Informatik IV

Theoretische Informatik

Ein Skriptum zur Vorlesung aus dem Sommersemester 1995
von Prof. Dr. W. Thomas an der Universität Kiel.

Ausarbeitung: M. Ackermann, W. Thomas

Inhaltsverzeichnis

Kapitel 1

Transitionssysteme, endliche Automaten

1.1 Nichtdeterministische endliche Automaten

Ausgangspunkt ist die Modellierung von Systemen durch (endlich viele) Zustände und Zustandsübergänge (Transitionen), die durch Aktionennamen beschriftet sind. Das Verhalten eines Systems ist gegeben durch die Folgen von Aktionen, die von „Anfangszuständen" in „Endzustände" führen.

Definition: Ein *Transitionssystem* hat die Form $\mathcal{A} = (Q, \Sigma, I, \Delta, F)$, wobei folgende Dinge gelten:

- Q *Zustandsmenge*, Σ disjunkt dazu *endliches Alphabet*
- $I, F \subseteq Q$ Mengen der *Anfangs-* bzw. *Endzustände*
- $\Delta \subseteq Q \times \Sigma \times Q$ *Transitionsrelation*

$\mathcal{A}$ heißt *endlich*, falls Q endlich ist.

Ein endliches Transitionssystem heißt *nichtdeterministischer endlicher Automat* (NEA), falls $I = \{q_0\}$ für ein $q_0 \in Q$. Notation: $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$.

Sei $\mathcal{A}$ ein Transitionssystem wie oben. Ein *Pfad* durch $\mathcal{A}$ ist eine Folge $\pi = p_0 a_1 p_1 a_2 p_2 \dots a_n p_n$ mit $(p_i, a_{i+1}, p_{i+1}) \in \Delta$ für $i < n$.

Die *Beschriftung* $\beta(\pi)$ sei $a_1 \dots a_n$, die Länge von π sei n. $\mathcal{A} : p \xrightarrow{w} q$ für $w \in \Sigma^*$ besage: Es existiert ein Pfad durch $\mathcal{A}$ von p nach q mit Beschriftung w.

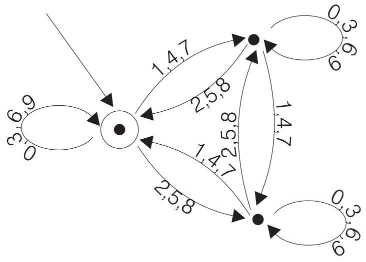
Ein Transitionssystem $\mathcal{A} = (Q, \Sigma, I, \Delta, F)$ *akzeptiert* $w \in \Sigma^*$, falls $\mathcal{A} : p \xrightarrow{w} q$ für ein $p \in I, q \in F$. Für einen NEA wird analog verlangt: $\mathcal{A} : q_0 \xrightarrow{w} q$ für ein $q \in F$. Für einen NEA $\mathcal{A}$ sei $L(\mathcal{A}) := \{w \in \Sigma^* | \mathcal{A} \text{ akzeptiert } w\}$ die durch $\mathcal{A}$ erkannte *Sprache*.
Zwei Automaten $\mathcal{A}, \mathcal{B}$ sind *äquivalent*, wenn $L(\mathcal{A}) = L(\mathcal{B})$.

Beispiel:

1. $\Sigma = \{0, \ldots, 9\}$, L =Menge der $w \in \Sigma^*$, so daß die durch w in Dezimaldarstellung bestimmte Zahl durch 3 teilbar ist. Gesucht: NEA $\mathcal{A}$ mit $L(\mathcal{A}) = L$.

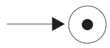
   Angabe durch Graphen: Zustände repräsentiert durch Knoten $\bullet$, Transitionen durch beschriftete Pfeile. Der Anfangszustand wird durch $\longrightarrow\bullet$ gekennzeichnet, der Endzustand durch $\longrightarrow\odot$.

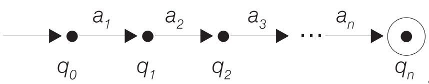
   Der Automat realisiert die Bildung der Quersumme modulo 3 mit den drei Zuständen für „Rest= 0“, „Rest= 1“, „Rest= 2“:

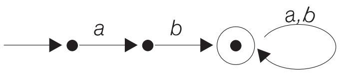
   Dieser Automat $\mathcal{A}$ ist *deterministisch*, d.h. für jeden Zustand p und jeden Buchstaben a existiert genau eine Transition der Form (p, a, q).

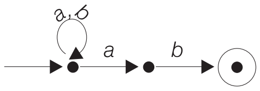
2. $L = \emptyset$, erkennbar durch NEA 

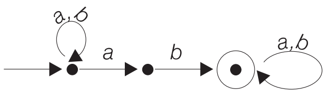
   $L = \{\varepsilon\}$, erkennbar durch NEA 

   $L = \{w\}$ mit $w = a_1 \ldots a_n$ erkennbar durch: 
.

3. Alphabet $\Sigma = \{a, b\}$

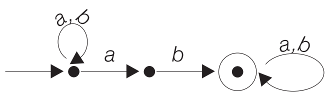
   $L_1 = \{w \in \Sigma^* | w$ hat *Präfix* $ab\}$, NEA: 

   $L_2 = \{w \in \Sigma^* | w$ hat *Suffix* $ab\}$, NEA: 

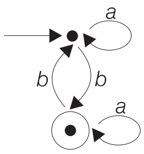
   $L_3 = \{w \in \Sigma^* | w$ hat *Infix* $ab\}$, NEA: 

4. $\{w \in \{a, b\}^* | w$ hat Infix ab und ungeradzahlig viele $b\}$

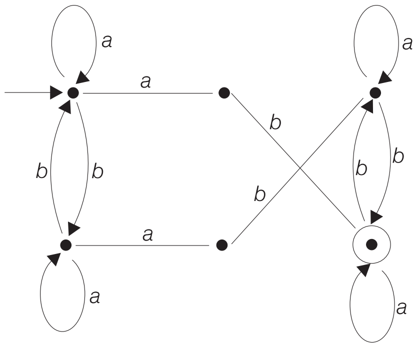
   Die 1. Eigenschaft wird erkannt durch:

2. Eigenschaft:



Kartesisches Produkt mit induzierten Transitionen gestattet simultanen Test auf beide Eigenschaften:



NEA $\mathcal{A}$ mit Test auf beide Eigenschaften: $L(\mathcal{A}) = L$.

Definition: Gegeben seien zwei Automaten

$$\mathcal{A}_1 = (Q_1, \Sigma, q_1, \Delta_1, F_1) \quad \text{und} \quad \mathcal{A}_2 = (Q_2, \Sigma, q_2, \Delta_2, F_2).$$

Der *Produktautomat* von $\mathcal{A}_1, \mathcal{A}_2$ ist definiert durch

$$\mathcal{A}_1 \times \mathcal{A}_2 = (Q_1 \times Q_2, \Sigma, (q_1, q_2), \Delta, F)$$

mit

$$((p, r), a, (p', r')) \in \Delta :\Leftrightarrow (p, a, p') \in \Delta_1, (r, a, r') \in \Delta_2,$$

sowie $F := F_1 \times F_2$.

Bemerkung: $L(\mathcal{A}_1 \times \mathcal{A}_2) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$

Beweis: Für $w = a_1 \dots a_n$ gilt

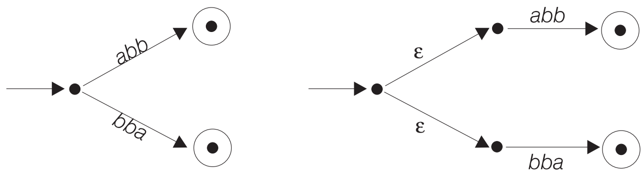
$$\begin{aligned}
a_1 \dots a_n \in L(\mathcal{A}_1 \times \mathcal{A}_2) \quad &\Leftrightarrow \quad \text{ex. Pfad } (p_0, r_0)a_1(p_1, r_1)a_2 \dots a_n(p_n, r_n), (p_n, r_n) \in F \\
& \quad (p_0, r_0) = (q_1, q_2), ((p_i, r_i), a_{i+1}, (p_{i+1}, r_{i+1})) \in \Delta \\
&\Leftrightarrow \quad \text{(nach Def. von } \Delta, F\text{) ex. Pfade } p_0 a_1 p_1 \dots a_n p_n \\
& \quad \text{mit } p_0 = q_1, (p_i, a_{i+1}, p_{i+1}) \in \Delta_1, p_n \in F_1 \\
& \quad \text{und } r_0 a_1 r_1 \dots a_n r_n \text{ mit } r_0 = q_2, (r_i, a_{i+1}, r_{i+1}) \in \Delta_2 \\
& \quad \text{und } r_n \in F_2 \\
&\Leftrightarrow \quad \mathcal{A}_1 \text{ akzeptiert } a_1 \dots a_n, \text{ und } \mathcal{A}_2 \text{ akzeptiert } a_1 \dots a_n \\
&\Leftrightarrow \quad a_1 \dots a_n \in L(\mathcal{A}_1) \cap L(\mathcal{A}_2).
\end{aligned}$$

Wir betrachten nun Verallgemeinerungen des NEA und zeigen, daß sich der Bereich der erkannten Sprachen dadurch nicht erweitert.

Definition: Ein NEA mit *Worttransitionen* hat die Form $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ analog wie NEA, aber mit $\Delta \subseteq Q \times \Sigma^* \times Q$ endlich.
Sonderfall: ε-NEA mit $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$.

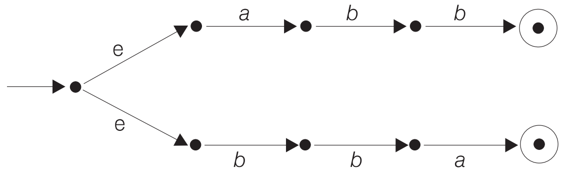
Pfade haben nun die Form $p_0 u_1 p_1 \dots p_n u_n p_{n+1}$ mit $(p_i, u_{i+1}, p_{i+1}) \in \Delta$; die entsprechende Beschriftung ist $u_1 \dots u_n$. $\mathcal{A} : p \xrightarrow{w} q$ werde analog zum Fall der NEA's definiert. $\mathcal{A}$ *akzeptiert* $w :\Leftrightarrow \mathcal{A} : p_0 \xrightarrow{w} F$ (d.h. ex. $q \in F$ mit $\mathcal{A} : p_0 \xrightarrow{w} q$).

Beispiel: $L = \{abb, bba\}$. Mögliche NEA's mit Worttransitionen, die L erkennen:



Lemma: *Zu jedem NEA mit Worttransitionen kann man einen äquivalenten ε-NEA konstruieren.*

Beweis: Ersetze in gegebenem ε-NEA $\mathcal{A}$ jede Transition $(p, b_1 \dots b_n, q)$, $n \geq 2$, durch Transitionen $(p, b_1, p_1), \dots, (p_{n-1}, b_n, q)$ mit den jeweils neuen Hilfszuständen $p_1, \dots, p_{n-1}$. Illustration zum letzten Beispiel:

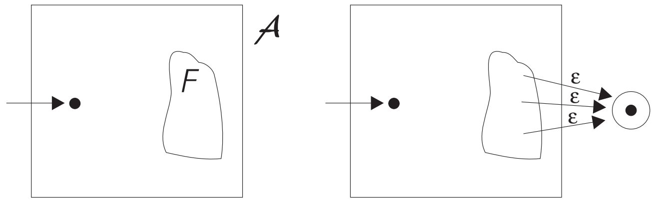


$\square$

Bevor wir die Reduktion von ε-NEA's auf NEA's durchführen, einige Bemerkungen zur *Motivation* und *Anwendungen* von ε-NEA's:

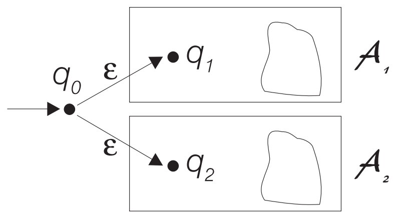
1. Strukturierung von endlichen Automaten:

   (a) Reduktion auf einen Endzustand:

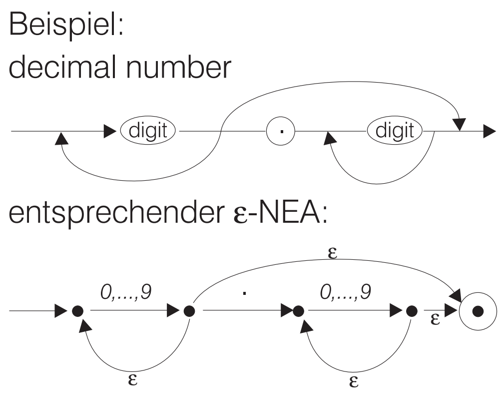


(b) Zusammensetzen von Automaten besser möglich.

Sind $\mathcal{A}_1 = (Q_1, \Sigma, q_1, \Delta_1, F_1)$, $\mathcal{A}_2 = (Q_2, \Sigma, q_2, \Delta_2, F_2)$ gegeben. Dann akzeptiert folgender ε-NEA die Sprache $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$ mit der Endzustandsmenge $F_1 \cup F_2$:



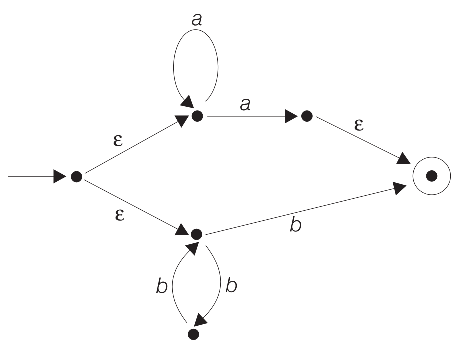
2. Syntaxdiagramme sind direkt als ε-NEA lesbar:



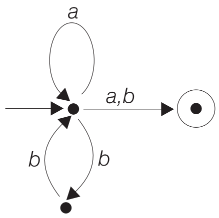
3. Modellierung paralleler Prozesse: Hier wird nach „sichtbaren" (für die Kommunikation wesentlichen) und „unsichtbaren" (prozeß-internen) Aktionen unterschieden. In Milner's CCS (Calculus of Communicating Systems) verwendet man einen festen Namen τ für die „unsichtbare Aktion", den ε-Transitionen entsprechend.

Lemma: *Zu jedem ε-NEA $\mathcal{A}$ kann man einen äquivalenten NEA $\mathcal{A}'$ konstruieren.*

Vorüberlegung am Beispiel:

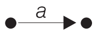


1. Idee: Zusammenziehen ε-verbundener Zustände: Es ergibt sich $\mathcal{A}'$:

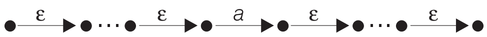


nicht äquivalent zu $\mathcal{A}$, denn $abba$ wird durch $\mathcal{A}'$, nicht jedoch durch $\mathcal{A}$ akzeptiert.

