berechnung; ohne Aufruf gilt $\ell[i] = 0$
- $t[i]$ ist die Nummer der Berechnung, mit der nach Beendigung der i-ten Berechnung fortgefahren werden muss; es gilt $t[1] = 0$

Steuerungsvariablen

- i ist die aktuelle Berechnungsnummer
- sr ist die aktuelle Nummer einer Wertzuweisung, bei der weitergerechnet wird; als Werte sind auch 0 und $r + 1$ zulässig, wobei $sr = 0$ bedeutet, dass alle Anweisungen ausgeführt werden, und $sr = r+1$ bedeutet, dass keine Anweisung mehr ausgeführt wird.
- p ist die kleinste freie Berechnungsnummer

Diese Variablen werden in der folgenden Rahmendeklaration zusammengeführt, die die prinzipielle Logik der Simulation beschreibt (Rahmendeklaration deshalb, weil die Blöcke $B(s)$ noch geeignet eingesetzt werden müssen):

```
function f(b_0, b_1, ..., b_n);
begin
    i := 1;
    p := 2;
    b_0[1] := b_0;
    ⋮
```

```
    b_n[1] := b_n;
    while (i > 0) do begin
       sr := rs[i];
       B(s);                                  /* Anweisungsblock, der s ersetzt */
       if (sr = 0) then                       /* i-te Berechnung abgeschlossen */
          i := t[i]
       else begin          /* es gilt sogar sr = r + 1, d.h., i-te Berechnung
                              wird unterbrochen und neuer Funktionsaufruf
                              mit Berechnungsnummer p simuliert */
          i := p;
          p := (p + 1)
       end
    end;
    f := f[1];
end
```

Es bleibt noch die Definition der Anweisungsblöcke $B(s')$, die eine Anweisung s' ersetzen. Dies führen wir induktiv über den Aufbau von Anweisungen durch. Zur Vereinfachung benutzen wir dabei folgende Notation: Ist c Wertausdruck, so sei $(c)_i$ derjenige Wertausdruck, der aus c entsteht, indem alle Vorkommen von b_j, f und $a_j[\ell]$ durch $b_j[i]$, $f[i]$ und $a_j[\ell, i]$ ersetzt werden.

- *Induktionsanfang*: Sind $c, c_0, \ldots, c_n, d$ Wertausdrücke ohne Funktionsaufrufe, so definieren wir

  $$B(d := c) =_{\text{def}} \texttt{if } (sr \leq \alpha(d := c)) \texttt{ then } (d)_i := (c)_i$$

  sowie

  $$B(d := f(c_0, \ldots, c_n)) =_{\text{def}}$$

  ```
  begin
     if (sr < α(d := f(c_0, ..., c_n))) then begin
        rs[i] := α(d := f(c_0, ..., c_n));
        ℓ[i] := p;
        t[p] := i;
        b_0[p] := (c_0)_i;
        ⋮
        b_n[p] := (c_n)_i;
        rs[p] := 0;
        sr := (r + 1)
     end;
     if (sr = α(d := f(c_0, ..., c_n))) then begin
        d[i] := f[ℓ[i]];
        sr := 0;
     end
  ```

```
end
```

- *Induktionsschritt*: Sind $s_1, \ldots, s_n$ Anweisungen und d eine Wertvariable, so definieren wir

  $$B(\texttt{begin } s_1; \ldots; s_n \texttt{ end}) =_{\text{def}} \texttt{begin } B(s_1); \ldots; B(s_n) \texttt{ end}$$

  sowie

  $$B(\texttt{while } (d \neq 0) \texttt{ do } s_1) =_{\text{def}}$$

  ```
  begin
     if (sr ≥ α(s1)) then B(s1);
     while ((d ≠ 0) and (sr ≤ β(s1))) do B(s1)
  end
  ```

  Um die wiedereingeführten Programmteile (`if-then-else` und `if-then`) zu eliminieren, führen wir die Schritte 1. bis 5. nochmals aus. Dabei werden keine Funktionsaufrufe eingeführt.

7. Um zweidimensionale Feldvariablen zu eliminieren, verwenden wir eine bijektive Funktion $f : \mathbb{N}^2 \to \mathbb{N}$. Dann können wir alle Vorkommen von $a[i][j]$ durch $a[f(i,j)]$ ersetzen. Als Bijektion können wir beispielsweise die Funktion

   $$f : \mathbb{N}^2 \to \mathbb{N} : (i,j) \mapsto \frac{(i+j)^2 + i + 3j}{2}$$

   verwenden. Somit ersetzen wir $a[i,j]$ durch $a[((((i+j)*(i+j))+(i+(3*j))):2)]$.

8. Nach den bisherigen Eiminationsschritten kommt ein Wertausdruck $a[b]$ nur noch in Wertzuweisungen s vor. Wir ersetzen s äquivalent durch

   ```
   begin
      c := b;                          /* c ist eine neue Wertvariable */
      s'
   end
   ```

   wobei s' aus s entsteht, indem $a[b]$ durch $a[c]$ ersetzt wird. Gegebenenfalls müssen wir die Ersetzung iterieren.

9. Ersetze alle Vorkommen von

   $$a[b] := c$$

   in denen c keine Wertvariable ist, äquivalent durch

   ```
   begin
      d := c;                          /* d ist eine neue Wertvariable */
   ```

```
    a[b] := d
end
```

Kommt $a[b]$ in $e := f$ auf der rechten Seite vor, so ersetze $e := f$ durch

```
begin
    g := a[b];                          /* g ist eine neue Wertvariable */
    e := f'
end
```

wobei f' aus f entsteht, indem $a[b]$ durch g ersetzt wird. Gegenenfalls müssen wir die Ersetzung iterieren.

10. Ersetze alle Vorkommen von

    $a := (c \circ d)$

    äquivalent durch

```
begin
    e := c;                             /* e ist eine neue Wertvariable */
    f := d;                             /* f ist eine neue Wertvariable */
    a := (e ∘ f)
end
```

    Gegebenenfalls muss iteriert werden.

11. Ersetze alle Vorkommen von

    $a := (c : d)$

    äquivalent durch

```
begin
    f := 1;
    c' := (c + f);
    g := (f − d);                   /* Test, ob d = 0; falls ja, rechne mit d = 1 */
    d' := (d + g);
    g := 0;
    while (c' ≠ 0) do begin
        c' := (c' − d');
        g := (g + f);
    end;
    a := (g − f)
end
```

    wobei c', d', f und g neue Wertvariablen sind.

12. Ersetze alle Vorkommen von

    $$a := (c * d)$$

    äquivalent durch

    ```
    begin
       e := d;
       f := 1;
       g := 0;
       while (e ≠ 0) do begin
          g := (g + c);
          e := (e − f);
       end;
       a := g
    end
    ```

    wobei e, f und g neue Wertvariablen sind.

13. Ersetze alle Vorkommen von

    $$a := b$$

    äquivalent durch

    ```
    begin
       c := 0;                          /* c ist ein neue Wertvariable */
       a := (b + c)
    end
    ```

Damit ist das Theorem bewiesen. ■

Mit den Theoremen 2.5, 2.3 und 2.6 haben wir somit die Inklusionskette

$$\textbf{MINI-RIES} \subseteq \textbf{RAM} \subseteq \textbf{RIES} \subseteq \textbf{MINI-RIES}$$

gezeigt. Folglich wissen wir, dass Gleichheit zwischen diesen Klassen besteht:

$$\textbf{RAM} = \textbf{RIES} = \textbf{MINI-RIES}$$

2.3 Turingmaschinen

Turingmaschinen sind ein einfaches Modell für den „rechnenden Menschen“. Benannt sind sie nach Alan Mathieson Turing, der sie 1936 eingeführt hat.

Eine Turingmaschine besteht aus folgenden Komponenten:

- k (mit $k \geq 1$) beidseitig unendliche, in Zellen unterteilte *Bänder*; in jeder Zelle steht genau ein Symbol aus dem *Bandalphabet* Σ, wobei das Leerzeichen $\square \in \Sigma$ zeigt an, dass in der Zelle eigentlich nichts steht
- für jedes Band ein *Schreib- und Lesekopf* (oder kurz einfach *Kopf*), der sich von Zelle zu Zelle bewegen kann, um Inhalte der Zellen zu lesen und zu verändern
- eine *Steuereinheit*, die sich in einem Zustand z aus einer endlichen Zustandsmenge Z befindet, die über die Köpfe Bandinhalte liest und die Aktivitäten entsprechend steuert

Schema wird noch eingefügt ...

Die Arbeitsweise einer Turingmaschine ist wie folgt festgelegt:

- eine Turingmaschine arbeitet taktweise
- das Verhalten pro Takt wird durch eine (totale) *Überführungsfunktion*

  $$f : Z \times \Sigma^k \to Z \times \Sigma^k \times \{L, 0, R\}^k$$

  beschrieben; die Beschreibung ist unabhängig von Taktnummer und Kopfpositionen
- pro Takt geht die Turingmaschine von einem aktuellen Zustand $z \in Z$ und k gelesenen Bandsymbolen $a_1, \ldots, a_k \in \Sigma$ in einen neuen Zustand $z' \in Z$ über, schreibt k Bandsymbole $a'_1, \ldots, a'_k \in \Sigma$ auf die Bänder und bewegt die Köpfe mit den Kopfbewegungen $\sigma_1, \ldots, \sigma_k \in \{L, 0, R\}$ um höchstens eine Zelle; formal

  $$f(z, a_1, \ldots, a_k) = (z', a'_1, \ldots, a'_k, \sigma_1, \ldots, a_k),$$

  wobei für Kopf i
  - $\sigma_i = L$ eine Kopfbewegung nach links,
  - $\sigma_i = 0$ keine Kopfbewegung und
  - $\sigma_i = R$ eine Kopfbewegung nach rechts

  bedeutet.
- die Turingmaschine beginnt stets im *Startzustand* $z_0 \in Z$ und hält stets beim ersten Erreichen des *Stoppzustandes* $z_1 \in Z$ an

Damit können Turingmaschinen formal eingeführt werden: Eine *Turingmaschine* ist ein Quintupel $M = (\Sigma, Z, f, z_0, z_1)$, wobei

- Σ das Bandalphabet,

- Z die Zustandsmenge,
- f die Überführungsfunktion,
- z_0 der Startzustand und
- z_1 der Stoppzustand

ist. Die Überführungsfunktion notieren wir auch in Form von *Befehlen*; an Stelle von $f(z, a_1, \ldots, a_k) = (z', a'_1, \ldots, a'_k)$ verwenden wir die Schreibweise:

$$za_1 \ldots a_k \to z'a'_1, \ldots a'_k\sigma_1 \ldots \sigma_k$$

Befehle, bei denen der Stoppzustand z_1 auf der linken Seite vorkommt, werden üblicherweise nicht angegeben, da sie keinen Einfluss auf den Ablauf besitzen. Um der Totalität der Überführungsfunktion genüge zu tun, können die Befehle beliebig ergänzt werden.

Beispiele:

1. *Inkrementierung*: Die Turingmaschine soll 1 zu einer Zahl addieren, die in dyadischer Darstellung gegeben ist. Dabei steht der Kopf zu Beginn auf dem ersten Symbol (von links). Beim Stopp soll der Kopf auf dem ersten Symbol von links der Reslutates stehen. Die Inkrementierung dyadischer Zahlen unterscheidet sich nicht von der üblichen Inkrementierung binärer Zahlen. Das Bandalphabet ist also $\Sigma = \{1, 2, \square\}$. Als Zustandsmenge verwenden wir $Z = \{z_0, z_1, z_2, z_3\}$ mit den folgenden Bedeutungen:
   - z_0 steht für: Start, Bewegung nach rechts
   - z_1 steht für: Stopp
   - z_2 steht für: Übertrag 0
   - z_3 steht für: Übertrag 1

   Die Überführungsfunktion wird durch folgendes Programm beschrieben:

$$\begin{array}{lll} z_0 1 \to z_0 1R & z_2 1 \to z_2 1L & z_3 1 \to z_2 2L \\ z_0 2 \to z_0 2R & z_2 2 \to z_2 2R & z_3 2 \to z_3 1L \\ z_0\square \to z_3\square L & z_2\square \to z_1\square R & z_3\square \to z_1 10 \end{array}$$

2. *Palindrome*: Die Turingmaschine soll testen, ob ein Wort $w \in \{a, b\}^*$ ein Palindrom ist, d.h., ob $w = w^R$ gilt. Dies gilt beispielsweise für *abba* oder *babbabbab*, aber nicht für *abab*. Das Wort w soll dabei gelöscht werden. Ist w ein Palindrom, so soll a auf dem Band stehen, anderenfalls b. Der Kopf steht zu Beginn auf dem ersten Symbol (von links) und beim Stopp auf dem Ergebnisbuchstaben. Die Konstruktionsidee für die Turingmaschine besteht darin, immer die beiden äußeren Buchstaben zu vergleichen und zu löschen. Das Bandalphabet ist $\Sigma = \{a, b, \square\}$. Als Zustandsmenge verwenden wir $Z = \{z_a, z_b, z'_a, z'_b, z_0, z_1, z_2, z_3\}$ mit folgenden Bedeutungen:
   - z_a steht für: a gemerkt, gehe nach rechts

- z_b steht für: b gemerkt, gehe nach rechts
- z'_a steht für: ein Schritt nach links, teste auf a
- z'_b steht für: ein Schritt nach links, teste auf b
- z_0 steht für: Start, lese ersten Buchstaben
- z_1 steht für: Stopp
- z_2 steht für: Test positiv, gehe nach links
- z_3 steht für: Test negativ, gehe nach links, lösche und stoppe

Die Überführungsfunktion wird durch folgendes Programm bechrieben:

$$
\begin{array}{lll}
z_a a \to z_a a R & z'_a a \to z_2 \square L & z_0 a \to z_a \square R \\
z_a b \to z_a b R & z'_a b \to z_3 \square L & z_0 b \to z_b \square R \\
z_a \square \to z'_a \square L & z'_a \square \to z_1 a 0 & z_0 \square \to z_1 a 0 \\
z_b a \to z_b a R & z'_b a \to z_3 \square L & z_2 a \to z_2 a L \\
z_b b \to z_b b R & z'_b b \to z_2 \square L & z_2 b \to z_2 b L \\
z_b \square \to z'_b \square L & z'_b \square \to z_1 a 0 & z_2 \square \to z_0 \square R \\
 & & z_3 a \to z_3 \square L \\
 & & z_3 b \to z_3 \square L \\
 & & z_3 \square \to z_1 b 0
\end{array}
$$

Für Funktionsberechnungen legen wir spezifische Start- und Endsituation für eine k-Band-Turingmaschine $M = (\Sigma, Z, f, z_0, z_1)$ fest. Für $z \in Z$, $a_1, \ldots, a_m \in \Sigma \setminus \{\square\}$ bezeichne $S(z, a_1 a_2 \ldots a_m)$ folgende Situation:

Abbildung wird noch eingefügt ....

Definition 2.7

1. *Es seien* $M = (\Sigma, Z, f, z_0, z_1)$ *eine Turingmaschine,* $\Sigma_1 \subseteq \Sigma \setminus \{\square, *\}$, $\Sigma_2 \subseteq \Sigma \setminus \{\square\}$, $n \in \mathbb{N}$. *Eine Funktion* $\varphi : (\Sigma_1^*)^n \to \Sigma_2^*$ *heißt* von der Turingmaschine M berechnet, *falls für alle* $x_1, \ldots, x_n \in \Sigma_1^*$ *gilt:*

   (a) *Ist* $\varphi(x_1, \ldots, x_n)$ *definiert, so gilt: Startet* M *in* $S(z_0, x_1 * x_2 * \ldots * x_n)$, *so stoppt* M *in* $S(z_1, \varphi(x_1, \ldots, x_n))$.

   (b) *Ist* $\varphi(x_1, \ldots, x_n)$ *nicht definiert, so gilt: Startet* M *in* $S(z_0, x_1 * x_2 * \ldots * x_n)$, *so stoppt* M *nicht oder* M *stoppt nicht in einer Situation der Form* $S(z_1, y)$ *mit* $y \in (\Sigma \setminus \{\square\})^*$.

2. *Eine Funktion* $\varphi : (\Sigma_1^*)^n \to \Sigma_2^*$ *heißt* Turing-berechenbar, *falls es eine Turingmaschine gibt, die* φ *berechnet.*

3. *Eine Funktion* $\varphi : \mathbb{N}^n \to \mathbb{N}$ *heißt* Turing-berechenbar, *falls sie in dyadischer Kodierung Turing-berechenbar ist, d.h. falls die durch*

$$\psi(x_1, \ldots, x_n) =_{\text{def}} \text{dya}\left(\varphi(\text{dya}^{-1}(x_1), \ldots, \text{dya}^{-1}(x_n))\right)$$

   definierte Funktion $\psi : (\{1,2\}^*)^n \to \{1,2\}^*$ *Turing-berechenbar ist.*

4. $\mathbf{TM} =_{\text{def}} \{\ \varphi \mid \varphi : \mathbb{N}^n \to \mathbb{N}$ *ist Turing-berechenbar* $\}$

Wir wollen zwei Bemerkungen zu obiger Definition nachtragen. Erstens ist das Verhalten von Turingmaschinen nicht festgelegt für Startsituationen, die von der Form $S(z_0, x_1 * \ldots * x_n)$ abweichen. Für die Funktion φ ist dies jedoch ohne Bedeutung. Und zweitens berechne die Turingmaschine $M = (\Sigma, Z, f, z_0, z_1)$ für alle $n \in \mathbb{N}$ und $\Sigma_1 \subseteq \Sigma \setminus \{\square, *\}$ eine Funktion $\varphi : (\Sigma_1^*)^n \to (\Sigma \setminus \{\square\})^*$.

> **Beispiel:** Die Turingmaschine für die Inkrementierung aus obigem Beispiel zeigt $\mathrm{S} \in \mathbf{TM}$ für $\mathrm{S} : \mathbb{N} \to \mathbb{N} : n \mapsto n + 1$.

Wir wollen zeigen, dass Turingmaschinen nicht mehr und nicht weniger berechnen können, als wir mit RAMs oder mit RIES-Programmen berechnen können. Als Zwischenschritt führen wir dazu zunächst 1-Turingmaschinen als eingeschränktes Modell von Turingmaschinen ein.

Eine 1-*Turingmaschine* besitzt nur ein *Halbband* (ein einseitig unendliches Band).

Abbildung hier einfügen ...

Da links vom Bandanfang keine Zellen vorhanden sind, wird ein Befehl $za \to z'a'L$ auf der Bandanfangszelle wie $za \to z'a'0$ ausgeführt. Die oben dargestellte Situation wird mit $S(z, a_1 \ldots, a_n)$ beschrieben, falls sich die 1-Turingmaschine im Zustand $z \in Z$ befindet.

Die Begriffsbildungen aus Definition 2.7 übertragen sich auf 1-Turingmaschinen. Damit ist festgelegt, ohne dass wir die Definitionen explizit anführen, was es bedeutet, wenn wir von Folgendem sprechen:

1. Eine Funktion $\varphi : (\Sigma_1^*)^n \to \Sigma_2^*$ wird *von einer* 1-*Turingmaschine berechnet.*
2. Eine Funktion $\varphi : (\Sigma_1^*)^n \to \Sigma_2^*$ ist 1-*Turing-berechenbar.*
3. Eine Funktion $\varphi : \mathbb{N}^n \to \mathbb{N}$ ist 1-*Turing-berechenbar.*
4. **1-TM** $=_{\text{def}} \{ \varphi \mid \varphi : \mathbb{N}^n \to \mathbb{N}$ ist 1-Turing-berechenbar $\}$

Lemma 2.8 *Es sei $n \in \mathbb{N}$, und es sei $\varphi : \mathbb{N}^n \to \mathbb{N}$ eine Turing-berechenbare bzw.* 1-*Turing-berechenbare Funktion. Dann gibt es eine Turingmaschine bzw.* 1-*Turingmaschine $M = (\Sigma, Z, f, z_0, z_1)$, die φ berechnet und für die gilt: Ist $\varphi(x_1, \ldots, x_n)$ für $x_1, \ldots, x_n \in \Sigma^*$ nicht definiert, so hält M bei Start in der Situation $S(z_0, x_1 * \ldots * x_n)$ nicht.*

Beweis: Übungsaufgabe. ∎

Theorem 2.9 TM $\subseteq$ 1-TM.