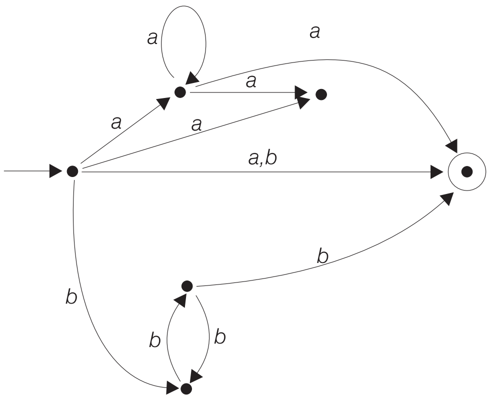
2. Idee: Eliminiere ε-Transitionen und kompensiere dies durch Einfügen von



für jeden Pfad



Falls $\mathcal{A} : q_0 \xrightarrow{\varepsilon} F$, muß in $\mathcal{A}$ q_0 auch als Endzustand deklariert sein. Ergebnis im Beispiel:



Beweis: Sei ε-NEA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ gegeben. Definiere $\mathcal{A}' = (Q, \Sigma, q_0, \Delta', F')$ mit $(p, a, q) \in \Delta' :\Leftrightarrow \mathcal{A} : p \xrightarrow{a} q$ (über ε-Pfad!).

$$F' := \begin{cases} F & \text{falls nicht } \mathcal{A} : q_0 \xrightarrow{\varepsilon} F \\ F \cup \{q_0\} & \text{falls } \mathcal{A} : q_0 \xrightarrow{\varepsilon} F \end{cases}$$

Zeige: $\mathcal{A} : q_0 \xrightarrow{w} F \Leftrightarrow \mathcal{A}' : q_0 \xrightarrow{w} F$.

„$\Rightarrow$“: 1. Fall: $w \neq \varepsilon$, etwa $w = b_1 \dots b_n$ $(n > 0)$. Wähle $p_0, \dots, p_n \in Q$ mit $p_0 = q_0$, $\mathcal{A} : p_i \xrightarrow{b_{i+1}} p_{i+1}$, $p_n \in F$. Dann ist $p_0 b_1 p_1 \dots b_n p_n$ Pfad durch $\mathcal{A}', \mathcal{A}'$ akzeptiert w.
2. Fall: $w = \varepsilon$. Der zweite Fall in Definition von F' trifft zu, also $\mathcal{A}'$ akzeptiert ε.

„$\Leftarrow$“: Gelte $\mathcal{A}' : q_0 \xrightarrow{w} F'$, $w = b_1 \dots b_n$ $(n \geq 0)$. Wähle Pfad $p_0 b_1 p_1 \dots b_n p_n, p_n \in F'$ durch $\mathcal{A}'$. Nach Definition von Δ' gilt: $\mathcal{A} : p_i \xrightarrow{b_{i+1}} p_{i+1}$. Falls $p_n \in F$, akzeptiert $\mathcal{A}$ das Wort w, und wir sind fertig. Falls $p_n \notin F$, dann gilt nach Definition von $F' : p_n = q_0$

und zweiter Fall von Definition F' trifft zu. Also $\mathcal{A} : q_0 \xrightarrow{w} p_9 = q_0 \xrightarrow{\varepsilon} F$, d.h. $\mathcal{A}$ akzeptiert w. $\square$

Es fehlt noch ein Verfahren zur Entscheidung, ob $\mathcal{A} : p \xrightarrow{a} q$ (bei Definition von Δ'). Hierzu genügt ein Verfahren zur Entscheidung, ob $\mathcal{A} : p' \xrightarrow{\varepsilon} q'$ für $p', q' \in Q$ durch einen ε-Pfad.

Bestimmung der ε-Pfad-verbundenen Zustandspaare in einem ε-NEA:

Gegeben: ε-NEA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ mit $Q = q_1, \ldots, q_n$
Sei (ε_{ij}) die $(n \times n)$-Matrix mit

$$\varepsilon_{ij} = \begin{cases} 1 & \text{falls } (q_i, \varepsilon, q_j) \in \Delta \\ 0 & \text{sonst} \end{cases}$$

Gesucht: $(n \times n)$-Matrix (c_{ij}) mit

$$c_{ij} = \begin{cases} 1 & \text{falls } \mathcal{A} : q_i \xrightarrow{\varepsilon} q_j \\ 0 & \text{sonst} \end{cases}$$

Modifikation:

$$c_{ij}^{(r)} = \begin{cases} 1 & \text{falls ex. Pfad von } q_i \text{ nach } q_j \text{ der Länge } \leq r \\ 0 & \text{sonst} \end{cases}$$

Vereinfachte Schreibweise für Elemente $c, d \in \{0, 1\}$:

- $c \oplus d$ statt $\max(c, d)$ („oder“)
- $c \otimes d$ statt $\min(c, d)$ („und“)
- $(c_{ij}) \oplus (d_{ij})$ für $(c_{ij} \oplus d_{ij})$
- $(c_{ij}) \otimes (d_{ij})$ für (e_{ij}) mit $e_{ij} = (c_{i1} \otimes d_{1j}) \oplus \ldots \oplus (c_{in} \otimes d_{nj}) =: (\sum_{k=1}^{n} c_{ik} \otimes d_{kj})$

Behauptung: $c_{ij} = c_{ij}^{n-1}$ für $n = |Q|$, d.h. es existiert ein ε-Pfad von q_i nach q_j $\Leftrightarrow$ es existiert ein ε-Pfad von q_i nach q_j der Länge $\leq n-1$.

Beweis: Dazu genügt „$\Rightarrow$“: Sei π ein ε-Pfad von q_i nach q_j minimaler Länge. Zeige: Länge von π ist $\leq n-1$. Wäre $\pi = (q_i =) p_0 \varepsilon p_1 \varepsilon \ldots \varepsilon p_m (= q_i)$ mit $m \geq n$, so existierte eine Zustandswiederholung, etwa $p_k = p_{k'}$ mit $k < k'$.
Dann ist $(q_i =) p_0 \varepsilon p_1 \varepsilon \ldots p_k \varepsilon p_{k'+1} \varepsilon \ldots \varepsilon p_m (= q_i)$ kürzerer ε-Pfad von q_i nach q_j als π, im Widerspruch zur minimalen Wahl von π. $\square$

Behauptung:

(a) $(c_{ij}^{(0)})$ =Einheitsmatrix $E_n (= (e_{ij}))$

(b) $(c_{ij}^{(r+1)}) = E_n \oplus ((c_{ij}^{(r)}) \otimes (\varepsilon_{ij}))$.

Beweis:

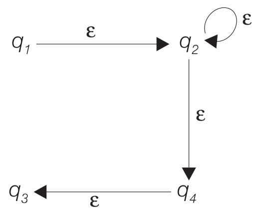
(a) Ein ε-Pfad von q_i nach q_j der Länge 0 existiert genau für $i = j$.

(b) $c_{ij}^{(r+1)} = 1 \Leftrightarrow i = j$, oder es existiert q_k und ε-Pfad von q_i nach q_k der Länge $\leq r$ und ε-Kante von q_k nach q_j $\Leftrightarrow i = j$ oder es existiert ein k mit $c_{ik}^{(r)} = 1$ und $\varepsilon_{kj} = 1 \Leftrightarrow e_{ij} \oplus \sum_{k=1}^{n} c_{ik}^{(r)} \otimes \varepsilon_{kj} = 1$.

$\square$

Behauptung 1 und 2 liefern ein Verfahren zur Berechnung von (c_{ij}): Falls $n = |Q|$, muß man nur $c_{ij}^{(n-1)}$ bestimmen.

Beispiel:



$$(\varepsilon_{ij}) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad (c_{ij}^{(1)}) = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

$$(c_{ij}) = (c_{ij}^{(3)}) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

Zeitaufwand für das Berechnungsverfahren (grobe Abschätzung nach oben):

- für den Übergang von $(c_{ij}^{(r)})$ zu $(c_{ij}^{(r+1)})$ pro Matrizeneintrag: n Multiplikationen, $n-1$ Additionen $+1$ Addition.
- für n^2 Einträge insgesamt $2n^3$ boolesche Operationen.

Insgesamt $(n-1) \cdot 2n^3$ boolesche Operationen, also Zeitaufwand $O(n^4)$. Dies ist mit ein wenig mehr Denkarbeit verbesserbar auf $O(n^3)$. Die Idee hierbei ist, nicht nach wachsender Pfadlänge vorzugehen, sondern die Verbindbarkeit von Zuständen mit Zwischenzuständen in den Mengen $\{q_1\}, \{q_1, q_2\}, \ldots, \{q_1, \ldots, q_n\}$ sukzessiv zu betrachten (Algorithmus zur Bestimmung der transitiven Hülle eines Graphen).

Auch die Bestimmung der durch NEA's erkannten Sprachen kann man in einem Matrizenkalkül durchführen. Dann stellt man einen NEA durch eine Matrix mit Einträgen $d_{ij} = \{w | (q_i, w, q_j) \in \Delta\}$ dar.

Andere Matrizen treten bei der Darstellung stochastischer Systeme auf, mit Zustandsraum $Q = \{q_1, \ldots, q_n\}$ und Transitionswahrscheinlichkeiten $p_{1n}, \ldots, p_{in}$ jeweils für den Übergang von q_i nach $q_1, \ldots, q_n$. Eine so bestimmte Matrix $P = (p_{ij})$ definiert (endlich-dimensionale) *Markoff-Kette*. Man nimmt hierbei $\sum_{j=1}^{n} p_{ij} = 1$ an. Ist $(a_1, \ldots, a_n)$ Vektor der Wahrscheinlichkeiten für $q_1, \ldots, q_n$ am Anfang, so ist $(a_1, \ldots, a_n) \cdot P^m$ der Vektor der Wahrscheinlichkeiten für $q_1, \ldots, q_n$ nach m Schritten.

Übungen

Aufgabe (1.1): (Produktautomat) Geben Sie einen NEA für die Sprache

$$\{w \in \{a,b\}^* \mid |w|_a \text{ ungerade und } |w|_b \text{ gerade}\}$$

an.

Aufgabe (1.2): (Endzustände)

1. Zeigen Sie: Jeder NEA ist äquivalent zu einem NEA mit höchstens zwei Endzuständen. (Warum reicht nicht einer?)

2. Sei $L \subseteq \Sigma^*$ durch einen NEA erkennbar. Zeigen Sie, daß auch die Sprache $L \cap \{w \in \Sigma^* \mid |w| \text{ gerade}\}$ durch einen NEA erkennbar ist.

Aufgabe (1.3): (Produktautomat) Zeigen Sie mit Hilfe der Produktautomatenkonstruktion, daß für je zwei Sprachen, die von NEA's erkannt werden, auch ihre Vereinigung von einem NEA erkannt wird.

Aufgabe (1.4): (Präfixabschluß) Ist u ein Präfix von v, so schreiben wir $u \sqsubseteq v$. Zu jeder Sprache L ist der Präfixabschluß $Prf(L)$ durch

$$Prf(L) := \{u \mid \exists v (v \in L \wedge u \sqsubseteq v)\}$$

definiert. Der Wiederholungsabschluß $Wdh(L)$ enthält alle Wörter, die aus Wörtern aus L durch Vermehrung vorkommender Buchstaben entstehen. D.h., ein Wort u gehört zu $Wdh(L)$ genau dann, wenn es ein Wort $b_1 \ldots b_m$ in L und positive natürliche Zahlen n_i gibt, so daß $u = b_1^{n_1} \ldots b_m^{n_m}$ gilt. (Die b_n Buchstaben des Alphabets.) Zeigen Sie: Wird L durch einen NEA erkannt, dann auch $Prf(L)$ und $Wdh(L)$.

Aufgabe (1.5): (NEA-Beispiele) Im folgenden sind drei natürlichsprachige Bedingungen angegeben, die jeweils die Zugehörigkeit eines Wortes zu einer formalen Sprache über dem Alphabet $\{a, b\}$ bestimmen. Konstruieren Sie nichtdeterministische endliche Automaten, die diese Sprache erkennen.

(a) Das Wort endet auf *baa*.

(b) Weder *aa* noch *bb* ist Infix des Wortes.

(c) Wenn an drei verschiedenen Stellen ein a auftritt, dann ist *aa* ein Infix des Wortes.

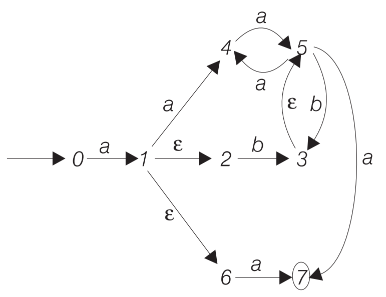
Zusatz: Versuchen Sie, für (a) einen deterministischen endlichen Automaten (DEA) zu konstruieren.

Aufgabe (1.6): (Mischprodukt oder Shuffle-Produkt) Das Mischprodukt (Shuffle Produkt) zweier Sprachen $L_1, L_2 \subseteq \Sigma^*$ ist definiert durch

$$\{w \in \Sigma^* | w = u_1 v_1 \dots u_n v_n, u_1 u_2 \dots u_n \in L_1, v_1 v_2 \dots v_n \in L_2, u_i, v_i \in \Sigma^* (1 \leq i \leq n)\}.$$

Konstruieren Sie zu gegebenen NEA's $\mathcal{A}_1, \mathcal{A}_2$ einen NEA $\mathcal{A}$, der das Mischprodukt der Sprachen $L(\mathcal{A}_1)$ und $L(\mathcal{A}_2)$ erkennt.

Aufgabe (1.7): (ε-Transitionen) Eliminieren Sie in dem folgenden ε-NEA die ε-Transitionen.



1.2 Deterministische endliche Automaten

Definition: Ein NEA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ heißt *deterministischer endlicher Automat* (DEA), falls zu jedem Paar $(q, a) \in Q \times \Sigma$ genau ein q' mit $(q, a, q') \in \Delta$ existiert. Dann ist Δ beschreibbar durch eine Funktion

$$\delta : Q \times \Sigma \to Q \text{ mit } \delta(p, a) = q \Leftrightarrow (p, a, q) \in \Delta.$$

Notation: $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$.

Definition: Die Fortsetzung von $\delta : Q \times \Sigma \to Q$ zu $\delta^* : Q \times \Sigma^* \to Q$ wird definiert durch $\delta^*(q, \varepsilon) = q$, $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$ für $w \in \Sigma^*, a \in \Sigma$.

Bemerkung: $\delta^*(p, w) = q \Leftrightarrow \mathcal{A} : p \xrightarrow{w} q$.