Beweis: Es sei φ eine Turing-berechenbare Wortfunktion, d.h., $\varphi : (\Sigma_1^*)^n \to \Sigma_2^*$ wird durch eine k-Band-Turingmaschine berechnet. Ohne Beeinträchtigung der Allgemeinheit stoppt M nicht, wenn $\varphi(x_1, \ldots, x_n)$ nich definiert ist (siehe Lemma 2.8). Wir konstruieren eine 1-Turingmaschine M', die M simuliert und dabei folgende Standardsituation nach jedem Simulationsschritt auf dem Halbband aufrecht erhält

$$\#a_1^1 a_2^1 \ldots \nabla a_{i_1}^1 \ldots a_{r_1}^1 \diamond a_1^2 a_2^2 \ldots \nabla a_{i_2}^2 \ldots a_{r_2}^2 \diamond \ldots \diamond a_1^k a_2^k \ldots \nabla a_{i_k}^k \ldots a_{r_k}^k \#\square\square\square \ldots$$

Hierbei sind $\#, \nabla$ und $\diamond$ neue Symbole, die im Bandalphabet von M nicht vorkommen. Die Folge von Bandinhalten entspricht der folgenden Situation von M nach einem Schritt:

$$\begin{array}{lll}
\ldots\square\square\square a_1^1 a_2^1 \ldots \overset{\nabla}{a_{i_1}^1} \ldots a_{r_1}^1 \square\square\square \ldots & & \text{Band 1} \\
\ldots\square\square\square a_1^2 a_2^2 \ldots \overset{\nabla}{a_{i_2}^2} \ldots a_{r_2}^2 \square\square\square \ldots & & \text{Band 2} \\
\vdots & & \\
\ldots\square\square\square a_1^k a_2^k \ldots \overset{\nabla}{a_{i_k}^k} \ldots a_{r_k}^k \square\square\square \ldots & & \text{Band } k
\end{array}$$

M' arbeitet in Phasen:

1. M' überführt die Startsituation (auf Band 1)

$$\ldots\square\square\square \overset{\nabla}{x_1} * \ldots * x_n \square\square\square \ldots$$

in die Standardsituation

$$\#\nabla x_1 * \ldots * x_n \diamond \nabla \square \diamond \ldots \diamond \nabla \square \# \square\square\square \ldots .$$

2. Für jeden Schritt von M, der S ind S' überführt, überführt M' die Standardsituation zu S in die Standardsituation zu S'.

3. Falls M stoppt, überführt M' die Standardsituation

$$\#\square \ldots \square \nabla \varphi(x_1, \ldots, x_n) \square \ldots \square \diamond \square \ldots \square \nabla \square \ldots \square \diamond \ldots \diamond \square \ldots \square \nabla \square \ldots \square \# \square\square\square \ldots$$

in die Endsituation

$$\begin{array}{l} \nabla \\ \varphi(x_1, \ldots, x)\ \square\square\square \ldots \end{array}$$

Zur Umsetzung der Phasen genügt es folgende Einzeloperationen von M' zu realisieren:

- Einfügen eines Symbols und verschieben des rechts befindlichen Bandinhalts um eine Zelle nach rechts
- Löschen eines Symbols und verschieben des rechts befindlichen Bandinhalts um eine Zelle nach links
- Merken von k Symbolen in Zuständen und ändern der Symbole neben ∇ im entsprechenden Bereich

Damit ist das Theorem bewiesen. ■

Theorem 2.10 RAM $\subseteq$ TM.

Beweis: (*Skizze*) Wir geben nur die Beweisidee an. Es sei $\varphi : \mathbb{N}^n \to \mathbb{N}$ eine von der RAM M berechnete Funktion. Wir konstruieren eine 3-Band-Turingmaschine, die M simuliert und dabei mit folgender Standardsituation auf Band 1 operiert:

$$\ldots \square\square\square \overset{\nabla}{\#}\, \mathrm{dya}(i_1) * \mathrm{dya}(\langle \mathtt{R}i_1 \rangle) \# \ldots \# \mathrm{dya}(i_r) * \mathrm{dya}(\langle \mathtt{R}i_r \rangle) \# \square\square\square \ldots,$$

wobei $\langle \mathtt{R}i \rangle$ der Registerinhalt von $\mathtt{R}i$ nach einem Schritt von M ist, $i_j \neq i_k$ für $1 \leq j < k \leq r$ gilt sowie $\langle \mathtt{R}i \rangle = 0$ für alle $i \notin \{i_1, \ldots, i_r\}$ gilt. $\langle \mathtt{BR} \rangle$ wird in den Zuständen gemerkt. Die Bänder 2 und 3 diene als Hilfsbänder für das Auffinden von Registern und zum Ausführen arithmetischer Operationen. Damit ist das Theorem bewiesen. ■

Theorem 2.11 1-TM $\subseteq$ RIES.

Beweis: Folgt ... ■

Mit den Theoremen 2.10, 2.9 und 2.11 haben wir somit die Inklusionskette

$$\mathbf{RAM} \subseteq \mathbf{TM} \subseteq \mathbf{1\text{-}TM} \subseteq \mathbf{RIES} = \mathbf{RAM}$$

gezeigt. Folglich wissen wir, dass Gleichheit zwischen diesen Klassen besteht:

$$\mathbf{RAM} = \mathbf{RIES} = \mathbf{MINI\text{-}RIES} = \mathbf{TM} = \mathbf{1\text{-}TM}$$

2.4 Partiell-rekursive Funktionen

Ziel dieses Abschnittes ist die Charakterisierung berechenbarer Funktionen mit Hilfe ihrer algebraischer Erzeugung. Die Aufgabe besteht also darin

- einfache Grundfunktionen $f_1, \ldots, f_m \in \mathbf{FUNK}(\mathbb{N})$ und
- einfache Operation $O_1, \ldots, O_k$ auf $\mathbf{FUNK}(\mathbb{N})$

zu finden mit

$$\mathbf{RIES} = \Gamma_{O_1,\ldots,O_k}(\{f_1, \ldots, f_m\})$$

Als Grundfunktionen betrachten wir die folgenden Funktionen:

- $\mathrm{S} : \mathbb{N} \to \mathbb{N} : x \mapsto x + 1$
- $\mathrm{I}_1^2 : \mathbb{N}^2 \to \mathbb{N} : (x, y) \mapsto x$
- die nullstellige 0-Konstante C_0^0

Als Operationen betrachten wir die in Kapitel 1 eingeführten Operationen:

- ZV ist die zyklische Vertauschung von Variablen
- LV ist die Vertauschung der letzten beiden Variablen
- ID ist Identifizierung der letzten beiden Variablen
- SUB ist die Substitution von Funktionen an letzter Stelle

Wir benötigen noch Operationen, um die in einer höheren Programmiersprache üblichen Schleifenkonstruktionen beschreiben zu können.

2.4.1 Primitive Rekursion

Es seien $\varphi : \mathbb{N}^n \to \mathbb{N}$ und $\psi : \mathbb{N}^{n+2} \to \mathbb{N}$. Dann ist die Funktion $\mathrm{PR}(\varphi, \psi) : \mathbb{N}^{n+1} \to \mathbb{N}$ wie folgt definiert:

$$\begin{aligned} \mathrm{PR}(\varphi,\psi)(x_1,\ldots,x_n,0) &=_{\text{def}} \varphi(x_1,\ldots,x_n) \\ \mathrm{PR}(\varphi,\psi)(x_1,\ldots,x_n,y) &=_{\text{def}} \psi(x_1,\ldots,x_n,y-1,\mathrm{PR}(\varphi,\psi)(x_1,\ldots,x_n,y-1)) \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{für } y > 0 \end{aligned}$$

PR steht für die Operation der *primitiven Rekursion*. Wir definieren die Menge der primitiv-rekursiven Funktionen wie folgt:

$$\mathbf{PRIM} =_{\text{def}} \Gamma_{\mathrm{ZV,LV,ID,SUB,PR}}(\{\mathrm{C}_0^0, \mathrm{I}_1^2, \mathrm{S}\})$$

Proposition 2.12 *Jede Funktion, die durch beliebige Permutation von Variablen, Identifizierung beliebiger Variablen oder simultane Substitution aus primitiv-rekursiven Funktionen entsteht, ist wieder primitiv-rekursiv.*

Beweis: Dies Aussage folgt sofort aus Lemma 1.2. ∎

Beispiele:

- Wir wissen bereits, dass sich I_m^n für alle $m, n \in \mathbb{N}_+$ aus I_1^2 mit Hilfe von $\mathrm{ZV}, \mathrm{LV}, \mathrm{ID}$ und SUB erzeugen lässt. Somit gilt

  $$\mathrm{I}_n^m \in \Gamma_{\mathrm{ZV,LV,ID,SUB}}\left(\{\mathrm{I}_1^2\}\right) \subseteq \Gamma_{\mathrm{ZV,LV,ID,SUB,PR}}\left(\{\mathrm{C}_0^0, \mathrm{I}_1^2, \mathrm{S}\}\right) = \mathbf{PRIM}$$

- Die Vorgängerfunktion $\mathrm{P} : \mathbb{N} \to \mathbb{N} : x \mapsto x \dot{-} 1$ ist primitiv-rekursiv:

  $$\begin{aligned} \mathrm{P}(0) &= 0 = \mathrm{C}_0^0 \\ \mathrm{P}(y) &= y - 1 = \mathrm{I}_1^2(y-1, \mathrm{P}(y-1)) \end{aligned}$$

  Mithin gilt $\mathrm{P} = \mathrm{PR}(\mathrm{C}_0^0, \mathrm{I}_1^2)$.

- Summenbildung ist primitiv-rekursiv wegen $\mathrm{sum} = \mathrm{PR}(\mathrm{I}_1^1, \mathrm{SUB}(\mathrm{S}, \mathrm{I}_3^3))$:

  $$\begin{aligned} \mathrm{sum}(x,0) &= x = \mathrm{I}_1^1 \\ \mathrm{sum}(x,y) &= x + (y-1) + 1 \\ &= \mathrm{sum}(x, y-1) + 1 \\ &= \mathrm{I}_3^3(x, y-1, \mathrm{sum}(x,y-1)) + 1 \\ &= \mathrm{S}\left(\mathrm{I}_3^3(x, y-1, \mathrm{sum}(x,y-1))\right) \\ &= \mathrm{SUB}(\mathrm{S}, \mathrm{I}_3^3)(x, y-1, \mathrm{sum}(x,y-1)) \end{aligned}$$

Lemma 2.13 *Die Funktionen* I_m^n *(für* $1 \leq m \leq n$*),* C_m^n *(für* $m, n \geq 0$*),* sum*,* md*,* prod*,* div*,* exp*,* prim *und* prex *sind primitiv-rekursiv.*

Beweis: Zum Beweis geben wir für die einzelnen Funktionen lediglich die entsprechenden Konstruktionen an. Die Überprüfung der Korrektheit der Konstruktionen bleibt als Übung überlassen.

- Für I_m^n siehe das erste Beispiel oben.
- Für C_m^n betrachten wir verschiedene Fälle:

$$\begin{aligned} \mathrm{C}_0^1 &= \mathrm{ID}(\mathrm{md}) \\ \mathrm{C}_0^n &= \mathrm{SUB}(\mathrm{C}_0^1, \mathrm{I}_1^n) \\ \mathrm{C}_m^n &= \mathrm{SUB}(\mathrm{S}_m, \mathrm{C}_0^n) \end{aligned}$$

  Hierbei ist die Funktion $\mathrm{S}_m : \mathbb{N} \to \mathbb{N} : x \mapsto x + m$ ebenfalls für alle $m \in \mathbb{N}$ primitv-rekursiv (wegen $\mathrm{S}_m = \mathrm{SUB}(\mathrm{S}, \mathrm{S}_{m-1})$ für $m \geq 2$, $\mathrm{S}_1 = \mathrm{S}$ und $\mathrm{S}_0 = \mathrm{I}_0^1$).
- Es gilt $\mathrm{sum} = \mathrm{PR}(\mathrm{I}_1^1, \mathrm{SUB}(\mathrm{S}, \mathrm{I}_3^3))$ (siehe auch das Beispiel oben).
- Es gilt $\mathrm{md} = \mathrm{PR}(\mathrm{I}_1^1, \mathrm{SUB}(\mathrm{P}, \mathrm{I}_3^3))$.
- Es gilt $\mathrm{prod} = \mathrm{PR}(\mathrm{C}_0^1, \mathrm{SIM}_2(\mathrm{sum}, \mathrm{I}_3^3, \mathrm{I}_1^3))$.
- Es gilt $\mathrm{div} = \mathrm{SIM}_3(d, \mathrm{I}_1^2, \mathrm{I}_2^2, \mathrm{I}_1^2)$ mit

$$d : \mathbb{N}^3 \to \mathbb{N} : (x, y, u) \mapsto \|\{\ z \mid z \leq u \text{ und } y \cdot u \leq x\ \}\| - 1.$$

  Es gilt $d = \mathrm{PR}(\mathrm{C}_0^2, g)$ mit

$$g : \mathbb{N}^4 \to \mathbb{N} : (x, y, u, v) \mapsto (1 \dot{-} ((u+1) \cdot y \dot{-} x)) + v.$$

  Es gilt $g = \mathrm{SIM}_2(\mathrm{sum}, \mathrm{I}_4^4, \mathrm{SIM}_2(\mathrm{md}, \mathrm{C}_1^4, \mathrm{SIM}_2(\mathrm{md}, \mathrm{SIM}_2(\mathrm{prod}, \mathrm{SUB}(\mathrm{S}, \mathrm{I}_3^4), \mathrm{I}_2^4), \mathrm{I}_1^4))))$.
- Es gilt $\mathrm{exp} = \mathrm{PR}(\mathrm{C}_1^1, \mathrm{SIM}_2(\mathrm{prod}, \mathrm{I}_1^3, \mathrm{I}_3^3))$.
- Für prim und prex siehe Übungsblatt 8.

Damit ist das Lemma bewiesen. ∎

Lemma 2.14 *Sind* $g_0, g_1, \ldots, g_r, h : \mathbb{N}^n \to \mathbb{N}$ *primitiv-rekursiv, so ist auch die durch*

$$f(x) =_{\mathrm{def}} \begin{cases} g_0(x) & \textit{falls } h(x) = 0 \\ g_1(x) & \textit{falls } h(x) = 1 \\ \vdots & \vdots \\ g_{r-1}(x) & \textit{falls } h(x) = r-1 \\ g_r(x) & \textit{falls } h(x) \geq r \end{cases}$$

definierte Funktion $f : \mathbb{N}^n \to \mathbb{N}$ *primitiv-rekursiv.*

Beweis: Wir geben wiederum nur die Idee für die Konstruktion an und überlassen die Ausführung als Übung. Es gilt

$$f(x) = \sum_{k=0}^{r-1} (1 \dot{-} ((h(x) \dot{-} k) + (k \dot{-} h(x)))) \cdot g_k(x) \quad + \quad (1 \dot{-} (r \dot{-} h(x))) \cdot g_r(x)$$

Damit ist das Lemma bewiesen. ■

2.4.2 Die Ackermann-Funktion

Die Ackermann-Funktion (benannt nach Wilhelm Ackermann, der sie 1928 publizierte) ist ein Beispiel für eine zwar berechenbare, aber nicht primitiv-rekursive Funktion. Dazu betrachten wir zunächst die folgende zweistellige Hilfsfunktion $A : \mathbb{N}^2 \to \mathbb{N}$, die wie folgt rekursiv definiert ist:

$$\begin{array}{lll} A(0,y) =_{\text{def}} y+1 & & \\ A(x,0) =_{\text{def}} x & & \text{für } x \geq 1 \\ A(x,y) =_{\text{def}} A(x-1, A(x,y-1)) & & \text{für } x \geq 1 \text{ und } y \geq 1 \end{array}$$

Beispiele: Wir wollen die Funktion $A(x,y)$ für einige Werte von x bestimmen. Vereinfachend nehmen wir im Folgenden an, dass $y > 0$ gilt. Die Aussage lassen sich aber auch leicht für $y = 0$ überprüfen.

- $A(1,y) = A(0, A(1,y-1)) = A(1,y-1) + 1 = \ldots = A(1,0) + y = y + 1.$
- $A(2,y) = A(1, A(2,y-1)) = A(2,y-1) + 1 = \ldots = A(2,0) + y = y + 2.$
- $A(3,y) = A(2, A(3,y-1)) = A(3,y-1) + 2 = \ldots = A(3,0) + 2y = 2y + 3.$
- $A(4,y) = 7 \cdot 2^y - 3$ lässt sich mittels vollständiger Induktion über y zeigen:
  - *Induktionsanfang* $y = 0$: Es gilt $A(4,0) = 4 = 7 \cdot 2^0 - 3$.
  - *Induktionsschritt* $y > 0$: Es gilt

$$\begin{array}{rcll} A(4,y) & = & A(3, A(4,y-1)) & \\ & = & 2 \cdot A(4,y-1) + 3 & \\ & = & 2 \cdot \left(7 \cdot 2^{y-1} - 3\right) + 3 & \text{(nach Induktionsvoraussetzung)} \\ & = & 7 \cdot 2^y - 3 & \end{array}$$

- $A(5,y) = A(4, A(5,y-1)) \geq 2^{A(5,y-1)} \geq \ldots \geq \left.2^{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}\right\} y\text{-mal}.$

Lemma 2.15 *Es sei $x \in \mathbb{N}$.*

1. *Für alle $y \in \mathbb{N}$ gilt $y < A(x, y)$.*
2. *Für alle $y \in \mathbb{N}$ gilt $A(x, y) < A(x, y + 1)$.*
3. *Für alle $y \in \mathbb{N}$ gilt $A(x, y) \leq A(x + 1, y)$.*
4. *Es gibt ein $z \in \mathbb{N}$ mit $A(x, y) \leq A(z, y - 1)$ für alle $y \in \mathbb{N}_+$.*
5. *Für alle $y \in \mathbb{N}$ gilt $A(x, 2y) < A(x + 3, y)$.*

Beweis: Wir beweisen die Aussagen einzeln.

1. Wir verwenden vollständige Induktion über x:
   - *Induktionsanfang* $x = 0$: Es gilt $A(0, y) = y + 1 > y$.
   - *Induktionsschritt* $x > 0$: Wir verwenden vollständige Induktion über y:
     - *Induktionsanfang* $y = 0$: Es gilt $A(x, 0) = x > 0 = y$.
     - *Induktionsschritt* $y > 0$: Es gilt

$$\begin{aligned} & A(x, y) \\ = \;& A(x - 1, A(x, y - 1)) && \text{(nach Definition von } A(x, y)) \\ > \;& A(x, y - 1) && \text{(nach Induktionsvoraussetzung für } x - 1) \\ \geq \;& y && \text{(nach Induktionsvoraussetzung für } y - 1) \end{aligned}$$

2. Wir betrachten zwei Fälle:
   - Für $x = 0$ gilt $A(0, y) = y+1 < y+2 = A(0, y+1)$ (nach Definition von $A(x, y)$ für $x = 0$).
   - Für $x > 0$ gilt $A(x, y + 1) = A(x - 1, A(x, y)) > A(x, y)$ (nach Definition von $A(x, y)$ für $x, y > 0$ und Aussage 1).