Beweis: durch Induktion über den Aufbau der Wörter über Σ.
$w = \varepsilon$: $\delta^*(p, \varepsilon) = q \Leftrightarrow p = q \Leftrightarrow \mathcal{A} : p \xrightarrow{\varepsilon} q$.
$w = va$ (Induktionsschritt):

$$\begin{array}{rcl} \delta^*(p, va) = q & \Leftrightarrow & \delta(\delta^*(p, v), a) = q \\ & \Leftrightarrow & \exists p' (\delta^*(p, v) = p' \wedge \delta(p', a) = q) \\ \text{(I.V.)} & \Leftrightarrow & \exists p' (\mathcal{A} : p \xrightarrow{v} p' \wedge \mathcal{A} : p' \xrightarrow{a} q) \\ & \Leftrightarrow & \mathcal{A} : p \xrightarrow{va} q. \end{array}$$

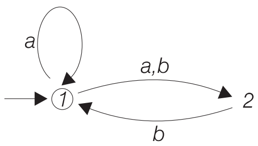
Damit läßt sich die durch einen DEA $\mathcal{A}$ erkannte Sprache so darstellen:

$$L(\mathcal{A}) = \{w \in \Sigma^* | \delta^*(q_0, w) \in F\}.$$

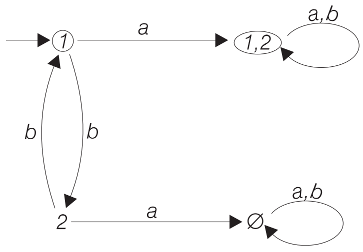
Wir schreiben in Zukunft einfach $\delta(q, w)$ statt $\delta^*(q, w)$.

Satz: (Myhill, Rabin / Scott, Potenzmengenkonstruktion) *Zu jedem NEA kann man einen äquivalenten DEA konstruieren.*

Idee anhand des folgenden Beispiels:



Bestimme längs aller Σ -Wörter die Mengen der so erreichbaren Zustände:



Beweis: Definiere zu jedem NEA $\mathcal{A} : (Q, \Sigma, q_0, \Delta, F)$ den DEA $\mathcal{A}' : (Q', \Sigma, q_0', \delta', F')$ so:

$$\begin{array}{rcl} Q' & := & \mathcal{P}(Q) \quad \text{(Potenzmenge von } Q) \\ q_0' & := & \{q_0\} \\ \delta'(P, a) & := & \{q \in Q | \exists p \in P : (p, a, q) \in \Delta\} \\ F' & := & \{P \subseteq Q | P \cap F \neq \emptyset\}. \end{array}$$

Behauptung: $q \in \delta'(\{p\}, w) \Leftrightarrow \mathcal{A} : p \xrightarrow{w} q$.
Beweis induktiv über$|w|$:
$|w| = 0$: $q \in \delta'(\{p\}, \varepsilon) \Leftrightarrow p = q \Leftrightarrow \mathcal{A} : p \xrightarrow{\varepsilon} q$.
$w = va$:

$$\begin{array}{rcl} q \in \delta'(\{p\}, va) & \Leftrightarrow & q \in \delta'(\delta'(\{p\}, v), a) \\ \text{(Def. von } \delta') & \Leftrightarrow & \exists r \in \delta'(\{p\}, v)_p : (r, a, q) \in \Delta \\ \text{(I.V.)} & \Leftrightarrow & \exists r : \mathcal{A} : p \xrightarrow{v} r, (r, a, q) \in \Delta \\ & \Leftrightarrow & \mathcal{A} : p \xrightarrow{va} q. \end{array}$$

Daraus folgt Korrektheit der Potenzmengenkonstruktion:

$$\begin{array}{rcl} \mathcal{A} \text{ akzeptiert } w & \Leftrightarrow & \exists q \in F : \mathcal{A} : q_0 \xrightarrow{w} q \\ & \Leftrightarrow & \exists q \in F : q \in \delta'(q_0, w) \\ & \Leftrightarrow & F \cap \delta'(q_0, w) \neq \emptyset \\ (\text{Def. von } F') & \Leftrightarrow & \delta'(q_0, w) \in F' \\ & \Leftrightarrow & \mathcal{A}' \text{ akzeptiert } w. \end{array}$$

□

Bemerkung: *Ist $L \subseteq \Sigma^*$ DEA-erkennbar, so auch $\Sigma^* \setminus L$.*

Beweis: $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ erkenne L. Dann: $w \in \Sigma^* \setminus L \Leftrightarrow \delta(q_0, w) \in Q \setminus F$. Also: $\mathcal{A}' = (Q, \Sigma, q_0, \delta, Q \setminus F)$ erkennt $\Sigma^* \setminus L$. □

Folgerung: *Die durch NEA's (bzw. DEA's) erkennbaren Sprachen über Σ bilden eine boolesche Algebra.*

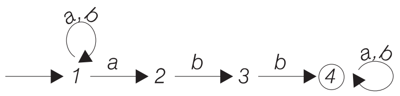
Von nun an verwenden wir auch die Redeweise: „L regulär“ statt „L von NEA (bzw. DEA) erkennbar“.

Die Potenzmengenkonstruktion enthält einen Größensprung bei den Automaten (von n auf 2^n, vgl. Aufgabe 1.16). Daher stellen wir eine Methode zur *Minimierung von DEA's* vor.

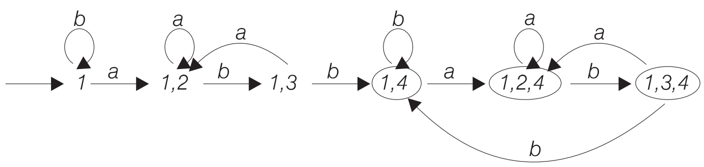
Wir führen zwei Schritte durch:

1. Bei der Potenzmengenkonstruktion Einschränkung auf erreichbare Zustände.
2. Zusammenfassen “äquivalenter Zustände“.

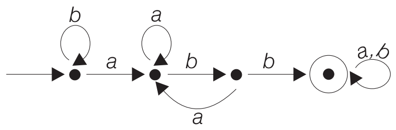
Beispiel: gegeben $\mathcal{A}$:



$\mathcal{P}$-Automat:



Wegen Beschränkung auf erreichbare Zustände ergeben sich 6 statt 16 Zustände. Da man von einem der drei Endzustände nur zu einem weiteren Endzustand kommt, lassen sich diese zusammenfassen. Ergebnis gemäß 2:



also ergeben sich nur 4 Zustände.

Allgemeine Fassung von Schritt 2 des Minimierungsverfahrens:

Definition: Für einen DEA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ gelte

$$p \sim_{\mathcal{A}} q :\Leftrightarrow \forall w \in \Sigma^* : (\delta(p, w) \in F \Leftrightarrow \delta(q, w) \in F).$$

Bemerkung:

1. *$\sim_{\mathcal{A}}$ ist Äquivalenzrelation auf Q, $q^{\sim}$ ist Äquivalenzklasse von q.*
2. *$a \in \Sigma, p \sim_{\mathcal{A}} q \Rightarrow \delta(p, a) \sim_{\mathcal{A}} \delta(q, a)$.*

Für 2 genügt die Kontraposition:

$$\neg\delta(p, a) \sim_{\mathcal{A}} \delta(q, a) \Rightarrow \neg p \sim_{\mathcal{A}} q.$$

Nach Voraussetzung wähle w mit

$$\delta(\delta(p, a), w) \in F, \qquad \delta(\delta(q, a), w) \notin F.$$

Dann

$$\delta(p, aw) \in F, \qquad \delta(q, aw) \notin F,$$

also $\neg p \sim_{\mathcal{A}} q$.

Definition: Nenne p, q *trennbar* über Länge l, falls für ein w mit $|w| = l$ die Äquivalenz $\delta(p, w) \in F \Leftrightarrow \delta(q, w) \in F$ nicht zutrifft.

Wir halten nach dem Beweis von 2 fest:
$(*)$ Sind $\delta(p, a), \delta(q, a)$ trennbar über Länge l, so p, q über Länge $l + 1$.

Definition: Der *reduzierte* DEA $\tilde{\mathcal{A}}$ zu $\mathcal{A}$ sei $\tilde{\mathcal{A}} = (\tilde{Q}, \Sigma, \tilde{q_0}, \tilde{\delta}, \tilde{F})$ mit $\tilde{Q} = \{\tilde{q} | q \in Q\}$, $\tilde{\delta}(\tilde{q}, a) = \widetilde{\delta(q, a)}$ (wohldefiniert nach Bemerkung 2), $\tilde{F} = \{\tilde{q} | q \in F\}$.

Es gilt:

(a) $\tilde{\delta}(\tilde{q}, w) = \widetilde{\delta(q, w)}$ (mit Induktion über $|w|$).

(b) $\tilde{\mathcal{A}}$ äquivalent zu $\mathcal{A}$, denn:

$$\begin{aligned} \mathcal{A} \text{ akzeptiert } w &\Leftrightarrow \delta(q_0, w) \in F \\ &\Leftrightarrow \widetilde{\delta(q_0, w)} \in \tilde{F} \\ &\Leftrightarrow \tilde{\delta}(\tilde{q_0}, w) \in \tilde{F} \\ &\Leftrightarrow \tilde{\mathcal{A}} \text{ akzeptiert } w. \end{aligned}$$

Um den Übergang $\mathcal{A} \to \tilde{\mathcal{A}}$ algorithmisch durchführen zu können, brauchen wir ein Verfahren zum Test, ob $p \sim_{\mathcal{A}} q$.
Idee: Bestimme für $l = 0, 1, 2, \ldots$ die (p, q), die über Länge l trennbar sind.

Ausführung: *Markierungsalgorithmus*

(0) Markiere alle Paare (p, q), die über Länge 0 trennbar sind
(d.h. $\neg(p \in F \Leftrightarrow q \in F)$).

(1) Solange im letzten Schritt ein zusätzliches Paar markiert wurde: Markiere alle noch nicht markierten Paare (p, q), für die mit geeignetem $a \in \Sigma$ das Paar $(\delta(p, a), \delta(q, a))$ schon markiert ist.

Korrektheitsbehauptung: Der Markierungsalgorithmus terminiert, und genau dann sind die trennbaren Paare markiert.

Beispiel:

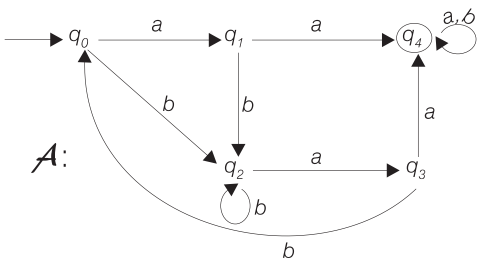
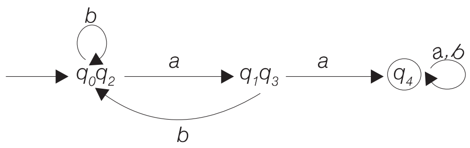


Tabelle der Paare:

q_1	*			
q_2		*		
q_3	*		*	
q_4	X	X	X	X
	q_0	q_1	q_2	q_3

X in (0) markiert
* in (1) markiert

Ergebnis: $q_0 \sim_{\mathcal{A}} q_2, q_1 \sim_{\mathcal{A}} q_3$ Es ergibt sich der Automat $\tilde{\mathcal{A}}$:



Beweis: (der Korrektheitsbehauptung)

1. Es werden nur trennbare Paare markiert

2. Termination gilt

3. Bei Termination sind alle trennbaren Paare markiert

zu 1: klar nach Bemerkung 2 (siehe oben $(*)$)

zu 2: klar, da nur endlich viele Paare markierbar, und jede Durchführung von (1) ein neues Paar markiert zu 3: Mit folgenden Punkten (a) und (b) sind wir offensichtlich fertig:

(a) nach l Schritten (1) sind alle Paare markiert, die über Länge l trennbar sind (der Beweis ist einfach, mit Induktion über l).

(b) Die Termination erfolge nach m Markierungsschritten (1). Dann sind alle überhaupt trennbaren Paare schon über Länge $\leq m$ trennbar (d.h. nach $\leq m$ Schritten (1)).

Beweis:

(A) Gemäß Terminationsbedingung liegt vor dem $(m+1)$-ten Schritt (1) kein Paar vor, das über Länge $m+1$ trennbar ist und noch nicht über Länge m. (Sonst gäbe es nämlich noch ein zu markierendes Paar.)

(B) Wir schließen noch die Trennbarkeit von Paaren über Längen $l > m+1$ aus. Annahme: Es gibt (p,q), über Länge $l > m+1$ trennbar, aber nicht über kleinere Längen. In einem trennbaren Wort sei v das Suffix der letzten $m+1$ Buchstaben:

$$\mathcal{A} : \begin{cases} p \xrightarrow{u} p' \xrightarrow{v} F \\ q \xrightarrow{u} q' \xrightarrow{v} \notin F \end{cases} \qquad |uv| = l, |v| = m+1.$$

Dann ist (p', q') über Länge $m+1$ trennbar, aber nicht über kleinere Länge (sonst wäre (p,q) über Länge $< l$ trennbar). Widerspruch zu (A). $\square$

Abschließend geben wir eine *Charakterisierung der regulären Sprachen* an, mit der man auch nachweisen kann, daß gewisse Sprachen nicht DEA-erkennbar sind. Schlüssel ist folgende Definition:

Definition: Sei $L \subseteq \Sigma^*$ (beliebig!) und $u, v \in \Sigma^*$. Wir definieren

$$u \sim_L v \Leftrightarrow \forall w \in \Sigma^* (uw \in L \Leftrightarrow vw \in L).$$

$\sim_L$ ist *Äquivalenzrelation*, $[u]_L$ sei die $\sim_L$-*Klasse* von u.

Beispiel: $L = \{w \in \{a,b\}^* | w \text{ hat } abb \text{ als Infix}\}$. Wir erhalten folgende $\sim_L$-Beziehungen:
$\neg \varepsilon \sim_L a$, denn $\varepsilon bb \notin L$, aber $abb \in L$.
$e \sim_L b$, denn $\varepsilon w \in L \Leftrightarrow w$ hat Infix $abb \Leftrightarrow bw \in L$.
$a \sim_L aa$, denn $aw \in L \Leftrightarrow w$ hat Infix abb oder beginnt mit $bb \Leftrightarrow aaw \in L$.
$\neg a \sim_L ab$, denn $ab \notin L$, aber $abb \in L$.
$\neg ab \sim_L abb$, denn $ab\varepsilon \notin L$, aber $abb\varepsilon \in L$.
Nach genauerer Analyse ergeben sich insgesamt vier $\sim_L$-Klassen:

$$[\varepsilon]_L, [a]_L, [ab]_L, [abb]_L.$$

Definition: Es sei *Index* $(\sim_L)$:=Anzahl der $\sim_L$-Klassen.