3. Wir verwenden vollständige Induktion über x:
   - *Induktionsanfang* $x = 0$: Es gilt $A(0, y) = y + 1 = A(1, y)$.
   - *Induktionsschritt* $x > 0$: Wir verwenden vollständige Induktion über y:
     - *Induktionsanfang* $y = 0$: Es gilt $A(x, 0) = x < x + 1 = A(x + 1, 0)$.
     - *Induktionsschritt* $y > 0$: Es gilt

$$\begin{aligned} & A(x, y) \\ = \;& A(x - 1, A(x, y - 1)) && \text{(nach Definition von } A(x, y)) \\ \leq \;& A(x - 1, A(x + 1, y - 1)) \\ & && \text{(nach Induktionsvoraussetzung für } y - 1 \text{ und Aussage 2)} \\ \leq \;& A(x, A(x + 1, y - 1)) && \text{(nach Induktionsvoraussetzung für } x - 1) \\ = \;& A(x + 1, y) && \text{(nach Definition von } A(x, y)) \end{aligned}$$

4. Wir müssen in Abhängigkeit von x (aber nicht von y) ein geeignetes $z \in \mathbb{N}$ finden. Dazu betrachten wir drei Fälle:

   - Für $x = 0$ gilt

     $$\begin{aligned} & A(0,y) \\ &= y+1 && \text{(nach Definition von } A(x,y) \text{ für } x=0) \\ &= (y-1)+2 \\ &= A(2,y-1) && \text{(wegen } A(2,y) = y+2) \end{aligned}$$

   - Für $x > 0$ und $y = 1$ gilt

     $$\begin{aligned} & A(x,1) \\ &= A(x-1, A(x,0)) && \text{(nach Definition von } A(x,y) \text{ für } x,y>0) \\ &= A(x-1,x) && \text{(nach Definition von } A(x,y) \text{ für } y=0) \\ &= A(A(x-1,x),0) && \text{(nach Definition von } A(x,y) \text{ für } y=0) \end{aligned}$$

   - Für $x > 0$ und $y > 1$ gilt

     $$\begin{aligned} & A(x,y) \\ &= A(x, A(2,y-2)) && \text{(wegen } A(2,y) = y+2) \\ &\leq A(x, A(x+1,y-2)) && \text{(nach Aussagen 2 und 3)} \\ &= A(x+1,y-1) && \text{(nach Definition von } A(x,y) \text{ für } x,y>0) \end{aligned}$$

   Somit und nach Aussage 3 können wir $z =_{\text{def}} \max\{2, A(x-1,x), x+1\}$ wählen.

5. Wir betrachten wieder drei Fälle.

   - Für $x \leq 1$ gilt

     $$A(0,2y) = A(1,2y) = 2y+1 < 2y+3 = A(3,y) \leq A(4,y)$$

     (nach Definition von $A(x,y)$ für $x = 0$, wegen $A(1,y) = y+1$ und nach Aussage 3). Damit sind beide Ungleichungen gezeigt.

   - Für $x > 1$ und $y = 0$ gilt

     $$A(x,0) = x < x+3 = A(x+3,0)$$

     (nach Definition von $A(x,y)$ für $x > 0$ und $y = 0$).

   - Für $x > 1$ und $y > 0$ gilt

     $$\begin{aligned} & A(x,2y) \\ &\leq A(x,2y+1) && \text{(nach Aussage 2)} \\ &= A(x, A(3,y-1)) && \text{(wegen } A(3,y) = 2y+3) \\ &\leq A(x, A(x+1,y-1)) && \text{(nach Aussage 3)} \\ &= A(x+1,y) && \text{(nach Definition von } A(x,y) \text{ für } x,y>0) \\ &< A(x+3,y) && \text{(nach Aussage 3)} \end{aligned}$$

Damit ist das Lemma bewiesen. ■

Lemma 2.16 *Es sei $f : \mathbb{N}^n \to \mathbb{N}$ eine primitiv-rekursive Funktion, $n \in \mathbb{N}_+$. Dann gibt es ein $x \in \mathbb{N}$, sodass für alle $y_1, \ldots, y_n \in \mathbb{N}$ gilt:*

$$f(y_1, \ldots, y_n) < A(x, y_1 + \cdots + y_n)$$

Beweis: (*Induktion*) Wir führen einen Induktionsbeweis über den Aufbau primitiv-rekursiver Funktionen.

- *Induktionsanfang*: Die Funktion f ist eine der Grundfunktionen I_1^2 oder S (die nullstellige Funktion C_0^0 entfällt). Wir betrachten dementsprechend zwei Fälle:
  - $f = \mathrm{I}_1^2$: Es gilt $\mathrm{I}_1^2(y_1, y_2) = y_1 < y_1 + y_2 + 1 = A(1, y_1 + y_2)$. Mithin gilt die Aussage für $x = 1$.
  - $f = \mathrm{S}$: Es gilt $\mathrm{S}(y_1) = y_1 + 1 < y_1 + 2 = A(2, y_1)$. Mithin gilt die Aussage für $x = 2$.
- *Induktionsschritt*: Die Funktion f entsteht aus den Funktionen $g : \mathbb{N}^m \to \mathbb{N}$ und $h : \mathbb{N}^k \to \mathbb{N}$ durch Anwendung von ZV, LV, ID, SUB oder PR. Als Induktionsvoraussetzung nehmen wir an, dass $g(y_1, \ldots, y_m) < A(x_g, y_1 + \cdots + y_n)$ und $h(y_1, \ldots, y_k) < A(x_h, y_1 + \cdots + y_k)$ für geeignete $x_g, x_h \in \mathbb{N}$ und alle $y_i \in \mathbb{N}$ gilt. Wir betrachten folgende Fälle:
  - $f = \mathrm{ZV}(g)$: Es gilt

    $$\begin{aligned} & f(y_1, \ldots, y_n) \\ &= g(y_2, \ldots, y_n, y_1) && \text{(nach Definition von ZV)} \\ &< A(x_g, y_2 + \cdots + y_n + y_1) && \text{(nach Induktionsvoraussetzung)} \\ &= A(x_g, y_1 + \cdots + y_n) \end{aligned}$$

    Mithin gilt die Aussage für $x = x_g$.
  - $f = \mathrm{LV}(g)$: Es gilt

    $$\begin{aligned} & f(y_1, \ldots, y_n) \\ &= g(y_1, \ldots, y_{n-2}, y_n, y_{n-1}) && \text{(nach Definition von LV)} \\ &< A(x_g, y_1 + \cdots + y_{n-2} + y_n + y_{n-1}) && \text{(nach Induktionsvoraussetzung)} \\ &= A(x_g, y_1 + \cdots + y_n) \end{aligned}$$

    Mithin gilt die Aussage für $x = x_g$.

– $f = \mathrm{ID}(g)$: Es gilt

$$\begin{aligned}
&f(y_1, \ldots, y_n) \\
&\quad = \quad g(y_1, \ldots, y_n, y_n) && \text{(nach Definition von ID)} \\
&\quad < \quad A(x_g, y_1 + \cdots + y_n + y_n) && \text{(nach Induktionsvoraussetzung)} \\
&\quad \leq \quad A(x_g, 2 \cdot (y_1 + \cdots + y_n)) && \text{(nach Lemma 2.15.2)} \\
&\quad < \quad A(x_g + 3, y_1 + \cdots + y_n) && \text{(nach Lemma 2.15.5)}
\end{aligned}$$

Mithin gilt die Aussage für $x = x_g + 3$.

– $f = \mathrm{SUB}(g, h)$: Somit erfüllen m, k die Gleichung $m + k - 1 = n$. Es gilt

$$\begin{aligned}
&f(y_1, \ldots, y_n) \\
&\quad = \quad g(y_1, \ldots, y_{m-1}, h(y_m, \ldots, y_n)) && \text{(nach Definition von SUB)} \\
&\quad < \quad A(x_g, y_1 + \cdots + y_{m-1} + h(y_m, \ldots, y_n)) \\
&\qquad\qquad && \text{(nach Induktionsvoraussetzung für } g\text{)} \\
&\quad < \quad A(x_g, y_1 + \cdots + y_{m-1} + A(x_h, y_m + \cdots + y_n)) \\
&\qquad\qquad && \text{(nach Induktionsvoraussetzung für } h \text{ und Lemma 2.15.2)} \\
&\quad \leq \quad A(x_g, y_1 + \cdots + y_n + A(x_h, y_1 + \cdots + y_n)) && \text{(nach Lemma 2.15.2)} \\
&\quad < \quad A(x_g, 2 \cdot A(x_h, y_1 + \cdots + y_n)) && \text{(nach Lemma 2.15.1 und Lemma 2.15.2)} \\
&\quad < \quad A(x_g + 3, A(x_h, y_1 + \cdots + y_n)) && \text{(nach Lemma 2.15.5)} \\
&\quad \leq \quad A(r - 1, A(r, y_1 + \cdots + y_n)) \\
&\qquad\qquad && \text{(mit } r = \max\{x_g + 4, x_h\}\text{, nach Lemma 2.15.2 und Lemma 2.15.3)} \\
&\quad = \quad A(r, y_1 + \cdots + y_n + 1) && \text{(nach Definition von } A(x, y) \text{ für } x, y > 0\text{)} \\
&\quad < \quad A(s, y_1 + \cdots + y_n) && \text{(mit geeignetem } s \text{ nach Lemma 2.15.4)}
\end{aligned}$$

Mithin gilt die Aussage für $x = s$.

– $f = \mathrm{PR}(g, h)$: Somit gilt für die Stelligkeiten $m = n - 1$ und $k = n + 1$. Wir zeigen die Aussage für $r =_{\mathrm{def}} \max\{x_h + 4, x_g\}$ mittels vollständiger Induktion über y_n:

  * *Induktionsanfang* $y_n = 0$: Es gilt

$$\begin{aligned}
&f(y_1, \ldots, y_{n-1}, 0) \\
&\quad = \quad g(y_1, \ldots, y_{n-1}) && \text{(nach Definition von PR)} \\
&\quad < \quad A(x_g, y_1 + \cdots + y_{n-1}) && \text{(nach Induktionsvoraussetzung für } g\text{)} \\
&\quad \leq \quad A(r, y_1 + \cdots + y_{n-1} + 0) && \text{(nach Lemma 2.15.3)}
\end{aligned}$$

  * *Induktionsschritt* $y_n > 0$: Es gilt

$$\begin{aligned}
&f(y_1, \ldots, y_{n-1}, y_n) \\
&\quad = \quad h(y_1, \ldots, y_{n-1}, y_n - 1, f(y_1, \ldots, y_{n-1}, y_n - 1))
\end{aligned}$$

(nach Definition von PR)

$$< A(x_h, y_1 + \cdots + y_{n-1} + y_n - 1 + f(y_1, \ldots, y_{n-1}, y_n - 1))$$

(nach Induktionsvoraussetzung für h)

$$< A(x_h, y_1 + \cdots + y_{n-1} + y_n - 1 + A(r, y_1 + \cdots + y_n - 1))$$

(nach Induktionsvoraussetzung für $y_n - 1$)

$$\leq A(x_h, 2 \cdot A(r, y_1 + \cdots + y_n - 1))$$

(nach Lemma 2.15.1 und Lemma 2.15.2)

$$< A(x_h + 3, A(r, y_1 + \cdots + y_n - 1)) \quad \text{(nach Lemma 2.15.5)}$$

$$\leq A(r - 1, A(r, y_1 + \cdots + y_n - 1)) \quad \text{(nach Lemma 2.15.3)}$$

$$= A(r, y_1 + \cdots + y_n) \quad \text{(nach Definition von } A(x, y) \text{ für } x, y > 0)$$

Mithin gilt die Aussage für $x = r$.

Damit ist das Lemma bewiesen. ■

Definition 2.17 *Die einstellige Funktion $A : \mathbb{N} \to \mathbb{N} : x \mapsto A(x, x)$ heißt* Ackermann-Funktion.

Theorem 2.18 *Die* Ackermann*-Funktion A ist berechenbar, aber nicht primitiv-rekursiv.*

Beweis: Zum Nachweis der Berechenbarkeit der Funktion A geben wir folgendes RIES-Programm an:

```
function a(x); a:=aa(x,x)
function aa(x,y);
if (x=0) then aa:=(y+1)
         else if (y=0) then aa:=x
                       else aa:=aa(x-1,aa(x,y-1))
```

Zum Nachweis, dass A nicht primitiv-rekursiv ist, führen wir einen Widerspruchsbeweis. Angenommen A ist primitiv-rekursiv. Dann gibt es nach Lemma 2.16 ein $x \in \mathbb{N}$ mit $A(y) < A(x, y)$ für alle $y \in \mathbb{N}$. Somit gilt insbesondere $A(x) < A(x, x) = A(x)$. Dies ist ein Widerspruch. Mithin ist die Annahme falsch, und A ist nicht primitiv-rekursiv. Damit ist der Satz bewiesen. ■

Ohne Beweis geben wir noch das folgende Theorem an, das eine Beziehung zwischen den primitiv-rekursiven und den RIES-berechenbaren Funktionen angibt.

Theorem 2.19 *Eine Funktion ist genau dann primitiv-rekursiv, wenn sie von einem RIES-Programm berechnet wird, das keine* `while`*-Schleifen und keine Funktionsaufrufe benutzt.*

2.4.3 μ-Rekursion

Im Lichte von Theorem 2.19 benötigen wir also noch eine geeignete Operationen, um `while`-Schleifen zu modellieren. Dies wird von der *μ-Rekursion* geleistet. Dazu führen wir den μ-Operator ein: Es sei E eine Eigenschaft (Aussageform) über dem Universum der natürlichen Zahlen. Wir definieren

$$\mu x(E(x)) =_{\text{def}} \text{kleinstes } x \in \mathbb{N} \text{ mit } E(x).$$

Anzumerken ist, dass $\mu x(E(x))$ im Allgemeinen nicht definiert sein muss.

Der μ-Operator wird in einer spezifischen Form genutzt, um die Operation der *Minimisierung* auf $\mathbf{FUNK}(\mathbb{N})$ zu definieren. Es sei $\varphi : \mathbb{N}^n \to \mathbb{N}$ eine n-stellige Funktion, $n \geq 1$. Dann ist die Funktion $\text{MIN}(\varphi) : \mathbb{N}^{n-1} \to \mathbb{N}$ wie folgt für alle $x_1, \ldots, x_{n-1} \in \mathbb{N}$ definiert:

$$\text{MIN}(\varphi)(x_1, \ldots, x_{n-1}) =_{\text{def}} \begin{cases} \mu y(\varphi(x_1, \ldots, x_{n-1}, y) = 0) & \text{falls } y \text{ existiert und der Wert} \\ & \varphi(x_1, \ldots, x_{n-1}, z) \text{ für alle} \\ & z < y \text{ definiert ist} \\ \text{nicht definiert} & \text{sonst} \end{cases}$$

Mit Hilfe von MIN definieren wir die Klasse der partiell-rekursiven Funktion

$$\mathbf{PART} =_{\text{def}} \Gamma_{\text{ZV,LV,ID,SUB,PR,MIN}}\left(\{\text{C}_0^0, \text{I}_1^2, \text{S}\}\right).$$

Es gilt ganz offensichtlich $\mathbf{PRIM} \subset \mathbf{PART}$ (für die Echtheit der Inklusion siehe das zweite Beispiel unten).

Beispiele:

- Für den Logarithmus $\log : \mathbb{N} \to \mathbb{N} : x \mapsto \lfloor \log_2 x \rfloor$ mit $\log(0) = 0$ gilt

$$\begin{aligned} \log(x) &= \max\{\ y \mid 2^y \leq x\ \} \\ &= \min\{\ y \mid 2^{y+1} > x\ \} \\ &= \mu y\left(2^{y+1} - 1 \geq x\right) \\ &= \mu y\left(x \dot{-} (2^{y+1} \dot{-} 1) = 0\right) \\ &= \mu y\left(\text{md}(x, \text{md}(\exp(2, \text{S}(y)), 1)) = 0\right) \end{aligned}$$

  also $\log = \text{MIN}(\text{SIM}_2(\text{md}, \text{I}_1^2, \text{SIM}_2(\text{md}, \text{SIM}_2(\exp, \text{C}_2^2, \text{SUB}(\text{S}, \text{I}_2^2)), \text{C}_1^2)))$. Mithin gilt $\log \in \mathbf{PART}$.

- Für die einstellige, nirgends definierte Funktion $\nu^1 : \mathbb{N} \to \mathbb{N}$ gilt

$$\nu^1(x) = \mu y(x + y + 1 = 0) = \mu y(\text{S}(\text{sum}(x, y)) = 0),$$

  also $\nu^1 = \text{MIN}(\text{SUB}(\text{S}, \text{sum}))$. Mithin gilt $\nu^1 \in \mathbf{PART}$.

- Das Problem von COLLATZ besteht in Folgendem: Wir betrachten Zahlenfolgen $(a_n)_{n\in\mathbb{N}}$ mit $a_0 \in \mathbb{N}$ und

$$a_n =_{\text{def}} \begin{cases} \dfrac{a_{n-1}}{2} & \text{falls } a_{n-1} > 0 \text{ gerade ist} \\ 3a_{n-1}+1 & \text{sonst} \end{cases}$$

  für $n \geq 1$. Die Vermutung ist nun, dass die Folge für alle Anfangsglieder $a_0 \in \mathbb{N}$ den Zyklus $4, 2, 1$ erreicht.

2.5 Hauptsatz der Algorithmentheorie

Wir fassen die bisher gezeigten Äquivalenzen von Berechnungsbegriffe im Hauptsatz der Algorithmentheorie zusammen.

Theorem 2.20 *Es gilt*

$$\mathbf{RAM} = \mathbf{RIES} = \mathbf{MINI\text{-}RIES} = \mathbf{TM} = \mathbf{1\text{-}TM} = \mathbf{PART}.$$

Mit anderen Worten sind für eine beliebige Funktion $\varphi \in \mathbf{FUNK}(\mathbb{N})$ die folgenden Aussagen äquivalent:

1. *φ ist* RAM*-berechenbar.*
2. *φ ist* RIES*-berechenbar.*
3. *φ ist* MINI-RIES*-berechenbar.*
4. *φ ist Turing-berechenbar.*
5. *φ ist* 1*-Turing-berechenbar.*
6. *φ ist partiell-rekursiv.*

Die Liste äquivalenter Berechnungsbegriffe in Theorem 2.20 kann, im Prinzip beliebig, erweitert werden, z.B. um folgende prominente Berechnungsmodelle:

- λ-Kalkül (mit den Programmiersprachen Haskell, Lisp, ...)
- Prolog
- Java, C, ...
- Wortersetzungssysteme (POST-, MARKOV-Algorithmen)
- Quantencomputer

Damit verfügen wir über Evidenz für die These von Church (manchmal auch Church-Turing-These), die besagt, dass mit der Turingmaschine das algorithmisch Berechenbare erfasst ist.

These 2.21 (Church) *Jede algorithmisch berechenbare Funktion ist Turing-berechenbar.*

Da der Algorithmenbegriff intuitiv und offen gegenüber zukünftigen Entwicklungen ist, ist die These prinzipiell nicht beweisbar (jedoch widerlegbar). Die These wird deshalb als Axiom angenommen. Wann immer wir eine Funktion als berechenbar voraussetzen, können wir einen der behandelten Algorithmenbegriffe auswählen, der uns am geeignetsten für die zu untersuchende Fragestellung erscheint.

Berechenbarkeitstheorie 3

Nachdem wir aus dem vorangegangenen Kapitel eine Vorstellung gewonnen haben, wie eine Funktion berechnet werden kann, wenden wir uns nunmehr der Frage zu, welche Funktionen überhaupt berechnet werden können. Dabei konzentrieren wir uns auf Mengen.

Es sei $A \subseteq \mathbb{N}^m$. Dann ist die *charakteristische Funktion* $c_A : \mathbb{N}^m \to \{0,1\}$ *von* A für alle $x_1, \ldots, x_m \in \mathbb{N}$ definiert als

$$c_A(x_1, \ldots, x_m) =_{\text{def}} \begin{cases} 1 & \text{falls } (x_1, \ldots, x_m) \in A \\ 0 & \text{sonst} \end{cases}$$

3.1 Entscheidbare Mengen

Definition 3.1 *Eine Menge $A \subseteq \mathbb{N}^m$ mit $m \geq 1$ heißt* entscheidbar*, falls c_A berechenbar ist.*

Beispiele: Folgende Mengen sind entscheidbar:

- die Menge der Quadratzahlen $\{ n^2 \mid n \geq 0 \}$
- die Menge der Zweierpotenzen $\{ 2^n \mid n \geq 0 \}$
- die Menge der Primzahlen
- die Menge $A = \{ n \mid n \geq 2$ und es gibt Primzahlen p, q mit $2n = p + q \}$ (unabhängig von der bisher unbewiesenen GOLDBACHschen Vermutung, nach der $A = \mathbb{N} \setminus \{0,1\}$ gelten würde).

Theorem 3.2 *Es seien $A, B \subseteq \mathbb{N}^m$ mit $m \in \mathbb{N}_+$.*

1. *Ist A endlich, so ist A entscheidbar.*
2. *Ist A entscheidbar, so ist $\overline{A}$ entscheidbar.*
3. *Sind A und B entscheidbar, so sind $A \cap B$ und $A \cup B$ entscheidbar.*
4. *Ist A entscheidbar und ist die Funktion $f : \mathbb{N}^m \to \mathbb{N}^n$ total und berechenbar, so ist $f^{-1}(A) = \{ x \mid f(x) \in A \}$ entscheidbar.*

Beweis: Wir führen den Bewis schrittweise.

1. Es sei $A = \{a_1, \ldots, a_k\}$ mit $k \in \mathbb{N}$. Dann entscheidet folgender Algorithmus A, ob $x \in A$ für die Eingabe x gilt: Auf x teste nacheinander, ob $x = a_1, x = a_2, \ldots$ oder

$x = a_k$ gilt; wenn ein Test positiv ist, so wird 1 ausgegeben, anderenfalls wird 0 ausgegeben. Mithin ist c_A berechenbar, also ist A entscheidbar.

2.-4. Es gelten folgende Beziehungen:

$$\begin{aligned} c_{\overline{A}}(x) &= 1 - c_A(x) && \text{für alle } x \in \mathbb{N}^m \\ c_{A\cup B}(x) &= \max\{c_A(x), c_B(x)\} && \text{für alle } x \in \mathbb{N}^m \\ c_{A\cap B}(x) &= \min\{c_A(x), c_B(x)\} && \text{für alle } x \in \mathbb{N}^m \\ c_{f^{-1}(A)}(x) &= c_A(f(x)) && \text{für alle } x \in \mathbb{N}^n \end{aligned}$$

Sind A und B entscheidbar, so sind c_A und c_B berechenbar. Somit sind auch $c_{\overline{A}}$, $c_{A\cup B}$, $c_{A\cap B}$ und $c_{f^{-1}(A)}$ berechenbar. Mithin sind $\overline{A}$, $A \cup B$, $A \cap B$ und $f^{-1}(A)$ entscheidbar.

Damit das Theorem bewiesen. ■

3.2 Aufzählbare Mengen

Definition 3.3 *Eine Menge $A \subseteq \mathbb{N}$ heißt* aufzählbar*, falls $A = \emptyset$ gilt oder es eine einstellige, totale und berechenbare Funktion $f : \mathbb{N} \to \mathbb{N}$ gibt mit $R_f = A$.*

Aufzählbarkeit ergibt sich durch das Verfahren der Reihe nach $f(0)$, $f(1)$, $f(2)$, ... zu berechnen, mit dem alle Elemente von A aufgelistet werden. Kommt x in der Folge $f(0), f(1), f(2), \ldots$ vor, so wissen wir, dass $x \in A$ gilt. Gilt jedoch $x \notin A$, so wissen wir zu keinem Zeitpunkt die Antwort auf die Frage „$x \in A$?“. Deshalb werden aufzählbare Mengen auch *semi-entscheidbar* genannt.

Theorem 3.4 *Eine Menge $A \subseteq N$ ist genau dann entscheidbar, wenn A und $\overline{A}$ aufzählbar sind.*

Beweis: Wir beweisen beide Richtungen einzeln.

($\Rightarrow$): Es sei $A \subseteq N$ eine entscheidbare Menge. Wir unterscheiden drei Fälle für A:

- *1. Fall*: Ist $A = \emptyset$, so ist A nach Definition aufzählbar und $\overline{A}$ wegen $\overline{A} = \mathbb{N} = R_{C_1^1}$ aufzählbar.
- *2. Fall*: Ist $A = \mathbb{N}$, so ist analog zum ersten Fall $\overline{A} = \emptyset$ nach Definition aufzählbar und A wegen $\overline{A} = R_{C_1^1}$ aufzählbar.

- *3. Fall*: Es sei $A \neq \emptyset$ and $A \neq \mathbb{N}$. Dann gibt es $a \in A$ und $b \in \overline{A}$. Wir definieren zwei Funktionen $f, g : \mathbb{N} \to \mathbb{N}$ wie folgt:

$$f(x) =_{\text{def}} \begin{cases} x & \text{falls } c_A(x) = 1 \\ a & \text{falls } c_A(x) = 0 \end{cases}, \qquad g(x) =_{\text{def}} \begin{cases} b & \text{falls } c_A(x) = 1 \\ x & \text{falls } c_A(x) = 0 \end{cases}$$

  Dann gilt $A = R_f$ und $\overline{A} = R_g$. Weiterhin sind f und g total und berechenbar. Somit sind A und $\overline{A}$ aufzählbar.

Mithin sind A und $\overline{A}$ stets aufzählbar.

($\Leftarrow$): Es seien A und $\overline{A}$ aufzählbar. Wir unterscheiden zwei Fälle:

- *1. Fall*: Ist $A = \emptyset$ oder $\overline{A} = \emptyset$, dann ist A entscheidbar (nach Theorem 3.2).
- *2. Fall*: Ist $A \neq \emptyset$ und $\overline{A} \neq \emptyset$, so existieren totale, berechenbare Funktionen $f, g : \mathbb{N} \to \mathbb{N}$ mit $R_f = A$ und $R_g = \overline{A}$. Ein Algorithmus, um A zu entscheiden, durchläuft auf Eingabe x die Folge $f(0), g(0), f(1), g(1), \ldots$, bis x in der Folge auftaucht. (Dies passiert stets wegen $R_f \cup R_g = \mathbb{N}$.) Kommt x als Wert von f vor, so gilt $x \in A$; kommt x als Wert von g vor, so gilt $x \notin A$. Der Algorithmus gibt einen entsprechenden Wert aus.

Mithin ist A stets entscheidbar.

Damit ist das Theorem bewiesen. ■

Theorem 3.5 *Es sei $\varphi : \mathbb{N} \to \mathbb{N}$ berechenbar. Dann sind D_φ und R_φ aufzählbar.*

Beweis: Sind D_φ und R_φ leer, dann sind D_φ und R_φ aufzählbar. Ansonsten gibt es $a \in D_\varphi$ und $b \in R_\varphi$. Es sei M eine Turingmaschine, die φ berechnet. Wir definieren zwei Funktionen $f, g : \mathbb{N} \to \mathbb{N}$ wie folgt für alle $x \in \mathbb{N}$:

$$f(x) =_{\text{def}} \begin{cases} \ell(x) & \text{falls } M \text{ bei Eingabe } \ell(x) \text{ nach } r(x) \text{ Takten stoppt} \\ a & \text{sonst} \end{cases}$$

$$g(x) =_{\text{def}} \begin{cases} \varphi(\ell(x)) & \text{falls } M \text{ bei Eingabe } \ell(x) \text{ nach } r(x) \text{ Takten stoppt} \\ b & \text{sonst} \end{cases}$$

Hierbei sind $\ell, r : \mathbb{N} \to \mathbb{N}$ totale, berechenbare Funktionen mit $(\ell(x), r(x)) = c^{-1}(x)$ für eine Bijektion $c : \mathbb{N}^2 \to \mathbb{N}$. Als Bijektion können beispielsweise $c(x, y) =_{\text{def}} \frac{1}{2} \cdot (x + y)^2 + x + 3y$ oder $c(x, y) =_{\text{def}} 2^x(2y+1)-1$ verwendet werden. Dann sind f und g total und berechenbar. Weiterhin gilt $D_\varphi = R_f$ sowie $R_\varphi = R_g$. Mithin sind D_φ und R_φ aufzählbar. Damit ist das Theorem bewiesen. ■

Korollar 3.6 *Für eine beliebige Menge $A \subseteq \mathbb{N}$ sind folgende Aussagen äquivalent:*

1. *A ist aufzählbar.*
2. *A ist Wertebereich einer einstelligen, berechenbaren Funktion.*
3. *A ist Definitionsbereich einer einstelligen, berechenbaren Funktion.*

Beweis: Die Implikationen 2. $\Rightarrow$ 1. und 3. $\Rightarrow$ 1. folgen aus Theorem 3.5. Die Implikation 1. $\Rightarrow$ 2. ist trivial. Für die Implikation 1. $\Rightarrow$ 3. unterscheiden wir zwei Fälle. Ist $A = \emptyset$, so gilt $A = D_{\nu^1}$. Ist $A \neq \emptyset$ mit $A = R_f$, so definieren wir $\varphi(x) =_{\text{def}} \mu y(f(y) = x)$. Mithin gilt $A = D_\varphi$. ■

Theorem 3.7 *Sind A und B aufzählbar, so sind $A \cup B$ und $A \cap B$ aufzählbar.*

Beweis: Übungsaufgabe. ■

Für eine Menge $B \subseteq \mathbb{N} \times \mathbb{N}$ definieren wir die *Projektion* von B als die Menge

$$\text{Proj}(B) =_{\text{def}} \{\ x \mid \text{es gibt ein } y \in \mathbb{N} \text{ mit } (x, y) \in B\ \}$$

Theorem 3.8 (Projektionssatz) *Es sei $A \subseteq \mathbb{N}$. Dann ist A genau dann aufzählbar, wenn A Projektion einer entscheidbaren Menge ist.*

Beweis: Wir zeigen beide Richtungen einzeln.

($\Rightarrow$): Es sei A ein aufzählbare Menge. Nach Korollar **??** gibt es eine berechenbare Funktion φ mit $A = D_\varphi$. Es sei M eine Turingmaschine, die φ berechnet. Wir definieren eine Menge B wie folgt:

$$B =_{\text{def}} \{\ (x, t) \mid M \text{ hält auf Eingabe } x \text{ nach } t \text{ Takten}\ \}$$

Dann ist B entscheidbar und es gilt für alle $x \in \mathbb{N}$:

$$\begin{aligned} x \in A &\iff x \in D_\varphi \\ &\iff M \text{ hält auf Eingabe } x \\ &\iff M \text{ hält auf Eingabe } x \text{ nach } t \text{ Takten mit geeigneter Taktzahl } t \\ &\iff \text{es gibt ein } t \in \mathbb{N} \text{ mit } (x, t) \in B \\ &\iff x \in \text{Proj}(B) \end{aligned}$$

Mithin ist A die Projektion einer entscheidbaren Menge.

($\Leftarrow$): Es sei B eine entscheidbare Menge mit $A = \text{Proj}(B)$. Wir definieren die Funktion φ für $x \in \mathbb{N}$ als

$$\varphi(x) =_{\text{def}} \mu y((x, y) \in B).$$

Dann ist φ berechenbar und es gilt für alle $x \in \mathbb{N}$:

$$\begin{aligned} x \in D_\varphi &\iff \text{es gibt ein } y \in \mathbb{N} \text{ mit } (x, y) \in B \\ &\iff x \in \text{Proj}(B) \\ &\iff x \in A \end{aligned}$$

Mithin ist A aufzählbar

Damit ist das Theorem bewiesen. ■

3.3 Das Halteproblem

Das Halteproblem ist ein Beispiel für ein aufzählbares Problem, das nicht entscheidbar ist. Zur Formulierung des Problem führen wir zunächst eine Kodierung von RAMs ein:

1. Wir kodieren Befehle einer RAM als Wörter über $\{\mathtt{R}, \mathtt{G}, |, \leftarrow, \rightarrow, +, -, =, \neq\}$:

Befehl	Kodierung
$\mathtt{R}i \leftarrow \mathtt{R}j$	$\mathtt{R}\underbrace{\|\|\ldots\|}_{i+1} \leftarrow \mathtt{R}\underbrace{\|\|\ldots\|}_{j+1}$
$\mathtt{R}i \leftarrow \mathtt{RR}j$	$\mathtt{R}\underbrace{\|\|\ldots\|}_{i+1} \leftarrow \mathtt{RR}\underbrace{\|\|\ldots\|}_{j+1}$
$\mathtt{RR}i \leftarrow \mathtt{R}j$	$\mathtt{RR}\underbrace{\|\|\ldots\|}_{i+1} \leftarrow \mathtt{R}\underbrace{\|\|\ldots\|}_{j+1}$
$\mathtt{R}i \leftarrow k$	$\mathtt{R}\underbrace{\|\|\ldots\|}_{i+1} \leftarrow \underbrace{\|\|\ldots\|}_{k+1}$
$\mathtt{R}i \leftarrow \mathtt{R}j + \mathtt{R}k$	$\mathtt{R}\underbrace{\|\|\ldots\|}_{i+1} \leftarrow \mathtt{R}\underbrace{\|\|\ldots\|}_{j+1} + \mathtt{R}\underbrace{\|\|\ldots\|}_{k+1}$
$\mathtt{R}i \leftarrow \mathtt{R}j - \mathtt{R}k$	$\mathtt{R}\underbrace{\|\|\ldots\|}_{i+1} \leftarrow \mathtt{R}\underbrace{\|\|\ldots\|}_{j+1} - \mathtt{R}\underbrace{\|\|\ldots\|}_{k+1}$
$\mathtt{GOTO}\ k$	$\mathtt{G}\underbrace{\|\|\ldots\|}_{k+1}$
$\mathtt{IF}\ \mathtt{R}i = 0\ \mathtt{GOTO}\ k$	$\mathtt{R}\underbrace{\|\|\ldots\|}_{i+1} = \mathtt{G}\underbrace{\|\|\ldots\|}_{k+1}$
$\mathtt{IF}\ \mathtt{R}i \neq 0\ \mathtt{GOTO}\ k$	$\mathtt{R}\underbrace{\|\|\ldots\|}_{i+1} \neq \mathtt{G}\underbrace{\|\|\ldots\|}_{k+1}$
$\mathtt{STOP}$	$\mathtt{GR}$

2. Wir kodieren die Symbole $\mathtt{R}, \mathtt{G}, |, \leftarrow, \rightarrow, +, -, =, \neq$ durch Wörter über $\{1, 2\}$:

Symbol s	R	G	|	$\leftarrow$	+	−	=	$\neq$
Kodierung $c(s)$	111	112	121	122	211	212	221	222

3. Wir kodieren eine RAM M als Wort über $\{1, 2\}$:

   - Ein Befehl $b = a_1 a_2 \ldots a_n$ wird kodiert als
   $$c(b) = c(a_1)c(a_2)\ldots c(a_n).$$
   - Eine aus den Befehlen $b_1, b_2, \ldots, b_m$ bestehende RAM M wird kodiert als
   $$c(M) = c(b_1)c(b_2)\ldots c(b_m).$$

Mit Hilfe der angegebenen Kodierung können wir die RAMs durchnummerieren. Eine *Gödelisierung* aller RAMs ist eine unendliche Folge $M_0, M_1, M_2, \ldots$ mit

$$M_i = \begin{cases} M & \text{falls } \mathrm{dya}(i) = c(M) \text{ für die RAM } M \\ M^* & \text{falls } \mathrm{dya}(i) \text{ keiner RAM entspricht} \end{cases}$$

Hierbei ist M^* die RAM, die nur aus dem Befehl `STOP` besteht. Außderdem gibt es einen Algorithmus, der die RAM M_i für gegebenes $i \in \mathbb{N}$ berechnet, sowie einen Algorithmus, der die *Gödelnummer* i für eine gegebene RAM $M = M_i (\neq M^*)$ berechnet.

Beispiele:

- Die RAMs $M_1, M_2, \ldots, M_{72}$ sind alle die RAM M^*.
- Wegen $\mathrm{dya}(73) = 112121$ verbirgt sich hinter der RAM M_{73} in unserer Gödelisierung diejenige RAM M, die die nirgends definierte Funktion ν^1 berechnet:

  0 `GOTO` 0

- Wir betrachten die RAM zur Multiplikation zweier Zahlen sowie ihre befehlsweise Kodierung als Wort über $\{1, 2\}$:

0	`R3` $\leftarrow$ 1	111 121 121 121 121 122 121 121
1	`IF` R1 $= 0$ `GOTO` 5	111 121 121 221 112 121 121 121 121 121 121
2	`R2` $\leftarrow$ `R2` $+$ `R0`	111 121 121 121 122 111 121 121 121 211 111 121
3	`R1` $\leftarrow$ `R1` $-$ `R3`	111 121 121 122 111 121 121 212 111 121 121 121 121
4	`GOTO` 1	112 121 121
5	`R0` $\leftarrow$ `R2`	111 121 122 111 121 121 121
6	`STOP`	112 111

  Zusammengesetzt wird diese RAM also durch ein Wort der Länge 168 beschrieben. Die zugehörige Gödelnummer ist somit größer als $2^{169} - 1$. Wie ist der genaue Zahlwert?

Definition 3.9 *Die Menge*

$$K_0 =_{\mathrm{def}} \{\ x \mid \textit{die RAM } M_x \textit{ hält auf Eingabe } x\ \}$$

heißt spezielles Halteproblem.

Das allgemeine Halteproblem ist $K =_{\mathrm{def}} \{\ (x, y) \mid \text{die RAM } M_x \text{ hält auf Eingabe } y\ \}$.

Theorem 3.10 *K_0 ist aufzählbar, aber nicht entscheidbar.*

Beweis: Wir zeigen die Aussages des Theorems in zwei Schritten.

1. Für Aufzählbarkeit betrachten wir folgenden Algorithmus A auf Eingabe x: Konstruiere die RAM M_x und simuliere M_x auf x. Die Formulierung des Algorithmus als RIES-Programm verbleibt als Übungsaufgabe. Es sei φ die von A berechnete Funktion. Dann gilt $K_0 = D_\varphi$. Mithin ist K_0 aufzählbar.

2. Wir führen einen Widerspruchsbeweis, um zu zeigen, dass K_0 nicht entscheidbar ist. Angenommen K_0 ist entscheidbar. Dann ist $\overline{K_0}$ aufzählbar. Es gilt also $\overline{K_0} = D_\varphi$ für eine berechenbare Funktion φ. Es sei M eine RAM, die φ berechnet, und es sei $i \in \mathbb{N}$ die Gödelnummer der RAM, d.h. $M = M_i$. Dann gilt:

$$\begin{aligned} i \in \overline{K_0} &\iff i \in D_\varphi \\ &\iff \text{die RAM } M \text{ hält bei Eingabe } i \\ &\iff \text{die RAM } M_i \text{ hält bei Eingabe } i \\ &\iff i \in K_0 \end{aligned}$$

   Dies ist jedoch ein Widerspruch. Somit ist $\overline{K_0}$ nicht aufzählbar und K_0 nicht entscheidbar.

Damit ist das Theorem bewiesen. ■

Korollar 3.11 *Die aufzählbaren Mengen sind nicht abgeschlossen unter Komplementbildung.*

Theorem 3.12 (Rice 1953) *Es sei $\mathcal{E} \subseteq$ **RAM** mit $\mathcal{E} \neq \emptyset$ und $\mathcal{E} \neq$ **RAM**. Dann ist die Menge*

$$\mathrm{C}(\mathcal{E}) =_{\text{def}} \{\ i \mid \textit{die RAM } M_i \textit{ berechnet eine Funktion } \varphi \in \mathcal{E}\ \}$$

unentscheidbar.