Satz: (Nerode) $L \subseteq \Sigma^*$ *regulär* $\Leftrightarrow$ *Index* $(\sim_L)$ *endlich.*

Beweis: „$\Rightarrow$" Sei $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ ein DEA, der L erkennt. Zeige: Verschiedene $\sim_L$-Klassen der Σ^* bestimmen verschiedene $\sim_{\mathcal{A}}$-Klassen in Q.
Dann $|Q| \geq$ Index $(\sim_{\mathcal{A}}) \geq$ Index $(\sim_L)$, und die Behauptung ist gezeigt, da Q endlich.
Gilt $\neg u \sim_L v$, d.h. $[u]_L \neq [v]_L$, so bilde $\delta(q_0, u), \delta(q_0, v)$, zeige $\neg\delta(q_0, u) \sim_{\mathcal{A}} \delta(q_0, v)$.
Wäre $\delta(q_0, u) \sim_{\mathcal{A}} \delta(q_0, v)$, so

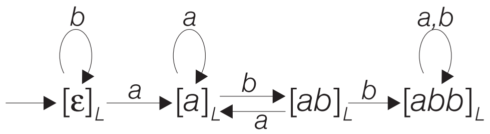
$$\forall w \in \Sigma^* : \underbrace{\delta(\delta(q_0, u), q)}_{uw \in L} \in F \Leftrightarrow \underbrace{\delta(\delta(q_0, v), q)}_{vw \in L} \in F,$$

also $uw \in L \Leftrightarrow vw \in L$, d.h. $u \sim_L v$. Widerspruch.

„$\Leftarrow$" (nur Idee) Sei Index $(\sim_L)$ endlich. Bilde DEA $\mathcal{A}_L$, der L erkennt mit $\sim_L$-Klassen als Zuständen (endlich viele).
Anfangszustand: $[\varepsilon]_L$,
Transitionsfunktion: $\delta_L([u]_L, a) := [ua]_L$,
Endzustände: $[u]_L \in F :\Leftrightarrow u \in L$.

Zusatz (ohne Beweis): Dieser „kanonische DEA" $\mathcal{A}_L$ ist isomorph zu jedem DEA $\tilde{\mathcal{A}}$, der gemäß Minimierungsverfahren aus einem beliebigen DEA $\mathcal{A}$ nur mit erreichbaren Zuständen, der L erkennt, entsteht. $\square$

Im oben genannten Beispiel der Sprache $L = \{w \in \{a, b\}^* | w$ hat abb als Infix$|\}$ erhalten wir den folgenden Automaten $\mathcal{A}_L$



Angabe nichtregulärer Sprachen: Aus dem Satz von Nerode (Richtung $\Rightarrow$) folgt: L ist nicht regulär, falls Index $(\sim_L)$ unendlich ist, d.h. falls es Wörter $u_0, u_1, \ldots$ gibt, so daß für $i \neq j$ jeweils ein w existiert mit $u_i w \in L, u_j w \notin L$.

Beispiel: $L_1 = \{a^i b^i | i \geq 0\}$. Wähle $u_i := a^i$. Für $i \neq j$ gilt mit

$$w := b^i, u_i w = a^i b^i \in L_1, u_j w = a^j b^i \notin L_1.$$

L_2 =die Menge der voll geklammerten arithmetischen Terme mit $+, *$ über Dezimalzahlen und Variablen xD, wobei D eine Dezimalzahl sei. $\Sigma = \{(,), +, *, x, 0, \ldots, 9\}$.
Wähle $u_i := (^i$. Für $i \neq j$ setze

$$w := 1 + \underbrace{1) + 1) + \ldots + 1)}_{i-\text{mal}},$$

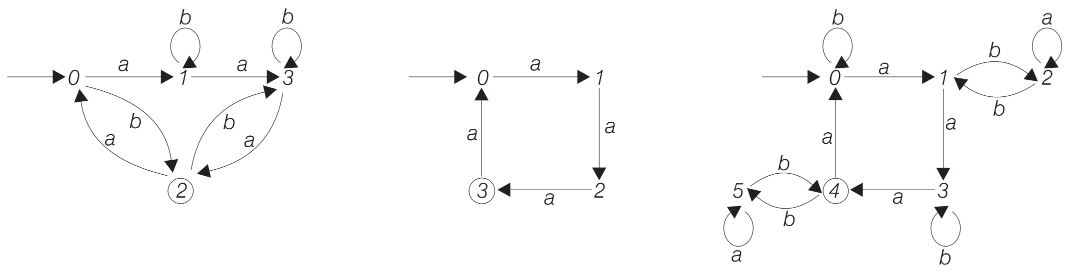
dann ist

$$u_i w = \underbrace{(\ldots(}_{i-\text{mal}} 1 + \underbrace{1) + 1) + \ldots + 1)}_{i-\text{mal}} \in L_2, u_j w \notin L_2.$$

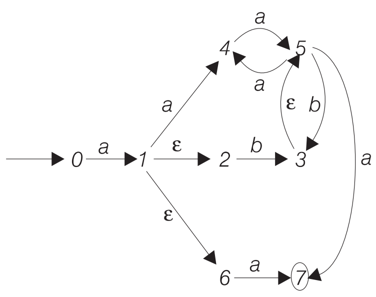
Die Sprachen L_1, L_2 sind also nicht regulär.

Übungen

Aufgabe (1.8): (Automatenreduktion) Führen Sie per Hand eine Reduktion der folgenden DEA's durch, indem Sie alle Zustandspaare auf Äquivalenz prüfen. Belegen Sie die Nichtäquivalenz jeweils durch Angabe eines geeigneten Wortes.



Aufgabe (1.9): (Potenzmengenkonstruktion, Minimierung) Führen Sie die Eliminierung der ε-Transitionen, die Potenzmengenkonstruktion und die Minimierung für den folgenden Automaten durch. Es genügt die graphische Darstellung aller Zwischenergebnisse.



Aufgabe (1.10): (Linksquotient und Rechtsquotient) Sei L eine Sprache, die von einem DEA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ erkannt wird. Geben Sie, ausgehend von $\mathcal{A}$, für fest vorgegebenes $w \in \Sigma^*$.

1. einen DEA an, der $w^{-1}L = \{v \in \Sigma^* | wv \in L\}$, d.h. den Linksquotienten von L bezüglich w erkennt.

2. einen DEA an, der $Lw^{-1} = \{v \in \Sigma^* | vw \in L\}$, d.h. den Rechtsquotienten von L bezüglich w erkennt.

Aufgabe (1.11): (Automaten mit Zusatzgedächtnis) Die Transitionsfunktion eines DEA bestimmt in Abhängigkeit *eines* Zustandes und *eines* Eingabesymbols den

Folgezustand. Wir verallgemeinern das Modell zu einem Automaten $\mathcal{A}^{(n)}$ mit Zusatzgedächtnis der Größe n (n eine positive natürliche Zahl), bei dem der Folgezustand von den letzten n Zuständen, die der Automat durchlaufen hat und dem Eingabesymbol abhängig ist, d.h., die Transitionsfunktion $\delta^{(n)}$ des Automaten $\mathcal{A}^{(n)} = (Q, \Sigma, \delta^{(n)}, q_0, F)$ ist eine Abbildung von $Q^n \times \Sigma$ in Q. Ein Wort $w = a_1 \dots a_k \in \Sigma^*$ wird genau dann akzeptiert, wenn eine Zustandsfolge $p_1, \dots, p_{n+k}$ existiert, so daß $p_1, \dots, p_n = q_0, p_i = \delta^{(n)}(p_{i-n}, \dots, p_{i-1}, a_{i-n})$ für $i \geq n+1$ und $p_{n+k} \in F$ ist.
Zeigen Sie, daß jede Sprache, die von einem Automaten mit Zusatzgedächtnis der Größe n erkannt wird, bereits von einem DEA mit Zusatzgedächtnis der Größe 1 erkannt wird.

Aufgabe (1.12): (Stochastische Automaten) Sei $P = (p_{i,j})_{1 \leq i,j \leq n}$ die Transitionsmatrix eines stochastischen Automaten mit den n Zuständen $q_1, \dots, q_n$; $p_{i,j}$ ist die Wahrscheinlichkeit, daß der Automat nach einem Schritt vom Zustand q_i in den Zustand q_j übergeht. Bestimmen Sie die Wahrscheinlichkeit $p_{i,j}^{(t)}$, daß der Automat vom Zustand q_i in t Schritten in den Zustand q_j übergeht. Beweisen Sie, daß die Matrix $(p_{i,j}^{(t)})_{1 \leq i,j \leq n}$ mit der Matrix P^t übereinstimmt.

Aufgabe (1.13): (Stochastische Akzeptoren) $\mathcal{A} = (Q, \Sigma, P(a)_{a \in S}, \pi, f)$ heißt stochastischer Akzeptor (stochastischer Automat) genau dann, wenn folgendes gilt:

- $P(a)$ ist für jedes $a \in \Sigma$ eine stochastische $n \times n$-Matrix $\{n = |Q|\}$
- π ist ein stochastischer (Zeilen-)Vektor, (Anfangsverteilung)
- f ist ein Spaltenvektor mit n Komponenten, die den Wert 0 oder 1 haben.

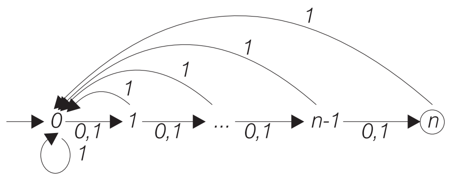
Die Sprache $L(\mathcal{A}) = \{w \in \Sigma^* | \pi \cdot P(\omega) \cdot f > \lambda\}$ ist die von $\mathcal{A}$ akzeptierte stochastische Sprache zum Schnittpunkt λ, wobei λ eine nicht-negative reelle Zahl ≤ 1 ist. Beweisen Sie, daß jede stochastische Sprache zum Schnittpunkt 0 von einem DEA erkannt wird.

Aufgabe (1.14): (Nicht-erkennbare Sprachen) Weisen Sie mit Hilfe des Satzes von Nerode nach, daß folgende Sprachen nicht erkennbar sind.

1. $\Sigma = \{a\}, L = \{a^m | m \text{ ist Potenz von } 2\}$
2. $\Sigma = \{a, b\}, L = \{w || w|_a = |w|_b\}$
3. $\Sigma = \{a, b, c\}, L = \{w || w| = n!, n * \mathbb{N}\}$

Aufgabe (1.15): (Pattern Matching) Zeigen Sie die folgende Behauptung: Zu jedem Wort w der Länge n gibt es einen DEA mit $n+1$ Zuständen, der genau die Wörter akzeptiert, die w als Infix haben. Hinweis: Als Zustände kann man die Präfixe von w wählen, dazu sind dann geeignete Transitionen zu wählen. Gibt es einen kleineren DEA, der dieselbe Sprache erkennt?

Aufgabe (1.16): (Optimalität der Potenzmengenkonstruktion) Sei $\Sigma = \{0, 1\}$. Für $n \geq 1$ sei $\mathcal{A}_n$ der folgende NEA mit der Zustandsmenge $\{0, \dots, n\}$:

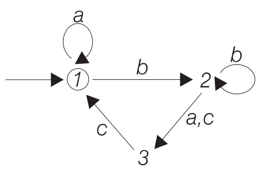


1. Beschreiben Sie umgangssprachlich die durch $\mathcal{A}_n$ akzeptierten Wörter.
2. Zeigen Sie, daß die Potenzmengenkonstruktion für $\mathcal{A}_n$ einen minimalen (!) Automaten mit 2^{n+1} Zuständen liefert.

   Hinweis: Zeigen Sie zuerst, daß jede Menge $P \subseteq \{0, \ldots, n\}$ im Potenzmengenautomaten durch ein Wort w erreichbar ist. Zeigen Sie dann, daß je zwei Mengen P und Q im Potenzmengenautomaten nicht äquivalent sind.

1.3 Reguläre Ausdrücke

Idee: Die durch $\mathcal{A}$



erkannte Sprache wird kompakter durch den Ausdruck

$$(a^*bb^*(a+c)c)^*a^*$$

beschrieben.

Definition: Die Menge der *regulären Ausdrücke* über einem Alphabet Σ $(0, 1 \notin \Sigma)$, ist induktiv definiert durch:

- 0,1, alle $a \in \Sigma$ sind reguläre Ausdrücke
- sind r, s reguläre Ausdrücke, so auch $(r + s), (r \bullet s), r^*$

Konventionen: Außenklammern fallen weg, $\bullet$ bindet stärker als $+$, * stärker als $\bullet$, $\bullet$ darf wegfallen.
Also: Statt $((a \bullet b^*) + b)$ schreiben wir auch $ab^* + b$

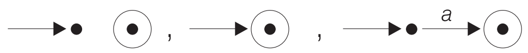
Definition: Die durch den regulären Ausdruck r definierte Sprache $L(r)$ wird induktiv festgelegt durch:

- $L(0) = \emptyset, L(1) = \{\varepsilon\}, L(a) = \{a\}$
- $L(r + s) = L(r) \cup L(s), L(r \bullet s) = L(r) \cdot L(s), L(r^*) = L(r)^*$

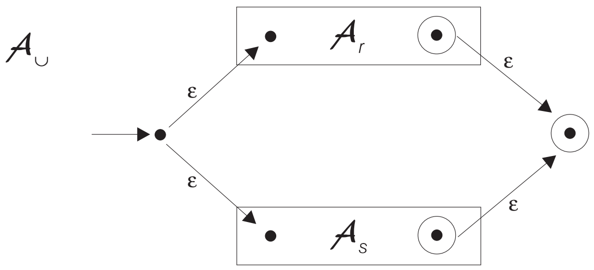
Häufig schreiben wir r statt $L(r)$ und machen keine Unterscheidung zwischen $\bullet$ und $\cdot$, $\ast$ und *.