Beweis: (*Widerspruch*) Angenommen $\mathrm{C}(\mathcal{E})$ ist entscheidbar. Dann ist auch $\overline{\mathrm{C}(\mathcal{E})}$ entscheidbar. Wir unterscheiden zwei Fälle:

1. Es sei $\nu^1 \in \mathcal{E}$. Wegen $\mathcal{E} \neq$ **RAM** gibt es eine Funktion $\psi \in$ **RAM**$\backslash\mathcal{E}$. Es werde ψ von der RAM M berechnet. Wir betrachten folgenden Algorithmus A auf der Eingabe x:

   (a) Berechne die Nummer w derjenigen RAM M_w, die wie folgt auf einer Eingabe y arbeitet:
      i. Bestimme die RAM M_x
      ii. Simuliere die RAM M_x auf Eingabe x
      iii. Falls M_x auf Eingabe x hält, simuliere die RAM M auf Eingabe y

(b) Berechne $c_{\mathrm{C}(\mathcal{E})}(w)$

Jeder Schritt dieses Algorithmus ist effektiv durchführbar. Insbesondere wird die RAM M_w im ersten Schritt nur konstruiert und nicht simuliert. Da nach Annahme $c_{\mathrm{C}(\mathcal{E})}$ berechenbar ist, ist die von A berechnete Funktion eine totale Funktion. Für jedes $x \in \mathbb{N}$ gilt

$$\begin{aligned} x \in \overline{K_0} &\implies x \notin K_0 \\ &\implies \text{die RAM } M_x \text{ hält nicht auf Eingabe } x \\ &\implies \text{die RAM } M_w \text{ berechnet die Funktion } v^1 \\ &\implies w \in \mathrm{C}(\mathcal{E}) \end{aligned}$$

beziehungsweise

$$\begin{aligned} x \notin \overline{K_0} &\implies x \in K_0 \\ &\implies \text{die RAM } M_x \text{ hält auf Eingabe } x \\ &\implies \text{die RAM } M_w \text{ berechnet die Funktion } \psi \\ &\implies w \notin \mathrm{C}(\mathcal{E}) \end{aligned}$$

Folglich gilt $x \in \overline{K_0} \iff c_{\mathrm{C}(\mathcal{E})}(w) = 1$. Der Algorithmus A berechnet also die Funktion $c_{\overline{K_0}}$. Somit ist $\overline{K_0}$ entscheidbar. Dies ist ein Widerspruch zu Theorem 3.10.

2. Es sei $\nu^1 \notin \mathcal{E}$. Analog zum ersten Fall kann hier gezeigt werden, dass K_0 entscheidbar ist. Dies ist wiederum ein Widerspruch zu Theorem 3.10.

In beiden Fällen haben wir eine Widerspruch erhalten. Mithin ist die Annahme falsch und die Menge $\mathrm{C}(\mathcal{E})$ ist unentscheidbar. Damit ist das Theorem bewiesen. ■

Komplexitätstheorie 4

In diesem Kapitel beschäftigen wir uns mit den Grundlagen der Theorie der Berechnungskomplexität. Der Schwerpunkt wird dabei auf dem $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Problem und der Theorie der **NP**-Vollständigkeit liegen.

4.1 Laufzeit von Algorithmen

Es sei M ein Algorithmus, der die totale Funktion $f : (\Sigma^*)^m \to \Sigma^*$ berechnet. Dann ist die *Laufzeitfunktion* $t_M : (\Sigma^*)^m \to \mathbb{N}$ wie folgt für alle $x_1, \ldots, x_m \in \Sigma^*$ definiert:

$$t_M(x_1, \ldots, x_m) =_{\text{def}} \text{„Rechenzeit“ von } M \text{ auf Eingabe } (x_1, \ldots, x_m)$$

Hierbei hängt „Rechenzeit“ vom gewählten Berechnungsmodell ab.

Definition 4.1 *Es sei $t : \mathbb{N} \to \mathbb{N}$ eine totale Funktion.*

1. *Ein Algorithmus M* berechnet $f : (\Sigma^*)^m \to \Sigma^*$ in der Zeit t, *falls folgende Bedingungen erfüllt sind:*

   - *M berechnet f.*
   - *Für alle $x_1, \ldots, x_m \in \Sigma^*$ gilt $t_M(x_1, \ldots, x_m) \leq t(|x_1| + \cdots + |x_m|)$.*

   Die Eingabelänge wird mit $n =_{\text{def}} |x_1| + \cdots + |x_m|$ bezeichnet.

2. *Ein Algorithmus M* berechnet $f : (\Sigma^*)^m \to \Sigma^*$ in der Zeit $O(t)$, *falls es ein $c > 0$ gibt, sodass M die Funktion f in der Zeit $c \cdot t(n) + c$ berechnet.*

3. *Ein Algorithmus M* entscheidet $A \subseteq (\Sigma^*)^m$ in der Zeit t *(bzw. $O(t)$), falls M die charakteristische Funktion c_A in der Zeit t (bzw. $O(t)$) berechnet.*

Als Bemerkung sei angeführt, dass obige Definition übertragbar ist auf Funktionen $f : \mathbb{N}^m \to \mathbb{N}$ mittels der Festlegung $|x| =_{\text{def}} |\text{dya}(x)|$ für $x \in \mathbb{N}$. Wie leicht einzusehen ist, gilt dabei stets $2^{|x|} - 1 \leq x \leq 2^{|x|+1} - 2$.

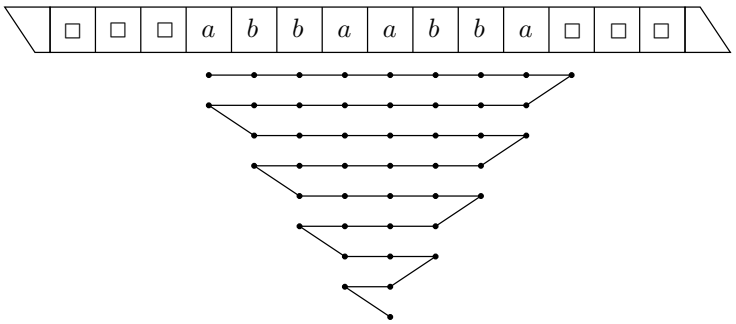
Um die Begriffsbildungen auszufüllen, müssen wir noch für die einzelnen Berechnungsmodelle festlegen, was wir jeweils unter „Rechenzeit“ verstehen wollen. Hierzu betrachten wir Turingmaschinen, RAMs und RIES-Programme.

Turingmaschinen

Naheliegend ist, jede Anwendung der Übergangsfunktion einer gegebenen Turingmaschine als einen Rechenschritt aufzufassen und dem gemäß die Rechenschritte aufzusummieren. Wir definieren deshalb für eine Turingmaschine M

$t_M(x_1, \ldots, x_m) =_{\text{def}}$ Anzahl der Rechenschritte von M auf Eingabe $(x_1, \ldots, x_m)$.

Beispiel: Wir betrachten die Turingmaschine M zur Entscheidung, ob ein gegebenens Wort eine Palindrom ist. M entscheidet also die Menge PALINDROM $=_{\text{def}} \{ w \mid w \in \{a,b\}^* \wedge w = w^R \}$. Ein Berechnungsverlauf ist in folgender Abbildung dargestellt:



Für alle $w \in \{a,b\}^*$ gilt

$$t_M(w) \leq \sum_{i=1}^{|w|+1} i = \frac{1}{2} \cdot (|w|+1)(|w|+2) = \frac{1}{2}|w|^2 + \frac{3}{2}|w| + 1.$$

Mithin entscheidet M die Menge PALINDROM in der Zeit $O(n^2)$.

Random-Access-Maschinen

Bei einer RAM ist ein natürliches Maß für die Rechenzeit die Anzahl der Befehle, die im Berechnungsverlauf ausgeführt werden. Dabei vernachlässigen wir die Größe der Zahlen, die in den Registern enthalten sind. Diese Kostenmaß wird auch als *uniformes* Kostenmaß bezeichnet. Wir definieren für eine RAM M

$t_M(x_1, \ldots, x_m) =_{\text{def}}$ Anzahl der Rechenschritte von M auf Eingabe $(x_1, \ldots, x_m)$.

Beispiel: Wir betrachten die RAM M zur Multiplikation zweier Zahlen.

```
0  R3 ← 1
1  IF R1 = 0 GOTO 5      ⎫
2  R2 ← R2 + R0          ⎪
3  R1 ← R1 − R3          ⎪  Schleife wird y-mal durchlaufen
4  GOTO 1                ⎭
5  R0 ← R2
6  STOP
```

Für $x, y \in \mathbb{N}$ gilt

$$t_M(x,y) = 4y + 3 \leq 4 \cdot 2^{|y|+1} - 8 + 3 \leq 8 \cdot 2^{|x|+|y|} - 5 \leq 8 \cdot 2^n$$

Mithin berechnet M das Produkt zweier Zahlen in der Zeit $O(2^n)$.

RIES-*Programme*

Für eine höhere Programmiersprache müssen die Kosten für die einzelnen Anweisungen definiert werden. Bei RIES legen wir fest:

- arithmetische Operationen + und − zählen jeweils als ein Schritt
- Vergleiche $\leq, <, \geq, >, =, \neq$ zählen jeweils als ein Schritt
- logische Operationen `not`, `and`, `or` zählen jeweils als ein Schritt
- Zuweisungen := zählen als ein Schritt
- arithmetische Operationen $*$ und : zählen jeweils als n Schritte.

Damit definieren wir für ein RIES-Programm P

$t_M(x_1, \ldots, x_m) =_{\text{def}}$ Anzahl der Rechenschritte von M auf Eingabe $(x_1, \ldots, x_m)$.

Die Festlegung der Kosten für die Multiplikation und Division in Abhängigkeit von der Größe der Operanden ist erklärungsbedürftig und soll in folgenden Beispiel plausibilisiert werden.

Beispiele: Wir betrachten verschiedene RIES-Programme für die Berechnung der Funktion prod.

- Für das Programm P_1 mit der Funktionsdeklaration (unter Verwendung von $*$)

```
function prod(x,y); prod:=(x*y)
```

  gilt $t_{P_1}(x,y) = |x| + |y| + 1 = n + 1$.
- Für das Programm P_2 mit der Funktionsdeklaration (ohne Verwendung von $*$)

```
function prod(x,y);
begin
  while (y>0) do begin
    z:=(z+x);
    y:=(y-1);
  end;
  prod:=z
end
```

  gilt $t_{P_2}(x,y) = 5y + 1 + 1 \leq 5 \cdot 2^{|y|+1} \leq 10 \cdot 2^{|x|+|y|} = 10 \cdot 2^n$.

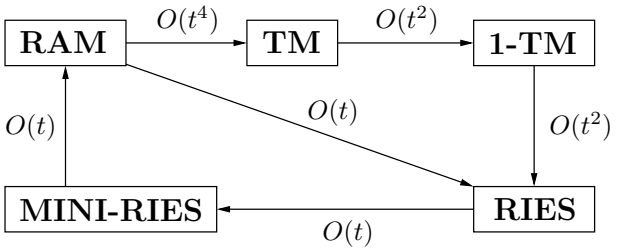
- Für das Programm P_3 mit der Funktionsdeklaration (ohne Verwendung von $*$)

```
function prod(x,y);
begin
  a[0]:=1;
  b[0]:=x;
  while (a[i]≤y) do begin
    a[(i+1)]:=(a[i]+a[i]);
    b[(i+1)]:=(b[i]+b[i]);
    i:=(i+1);
  end;
  while (i>0) do begin
    i:=(i-1);
    if (a[i]≤y) then begin
      y:=(y-a[i]);
      z:=(z+b[i]);
    end;
  end;
  prod:=z
end
```

gilt $t_{P_3}(x,y) \leq 17|y| + 5 \leq 17(|x| + |y|) + 5 = 17n + 5$. Mithin berechnet P_3 die Funktion prod in der Zeit $O(n)$.

4.2 Die Klasse P

Aus der Analyse der Simulationslaufzeiten aus Kapitel 2 ergibt sich folgendes Diagramm.



Ein Pfeil bedeutet, dass eine Berechnung, die im Quellberechnungsmodell in der Zeit t läuft, auf dem Zielberechnungsmodell in der Zeit, die auf dem Pfeil angegeben ist, simuliert werden kann.

Aus dem Diagramm kann zum Beispiel herausgelesen werden, dass alles, was auf einer RAM in der Zeit $O(t)$ berechnet werden kann, sowohl mit RIES als auch mit MINI-Ries in der Zeit $O(t)$ berechnet werden kann. Mit linearer Rechenzeit können in diesen drei Berechnungsmodellen genau dieselben Mengen entschieden werden. Turingmaschinen mit

linearer Laufzeit besitzen dagegen eine andere Berechnungsstärke. Linearzeit ist in diesem Sinne nicht robust unter dem Wechsel des Berechnungsmodells. Das ändert sich, wenn wir zu Polynomialzeit übergehen.

Definition 4.2 *Ein Algorithmus M* berechnet *$f : (\Sigma^*)^m \to \Sigma$* in Polynomialzeit*, falls ein Polynom $p(n) = a_0 + a_1 n + \cdots + a_r n^r$ mit $r, a_0, \ldots, a_r \in \mathbb{N}$ gibt, sodass M die Funktion f in der Zeit p berechnet.*

Theorem 4.3 *Für eine Funktion f sind folgende Aussagen äquivalent:*

1. *f ist durch eine RAM in Polynomialzeit berechenbar.*
2. *f ist durch ein RIES-Programm in Polynomialzeit berechenbar.*
3. *f ist durch ein MINI-RIES-Programm in Polynomialzeit berechenbar.*
4. *f ist durch eine Turingmaschine in Polynomialzeit berechenbar.*
5. *f ist durch eine* 1*-Turingmaschine in Polynomialzeit berechenbar.*

Beweis: Folgt sofort aus obigem Diagramm. ∎

Damit definieren wir:

$$\begin{aligned} \mathbf{FP} &=_{\text{def}} \{\ f \mid f \text{ ist eine in Polynomialzeit berechenbare Funktion } \} \\ \mathbf{P} &=_{\text{def}} \{\ A \mid A \text{ ist eine in Polynomialzeit entscheidbare Menge } \} \end{aligned}$$

Proposition 4.4

1. *Für alle Mengen A gilt:*
$$A \in \mathbf{P} \iff c_A \in \mathbf{FP}$$
2. *Es gilt* $\mathrm{sum}, \mathrm{md}, \mathrm{prod}, \mathrm{div} \in \mathbf{FP}$.

Theorem 4.5

1. **FP** *ist abgeschlossen unter Addition, modifizierter Subtraktion, Multiplikation, ganzzahliger Division und Hintereinanderausführung, d.h. mit $f, g \in \mathbf{FP}$ gilt auch $f + g, f \dot{-} g, f * g, f : g, f \circ g \in \mathbf{FP}$.*
2. **P** *ist abgeschlossen unter Vereinigung, Durchschnitt und Komplementbildung, d.h. mit $A, B \in \mathbf{P}$ gilt auch $A \cup B, A \cap B, \overline{A} \in \mathbf{P}$.*

Beweis: Klar (oder Übungsaufgabe). ∎

Beispiele:

1. Die Polynome $p(x) = a_0 + a_1 x + \cdots + a_m x^m$ mit $a_0, a_1, \ldots, a_m, m \in \mathbb{N}$ liegen in **FP**.
2. Es gilt $\exp \notin \mathbf{FP}$, denn aus $|\exp(2,x)| = |\mathrm{dya}(2^x)| = x > 2^{|x|} - 1$ folgt, dass eine Turingmaschine allein mindestens $2^n - 1$ Schritte benötigt, um die Ausgabe auf das Band zu schreiben.
3. Es gilt $\mathrm{mod} \in \mathbf{FP}$, denn mit dem RIES-Programm

   ```
   function mod(x,y); mod:=(x-((x:y)*y))
   ```

   kann mod in der Zeit $2 + (|x| + |y|) + |y| \leq 2(n+1) = O(n^2)$ berechnet werden.
4. Das Mustererkennungsproblem

   $$\mathrm{ME} =_{\mathrm{def}} \{\ (m,t) \mid m,t \in \{a,b\}^* \text{ und es gibt } x,y \text{ mit } t = xmy\ \}$$

   liegt in **P**, da es z.B. mit dem (als RIES-Programm beschriebenen) Algorithmus von Knuth, Morris und Pratt in der Zeit $O(n)$ entschieden werden kann.
5. Das Knotenfärbungsproblem mit zwei Farben

   $$\text{2-Colorability} =_{\mathrm{def}} \{\ G \mid G \text{ ist ein mit 2 Farben färbbarer Graph }\}$$

   liegt in **P**, wobei die Kantenmenge E eines Graphen $G = (\{1,\ldots,m\}, E)$ mit m Knoten als Adjazenzmatrix in einem Wort

   $$w_G =_{\mathrm{def}} e_{11}e_{12}\ldots e_{1m}e_{21}e_{22}\ldots e_{2m}\ldots e_{m1}e_{m2}\ldots e_{mm}$$

   mit

   $$e_{ij} =_{\mathrm{def}} \begin{cases} 2 & \text{falls } (i,j) \in E \\ 1 & \text{falls } (i,j) \notin E \end{cases}$$

   kodiert wird.

4.3 Die Klasse NP

Es gibt eine Reihe von Problemen, von denen nicht bekannt ist, ob sie in Polynomialzeit lösbar sind. Dazu gehören beispielsweise die folgenden drei Probleme:

$$\begin{aligned} k\text{-Colorability} &=_{\mathrm{def}} \{\ G \mid G \text{ ist ein mit } k \text{ Farben färbbarer Graph }\} \\ \text{Hamilton Cycle} &=_{\mathrm{def}} \{\ G \mid G \text{ ist ein hamiltonscher Graph }\} \\ \text{Sum of Subsets} &=_{\mathrm{def}} \{\ (a_1,\ldots,a_m,b) \mid \text{es gibt } I \subseteq \{1,\ldots,m\} \text{ mit } \textstyle\sum_{i\in I} a_i = b\ \} \end{aligned}$$

Gemeinsam ist diesen Problemen eine gewisse strukturelle Ähnlichkeit der Lösungen:

$$(\{1,\ldots,m\},E) \in k\text{-Colorability}$$
$$\iff (\exists c_1,\ldots,c_m)\,[c_1,\ldots,c_m \in \{1,\ldots,k\} \wedge (\forall i)(\forall j)[(i,j)\in E \rightarrow c_i \neq c_j]]$$

$$(\{1,\ldots,m\},E) \in \text{Hamilton Cycle}$$
$$\iff (\exists v_1,\ldots,v_m)\,[\{v_1,\ldots,v_m\} = \{1,\ldots,m\} \wedge (\forall i)[1 \leq i < m \rightarrow (v_i,v_{i+1})\in E] \wedge (v_m,v_1)\in E]$$

$$(a_1,\ldots,a_m,b) \in \text{Sum of Subsets}$$
$$\iff (\exists c_1,\ldots,c_m)\,[c_1,\ldots,c_m \in \{0,1\} \wedge \textstyle\sum_{i=1}^{m} c_i \cdot a_i = b]$$

Folgende Beobachtungen können dabei festgehalten werden:

- Die Charakterisierung basiert auf existenzieller Quantifizierung ($\exists\ldots$) über einer Menge von Lösungen; die Berechnungsaufgabe besteht somit in der Suche nach einer korrekten Lösung.
- Die Länge der Lösungen ist polynomiell beschränkt in der Länge der Eingabe, wobei $|(x_1,\ldots,x_m)| =_{\text{def}} |x_1| + \cdots + |x_m|$ verwendet wird.
- Die Überprüfung der Korrektheit von Lösungen ($[\ldots]$) kann in Polynomialzeit vollzogen werden.

Diese Beobachtungen verdichten wir zur Definition der Klasse **NP**.

Definition 4.6 *Eine Menge $A \subseteq \Sigma_1^*$ liegt genau dann in der Klasse* **NP***, wenn ein Polynom p und eine Menge $B \in \mathbf{P}$ existieren, sodass für alle $y \in \Sigma_2^*$ gilt:*

$$x \in A \iff (\exists y)[|y| \leq p(|x|) \wedge (x,y) \in B]$$

Das verwendete Polynom kann als monoton wachsendes Polynom mit natürlichzahligen Koeffizienten aufgefasst werden.

Proposition 4.7 *Die Probleme k-*Colorability *(mit $k \geq 3$),* Hamilton Cycle *und* Sum of Subsets *liegen in* **NP***.*

Theorem 4.8 $\mathbf{P} \subseteq \mathbf{NP}$.

Beweis: Es sei $A \in \mathbf{P}$. Wir definieren $B =_{\text{def}} \{\ (x,y) \mid x \in A\ \}$ und $p(n) =_{\text{def}} n$. Dann gilt $B \in \mathbf{P}$ und $x \in A \Leftrightarrow (x,x) \in B$. Mithin gilt $A \in \mathbf{NP}$. Damit ist die Proposition beweisen. ∎

Theorem 4.9 *Die Klasse* **NP** *ist abgeschlossen unter Vereinigung und Durchschnitt, d.h. mit* $A, B \in \mathbf{NP}$ *gilt auch* $A \cup B, A \cap B \in \mathbf{NP}$.

Wir wollen den abstrakten, logischen Ansatz von **NP** um einen algorithmischen ergänzen. Ziel ist also, Antworten auf folgende Frage zu geben: Mit welchen Ressourcen auf welchen Berechnungsmodellen können **NP**-Mengen entschieden werden?

Zunächst definieren wir dazu die Menge der in Exponentialzeit entscheidbaren Mengen:

$$\mathbf{EXP} =_{\text{def}} \{\ A \mid \text{es gibt ein Polynom } p, \text{sodass } A \text{ in der Zeit } 2^{p(n)} \text{ entscheidbar ist } \}$$

Theorem 4.10 $\mathbf{NP} \subseteq \mathbf{EXP}$.

Beweis: Es sei $A \in \mathbf{NP}$ via einem Polynom p und einer Menge $B \in \mathbf{P}$ mit $x \in A \Leftrightarrow (\exists y)[|y| \leq p(|x|) \wedge (x, y) \in B]$. Wir betrachten nun den Algorithmus, der der Reihe nach alle durch p längenbeschränkte y danach überprüft, ob $(x, y) \in B$ gilt. Ist $B \subseteq \Sigma_1^* \times \Sigma_2^*$ mit $\|\Sigma_2\| = k$, so gibt es maximal

$$\sum_{i=0}^{p(|x|)} k^i = \frac{k^{p(|x|)+1} - 1}{k - 1} \leq k^{p(|x|)+1} \leq 2^{k \cdot p(|x|)+k}$$

zu überprüfende y. Weiterhin sei B in Polynomialzeit q entscheidbar, wobei ohne Beeinträchtigung der Allgemeinheit q monoton ist. Somit benötigt eine einzelne Überprüfung, ob $(x, y) \in B$ gilt, die Laufzeit $q(|x| + |y|) \leq q(|x| + p(|x|))$. Insgesamt erhalten wir für die Laufzeit des obigen Algorithmus

$$O\left(2^{k \cdot p(|x|)+k} \cdot q(|x| + p(|x|))\right)$$

Mithin gilt $A \in \mathbf{EXP}$ mit dem Polynom $c \cdot (k \cdot p(n) + k) \cdot q(n + p(n))$, wobei $c > 0$ die in O versteckte Konstante ist. Damit ist das Theorem bewiesen. ∎

Algorithmen mit exponentieller Laufzeit sind ineffizient. Wenn wir beispielsweise annehmen, dass ein Rechner mit 10^8 Operationen pro Sekunde (also 100 MIPS) arbeitet, so ergeben sich ungefähr folgende physikalisch messbaren Berechnungsdauern für unterschiedliche Eingabelängen im Vergleich mit Laufzeitschranken geringerer Größenordnung:

Eingabelänge	20	40	60	100	300
n	10^{-7} sec	10^{-7} sec	10^{-6} sec	10^{-6} sec	10^{-6} sec
n^2	10^{-6} sec	10^{-5} sec	10^{-5} sec	10^{-4} sec	10^{-3} sec
n^3	10^{-4} sec	10^{-3} sec	10^{-3} sec	10^{-2} sec	10^{-1} sec
$2^{\frac{n}{\log n}}$	10^{-6} sec	10^{-5} sec	10^{-4} sec	10^{-2} sec	788 Tage
2^n	10^{-2} sec	183 min	365 Jahre	–	–

Folgende Probleme, die die Lage von **NP** zwischen **P** und **EXP** betreffen, sind derzeit noch ungelöst:

- Gilt $\mathbf{P} = \mathbf{NP}$? – Die Vermutung ist $\mathbf{P} \neq \mathbf{NP}$. Dies ist das berühmte $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Problem.

- Gilt $\mathbf{NP} = \mathbf{EXP}$? – Die Vermutung ist ebenfalls $\mathbf{NP} \neq \mathbf{EXP}$. Die Frage ist eng mit dem $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Problem verbunden, denn es gilt: $\mathbf{P} = \mathbf{NP} \Rightarrow \mathbf{NP} \neq \mathbf{EXP}$.

- Ist **NP** abgeschlossen unter Komplementbildung? – Auch hier ist die Vermutung, dass dies nicht gilt. Formal kann das Problem auch anders ausgedrückt werden: Gilt $\mathbf{NP} = \text{co-}\mathbf{NP}$? Hierbeit ist $\text{co-}\mathbf{NP} =_{\text{def}} \{ \overline{A} \mid A \in \mathbf{NP} \}$. Die Verbindung zum $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Problem ist durch folgende Implikation gegeben: $\mathbf{NP} \neq \text{co-}\mathbf{NP} \Rightarrow \mathbf{P} \neq \mathbf{NP}$.

Der Name „NP“ steht für „nichtdeterministische Polynomialzeit“. Wir führen nun nichtdeterministische Algorithmen ein:

1. *Nichtdeterministische Turingmaschine:* Eine Turingmaschine wird nichtdeterministisch, wenn wir mehrere Befehle mit gleicher linker Seite erlauben, d.h. es gibt Zustand-Bandinhalt-Kombinationen sa mit $s \in Z$, $a \in \Sigma$ und

$$\begin{aligned} sa &\rightarrow s'_1 a'_1 \sigma_1 \\ sa &\rightarrow s'_2 a'_2 \sigma_2 \\ &\vdots \\ sa &\rightarrow s'_k a'_k \sigma_k \end{aligned}$$

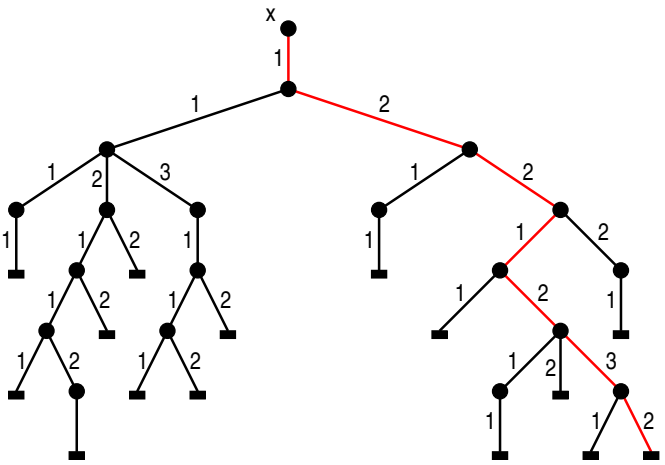
2. *Nichtdeterministische Random-Access-Maschine:* Eine RAM wird nichtdeterministisch, wenn wir mehrere Befehle mit gleicher Befehlsnummer erlauben.

3. *Nichtdeterministisches RIES:* Wir ergänzen RIES um das folgende Sprachelement. Sind $s_1, \ldots, s_k$ Anweisungen, so ist auch

   `begin` s_1 `|` s_2 `|` ... `|` s_k `end`

   eine Anweisung.

Nichtdeterministische Berechnungen lassen sich in einem Berechnungsbaum darstellen.



Hierbei steht jeder Knoten für einen nichtdeterministischen Befehl im Ablauf der Berechnung. Die Verzweigung gibt an, welche Alternative gewählt wird. Ein Verzweigung mit nur einer Alternative entspricht der Ausführung eines deterministischen Befehls. Ein Pfad von der Wurzel zu einem Blatt heißt *Berechnungspfad*.

Definition 4.11 *Es sei M ein nichtdeterministscher Algorithmus.*

1. M akzeptiert $x \in (\Sigma^*)^m$ *genau dann, wenn es einen Berechnungspfad von M auf x gibt, auf dem der Wert* 1 *ausgegeben wird.*
2. $L(M) =_{\text{def}} \{\ x \mid M \text{ akzeptiert } x\ \}$ *heißt die* von M akzeptierte Menge.

Einen Berechnungspfad, auf dem 1 ausgegeben wird, nennen wir auch einen *akzeptierenden* Berechnungspfad.

Definition 4.12 *Es sei $t : \mathbb{N} \to \mathbb{N}$ eine totale Funktion.*

1. *Ein nichtdeterministischer Algorithmus M* akzeptiert A in der Zeit t, *falls gilt:*
   - $L(M) = A$.
   - *Die Rechenzeit entlang eines jeden Berechnungspfades von M auf x ist höchsten $t(|x|)$.*
2. *Ein nichtdeterministischer Algorithmus M* akzeptiert A in der Zeit $O(t)$, *falls ein $c > 0$ existiert, sodass M die Menge A in der Zeit $c \cdot t(n) + c$ akzeptiert.*

Beispiel: Das folgende nichtdeterministische RIES-Programm akzeptiert die Menge Sum of Subsets:

```
function sos(a[],b,m)
begin
  for i:=1 to m do begin
    s:=s | s:=(s+a[i])
  end
  if (s=b) then sos:=1
end
```

Die Laufzeitanalyse ergibt, dass jeder Berechnungspfad in $O(m)$ abgearbeitet wird. Wegen $m \leq |a[]|$ (m ist die Größe des Feldes $a[]$) akzeptiert das Programm SUM OF SUBSETS in $O(n)$.

Theorem 4.13 *Für eine Menge A sind folgende Aussagen äquivalent:*

1. $A \in \mathbf{NP}$.
2. *A wird durch eine nichtdeterministische RAM in Polynomialzeit akzeptiert.*
3. *A wird durch eine nichtdeterministische Turingmaschine in Polynomialzeit akzeptiert.*
4. *A wird durch eine nichtdeterministische 1-Turingmaschine in Polynomialzeit akzeptiert.*
5. *A wird durch ein nichtdeterministisches RIES-Programm in Polynomialzeit akzeptiert.*

Beweis: Wir zeigen die Äquivalenzen schrittweise:

- Die Äquivalenzen 2. $\Leftrightarrow$ 3. 2. $\Leftrightarrow$ 4., 2. $\Leftrightarrow$ 5., 3. $\Leftrightarrow$ 4., 3. $\Leftrightarrow$ 5. und 4. $\Leftrightarrow$ 5. folgen aus Theorem 4.3.
- Für die Implikation 1. $\Rightarrow$ 5. sei $A \in \mathbf{NP}$, d.h. es gibt ein Polynom p und eine Menge $B \in \mathbf{P}$ mit $x \in A \Leftrightarrow (\exists y)[|y| \leq p(|x|) \wedge (x,y) \in B]$. Es sei $B \subseteq \Sigma_1^* \times \Sigma_2^*$ mit $\Sigma_2 = \{a_1, \ldots, a_k\}$. Da $B \in \mathbf{P}$ gilt, gibt es ein RIES-Programm

  `function b(x,y); s`,

  das B in Polynomialzeit entscheidet. Wir betrachten nun das folgende (nicht ganz vollständig ausformulierte) nichtdeterministische RIES-Programm P_A:

  ```
  function a(x);
  begin
    y:=ε;                          /* Kodierung des leeren Wortes verwenden */
    for i:=1 to p(|x|) do begin
      y:=y | y:=ya_1 | ... | y:=ya_k
                    /* Konkatenation als Operation auf Zahlen ausdrücken */
  ```

```
    end;
    a:=b(x,y)
  end;
  function b(x,y);s
```

Klarerweise akzeptiert P_A die Menge A in Polynomialzeit.

- Für die Implikation 3. $\Rightarrow$ 1. sei M eine nichtdeterministische Polynomialzeit-Turingmaschine, die die Menge A in der Zeit p akzeptiert, wobei p ein Polynom ist. Es sei r die maximale Zahl nicht deterministischer Befehle. Ein Berechnungspfad kann somit als ein Wort s aus der Menge $\{1, \ldots, r\}^*$ aufgefasst werden. Es gilt stets $|s| \leq p(|x|)$. Es muss allerdings nicht jedes Wort mit dieser Eigenschaft auch einem Berechnungspfad entsprechen. Dies ist jedoch leicht zu testen. Wir definieren nun eine Menge B wie folgt:

  $B =_{\text{def}} \{ (x, y) \mid y \in \{1, \ldots, r\}^*$ ist ein Berechnungspfad von $M(x)$ mit Ausgabe 1 $\}$

  Dann gilt $B \in \mathbf{P}$ und

  $$\begin{aligned} & x \in A \\ & \quad \Longleftrightarrow \quad \text{es gibt einen Berechnungspfad } y \text{ mit } |y| \leq p(|x|), \text{ auf dem } M(x) \text{ den Wert 1 ausgibt} \\ & \quad \Longleftrightarrow \quad (\exists y)[|y| \leq p(|x|) \wedge (x, y) \in B] \end{aligned}$$

  Mithin gilt $A \in \mathbf{NP}$.

Damit ist das Theorem bewiesen. ■

4.4 NP-vollständige Mengen

Wegen $\mathbf{P} \subseteq \mathbf{NP}$ können die Menge aus $\mathbf{P}$ als die einfachsten Mengen in $\mathbf{NP}$ aufgefasst werden. In diesem Abschnitt wollen wir uns um die schwierigsten Mengen in $\mathbf{NP}$ kümmern. Dazu benötigen wir eine Möglichkeit, Mengen hinsichtlich ihres algorithmischen Schwierigkeitsgrad vergleichen zu können.

Definition 4.14 *Es seien $A \subseteq \Sigma_1^*$ und $B \subseteq \Sigma_2^*$. Dann heißt A* auf B in Polynomialzeit reduzierbar*, symbolisch $A \leq_m^p B$, falls eine Funktion $f : \Sigma_1^* \to \Sigma_2^*$ aus* $\mathbf{FP}$ *existiert, sodass*

$$x \in A \Longleftrightarrow f(x) \in B$$

für alle $x \in \Sigma_1^*$ gilt.

In Ergänzung zu dieser Definition halten wir fest:

$$A \equiv_m^p B \Longleftrightarrow_{\text{def}} A \leq_m^p B \wedge B \leq_m^p A$$

Beispiel: Wir betrachten als weiteres Problem das Rucksack-Problem

$$\begin{aligned} \textsc{Knapsack} =_{\text{def}} \{\, (g_1, \ldots, g_m, k_1, \ldots, k_m, G, K \mid \text{es gibt } I \subseteq \{1, \ldots, m\} \\ \text{mit } \textstyle\sum_{i \in I} g_i \leq G \text{ und } \sum_{i \in I} k_i \geq K \} \end{aligned}$$

Wir wollen zeigen, dass Sum of Subsets $\leq_m^p$ Knapsack gilt. Dazu definieren wir eine Funktion f wie folgt:

$$f(a_1, \ldots, a_m, b) =_{\text{def}} (a_1, \ldots, a_m, a_1, \ldots, a_m, b, b)$$

Es gilt $f \in \mathbf{FP}$ und für alle Instanzen $(a_1, \ldots, a_m, b)$:

$$\begin{aligned} &(a_1, \ldots, a_m, b) \in \textsc{Sum of Subsets} \\ &\quad \Longleftrightarrow \quad \text{es gibt } I \subseteq \{1, \ldots, m\} \text{ mit } \textstyle\sum_{i \in I} a_i = b \\ &\quad \Longleftrightarrow \quad \text{es gibt } I \subseteq \{1, \ldots, m\} \text{ mit } \textstyle\sum_{i \in I} a_i \leq b \text{ und } \sum_{i \in I} a_i \geq b \\ &\quad \Longleftrightarrow \quad (a_1, \ldots, a_m, a_1, \ldots, a_m, b, b) \in \textsc{Knapsack} \\ &\quad \Longleftrightarrow \quad f(a_1, \ldots, a_m, b) \in \textsc{Knapsack} \end{aligned}$$

Proposition 4.15

1. *Die Relation $\leq_m^p$ ist reflexiv und transitiv.*
2. *Für alle Mengen A und B gilt:*

$$A \leq_m^p B \Longleftrightarrow \overline{A} \leq_m^p \overline{B}$$

Beweis: Wir zeigen die Aussagen einzeln.

1. Zur Reflexivität: Wegen $x \in A \Leftrightarrow x \in A$ gilt $A \leq_m^p A$ mittels $\mathrm{I}_1^1 \in \mathbf{FP}$. Zum Nachweis der Transitivität gelte $A \leq_m^p B$ via $f \in \mathbf{FP}$ und $B \leq_m^p C$ via $g \in \mathbf{FP}$. Wir erhalten:

$$\begin{aligned} x \in A \quad &\Longleftrightarrow \quad f(x) \in B \\ &\Longleftrightarrow \quad g(f(x)) \in C \\ &\Longleftrightarrow \quad (g \circ f)(x) \in C \end{aligned}$$

   Nach Theorem 4.5 gilt $g \circ f \in \mathbf{FP}$. Somit folgt $A \leq_m^p C$ via $g \circ f$.

2. Gilt $x \in A \Leftrightarrow f(x) \in B$ für $f \in \mathbf{FP}$, so gilt auch $x \in \overline{A} \Leftrightarrow f(x) \in \overline{B}$ für $f \in \mathbf{FP}$.

Damit ist die Proposition bewiesen. ■

Theorem 4.16 **P** *und* **NP** *sind abgeschlossen unter* $\leq_m^p$. *Mit anderen Worten:*

1. *Ist* $B \in \mathbf{P}$ *und gilt* $A \leq_m^p B$, *so ist* $A \in \mathbf{P}$.
2. *Ist* $B \in \mathbf{NP}$ *und gilt* $A \leq_m^p B$, *so ist* $A \in \mathbf{NP}$.

Beweis: Wir zeigen beide Aussagen einzeln.

1. Es seien $B \in \mathbf{P}$ und $A \leq_m^p B$ via $f \in \mathbf{FP}$. Somit gilt $c_B \in \mathbf{FP}$ und auch $c_A = c_B \circ f \in \mathbf{FP}$ (nach Theorem 4.5). Mithin ist $A \in \mathbf{P}$.
2. Es seien $B \in \mathbf{NP}$ und $A \leq_m^p B$ via $f \in \mathbf{FP}$. Es sei M eine nichtdeterministische Turingmaschine, die B in Polynmialzeit mit dem Polynom p akzeptiert. Weiterhin sei q das Polynom, das die Laufzeit zur Berechnung von f beschränkt. Wir betrachten die nichtdeterministische Turingmaschine M', die auf Eingabe x zunächst $f(x)$ berechnet und anschließend M auf $f(x)$ simuliert. Dann gilt:

$$\begin{aligned} M' \text{ akzeptiert } x &\iff M \text{ akzeptiert } f(x) \\ &\iff f(x) \in B \\ &\iff x \in A \end{aligned}$$

   Die Rechenzeit von M' schätzen wir wie folgt ab: Es gilt $|f(x)| \leq q(|x|)$. Somit hat jeder Berechnungspfad eine Laufzeit von höchstens $q(|x|) + p(|x| + q(|x|))$. Damit akzeptiert M' die Menge A in der Polynomialzeit $q(n) + p(n + q(n))$. Mithin ist $A \in \mathbf{NP}$.