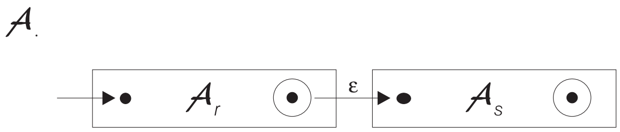
Satz: **(Kleene)** *Eine Sprache* $L \subseteq \Sigma^*$ *ist durch einen NEA erkennbar* $\Leftrightarrow$ L *ist durch einen regulären Ausdruck definierbar.*

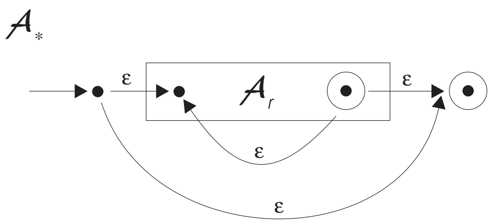
Beweis: „$\Leftarrow$“ durch Induktion über den Aufbau der regulären Ausdrücke. Finde für jeden regulären Ausdruck r einen ε-NEA $\mathcal{A}_r$ mit nur einem Endzustand, so daß $L(r) = L(\mathcal{A}_r)$.
Induktionsanfang: Die Fälle $r = 0, 1, a \in \Sigma$ sind trivial: Entsprechende ε-NEA's:



Induktionsschritt: I.V. Zu r, s mögen ε-NEA's mit $\mathcal{A}_r, \mathcal{A}_s$ mit nur einem Endzustand existieren, so daß $L(r) = L(\mathcal{A}_r), L(s) = L(\mathcal{A}_s)$.
Induktionsbehauptung: Zu $(r + s), (r \cdot s), r^*$ existieren ε-NEA's $\mathcal{A}_\cup, \mathcal{A}_., \mathcal{A}_*$ mit nur einem Endzustand, so daß $L(r + s) = L(\mathcal{A}_\cup), L(r \cdot s) = L(\mathcal{A}_.), L(r^*) = L(\mathcal{A}_*)$. Wir wählen







Vorbereitung für „$\Rightarrow$“: Reguläre Ausdrücke (mit ihrer Standardbedeutung) genügen folgenden Gleichungen:

(1) $X + Y = Y + X$

(2) $X(Y + Z) = XY + XZ$

(3) $(X + Y)Z = XZ + YZ$

(4) $X + 0 = 0 + X = X$

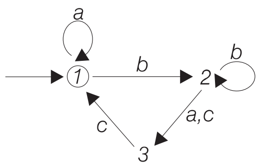
(5) $X \cdot 1 = 1 \cdot X = X$

Wir geben den Beweis zu (2):

$$\begin{aligned}
w \in K \cdot (L \cup M) \quad &\Leftrightarrow \quad \text{ex. } u, v : u \in K, v \in L \cup M, uv = w \\
&\Leftrightarrow \quad \text{ex. } u, v : u \in K, (v \in L \vee v \in M), uv = w \\
&\Leftrightarrow \quad \text{ex. } u, v : (u \in K, v \in L, uv = w) \\
& \quad \text{oder } (u \in K, v \in M, uv = w) \\
&\Leftrightarrow \quad \text{ex. } u, v : (u \in K, v \in L, uv = w) \\
& \quad \text{oder ex. } u, v : (u \in K, v \in M, uv = w) \\
&\Leftrightarrow \quad w \in K \cdot L \text{ oder } w \in K \cdot M \\
&\Leftrightarrow \quad w \in KL \cup KM
\end{aligned}$$

Idee zum Satz von Kleene (Richtung „⇒“):
Jeder NEA $\mathcal{A} = (q_1, \ldots, q_n, \Sigma, q_1, \Delta, F)$ bestimmt Gleichungen für die Sprachen $L_1, \ldots, L_n$, wobei $L_i = L(q_1, \ldots, q_n, \Sigma, q_i, \Delta, F)$. □

Beispiel:



Für L_1, L_2, L_3 gelten die Gleichungen

$$\begin{aligned}
L_1 &= \{a\} \cdot L_1 \cup \{b\} \cdot L_2 \cup \{\varepsilon\} \\
L_2 &= \{b\} \cdot L_2 \cup \{a, c\} \cdot L_3 \\
L_3 &= \{c\} \cdot L_1
\end{aligned}$$

$\mathcal{A}$ induziert also Gleichungssystem $GS(\mathcal{A})$ mit regulären Ausdrücken auf den rechten Seiten und Variablen X_i für Sprachen:

(1) $X_1 = aX_1 + bX_2 + 1$

(2) $X_2 = bX_2 + (a + c)X_3$

(3) $X_3 = cX_1$

Allgemein: Zu $\mathcal{A} = (q_1, \ldots, q_n, \Sigma, q_1, \Delta, F)$ sei $GS(\mathcal{A})$ das Gleichungssystem

$$X_i = \sum_{j=1}^{n} A_{ij} X_j + E_i, (i = 1, \ldots, n)$$

mit

$$A_{ij} = \{a | q_i, a, q_j) \in \Delta\}, E_i = \begin{cases} 0 & q_i \notin F \\ 1 & q_i \in F \end{cases}$$

Lemma: (Resolutionslemma): *Die Gleichung $X = AX + B$ mit $A, B \subseteq \Sigma^*$ wird gelöst durch $X = A^*B$.*

Beweis: $A^*B = A^+B \cup B = (AA^*)B \cup B = A(A^*B) \cup B$. □

Anwendung im Beispiel: Einsetzen von (3) in (2) liefert $X_2 = bX_2 + (a+c)cX_1$.
Resolutionslemma liefert: $X_2 = b^*(a+c)cX_1$
Einsetzen in (1) liefert: $X_1 = aX_1 + bb^*(a+c)cX_1 + 1 = (a + bb^*(a+c)c)X_1 + 1$
Resolutionslemma liefert: $X_1 = (a+bb^*(a+c)c)^*$ als regulären Ausdruck für L_1, d.h. für $L(\mathcal{A})$.

Allgmeines Verfahren der Auflösung von $GS(\mathcal{A})$:

- für eine einzige Gleichung in $GS(\mathcal{A})$ wende das Resolutionslemma an.
- für $m+1$ Gleichungen in $\leq m+1$ Variablen nehme Reduktion auf m Gleichungen in $\leq m$ Variablen vor: Betrachte

  $$X_{m+1} = A_{m+1,m+1}X_{m+1} + \sum_{j=1}^{m} A_{m+1,j}X_j + E_{m+1}.$$

  Resolutionslemma liefert

  $$X_{m+1} = A^*_{m+1,m+1}\left(\sum_{j=1}^{m} A_{m+1,j}X_j + E_{m+1}\right).$$

  Einsetzen in die ersten m Gleichungen liefert die gewünschte Reduktion.
- Iterierte Anwendung liefert regulären Ausdruck für X_1.

Zusatz Ist (wie bei NEA's) in keinem A_{ij} das leere Wort enthalten, sind die Lösungen eindeutig.

Übungen

Aufgabe (1.17): (Reguläre Ausdrücke) Es seien L und M formale Sprachen. Beweisen Sie die Gültigkeit der folgenden Gleichung bzw. Ungleichung:

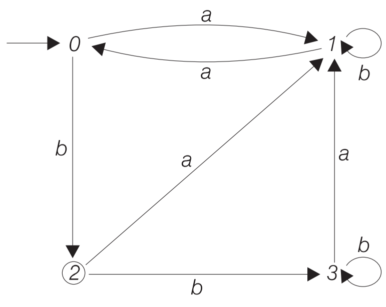
$$(L \cup M)^* = (L^* \cdot M^*)^*$$

$$(L \cup M)^* = (L^* \cdot M)^* \cup (M^* \cdot L)^*(L^* \cdot M \cdot L^*)^* \subseteq (L^* \cdot M)^* \cdot L^*$$

Geben Sie konkrete reguläre Sprachen L und M an, so daß in der Ungleichung die echte Inklusion eintritt.

Aufgabe (1.18): (Resolutionslemma) Es seien L, M und R formale Sprachen, M enthalte nicht das leere Wort. Beweisen Sie, daß $L = (M \cdot L) \cup R$ genau dann gilt, wenn $L = M^*R$ ist.

Aufgabe (1.19): (Konstruktion regulärer Ausdrücke) Berechnen Sie für die durch den folgenden Automaten erkannte Sprache einen regulären Ausdruck nach dem obigen Verfahren.



1.4 Sequentielle Maschinen

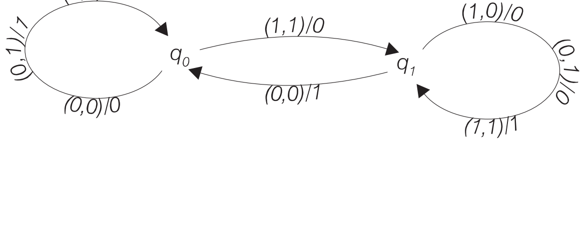
Idee: Wir verwenden verallgemeinerte Transitionen $q \xrightarrow{a/v} q'$: mit *Eingabebuchstabe* a, *Ausgabewort* v.

Definition: Eine *verallgemeinerte sequentielle Maschine* (kurz GSM: generalized sequential machine) hat die Form $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \delta, \lambda)$ mit Q endlich, Σ, Γ Eingabe- bzw. Ausgabealphabet, $q_0 \in Q$, $\delta : Q \times \Sigma \to Q$, $\lambda : Q \times \Sigma \to \Gamma^*$. Variante: *Sequentielle Maschine* oder *Mealy-Automat*, falls $\lambda : Q \times \Sigma \to \Gamma$.
Erweitere δ auf $Q \times \Sigma^*$ wie zuvor, entsprechend definiere $\lambda^* : Q \times \Sigma^* \to \Gamma^*$ wie folgt: $\lambda^*(q, \varepsilon) = \varepsilon, \lambda^*(q, wa) = \lambda^*(q, w)\lambda(\delta(q, w), a)$. Dann sei $f_{\mathcal{A}} : \Sigma^* \to \Gamma^*$ definiert durch $f_{\mathcal{A}}(w) := \lambda^*(q_0, w)$
$f_{\mathcal{A}}(w)$ ist also das durch $\mathcal{A}$ bei Lesen von w gelieferte Ausgabewort.

Beispiel: *Addition gespiegelter Binärzahlen* mit wenigstens einer führenden 0. Beispiel (führende Nullen stehen rechts!):

$$\begin{array}{c} \texttt{101100} \\ \underline{\texttt{100010}} \\ \texttt{011110} \end{array}$$

Ein- und Ausgabealphabet sind $\Sigma = \{(0,0), (0,1), (1,0), (1,1)\}, \Gamma = \{0,1\}$. Die entsprechende Funktion $f_+ : \Sigma^* \to \Gamma^*$ wird berechnet durch:



Zustand q_0 repräsentiert Übertrag 0, q_1 Übertrag 1.

Eigenschaften verallgemeinerter sequentieller Funktionen:
$f : \Sigma^* \to \Gamma^*$ sei verallgemeinert sequentiell, d.h. es existiert eine verallgemeinerte sequentielle Maschine $\mathcal{A}$ mit $f = f_{\mathcal{A}}$. Dann gilt:

1. f ist *präfixtreu*, d.h. $f(uv)$ hat jeweils $f(u)$ als Präfix.
2. f ist *längenbeschränkt*, d.h. es existiert $k \geq 0$ mit $|f(w)| \leq k \cdot |w|$ für alle $w \in \Sigma^*$.
3. $L \subseteq \Sigma^*$ regulär $\Rightarrow f(L)$ regulär.

Der Beweis von 1., 2. ist einfach. Behauptung 3. beweise der Leser als Übung.

Anwendung: Die *Multiplikation gespiegelter Binärzahlen* liefert eine *nicht* verallgemeinert sequentielle Funktion $f_{\cdot} : \Sigma^* \to \Gamma^*$ mit Σ, Γ wie vorher.
Wegen 3. genügt Angabe von $L \subseteq \Sigma^*$ regulär mit $f_{\cdot}(L)$ nicht regulär.
Betrachte die reguläre Sprache $L = \{(0,1)^n(1,0) | n \geq 0\}$. Für $n = 3$ ergeben sich die Eingabewörter `1000` und `0111`. Anhand der Beispielmultiplikation

```
1000·0111
 1000
  1000
   1000
 111000
```

sehen wir: $f_{\cdot}(L) = \{0^n1^n | n > 0\} \cup \{0\}$, also ist $f_{\cdot}(L)$ nicht regulär.

Kapitel 2

Grammatiken, kontextfreie Sprachen

2.1 Grammatiken: Beispiele und Klassifikation

Automaten sind Akzeptoren gegebener Wörter.
Grammatiken sind Regelsysteme zur Erzeugung von Wörtern.

Beispiel: BNF-Regeln:
$\langle\text{Term}\rangle ::= 0 \,|\, 1 \,|\, (\langle\text{Term}\rangle + \langle\text{Term}\rangle) \,|\, (\langle\text{Term}\rangle * \langle\text{Term}\rangle)$
Beispielableitung:

$$\begin{array}{rcl}
\langle\text{Term}\rangle & \vdash & (\langle\text{Term}\rangle + \langle\text{Term}\rangle) \\
 & \vdash & (0 + \langle\text{Term}\rangle) \\
 & \vdash & (0 + (\langle\text{Term}\rangle * \langle\text{Term}\rangle)) \\
 & \vdash & (0 + (1 * \langle\text{Term}\rangle)) \\
 & \vdash & (0 + (1 * 1))
\end{array}$$

Definition: Eine *Chomsky-Grammatik* hat die Form $G = (N, \Sigma, P, S)$ mit:
N: endliche Menge von *Nichtterminalsymbolen* [z.B. $\langle\text{Term}\rangle$]
Σ: endliches zu N disjunktes Alphabet von *Terminalsymbolen* [z.B. $(,), +, *$]
P: endliche Menge von *Regeln* oder *Produktionen*, $P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
$S \in N$: *Startsymbol*

Schreibweise „$\alpha \to \beta \in P$“ statt „$(\alpha, \beta) \in P$“ (BNF: $\alpha ::= \beta$)

Beispiel: $\langle\text{Term}\rangle \to (\langle\text{Term}\rangle + \langle\text{Term}\rangle)$

Symbole in N: $A, B, \ldots, S$
Symbole in Σ: $a, b, c, \ldots$
Wörter über $N \cup \Sigma$: $\alpha, \beta, \gamma, \ldots$ (Satzformen)

Wörter über Σ: $u, v, w, \ldots$
$\alpha \vdash_G \beta$ besage: ex. $\gamma, \delta, \alpha_1, \beta_1$ mit $\alpha = \gamma\alpha_1\delta, \alpha_1 \to \beta_1 \in P, \beta = \gamma\beta_1\delta$

Beispiel: $\underbrace{(}_{\gamma}\underbrace{\langle\text{Term}\rangle}_{\alpha_1}\underbrace{+\langle\text{Term}\rangle)}_{\delta} \vdash \underbrace{(}_{\gamma}\underbrace{0}_{\beta_1}\underbrace{+\langle\text{Term}\rangle)}_{\delta}$

Weitere Schreibweisen:
$\alpha \vdash^n_G \beta$ für ex. $\alpha_0, \ldots, \alpha_n$ mit $\alpha_0 = \alpha, \alpha_i \vdash_G \alpha_{i+1} (i < n), \alpha_n = \beta$
$\alpha \vdash^*_G \beta$ für ex. $n \geq 0$ mit $\alpha \vdash^n_G \beta$

Definition: Die durch $G = (N, \Sigma, P, S)$ erzeugte Sprache sei

$$L(G) = \{w \in \Sigma^* | S \vdash^* w\}.$$

Beispiel: *Grammatiken* (in Kurznotation, wobei rechte Regelseiten bei gleicher linker Regelseite durch | getrennt werden, wie bei BNF):

$$\begin{aligned} G_1 : S &\to abc|aAbc \\ Ab &\to bA \\ Ac &\to Bbcc \\ bB &\to Bb \\ aB &\to aaA|aa \end{aligned}$$

$$G_2 : S \to 0|1|(S+S)|(S*S)$$

$$\begin{aligned} G_2' : S &\to aB|bA \\ A &\to a|aS|bAA \\ B &\to b|bS|aBB \end{aligned}$$

$$\begin{aligned} G_3 : S &\to aA|a, \\ A &\to aA|a|bB, \\ B &\to aB|bA|b \end{aligned}$$

Beispiel: Ableitungen:
$G_1 : S \vdash aAbc \vdash abAc \vdash abBbcc \vdash aBbbcc \vdash aabbcc$
$G_2' : S \vdash aB \vdash abS \vdash abbA \vdash abbaS \vdash abbaaB \vdash abbaab$
$G_3 : S \vdash aA \vdash aaA \vdash aabB \vdash aabbA \vdash aabba$

Behauptung: über die erzeugten Sprachen:
$L(G_1) = \{a^n b^n c^n | n > 0\}$
$L(G_2) = T$, wobei T=kleinste Menge M, die 0,1 enthält und die mit $t_1, t_2 \in M$ jeweils auch $(t_1 + t_2), (t_1 * t_2)$ enthält (Menge der $+/*$ -Terme über 0,1).
$L(G_2') = \{w \in \{a, b\}^* || w|_a = |w|_b\}$
$L(G_3) = \{w \in \{a, b\}^* | w$ beginnt mit a und hat geradzahlig viele $b\}$

Wir geben die (etwas mühseligen) Beweise. Von rechts nach links gilt es nur, Ableitungen mit bekannten Zielen zu finden; von links nach rechts muß man alle Ableitungsmöglichkeiten in den gegebenen Grammatiken verfolgen.