Damit ist das Theorem bewiesen. ■

Theorem 4.17 *Es sei* $B \neq \emptyset$ *und* $\overline{B} \neq \emptyset$. *Dann gilt* $A \leq_m^p B$ *für alle* $A \in \mathbf{P}$.

Beweis: Wir wählen zwei Elemente $a \in B$ und $b \in \overline{B}$, die nach Voraussetzung an B existieren müssen. Dann definieren wir eine Funktion f wie folgt:

$$f(x) =_{\text{def}} \begin{cases} a & \text{falls } x \in A \\ b & \text{falls } x \notin A \end{cases}$$

Somith gilt $x \in A \Leftrightarrow f(x) \in B$ für alle x. Außderdem gilt $f(x) = a \cdot c_A(x) + b \cdot (1 \dot{-} c_A(x))$, d.h. $f \in \mathbf{FP}$ (nach Theorem 4.5). Mithin gilt $A \leq_m^p B$. Damit ist das Theorem bewiesen. ■

Definition 4.18 *Eine Menge* B *heißt* **NP**-vollständig, *falls* $B \in \mathbf{NP}$ *gilt und* $A \leq_m^p B$ *für alle* $A \in \mathbf{NP}$ *gilt.*

Theorem 4.19 *Es seien A und B Mengen.*

1. *Sind A und B **NP**-vollständig, so gilt $A \equiv_m^p B$.*
2. *Ist A **NP**-vollständig, so gilt:*
$$A \in \mathbf{P} \Longleftrightarrow \mathbf{P} = \mathbf{NP}$$
3. *Ist $B \in \mathbf{NP}$, ist A **NP**-vollständig und gilt $A \leq_m^p B$, so ist B **NP**-vollständig.*

Beweis: Die Aussagen folgen sofort aus Definition 4.18 mit Hilfe von Proposition 4.15 und Theorem 4.16. ■

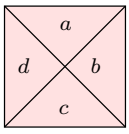
Theorem 4.19 hat einige Konsequenzen, die wir mit folgenden Bemerkungen festhalten wollen.

1. Um das $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Problem zu lösen, genügt es, für ein konkretes **NP**-vollständiges Problem A entweder einen Polynomialzeitalgorithmus anzugeben oder zu zeigen, dass es für A keinen Polynomialzeitalgorithmus geben kann. Hierbei kann unter einigen tausend, natürlichen Problemen wählen, für die die **NP**-Vollständigkeit nachgewiesen worden ist (siehe z.B. [GJ79]).

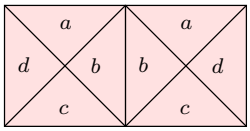
2. Andersherum zeigt der Nachweis der **NP**-Vollständigkeit für ein Problem A eine bedingte untere Schranke für A: Ist $\mathbf{P} \neq \mathbf{NP}$, so gibt es keinen Polynomialzeitalgorithmus für A.

3. Zur Festellung der **NP**-Vollständigkeit für ein gegebenes Problem können wir ein beliebiges **NP**-vollständiges Problem wählen und eine Reduktion auf unser Problem nachweisen; der Nachweis ist nicht für alle **NP**-Mengen zu erbringen.

Wir wollen uns davon überzeugen, dass es tatsächlich **NP**-vollständige Mengen gibt. Dazu betrachten wir folgendes *Puzzleproblem*: Puzzleteile haben die Form



Zwei Puzzleteile können aneinander gelegt werden, wenn die sich berührenden Seiten übereinsteimmen, z.B.



Wir verfügen jedoch nur über Puzzleteile aus einer Galerie $T_1, \ldots, T_k$ von Typen, dafür aber von jedem Typ T_i beliebig viele Teile. Das zu lösende Problem besteht darin, einen gegebenen Rahmen R zu füllen, wenn dies möglich ist.

Formal repräsentieren wir ein Puzzlespiel wie folgt: Die auf den Puzzleteilen vorkommenden Symbole kodieren wir als Wörter über dem Alphabet $\Sigma_m =_{\text{def}} \{0,1\}^m$, wobei m groß genug ist, um alle Symbole zu kodieren. Ein Puzzleteil T ist dann gegeben als Wort aus $(\Sigma_m)^4$ (wobei wir eine feste Orientierung in Uhrzeigersinn annehmen). Ein Rahmen R der Größe $t \times t$ ist ein Wort aus $(\Sigma_m)^{4t}$ (wobei wir auch hier den Rahmen in Uhrzeigersinn lesen). Alle lösbaren Puzzlespiele fassen wir in der Menge PUZZLE zusammen:

$$\begin{aligned}\text{PUZZLE} =_{\text{def}} \{\ (m,k,t,R,T_1,\ldots,T_k) \mid\ & m,k,t \geq 1, R \in (\Sigma_m)^{4t}, T_1,\ldots,T_k \in (\Sigma_m)^4 \\ & \text{und } R \text{ kann mit Puzzleteilen der Typen} \\ & T_1,\ldots,T_k \text{ gefüllt werden } \}\end{aligned}$$

Theorem 4.20 PUZZLE *ist* **NP***-vollständig.*

Beweis: Es sind zwei Eigenschaften für PUZZLE zu zeigen.

1. PUZZLE $\in$ **NP**: Eine Füllung eines Rahmens R der Größe $t \times t$ mit Puzzleteilen vom Typ $T_1, \ldots, T_k$ ist eine totale Funktion

   $$s : \{1,\ldots,t\}^2 \to \{1,\ldots,k\}.$$

   Wie definieren eine Menge B als

   $$\begin{aligned}B =_{\text{def}} \{\ (m,k,t,R,T_1,\ldots,T_k,s) \mid\ & m,k,t \geq 1, R \in (\Sigma_m)^{4t}, T_1,\ldots,T_k \in (\Sigma_m)^4 \\ & s : \{1,\ldots,t\}^2 \to \{1,\ldots,k\} \text{ und } s \text{ füllt den} \\ & \text{Rahmen } R \text{ korrekt aus } \}\end{aligned}$$

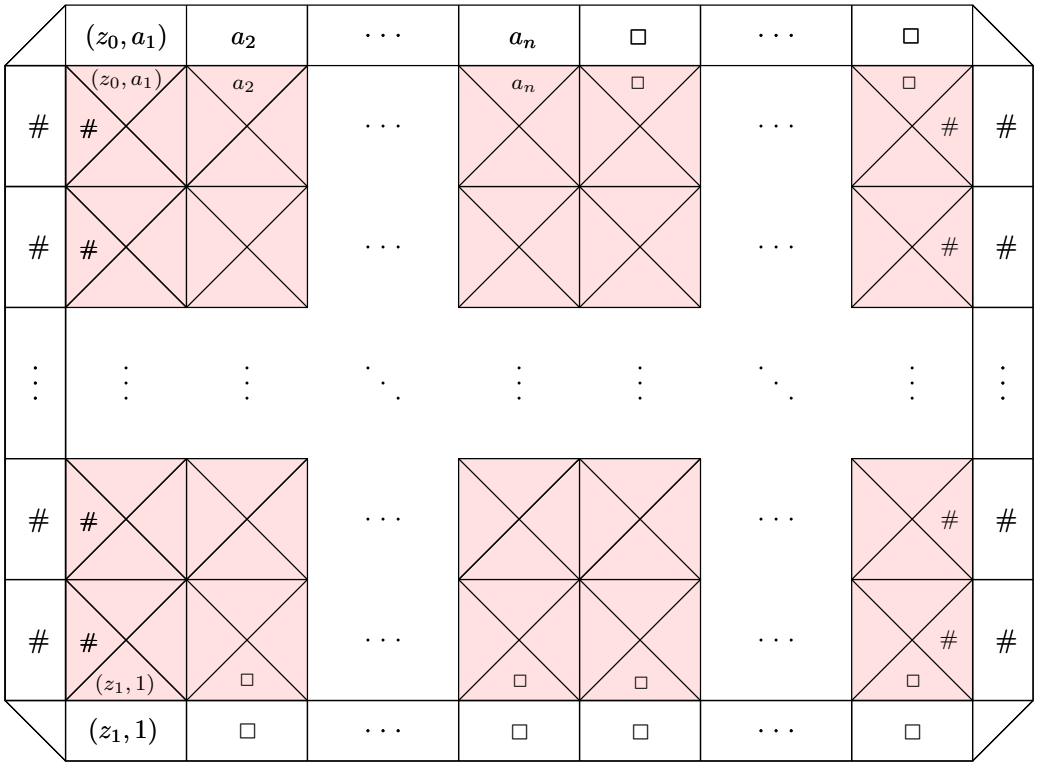
   Dann gilt $B \in \mathbf{P}$ (es sind t^2 Felder zu überprüfen) sowie

   $$\begin{aligned}&(m,k,t,R,T_1,\ldots,T_k) \in \text{PUZZLE} \\ &\qquad \iff (\exists s)[s : \{1,\ldots,t\}^2 \to \{1,\ldots,k\} \wedge (m,k,t,R,T_1,\ldots,T_k,s) \in B]\end{aligned}$$

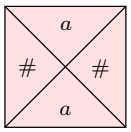
   mit $|s| \leq |k| \cdot t^2 \leq |R|^2 \leq |(m,k,t,R,T_1,\ldots,T_k)|^2$. Mithin gilt PUZZLE $\in$ **NP**.

2. $A \leq_m^p$ PUZZLE für beliebige $A \in$ **NP**: Für eine Menge $A \in$ **NP** gibt es eine Polynom p und eine nichtdeterministische 1-Turingmaschine M, die A in der Zeit p akzeptiert. Ohne Beeinträchtigung der Allgemeinheit führt M in der linkesten Zelle nie einen Linksbefehl aus. Wir beschreiben eine (akzeptierende) Berechnung von M auf

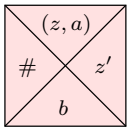
Eingabe $x = a_1 \dots a_n$ durch folgenden Rahmen (und einer angedeuteten korrekten Füllung):



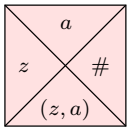
Um die Übergänge zwischen den Konfigurationen in den einzeln Schritten zu beschreiben, verwenden wir Puzzleteile von Typen folgender Kategorien:



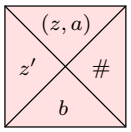
Kopf zeigt nicht auf diese Zelle



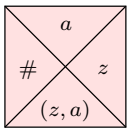
Kopf zeigt auf diese Zelle, Befehl ist $za \to z'bR$



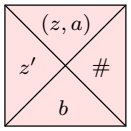
Kopf gelangt mit Zustand z durch Rechtsbewegung auf diese Zelle



Kopf zeigt auf diese Zelle, Befehl ist $za \to z'bL$



Kopf gelangt mit Zustand z durch Linksbewegung auf diese Zelle



Kopf bleibt stehen, Befehl ist $za \to z'b0$

Die Puzzletypen hängen nur von den Befehlen von M ab. Somit gibt es eine feste Anzahl von Typen $T_1^M, \ldots, T_k^M$. Der $p(n) \times p(n)$-Rahmen hängt von M und x ab:

$$R_{M,x} =_{\text{def}} \underbrace{(z_0, a_1)a_2 \ldots, a_n \square \ldots \square}_{p(n)} \underbrace{\# \ldots \#}_{p(n)} \underbrace{\square \ldots \square (z_1, 1)}_{p(n)} \underbrace{\# \ldots \#}_{p(n)}$$

Es gilt

$x \in A$

$\iff$ es gibt eine akzeptierenden Berechnungspfad von $M(x)$ der Länge $\leq p(|x|)$

$\iff$ es gibt eine Füllung vom $R_{M,x}$ mit Puzzleteilen der Typen $T_1^M, \ldots, T_k^m$

$\iff$ $(m, k, p(|x|), R_{M,x}, T_1^M, \ldots, T_k^M) \in$ Puzzle

Hierbei ist m groß genug, um alle Puzzleteile zu kodieren (d.h. Zustände, Bandalphabet, Paare, #). Die Funktion f mit

$$f : x \mapsto (m, k, p(|x|), R_{M,x}, T_1^M, \ldots, T_k^M)$$

ist in Polynomialzeit berechenbar, und es gilt

$$x \in A \iff f(x) \in \text{Puzzle}.$$

Somit gilt $A \leq_m^p$ Puzzle via f.

Damit ist das Theorem bewiesen. ∎

Automatentheorie 5

5.1 Endliche Automaten

Definition 5.1 *Ein Tupel $A = (\Sigma, Z, f, z_0, Z')$ heißt* endlicher Automat*, falls gilt:*

- Σ *ist eine endliche, nichtleere Menge, das* Eingabealphabet*,*
- Z *ist eine endliche, nichtleere Menge, die* Zustandsmenge*,*
- $f : Z \times \Sigma \to Z$ *ist eine totale Funktion, die* Überführungsfunktion*,*
- $z_0 \in Z$ *ist der* Startzustand *und*
- $Z' \subseteq Z$ *ist die Menge der* akzeptierenden Zustände*.*

Für einen gegebenen endlichen Automaten $A = (\Sigma, Z, f, z_0, Z')$ definieren wir induktiv die *erweiterte Überführungsfunktion* $\bar{f} : Z \times \Sigma^* \to Z$:

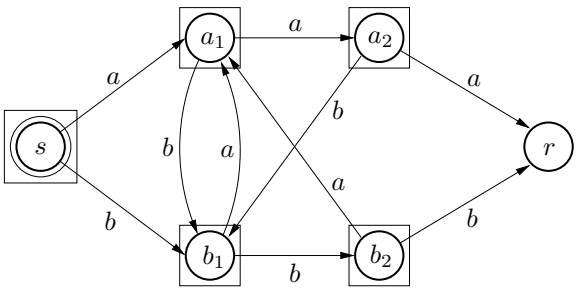
$$\begin{array}{rcll} \bar{f}(z, \varepsilon) & =_{\text{def}} & z & \text{für alle } z \in Z \\ \bar{f}(z, wa) & =_{\text{def}} & f\left(\bar{f}(z, w), a\right) & \text{für alle } z \in Z, w \in \Sigma^*, a \in \Sigma \end{array}$$

Definition 5.2 *Es sei $A = (\Sigma, Z, f, z_0, Z')$ ein endlicher Automat.*

1. *Ein Wort $w \in \Sigma^*$ heißt* von A akzeptiert*, falls $\bar{f}(z_0, w) \in Z'$ gilt.*
2. $L(A) =_{\text{def}} \{\ w \mid w \in \Sigma^*$ *und* $\bar{f}(z_0, w) \in Z'\ \}$ *heißt* die von A akzeptierte Wortmenge*.*

Beispiele:

- Folgende Abbildung veranschaulicht einen endlichen Automaten, der genau die Wörter über $\{a, b\}$, in denen kein Buchstabe dreimal hintereinander vorkommt, akzeptiert:



Formal ist der Automat definiert als

$$A =_{\text{def}} (\{a, b\}, \{s, a_1, a_2, b_1, b_2, r\}, f, s, \{s, a_1, a_2, b_1, b_2\}),$$

wobei die Überführungsfunktion f den Pfeilen in obiger Abbildung entnommen werden kann.

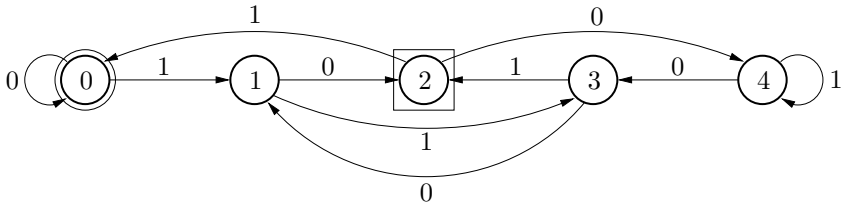
- Wir wollen einen endlichen Automaten konstruieren, der die Menge aller Binärzahlen, die den Divisionrest 2 bei Division durch 5 ergeben, akzeptiert. Dazu verwenden wir, dass für jedes Wort $w \in \{0,1\}^*$ und jeden Buchstaben $a \in \{0,1\}$ gilt:

  $$\text{val}(wa) = 2 \cdot \text{val}(w) + a,$$

  wobei $\text{val} : \{0,1\}^* \to \mathbb{N}$ jedem Binärwort den Dezimalwert zuordnet. Mit den Regeln der Modulo-Arithmik folgt weiterhin:

  $$\text{mod}(\text{val}(wa), 5) = \text{mod}(2 \cdot \text{mod}(\text{val}(w), 5) + \text{mod}(a, 5), 5).$$

  Damit ergibt sich als endlicher Automat:



Mit **EA** bezeichnen wir die Klasse der von endlichen Automaten akzeptierten Mengen:

$$\textbf{EA} =_{\text{def}} \{\ L \mid \text{es gibt einen endlichen Automaten } A \text{ mit } L = L(A)\ \}$$

5.2 Nichtdeterministische endliche Automaten

Definition 5.3 *Ein Tupel* $A = (\Sigma, Z, f, z_0, Z')$ *heißt* nichtdeterministischer endlicher Automat*, falls gilt:*

- Σ *ist eine endliche, nichtleere Menge, das* Eingabealphabet*,*
- Z *ist eine endliche, nichtleere Menge, die* Zustandsmenge*,*
- $f : Z \times \Sigma \to \mathcal{P}(Z)$ *ist eine totale Funktion, die* Überführungsfunktion*,*
- $z_0 \in Z$ *ist der* Startzustand *und*
- $Z' \subseteq Z$ *ist die Menge der* akzeptierenden Zustände*.*

Auch für einen nichtdeterministische Automaten $A = (\Sigma, Z, f, z_0, Z')$ führen wir wieder eine *erweiterte Überführungsfunktion* $\bar{f} : Z \times \Sigma^* \to \mathcal{P}(Z)$ ein:

$$\begin{aligned} \bar{f}(z, \varepsilon) &=_{\text{def}} \{z\} && \text{für alle } z \in Z \\ \bar{f}(z, wa) &=_{\text{def}} \bigcup_{z' \in \bar{f}(z,w)} f(z', a) && \text{für alle } z \in Z, w \in \Sigma^*, a \in \Sigma \end{aligned}$$

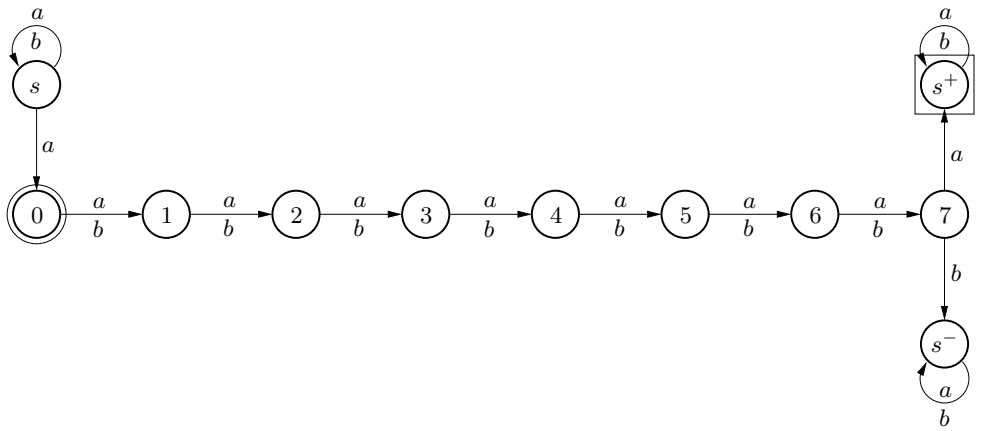
Definition 5.4 *Es sei $A = (\Sigma, Z, f, z_0, Z')$ ein nichtdeterministischer endlicher Automat.*

1. *Ein Wort $w \in \Sigma^*$ heißt* von A akzeptiert*, falls $\bar{f}(z_0, w) \cap Z' \neq \emptyset$ gilt.*
2. *$L(A) =_{\text{def}} \{\ w \mid w \in \Sigma^*$ und $\bar{f}(z_0, w) \cap Z' \neq \emptyset\ \}$ heißt* die von A akzeptierte Wortmenge.

Beispiel: Die wie folgt definierte Menge

$$L_7 =_{\text{def}} \{a, b\}^* \cdot \{a\} \cdot \{a, b\}^7 \cdot \{a\} \cdot \{a, b\}^*$$

kann mit dem in der Abbildung dargestellten nichtdeterministischen endlichen Automaten akzeptiert werden:



Der Automat ist nur im Startzustand nichtdeterministisch. Damit raten wir uns ein a, für das wir uns in den Zustïden $0, 1, \ldots, 7$ die Anzahl der trennenden Symbole merken, bevor wieder ein a kommen muss.

Mit **NEA** bezeichnen wir die Klasse der von nichtdeterministischen endlichen Automaten akzeptierten Mengen:

$$\textbf{NEA} =_{\text{def}} \{\ L(A) \mid A \text{ ist ein nichtdeterministischer endlicher Automat}\ \}$$

Theorem 5.5 EA = NEA

Beweis: Wir zeigen beide Inklusion einzeln.

($\subseteq$): Es sei $L \in \mathbf{EA}$ mittels eines endlichen Automaten $A = (\Sigma, Z, f, z_0, Z')$. Wir definieren einen nichtdeterministischen endlichen Automaten $A' = (\Sigma, Z, f', z_0, Z')$ mit

$$f' : (z, a) \mapsto \{f(z, a)\}$$

Somit gilt $\|f'(z, a)\| = 1$ für alle $z \in Z$ und $a \in \Sigma$. Es folgt

$$\bar{f}'(z_0, w) = \{\bar{f}(z_0, w)\}$$

für alle $w \in \Sigma^*$. Mithin ist $L = L(A) = L(A')$, also $L \in \mathbf{NEA}$.

($\subseteq$): Es sei $L \in \mathbf{NEA}$ mittels eines nichtdeterministischen endlichen Automaten $A = (\Sigma, Z, f, z_0, Z')$. Ein zu A äquivalenter, deterministischer endlicher Automat wird unter Verwendung der *Potenzmengenkonstruktion* angegeben. Wir definieren den endlichen Automaten $A' =_{\text{def}} (\Sigma, \mathcal{P}(Z), f', \{z_0\}, Z'')$ mit

$$f'(S, a) =_{\text{def}} \bigcup_{z \in S} f(z, a) \qquad \text{für alle } S \subseteq Z,\ a \in \Sigma$$

und $Z'' =_{\text{def}} \{\ S \mid S \subseteq Z \text{ und } S \cap Z' \neq \emptyset\ \}$. Dann gilt die Gleichung

$$\bar{f}'(\{z\}, w) = \bar{f}(z, w) \quad \text{für alle } z \in Z \text{ und } w \in \Sigma^* \tag{5.1}$$

Dies ist mittels Induktion über die Länge n der Wörter w einzusehen:

- *Induktionsanfang:* Es sei $n = 0$, d.h. $w = \varepsilon$. Nach Definition der erweiterten Überführungsfunktionen gilt $\bar{f}'(\{z\}, \varepsilon) = \{z\} = \bar{f}(z, \varepsilon)$.
- *Induktionsschritt:* Es sei $n > 0$, d.h. $w = ua$ mit $u \in \Sigma^*$ und $a \in \Sigma$. Dann gilt

$$\begin{aligned}
\bar{f}'(\{z\}, w) &= \bar{f}'(\{z\}, ua) \\
&= f'(\bar{f}'(\{z\}, u), a) && \text{(nach Definition von } \bar{f}') \\
&= f'(\bar{f}(z, u), a) && \text{(nach Induktionsvoraussetzung)} \\
&= \bigcup_{z' \in \bar{f}(z,u)} f(z', a) && \text{(nach Definition von } f') \\
&= \bar{f}(z, ua) && \text{(nach Definition von } \bar{f}) \\
&= \bar{f}(z, w)
\end{aligned}$$

Aus Gleichung (5.1) folgt nun:

$$\begin{aligned}
w \in L(A) &\iff \bar{f}(z_0, w) \cap Z' \neq \emptyset \\
&\iff \bar{f}'(\{z_0\}, w) \cap Z' \neq \emptyset && \text{(nach Gleichung (5.1))} \\
&\iff \bar{f}'(\{z_0\}, w) \in Z'' && \text{(nach Definition von } Z'') \\
&\iff w \in L(A')
\end{aligned}$$

Also gilt $L = L(A) = L(A')$. Mithin ist $L \in \mathbf{EA}$.

Damit ist das Theorem bewiesen. ■

Ohne Beweis geben wir die Abschlusseigenschaften für die Klasse **EA** an.

Theorem 5.6 **EA** *ist abgeschlossen unter Vereinigung, Durchschnitt, Komplement, Konkatenation und Iteration, d.h. mit* $L, L' \in$ **EA** *sind auch* $L \cup L', L \cap L', \overline{L}, L \cdot L', L^* \in$ **EA**.

5.3 Reguläre Mengen

Definition 5.7

1. *Die* regulären Mengen über dem Alphabet Σ *sind wie folgt induktiv definiert:*
   - Induktionsanfang: $\emptyset$ *und* $\{a\}$ *für alle* $a \in \Sigma$ *sind reguläre Mengen über* Σ.
   - Induktionsschritt: *Sind* L *und* L' *reguläre Mengen über* Σ*, so sind auch* $L \cup L'$, $L \cdot L'$ *und* L^* *reguläre Mengen über* Σ.
2. *Eine Wortmenge heißt* regulär*, falls sie eine reguläre Menge über einem geeigneten Alphabet* Σ *ist.*

Mit **REG** bezeichnen wir die Klasse der regulären Mengen.

Beispiele:

- Die Menge L_7 aus obigem Beispiel ist regulär, denn L_7 lässt sich schreiben als

$$L_7 = (\{a\} \cup \{b\})^* \cdot \{a\} \cdot \underbrace{(\{a\} \cup \{b\}) \cdot \ldots \cdot (\{a\} \cup \{b\})}_{\text{7-mal}} \cdot \{a\} \cdot (\{a\} \cup \{b\})^*.$$

  Eine kompaktere Notation für reguläre Mengen bieten *reguläre Ausdrücke*: Die Einermenge $\{a\}$ wird durch a notiert, die Vereinigung $\cup$ wird mit $+$ notiert, die Konkatenation üblicherweise ganz weggelassen. Daneben wird $\emptyset$ als Symbol für die leere Menge $\emptyset$ benutzet, und ε wird für die Einermenge $\{\varepsilon\} = \emptyset^*$ verwendet. L_7 lässt sich somit durch den folgenden Ausdruck beschreiben:

$$L_7 = (a+b)^* a (a+b)^7 a (a+b)^*$$

  Zu beachten ist, dass die Iteration * stärker bindet als die Konkatenation $\cdot$ und die Konkatenation $\cdot$ stärker bindet als $\cup$.

- Der reguläre Ausdruck $((a+b)(a+b))^*$ beschreibt die (reguläre) Menge aller Wörter gerader Länge über dem Alphabet $\{a, b\}$.

- Der reguläre Ausdruck $a^*(baa^*)^* + a^*(baa^*)b$ beschreibt die (reguläre) Menge aller Wörter über dem Alphabet $\{a, b\}$, bei denen b nicht hintereinander vorkommt.

Im Folgenden wollen wir zeigen, dass die regulären Mengen genau die von endlichen Automaten akzeptierten Mengen sind. Dafür benötigen wir folgendes Lemma.

Lemma 5.8 *Es seien $B, C \subseteq \Sigma^*$. Gilt $\varepsilon \notin B$, so ist die Menge $L = B^* \cdot C$ die einzige Lösung für die Gleichung $L = B \cdot L \cup C$.*

Beweis: Es sei L die Menge $L = B \cdot L \cup C$. Für die Gleichheit $L = B^* \cdot C$ müssen zwei Inklusionen gezeigt werden.

($\supseteq$) : Wir zeigen zunächst mittels Induktion über k, dass $B^k \cdot C \subseteq L$ für alle $k \in \mathbb{N}$ gilt:

- *Induktionsanfang:* Es sei $k = 0$. Dann gilt

$$B^0 \cdot C = \{\varepsilon\} \cdot C = C \subseteq B \cdot L \cup C = L.$$

- *Induktionsschritt:* Für $k > 0$ gilt

$$B^k \cdot C = B \cdot (B^{k-1} \cdot C) \subseteq B \cdot L \subseteq B \cdot L \cup C = L,$$

  wobei wir für die erste Abschätzung die Induktionsvoraussetzung angewendet haben.

Insgesamt gilt somit

$$B^* \cdot C = \left(\bigcup_{k=0}^{\infty} B^k \right) \cdot C = \bigcup_{k=0}^{\infty} B^k \cdot C \subseteq \bigcup_{k=0}^{\infty} L = L.$$

($\subseteq$): Wir zeigen mittels Induktion über die Länge der Wörter w, dass $w \in B^* \cdot C$ für $w \in L$ gilt.

- *Induktionsanfang:* Es sei $|w| = 0$, d.h. $w = \varepsilon$. Gilt $\varepsilon \in L$, so ist $\varepsilon \in C$, da $\varepsilon \notin B$. Mithin gilt $\varepsilon = \varepsilon \cdot \varepsilon \in B^* \cdot C$.
- *Induktionsschritt:* Es sei $|w| > 0$ mit $w \in L = B \cdot L \cup C$. Dann gibt es zwei Fälle:
  (a) Ist $w \in C$, so gilt $w = \varepsilon \cdot w \in B^* \cdot C$.
  (b) Ist $w \in B \cdot L$, dann gibt es $u \in B$ und $v \in L$ mit $w = u \cdot v$. Wegen $\varepsilon \notin B$ gilt dann $|u| > 0$ und $|v| < |w|$. Nach Induktionsvoraussetzung gilt $v \in B^* \cdot L$. Mithin gilt $w = u \cdot v \in B \cdot B^* \cdot L = B^+ \cdot L \subseteq B^* \cdot L$.

Damit ist das Lemma bewiesen. ■

Theorem 5.9 EA = REG.

Beweis: Wir zeigen beide Inklusionen einzeln.

($\supseteq$): Wir führen einen Induktionsbeweis entlang des Aufbau regulärer Mengen:

- *Induktionsanfang:* Für die einfachsten regulären Mengen $\emptyset$ und $\{a\}$ für $a \in \Sigma$ geben wir direkt die akzeptierenden endlichen Automaten an:
  Abbildung einfügen ...
  Somit gilt $\emptyset \in \mathbf{EA}$ und $\{a\} \in \mathbf{EA}$ für alle $a \in \Sigma$.
- *Induktionsschritt:* Mit $L, L' \in \mathbf{EA}$ sind auch $L \cup L', L \cdot L', L^* \in \mathbf{EA}$ (nach Theorem 5.6).

($\subseteq$): Es sei $L \in \mathbf{EA}$ mittels des endlichen Automaten $A = (\Sigma, Z, f, z_0, Z')$ mit $\Sigma = \{a_1, \ldots, a_k\}$. Wir definieren für jedes $z \in Z$ die Menge

$$L_z =_{\text{def}} \{\ w \mid \bar{f}(z, w) \in Z'\ \}$$

Es kann nun gezeigt werden, dass L_z für alle $z \in Z$ regulär ist (also insbesondere auch L_{z_0}). Denn es gilt

$$L_z = \bigcup_{i=1}^{k} \{a_i\} \cdot L_{f(z,a_i)} \qquad \text{für } z \notin Z'$$

$$L_z = \bigcup_{i=1}^{k} \{a_i\} \cdot L_{f(z,a_i)} \cup \{\varepsilon\} \qquad \text{für } z \in Z'$$

Wir betrachten L_z als Mengenvariable und lösen das Gleichungssystem mit Hilfe von Lemma 5.8, d.h., wir eliminieren der Reihe nach alle L_z bis L_{z_0} übrig bleibt. Mithin gilt $L = L(A) = L_{z_0} \in \mathbf{REG}$.

Damit ist das Theorem bewiesen. ■

Beispiel: Wir wollen die Konstruktion und die Lösung des Gleichungssystems auf obigem Beweis an einem Beispielautomaten nachvollziehen. Dazu betrachten wir folgenden endlichen Automaten über dem Alphabet $\{a, b\}$

Abbildung einfügen ...

Als Gleichungssystem für die Mengen L_{z_0}, L_{z_1} und L_{z_2} ergibt sich:

$$\begin{aligned} L_{z_0} &= \{a\} \cdot L_{z_0} \cup \{b\} \cdot L_{z_2} \\ L_{z_1} &= \{a\} \cdot L_{z_0} \cup \{b\} \cdot L_{z_1} \\ L_{z_2} &= \{a\} \cdot L_{z_1} \cup \{b\} \cdot L_{z_0} \cup \{\varepsilon\} \end{aligned}$$

In der letzten Gleichung kommt L_{z_2} nicht auf der rechten Seite der Gleichung vor. Damit können wir alle Vorkommen von L_{z_2} in den anderen Gleichungen durch die rechte Seite ersetzen und erhalten:

$$\begin{aligned} L_{z_0} &= \{a\} \cdot L_{z_0} \cup \{b\} \cdot (\{a\} \cdot L_{z_1} \cup \{b\} \cdot L_{z_0} \cup \{\varepsilon\}) \\ &= (\{a\} \cup \{b\} \cdot \{b\}) \cdot L_{z_0} \cup \{b\} \cdot \{a\} \cdot L_{z_1} \cup \{b\} \\ L_{z_1} &= \{a\} \cdot L_{z_0} \cup \{b\} \cdot L_{z_1} \end{aligned}$$

Die letzte Gleichung lässt sich mit Lemma 5.8 lösen:

$$L_{z_1} = \{b\}^* \cdot \{a\} \cdot L_{z_0}$$

Wir ersetzen L_{z_1} in der ersten Gleichung und erhalten:

$$\begin{aligned} L_{z_0} &= (\{a\} \cup \{b\} \cdot \{b\}) \cdot L_{z_0} \cup \{b\} \cdot \{a\} \cdot \{b\}^* \cdot \{a\} \cdot L_{z_0} \cup \{b\} \\ &= (\{a\} \cup \{b\} \cdot \{b\} \cup \{b\} \cdot \{a\} \cdot \{b\}^* \cdot \{a\}) \cdot L_{z_0} \cup \{b\} \end{aligned}$$

Mit Lemma 5.8 ergibt sich somit

$$L_{z_0} = (\{a\} \cup \{b\} \cdot \{b\} \cup \{b\} \cdot \{a\} \cdot \{b\}^* \cdot \{a\})^* \cdot \{b\}$$

als Lösung für L_{z_0}. Aus der Darstellung ist ersichtlich, dass die Menge regulär ist. Der äquivalente reguläre Ausdruck ist $(a + bb + bab^*a)^*b$.

5.4 Das Pumping-Lemma für reguläre Mengen

Eine Frage bleibt: Wie können wir zeigen, dass eine konkrete Menge nicht regulär ist? Eine notwendige Bedingung für die Regularität einer Sprache liefert das folgende, sogenannte Pumping-Lemma.

Theorem 5.10 (Pumping-Lemma für reguläre Sprachen, uvw-Theorem) *Für jede reguläre Menge L gibt es eine natürliche Zahl $n_0 > 0$ mit folgender Eigenschaft: Für jedes $x \in L$ mit $|x| \geq n_0$ gibt es eine Zerlegung $x = uvw$ mit $|uv| \leq n_0$, $|v| > 0$ und $uv^kw \in L$ für alle $k \geq 0$.*

Beweis: Es sei L eine reguläre Menge. Nach Satz 5.9 gibt es einen endlichen Automaten $A = (\Sigma, Z, f, z_0, Z')$ mit $L(A) = L$. Wir setzen $n_0 =_{\text{def}} 1 + \|Z\|$. Es sei nun $x = x_1x_2 \ldots x_n \in L$ mit $n \geq n_0$. Beim Lesen des Wortes x durchläuft der Automat A eine Folge von $n > \|Z\|$ Zuständen an. Nach dem Schubfachschluss gibt es somit (mindestens) einen Zustand in dieser Folge, der (mindestens) doppelt besucht wird. Mit anderen Worten gibt es $1 \leq i < j \leq n_0$ mit $\bar{f}(z_0, x_1 \ldots, x_i) = \bar{f}(z_0, x_1 \ldots x_j)$. Wir wählen ein solches Paar (i, j) und definieren die Zerlegung als:

$$u =_{\text{def}} x_1 \ldots x_i, \qquad v =_{\text{def}} x_{i+1} \ldots x_j, \qquad w =_{\text{def}} x_{j+1} \ldots x_n$$

Dann gilt $x = uvw$, $|uv| = j \leq n_0$ sowie $|v| = j - i > 0$. Weiterhin gilt

$$\bar{f}(z_0, uv^k w) = \bar{f}(z_0, uv) \quad \text{für alle } k \in \mathbb{N}$$

Dies ist wie folgt induktiv einzusehen:

- *Induktionsanfang:* Für $k = 0$ gilt

$$\bar{f}(z_0, u) = \bar{f}(z_0, x_1 \dots x_i) = \bar{f}(z_0, x_1 \dots x_j) = \bar{f}(z_0, uv).$$

- *Induktionsschritt:* Für $k > 0$ erhalten wir

$$\begin{aligned} \bar{f}(z_0, uv^k) &= \bar{f}(\bar{f}(z_0, uv^{k-1}), v) \\ &= \bar{f}(\bar{f}(z_0, uv), v) && \text{(nach Induktionsvoraussetzung)} \\ &= \bar{f}(z_0, uv^2) \\ &= \bar{f}(z_0, uv) && \text{(nach Induktionsvoraussetzung)} \end{aligned}$$

Somit ergibt sich für alle $k \in \mathbb{N}$:

$$\bar{f}(z_0, uv^k w) = \bar{f}(\bar{f}(z_0, uv^k), w) = \bar{f}(\bar{f}(z_0, uv)w) = \bar{f}(z_0, uvw) = \bar{f}(z_0, x) \in Z'$$

Mithin gilt $uv^k w \in L$ und das Theorem ist bewiesen. ■

Beispiel: Wir wollen mit Hilfe des Pumping-Lemma zeigen, dass die Menge PALINDROM aller Palindrome über dem Alphabet $\Sigma = \{a, b\}$ nicht regulär ist. Angenommen PALINDROM ist regulär. Dann gibt es nach Theorem 5.10 ein $n_0 > 0$, sodass die angegebene Zerlegung für jedes Palindrom existiert. Wir betrachten das Wort $x = a^{n_0} b^{2n_0} a^{n_0} \in$ PALINDROM. Dann gibt es also eine Zerlegung $a^{n_0} b^{2n_0} a^{n_0} = uvw$, sodass $|uv| \leq n_0$, $|v| > 0$ und $uv^0 w = uw = a^{n_0 - |v|} b^{2n_0} a^{n_0} \in$ PALINDROM gilt. Wegen $|v| > 0$ ist das Wort uw aber kein Palindrom. Dies ist ein Widerspruch, und somit ist PALINDROM nicht regulär.

Literaturverzeichnis

[GJ79] Michael R. Garey, David S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-completeness.* W. H. Freeman, San Francisco, CA, 1979.

[HMU06] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* 3. Auflage. Prentice Hall, Upper Saddle River, NJ, 2006.

[Sch08] Uwe Schöning. *Theoretische Informatik – kurz gefasst.* 5. Auflage. Spektrum Akademischer Verlag, Heidelberg, 2008.

[Wag03] Klaus W. Wagner. *Theoretische Informatik. Eine kompakte Einführung.* 2., überarbeitete Auflage. Springer-Verlag, Berlin, 2003.