Beweis: zu G_1, „$\supseteq$“: Es gilt $S \vdash abc$ (mit erster Regel). Wir zeigen:

$$S \vdash^* a^i B b^{i+1} c^{i+1} \quad \forall i \geq 1.$$

Dann $S \vdash^* a^i b^i c^i$ für $i \geq 2$ mit letzter Regel.
Beweis durch Induktion über $i \geq 1$:
$i = 1$, klar nach Definition von G_1 (siehe obige Beispiel-Ableitung bis zum vorletzten Wort).
Induktionsschritt: Gelte $S \vdash^* a^i B b^{i+1} c^{i+1}$, zeige $S \vdash^* a^{i+1} B b^{i+2} c^{i+2}$. klar, wegen $a^i B b^{i+1} c^{i+1} \vdash^* a^{i+1} B b^{i+2} c^{i+2}$ mit den Regelanwendungen:

$$\begin{array}{rrcll} & aB & \to & aaA & (a^{i+1} A b^{i+1} c^{i+1}) \\ (i+1)\text{mal} & Ab & \to & bA & (a^{i+1} b^{i+1} A c^{i+1}) \\ & Ac & \to & Bbcc & (a^{i+1} b^{i+1} B b c^{i+2}) \\ (i+1)\text{mal} & bB & \to & Bb & (a^{i+1} B b^{i+2} c^{i+2}) \end{array}$$

zu G_1, „$\subseteq$“: Gelte $S \vdash^* w$. Falls $S \vdash w$, dann $w = abc$, fertig.
Sonst $S \vdash aAbc \vdash \ldots \vdash aBbbcc$ eindeutig. Von $a^i B b^{i+1} c^{i+1}$ Übergang direkt zu $a^{i+1} b^{i+1} c^{i+1}$ möglich oder Übergang zu $a^{i+1} A b^{i+1} c^{i+1}$ und dann weiter eindeutig bis $a^{i+1} B b^{i+2} c^{i+2}$.
Also Terminalwörter nur aus Satzformen $a^n B b^{n+1} c^{n+1}$ direkt herstellbar, somit von gewünschter Form $a^{n+1} b^{n+1} c^{n+1}$.

zu G_2, „$\supseteq$“: $\forall t \in T$ ist $S \vdash^* t$ zu zeigen. Nachweis durch Induktion über Aufbau von T: $t = 0$ bzw. 1: $S \vdash 0, S \vdash 1$ klar.
Induktionsschritt: $t = (t_1 + / * t_2)$, nach I.V. gelte $S \vdash^* t_1, S \vdash^* t_2$. Wir zeigen die I.B. $S \vdash^* (t_1 + / * t_2)$ Dies ergibt sich mit der Ableitung $S \vdash (S + / * S) \vdash^* (t_1 + / * S) \vdash^* (t_1 + / * t_2)$

zu G_2, „$\subseteq$“: Zeige: $(\forall n : S \vdash^n w) \Rightarrow w \in T$.
$n = 1$: $w = 0$ oder $w = 1$, also $w \in T$.
Induktionsschritt: Gelte $S \vdash^{n+1} w$, zeige $w \in T$. Die gegebene Ableitung hat die Form $S \vdash (S + / * S) \vdash \ldots \vdash (w_1 + / * w_2) = w$ mit $S \vdash^{\leq n} w_1, S \vdash^{\leq n} w_2$. I.V. liefert $w_1 \in T, w_2 \in T$. Also $w = (w_1 + / * w_2) \in T$.

zu G_2': Zeige durch Induktion über $|w| \geq 1$

(1) $S \vdash^* w \Leftrightarrow |w|_a = |w|_b$

(2) $A \vdash^* w \Leftrightarrow |w|_a = |w|_b + 1$

(3) $B \vdash^* w \Leftrightarrow |w|_a + 1 = |w|_b$

Dann ergibt sich aus (1) die Behauptung.
$|w| = 1$:

(1) gilt, da beide Seiten falsch

(2) gilt, da $w = a$, beide Seiten gelten

(3) analog

Induktionsschritt: (1),(2),(3) mögen gelten für Wortlänge k (I.V.). Sei $|w| = k+1$. Zeige hier (1): Für „$\Rightarrow$" gelte $S \vdash^* w$, z.B. mit Ableitung $S \vdash aB \vdash \ldots \vdash av = w$, dann auch $B \vdash^* v$. Also wegen I.V. (3) gilt $|v|_b = |v|_a + 1$, also $|w|_a = |w|_b$.
Für „$\Leftarrow$" gelte $|w| = k+1, |w|_a = |w|_b$, z.B. $w = av$ (analog für bv). Es gilt $|v| = k, |v|_b = |v|_a + 1$. I.V. (3) liefert $B \vdash^* v$, folglich $S \vdash aB \vdash^* av = w$, also $S \vdash^* w$.
(2) und (3) zeigt man analog.

Auf G_3 gehen wir weiter unten ein. □

Definition: Eine Grammatik $G = (N, \Sigma, P, S)$ heißt

- vom *Typ* 0 im allgemeinen Fall.
- vom Typ 1 (oder *kontextsensitiv*), falls für jede Regel $\alpha \to \beta$ gilt: $|\alpha| \leq |\beta|$.
- vom Typ 2 (oder *kontextfrei*), falls jede Regel die Form $A \to \alpha$ hat.
- vom Typ 3 (oder *rechtslinear*), falls jede Regel die Form $A \to w$ oder $A \to wB$ hat, $w \neq \varepsilon$.

In den Fällen Typ 1-3 verlangen wir die ε-Bedingung: ε ist auf rechter Seite nur in der Regel $S \to \varepsilon$ erlaubt, und dann darf S in keiner rechten Regelseite auftauchen. (Also ist das ε nur mit einer Ableitung $S \vdash \varepsilon$ direkt aus S herstellbar.)

Die Beispielgrammatik G_1 ist vom Typ 1, G_2 und G_2' vom Typ 2 und G_3 vom Typ 3.

Definition: Eine Sprache $L \subseteq \Sigma^*$ heißt vom Typ 0 (*rekursiv aufzählbar*), 1 (*kontextsensitiv*), 2 (*kontextfrei*), 3 (*rechtslinear*), falls es eine Grammatik G entsprechenden Typs gibt mit $L = L(G)$.

Bemerkung: *Es gilt offensichtlich nach Definition: L rechtslinear $\Rightarrow$ L kontextfrei $\Rightarrow$ L kontextsensitiv $\Rightarrow$ L rekursiv aufzählbar (*Chomsky-Hierarchie*)*

Satz: *L rechtslinear $\Leftrightarrow$ L regulär*

Beweis: durch effektive Transformation von rechtslinearen Grammatiken in NEA's und umgekehrt. „$\Rightarrow$": Sei L rechtslinear, $L = L(G)$ für $G = (N, \Sigma, P, S)$ rechtslinear.
Dann gilt $w \in L(G) \Leftrightarrow$ ex. G-Ableitung

$$S =: B_0 \vdash w_1 B_1 \vdash w_1 w_2 B_2 \vdash \ldots \vdash w_1 \ldots w_{n-1} B_{n-1} \vdash w_1 \ldots w_n,$$

wobei

$$w = w_1 \ldots w_n, \quad B_i \to w_{i+1} B_{i+1} \in P (i < n-1), \quad B_{n-1} \vdash w_n \in P. (*)$$

Definiere $\mathcal{A} := (N \cup \Omega, \Sigma, S, \Delta, \Omega)$ mit

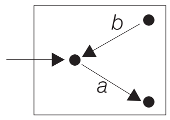
$$(A, w, B) \in \Delta :\Leftrightarrow A \to wB \in P, \quad (A, w, \Omega) \in \Delta :\Leftrightarrow A \to w \in P.$$

Dies ist ein NEA mit Worttransitionen. Dann gilt

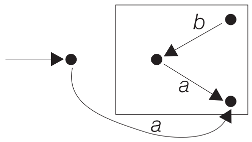
$$(*) \Leftrightarrow \text{ex. Pfad } S = B_0 w_1 B_1 \dots w_{n-1} B_{n-1} w_n \Omega$$

durch $\mathcal{A}$ mit $w = w_1 \dots w_n \Leftrightarrow \mathcal{A}$ akzeptiert w, wie gewünscht.

„$\Leftarrow$“: Sei NEA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ gegeben, oBdA ohne Transitionen nach q_0. (Andernfalls gehe zunächst von



über zu



.) Definiere Grammatik $G = (N, \Sigma, P, S)$ durch

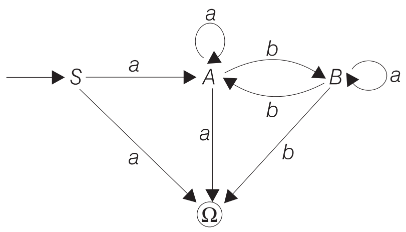
$$N := Q, S := q_0, A \to aB \in P :\Leftrightarrow (A, a, B) \in \Delta, A \to a \in P :\Leftrightarrow (A, a, B) \in \Delta$$

für ein $B \in F$. Falls $q_0 \in F$, nehme auch $S \to \varepsilon$ hinzu. Nach der oBdA-Annahme gilt die ε-Bedingung.
Dann gilt für nichtleeres $w = a_1 \dots a_n$: $\mathcal{A}$ akzeptiert $w \Leftrightarrow \exists$Pfad $q_0 a_1 q_1 \dots a_n q_n \in F$ in $\mathcal{A} \Leftrightarrow$ ex. G-Ableitung

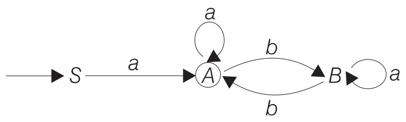
$$S = B_0 \vdash a_1 B_1 \vdash a_1 a_2 B_2 \vdash \dots \vdash a_1 \dots a_{n-1} B_{n-1} \vdash a_1 \dots a_n$$

mit $B_i = q_i \Leftrightarrow w \in L(G)$.
Für $w = \varepsilon$ gilt $\mathcal{A}$ akzeptiert $\varepsilon \Leftrightarrow q_0 \in F \Leftrightarrow S \to \varepsilon \in P \Leftrightarrow S \vdash_G^* \varepsilon$ $\square$

Beispiel: für den Übergang von G_3 (s.o.) zu einem NEA:



vereinfacht



Also werden die Wörter erkannt, die mit a beginnen und geradzahlig viele b haben.

Zusatz: Jede rechtslineare Grammatik ist äquivalent zu einer rechtslinearen Grammatik nur mit Regeln $A \to a, A \to aB$ (evtl. $S \to \varepsilon$).

Beweis: durch folgende Übergänge von gegebener Grammatik G aus: $G \to$ NEA mit Worttransitionen $\to$ NEA $\to$ rechtslineare Grammatik G' (nach obigem Beweis).
□

Grundsätzliche Fragen zu Grammatiken

1. Vergleich der *Ausdrucksstärke* der verschiedenen Typen
2. Zusammenhänge *Grammatiken - Automaten*
3. Reduktion auf einfache Regelformen (*Normalformen*)
4. Entscheidungsprobleme, z.B.
   - *Wortproblem*: Gegeben G, w, teste, ob w in G ableitbar
   - *Leerheitsproblem*: Gegeben G, teste, ob $L(G) = \emptyset$
   - *Äquivalenzproblem*: Gegeben G_1, G_2, teste, ob $L(G_1) = L(G_2)$

Wir diskutieren hier nur einige elementare Ergebnisse, Näheres wird in der Vorlesung „Automaten und formale Sprachen“ gezeigt.

Übungen

Aufgabe (2.1): (kontextfreie Grammatiken) Geben Sie kontextfreie Grammatiken G_1, G_2 für die Sprachen $L_1 = \{a^i b^j c^j | i, j > 0\}$, $L_2 = \{a^i b^j c^k | i = j + k, i > 0\}$ an und zeigen Sie $L_1 = L(G_1), L_2 = L(G_2)$.

Aufgabe (2.2): (linkslineare Grammatiken) Eine Grammatik $G = (N, \Sigma, P, S)$ heißt vom Typ 3 (linkslinear), falls jede Regel die Form $A \to w$ oder $A \to Bw$ hat ($w \in \Sigma^*, B \in N$). Regeln dieser Form nennen wir ebenfalls linkslinear (analog für rechtslinear):

1. Zeigen Sie: Eine Sprache ist genau dann linkslinear, wenn sie rechtslinear ist.
2. Beweisen Sie oder widerlegen Sie die folgende Behauptung: Jede Sprache, die von einer Grammatik erzeugt wird, so daß jede Regel die Form $A \to a$, $A \to aB$ oder $A \to Ba$ hat, ist regulär.

2.2 Chomsky-Normalform

Satz: (Chomsky-Normalform) *Zu jeder kontextfreien Grammatik G kann man effektiv eine kontextfreie Grammatik G' herstellen nur mit Regeln $A \to a$, $A \to BC$ (evtl. außerdem $S \to \varepsilon$).*

Beweis: 1. Etappe: Reduktion der Terminalsymbole auf Regeln $A \to a$. Führe für $a \in \Sigma$ jeweils Nichtterminalsymbol X_a ein, ersetze in Regeln a durch X_a (Dann $S \vdash_G a_1 \dots a_n \Leftrightarrow S \vdash_{\text{neue Gr.}} X_{a_1} \dots X_{a_n}$) Füge Regeln $X_a \to a$ hinzu, erhalte so G_1. Dann $L(G) = L(G_1)$.

Beispiel: $G : S \to aSb|ab, L(G) = \{a^n b^n | n > 0\}$ (eine kontextfreie, nicht reguläre Sprache) Wir erhalten $G_1 : S \to X_a S X_b | X_a X_b, X_a \to a, X_b \to b$.

2. Etappe: Elimination von Regeln $A \to B$

(1) Bestimme alle Ableitungen $A_1 \vdash A_2 \vdash \dots \vdash A_k \vdash \alpha \notin N$ ohne Nichtterminal-Wiederholungen (effektiv bestimmbar).

(2) Streiche alle Regeln $A \to B$

(3) Füge für $\alpha \notin N$ Regel $A_1 \to \alpha$ hinzu, falls Ableitung $A_1 \vdash \dots \vdash A_k \vdash \alpha$ mit gewissen $A_2, \dots, A_k$ existiert.

Erhalte so G_2. Beh: $L(G_1) = L(G_2)$

„$\subseteq$“: Gelte $S \vdash_{G_1} \dots \vdash_{G_1} w$. Transformiere diese G_1-Ableitung schrittweise in G_2-Ableitung durch Elimination der Regelanwendungen $A \to B$ von links nach rechts. Betrachte erstes Vorkommen einer solchen Regelanwendung:

$$S \vdash_{G_1}^* \beta A_1 \gamma \vdash_{G_1} \beta A_2 g \vdash_{G_1}^* \beta' \alpha \gamma' \vdash_{G_1}^* w, \qquad \alpha \notin N$$

mit

$$A_1 \vdash A_2 \vdash \dots \vdash A_k \vdash \alpha, b \vdash_{G_1}^* \beta', \qquad \gamma \vdash_{G_1}^* \gamma'.$$

Erhalte neue Ableitung

$$S \vdash_{G_2}^* \beta A_1 \gamma \vdash_{G_2} \beta \alpha \gamma \vdash_{G_1}^* \beta' \alpha \gamma' \vdash_{G_1}^* w$$

mit G_2-Anfangsstück gemäß Definition von G_2. Sukzessives Anwenden dieser Ersetzung liefert $S \vdash_{G_2}^* w$.

„$\supseteq$“: Ersetze neue Regel $A_1 \to \alpha$ durch Anwendungen $A_1 \vdash_{G_1} \dots \vdash_{G_1} A_k \vdash_{G_1} \alpha$ in G_1.

3. Etappe: Elimination von Regeln $A \to B_1 \dots B_n$ $(n \geq 3)$. Ersatz durch Regeln

$$A \to B_1 C_1, \qquad C_1 \to B_2 C_2, \qquad C_{n-2} \to B_{n-1} B_n$$

mit jeweils neuen Hilfs-Nichtterminalsymbolen $C_1, \dots, C_{n-2}$. Erhalte so $G_3 := G'$. Es gilt $L(G_2) = L(G_3)$. □

Beispiel: Aus

$$S \to X_a S X_b | X_a X_b, \qquad X_a \to a, \qquad X_b \to b$$

ergibt sich

$$S \to X_a C | X_a X_b, \qquad X_a \to a, \qquad X_b \to b, \qquad C \to S X_b.$$

Übungen

Aufgabe (2.3): (Chomsky-Normalform) Bestimmen Sie eine Chomsky-Normalform für die folgende Grammatik: $S \rightarrow aAB, A \rightarrow AAB|BA|aba, B \rightarrow a|aBa$.

Aufgabe (2.4): (Beweisdetail) Zeigen Sie ausführlich, daß für die in der 3. Etappe des Beweises des Chomsky-Normalform-Theorems angegebene Grammatik G_3 $L(G_2) = L(G_3)$ gilt.

2.3 Wort- und Leerheitsproblem für kontextfreie Grammatiken

Wir beginnen mit einem Rückblick auf endliche Automaten (NEA's) und zeigen, daß Wort-, Leerheits- und Äquivalenzproblem entscheidbar sind.

Wortproblem: Von NEA $\mathcal{A}$ gehe zu DEA $\mathcal{A}'$ über, berechne den Zustand, der durch das Eingabewort w bestimmt wird und teste, ob Endzustand vorliegt.

Leerheitsproblem: Zu gegebenem NEA ist zu testen, ob ein Pfad vom Anfangszustand zu einem Endzustand existiert. Vorgehen analog zur Bestimmung der ε-Pfade in Abschnitt 1.1. Gegeben sei eine Transitionsmatrix (t_{ij}) mit

$$t_{ij} = \begin{cases} 1 & \text{falls ex. Transition } (q_i, a, q_j) \in \Delta \\ 0 & \text{sonst} \end{cases}$$

Berechne hieraus (c_{ij}) wie in Abschnitt 1.1:

$$c_{ij} = \begin{cases} 1 & \text{falls ex. Pfad von } q_i \text{ nach } q_j \\ 0 & \text{sonst} \end{cases}$$

Prüfe in (c_{ij}) für Zeile des Anfangszustandes (z.B. $i = 1$), ob eine 1 in einer Endzustandsspalte existiert.

Äquivalenzproblem: Zu gegebenen NEA's $\mathcal{A}, \mathcal{B}$ konstruiere NEA $\mathcal{C}$, der die symmetrische Differenz $(L(\mathcal{A}) \setminus L(\mathcal{B})) \cup (L(\mathcal{B}) \setminus L(\mathcal{A}))$ erkennt. (Beachte: diese ist leer $\Leftrightarrow L(\mathcal{A}) = L(\mathcal{B})$). Genügt also Leerheitstest für $\mathcal{C}$.
Zur Konstruktion von $\mathcal{C}$ über Σ: $L(\mathcal{A}) \setminus L(\mathcal{B}) = L(\mathcal{A}) \cap (\Sigma^* \setminus L(\mathcal{B}))$. Konstruktion klar wegen Abschlußeigenschaften von DEA's unter $\cup, \cap$ und Komplement.

Wir betrachten nun das Wort- und das Leerheitsproblem für kontextfreie Grammatiken:

Satz: (Entscheidbarkeit des Wortproblems) *Sei G eine kontextfreie Grammatik über Σ in Chomsky-Normalform. Dann entscheidet der sogenannte* Cocke-Younger-Kasami-Algorithmus *(CYK-Algorithmus) zu $w \in \Sigma^*$ mit dem Zeitaufwand $O(|w|^3)$, ob $w \in L(G)$.*

Beweis: Sei $G = (N, \Sigma, P, S)$. $w = a_1 \dots a_n$ gegeben. Idee: Bestimme für i, j $(1 \leq i \leq j \leq n)$ jeweils die Nichtterminalmenge $N_{ij} := \{A \in N | A \vdash_G^* a_i \dots a_j\}$. Dann

$$w \in L(G) \Leftrightarrow S \vdash^* w = a_1 \dots a_n \Leftrightarrow S \in N_{1n}.$$

Die Berechnung der N_{ij} nach wachsenden Differenzen $d = j - i$.
Es gilt: $A \in N_{ii} \Leftrightarrow A \to a_i$ in P, da G in Chomsky-Normalform.
Für $i < j$:

$$\begin{aligned} A \in N_{ij} \quad &\Leftrightarrow \quad A \vdash^* a_i \dots a_j \\ &\Leftrightarrow \quad \text{ex. } A \to BC \text{ in } P, \text{ ex. } k (i \leq k < j) \\ &\qquad \text{mit } B \vdash^* a_i \dots a_k, C \vdash^* a_{k+1} \dots a_j, \\ &\qquad \text{d.h. mit } B \in N_{ik}, C \in N_{k+1,j}. \end{aligned}$$

Beachte $k - i$, $j - (k+1)$ sind $< j - i$. Also stehen $N_{ik}, N_{k+1,j}$ bei der Bestimmung von N_{ij} zur Verfügung.

Es ergibt sich der CYK-Algorithmus:
$\{$Gegeben $G = (N, \Sigma, P, S)$ in CNF, $w = a_1 \dots a_n \in \Sigma^*\}$

für $i = 1$ bis n setze $N_{ii} := \{A | A \to a_i \in P\}$
für $d = 1$ bis $n - 1$
    für $i = 1$ bis $n - d$ $\{$baue $N_{i,i+d}$ auf$\}$
        $j := i + d$
        $N_{ij} := \emptyset$
        für $k = i$ bis $j - 1$
            $N_{ij} := N_{ij} \cup \{A | \exists A \to BC \text{ in } P : B \in N_{ik}, C \in N_{k+1,j}\}$
$\{N_{ij} = \{A | A \vdash^* a_i \dots a_j\}\}$

□

Zeitaufwand:
für innerste Schleife über k: $\leq n \cdot const$
für Schleife über i: $\leq (n - d) \cdot n \cdot const$
für Schleife über d: $\leq \sum_{d=1}^{n-1} (n - d) \cdot n \cdot const \leq n \cdot n \cdot n \cdot const$
Initalisierung: $\leq n \cdot const'$
insgesamt $O(n^3)$

Beispiel: G gegeben durch $S \to SA | a, A \to BS, B \to BB | BS | b | c$, $w = abaaba$ (notiert unter der Diagonalen im folgenden Diagramm).

i/j	1	2	3	4	5	6
1	S	$\emptyset$	S	S	$\emptyset$	S
2	a	B	A, B	A, B	B	A, B
3		b	S	$\emptyset$	$\emptyset$	$\emptyset$
4			a	S	$\emptyset$	S
5				a	B	A, B
6					b	S
						a

Es ist $S \in N_{16}$, also gilt $S \vdash w$.

Satz: *(zur Entscheidbarkeit des Leerheitsproblems) Es gibt einen Algorithmus, der zu gegebener kontextfreier Grammatik G jeweils entscheidet, ob $L(G) = \emptyset$.*

Beweis: Nur Idee: Berechnung der Menge $T \subseteq N$ der terminierenden Variablen: $T = \{A \in N | ex.w \in \Sigma^* \text{ mit } A \vdash^* w\}$. Dann $L(G) \neq \emptyset$ gdw. $S \in T$.
Folgender **Markierungsalgorithmus** bestimmt T:

(0) Markiere (z.B. durch Unterstreichen) auf den rechten Regelseiten alle Terminalsymbole.

(1) Solange Regel mit unmarkierter linker Seite A existiert, so daß eine rechte Seite voll markiert ist, markiere A überall in der Grammatik.

$\{T =$Menge der markierten Variablen$\}$

Korrektheit:

- Termination des Algorithmus: klar, da N endlich und in jeder Schleife eine Variable markiert wird.
- Es werden nur terminierende Variablen markiert (klar mit Induktion über Anzahl $n \geq 1$ von Schleifendurchläufen)
- Es werden alle terminierenden Variablen markiert.

Zeige durch Induktion über $n \geq 1$: $A \vdash^{\leq n} w$ für ein $w \Rightarrow A$ wird markiert:
$n = 1$: $A \vdash_G w : A \to w$ in G, A wird markiert.
$n + 1$: Gelte $A \vdash^{\leq n+1} w$ für ein w. Dann

$$A \vdash u_0 B_1 u_1 \ldots B_m u_m \vdash^{\leq n} u_0 v_1 u_1 \ldots v_m u_m = w$$

mit $B_i \vdash^{\leq n} v_i$. I.V. besagt: $B_1, \ldots, B_m$ werden markiert, also wird auch A markiert, da Regel $A \to u_0 B_1 u_1 \ldots B_m u_m$ in G vorhanden.

Zeitaufwand in Größe der Grammatik (Länge des Wortes aus allen Regeln, u.a. mit den Zeichen $\to$ $|;$): Bei Wortlänge n zur Darstellung der Grammatik genügt Zeitaufwand $O(n^2)$. □

Der CYK-Algorithmus und der Markierungsalgorithmus sind typische Beispiele für zwei wichtige Ansätze des Algorithmenentwurfs. Im CYK-Algorithmus werden „alle Teilprobleme nach wachsender Größe“ gelöst und Lösungen für Teilprobleme zu Lösungen für größere Teilprobleme zusammengesetzt („*dynamisches Programmieren*“), im Markierungsalgorithmus erfolgt der *induktive Aufbau* einer Menge, die unter gewissen Operationen abgeschlossen ist (hier: Operation (1)).
In der Vorlesung „Automaten und formale Sprachen“ wird gezeigt, daß das Äquivalenzproblem für kontextfreie Grammatiken nicht entscheidbar ist.

Übungen

Aufgabe (2.5): (CYK-Algorithmus) Wenden Sie den CYK-Algorithmus auf die Grammatik mit den Regeln $S \to AB|BC, A \to BA|a, B \to CC|b, C \to AB|a$ und das Wort $w = baaba$ an.

Aufgabe (2.6): (Kodierung kontextfreier Grammatiken) Geben Sie über einem geeignet gewählten (festen!) Alphabet eine Kodierung der kontextfreien Grammatiken über dem Terminalalphabet $\Sigma = \{a, b\}$ an. Die Menge der Kodierungen kontextfreier Grammatiken sollte eine reguläre Sprache bilden. Weisen Sie dies für Ihre Kodierung nach.

2.4 Kellerautomaten

Ziel: Wir suchen ein Modell von Automaten $\mathcal{A}$, die zu kontextfreien Grammatiken G in folgender Korrespondenz stehen:

$$\mathcal{A} \text{ akzeptiert } w \Leftrightarrow G \text{ erzeugt } w$$

Eine Reduktion von w durch $\mathcal{A}$ entspricht dann dem Aufbau von w durch G.

Die Idee erläutert folgendes Beispiel über dem Alphabet $\mathcal{D}_2 = \{(,),[,]\}$: Die *Dyck-Sprache* D_2 bestehe aus den korrekten Klammerwörtern über $\mathcal{D}_2$. (Der Index 2 deutet an, daß zwei Klammertypen benutzt werden.) $\mathcal{D}_2$ werde formal definiert durch die Grammatik

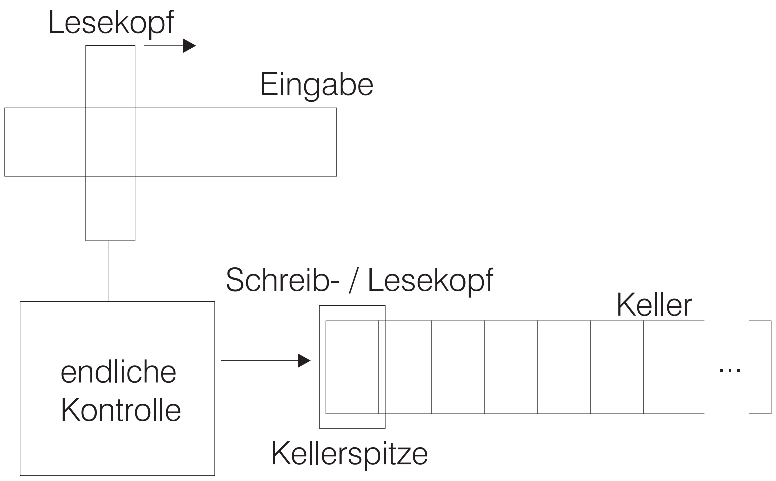
$$S \to (\,)|[\,]|(S)|[S]|SS$$

Beispielwörter: $(\,)((\,))[\,], ([(\,)((\,))[\,]])$
Beispielableitung: $S \vdash SS \vdash (\,)S \vdash (\,)SS \vdash (\,)(S)S \vdash (\,)((\,))S \vdash (\,)((\,))[\,]$

Ein entsprechendes Reduktionsverfahren geht von $(\,)((\,))[\,]$ aus und benutzt einen Hilfsspeicher für passende Satzformen, die wie das Inputwort beginnen und aus S herleitbar sind. Die Abarbeitung des Inputwortes schreitet bei Herstellung entsprechender Terminalsymbole im Hilfsspeicher fort, bis das Eingabewort verarbeitet ist und der Hilfsspeicher leer.

Restwort	Hilfsspeicher für Regelanwendungen (Kellerspeicher)
$(\,)((\,))[\,]$	S
$(\,)((\,))[\,]$	SS
$(\,)((\,))[\,]$	$(\,)S$
$((\,))[\,]$	S
$((\,))[\,]$	SS
$((\,))[\,]$	$(S)S$
$((\,))[\,]$	$((\,))S$
$[\,]$	S
$[\,]$	$[\,]$
ε	ε

Dies motiviert folgendes Automatenmodell (Kellerautomat):



Definition: Ein *Kellerautomat* (*Pushdown-Automat*, PDA) hat die Form

$$\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta)$$

mit Q endlicher Zustandsmenge, Σ Eingabealphabet, Γ *Kelleralphabet*, q_0 Anfangszustand, $Z_0 \in \Gamma$ *Kellerstartsymbol*, Transitionsrelation $\Delta \subseteq Q \times (\Sigma \cup \varepsilon)\} \times Q$.
$(q, a/\varepsilon, Z, g, q') \in \Delta$ erlaubt den Übergang von q zu q' bei Einlesen von a/ε von Eingabe und Ersatz von Z durch γ auf der Kellerspitze, Transition im Fall $\gamma = \varepsilon$ heißt *pop-Schritt*, im Fall $\gamma \neq \varepsilon$ *push-Schritt*.

Formaler: Eine Konfiguration hat die Form $(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*$ mit q Zustand, w Restinput, α Kellerinhalt. Es gelte:

$$\begin{aligned} (q, aw, Z\alpha) &\vdash_{\mathcal{A}} (q', w, \gamma\alpha), \text{ falls } (q, a, Z, \gamma, q') \in \Delta, \\ (q, w, Z\alpha) &\vdash_{\mathcal{A}} (q', w, \gamma\alpha), \text{ falls } (q, \varepsilon, Z, \gamma, q') \in \Delta \end{aligned}$$

Für Konfigurationen κ, κ' gelte $\kappa \vdash_{\mathcal{A}}^* \kappa'$, falls ex. $\kappa_1, \ldots, \kappa_n$ mit $\kappa_0 = \kappa$, $\kappa_i \vdash_{\mathcal{A}} \kappa_{i+1}$, $\kappa_n = \kappa'$.

$\mathcal{A}$ akzeptiert w *mit leerem Keller* $:\Leftrightarrow (q_0, w, Z_0) \vdash_{\mathcal{A}}^* (q, \varepsilon, \varepsilon)$ für ein $q \in Q$.
Variante: Akzeptieren *mit Endzustand* durch Konfigurationen (q, ε, α) mit Endzustand q und α beliebig.

$N(\mathcal{A}) := \{w \in \Sigma^* | \mathcal{A} \text{ akzeptiert } w \text{ mit leerem Keller}\}$

$\mathcal{A}$ heißt *deterministisch* (DPDA), falls für $(q, a/\varepsilon, Z)$ ex. jeweils höchstens eine Transition $(q, a/\varepsilon, Z, \cdot, \cdot)$ in $\mathcal{A}$, und es ex. nicht zwei Transitionen $(q, a, Z, \cdot, \cdot), (q, \varepsilon, Z, \cdot, \cdot)$.

Beispiel: Das eingangs genannte Beispiel führt auf den PDA

$$\mathcal{A}_{D_2} = (\{q_0\}, \mathcal{D}_2, \mathcal{D}_2 \cup S, q_0, S, \Delta)$$

mit folgenden Transitionen in Δ:

$$\begin{array}{ccc} (q_0, \varepsilon, S, (\,), q_0) & (q_0, (, (, \varepsilon, q_0) & (q_0, \varepsilon, S, [\,], q_0) \\ (q_0,),), \varepsilon, q_0) & (q_0, \varepsilon, S, (S), q_0) & (q_0, [, [, \varepsilon, q_0) \\ (q_0, \varepsilon, S, [S], q_0) & (q_0,],], \varepsilon, q_0) & (q_0, \varepsilon, S, SS, q_0) \end{array}$$

Es gilt $N(\mathcal{A}_{D_2}) = \mathcal{D}_2$. (Diese Gleichheit ergibt sich aus dem Hauptsatz dieses Abschnitts; der Aufbau von $\mathcal{A}_{D_2}$ ist ein Beispiel für die dort angegebene Konstruktion.)

Beispiel: $L_2 = \{a^n b^n | n > 0\}$, betrachte $\mathcal{A}_2 := (\{q_0, q_1\}, \{a, b\}, \{Z, Z_0\}, q_0, Z_0, \Delta)$ mit folgenden Transitionen in Δ:

(1) (q_0, a, Z_0, ZZ_0, q_0)

(2) (q_0, a, Z, ZZ, q_0)

(3) $(q_0, b, Z, \varepsilon, q_1)$

(4) $(q_1, b, Z, \varepsilon, q_1)$

(5) $(q_1, \varepsilon, Z_0, \varepsilon, q_1)$

Es gilt dann $N(\mathcal{A}_2) = L_2$

Beweis: „$\supseteq$" ist klar mit den Transitionen
(1), $(n-1)$ mal (2) [liefert Konfiguration $(q_0, b^n, Z^n Z_0)$]
(3), $(n-1)$ mal (4) [liefert Konfiguration (q_1, ε, Z_0)]
(5), liefert $(q_1, \varepsilon, \varepsilon)$

„$\subseteq$": Gelte $w \in N(\mathcal{A}_2)$, d.h. $(q_0, w, Z_0) \vdash^*_{\mathcal{A}_2} (q_{0,1}, \varepsilon, \varepsilon)$. Es müssen dann (in dieser Reihenfolge) die Transitionen (1), i-mal (2) mit, $i \geq 0$, (3), j-mal (4) mit, $j \geq 0$ (5) angewendet werden. Da vor Anwendung von (5) Z_0 auf Kellerspitze ist, muß $i = j$ sein, d.h. $w = a^{i+1} b^{i+1}$ und somit $w \in L_2$. □

Als Vorbereitung für den Übergang Grammatik $\rightarrow$ PDA benötigen wir einen weiteren Begriff:

Definition: Eine Ableitung $\alpha_0 \vdash_G \alpha_1 \vdash \ldots \vdash_G \alpha_n$ (G kontextfreie Grammatik) heißt *Linksableitung*, falls beim Übergang von $\alpha_i zu \alpha_{i+1}$ jeweils das am weitesten links stehende Nichtterminalsymbol in α_i ersetzt wird.

Lemma: *Gilt $S \vdash^*_G w$, dann existiert eine Linksableitung $S \vdash \ldots \vdash w$.*

Beweis: Betrachte Ableitung $S \vdash \ldots \vdash w$ und hierbei die erste Stelle, wo Linksableitung verletzt (sonst fertig). Wir isolieren diese Stelle bei $\underline{\vdash}$:

$$S \vdash^* u_0 B_1 u_1 \ldots B_m u_m \underline{\vdash} u_0 B_1 \ldots \beta_k u_k \ldots B_m u_m \vdash^* u_0 v_1 u_1 \ldots v_m u_m = w$$

mit $B_k \rightarrow \beta_k, \beta_k \vdash^* v_k, B_i \vdash^* v_i$, $(i \neq k)$. Gehe über zu Ableitung

$$S \vdash^* u_0 B_1 u_1 \ldots B_m u_m \vdash u_0 v_1 u_1 B_2 \ldots B_k u_k \ldots B_m u_m \vdash^* u_0 v_1 \ldots u_m v_m = w$$

mit $B_i \vdash^* v_i$; hier ist die Eigenschaft der Linksableitung erst später verletzt. Fährt man so fort, erhält man schließlich eine Linksableitung von w. $\Box$

Satz: *Zu jeder kontextfreien Grammatik G kann man einen PDA $\mathcal{A}$ konstruieren mit $L(G) = N(A)$.*

Beweis: $G = (N, \Sigma, P, S)$ gegeben. Konstruiere $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta)$ über der einelementigen (!) Zustandsmenge $Q = \{q_0\}$ mit $\Gamma = \Sigma \cup N$, $Z_0 = S$, Δ mit Transitionen $(q_0, \varepsilon, A, \gamma, q_0)$ für $A \to \gamma \in P$ und $(q_0, a, a, \varepsilon, q_0)$ für $a \in \Sigma$.
Notation: $S \ {}^*\!\vdash \alpha$ für Existenz einer Linksableitung.
Behauptung: $(q_0, u, S) \vdash^*_{\mathcal{A}} (q_0, \varepsilon, \alpha) \Leftrightarrow S \ {}^*\!\vdash u\alpha$ (wobei für $\Leftarrow$ vorausgesetzt sei, daß $\alpha = \varepsilon$ oder α beginnt mit Nichtterminal).
Für $\alpha = \varepsilon$ ist dann der Satz gezeigt.

Nachweis von „$\Rightarrow$“ durch Induktion über Schrittzahl $l \geq 0$ von $\mathcal{A}$:
$l = 0$: Voraussetzung liefert $u = \varepsilon$, $\alpha = S$, wegen $S \ {}^*\!\vdash S$ ist die Behauptung klar.
$l + 1$: Gelte $(q_0, u, S) \vdash^{l+1}_{\mathcal{A}} (q_0, \varepsilon, \alpha)$.
1. Fall: letzte $\mathcal{A}$-Transition ein $A \to \gamma$ Schritt. Also

$$(q, u, S) \vdash^l_{\mathcal{A}} (q_0, \varepsilon, A\alpha') \vdash (q_0, \varepsilon, g\alpha')$$

mit $\gamma\alpha' = \alpha$. I.V. liefert $S \ {}^*\!\vdash uA\alpha' \vdash u\gamma\alpha' \vdash u\alpha$.
2. Fall: letzte $\mathcal{A}$-Transition Reduktion eines Eingabebuchstaben

$$(q_0, u'a, S) \vdash^l_{\mathcal{A}} (q_0, a, a\alpha) \vdash_{\mathcal{A}} (q_0, \varepsilon, \alpha)$$

mit $u = u'a, (q_0, u', S) \vdash^l_{\mathcal{A}} (q_0, \varepsilon, a\alpha)$. I.V. liefert $S \ {}^*\!\vdash u'a\alpha = u\alpha$.

Nachweis von „$\Leftarrow$“ durch Induktion über Länge $l \geq 0$ von Linksableitungen:
$l = 0$: Voraussetzung besagt $u = \varepsilon$, $\alpha = S$. Dann gilt auch linke Seite.
$l + 1$: Gelte $S \ {}^{l+1}\!\vdash u\alpha$, im letzten Schritt etwa $A \to \gamma$. Dann $S \ {}^{l}\!\vdash vA\beta \vdash v\gamma\beta = u\alpha$ $(*)$. Beachte, daß links von A nur Terminalsymbole stehen können, also $v \in \Sigma^*$. Nach I.V. gilt:

$$(q_0, v, S) \vdash^*_{\mathcal{A}} (q_0, \varepsilon, A\beta) \vdash (q_0, \varepsilon, \gamma\beta) \qquad (**)$$

Da $\alpha = \varepsilon$ oder mit Nichtterminal beginnt, folgt nun aus $(*)$: $|u| \geq |v|$. Definiere w durch $vw = u$, also $w\alpha = \gamma\beta$, da $u\alpha = v\gamma\beta$. Aus $(**)$ folgt nun

$$(q_0, vw, S) \vdash^*_{\mathcal{A}} (q_0, w, w\alpha) = (q_0, w, \gamma\beta)$$

und nun weiter mit Reduktionsschritten (Löschen von w auf Eingabe und Keller) $\vdash^*_{\mathcal{A}} (q_0, \varepsilon, \alpha)$. $\Box$

Es gilt auch die Umkehrung des Satzes. Dies wird hier nicht gezeigt. Ansatz: Zu $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta)$ definiere Nichtterminalsymbole für G: $[pZq]$ mit $p, q \in Q$ und $Z = \Gamma$, Regeln so, daß $[pZq] \vdash^*_G w \Leftrightarrow \mathcal{A}$ kann über w von p zu q gelangen bei Löschen der Kellerspitze Z.

Rückblick auf Zusammenhang Grammatik $\leftrightarrow$ PDA:

Wir betrachten dies anhand des Beispiels von

AAP=Menge der aussagenlogischen Ausdrücke in Präfix-Notation

(hier nur mit Aussagevariablen $a1, \ldots, a9$). Das Alphabet ist

$$\Sigma = \{\wedge, \vee, \neg, a, 1, \ldots, 9, \#\},$$

wobei # zur Endmarkierung dient. Ein Beispielausdruck ist $\wedge \vee \wedge a1a2\neg a3 \wedge a2a3\#$; in Infix-Notation würde man $(((a1 \wedge a2) \vee \neg a3) \wedge (a2 \wedge a3))$ schreiben.
AAP wird durch folgende Grammatik erzeugt:

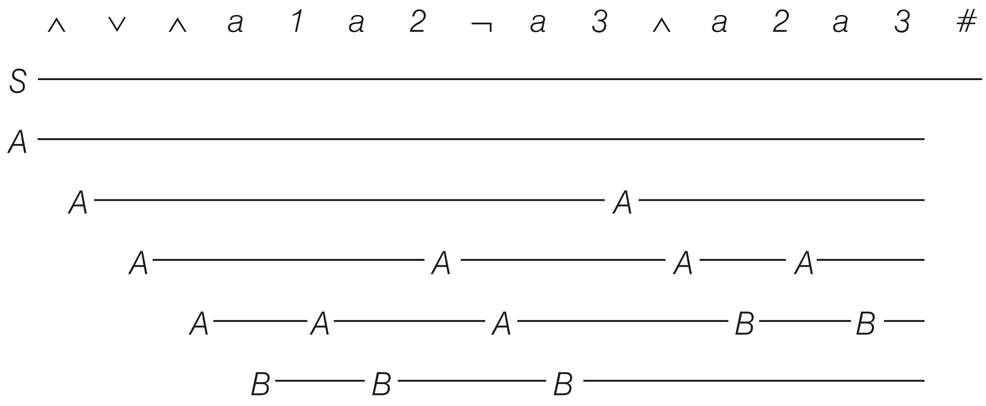
$$S \to A\#, \quad A \to aB|\neg A| \vee AA| \wedge AA, \quad B \to 1|\ldots|9$$

Die Grammatik entspricht unmittelbar einem rekursiven Parsing-Programm, welches Eingabewörter auf Mitgliedschaft in AAP testet.
Und zwar entspricht allgemein eine Regel $A \to u_0B_1u_1 \ldots B_m u_m$ einer Prozedur „A“ zum Akzeptieren der aus A ableitbaren Wörter: lese u_0 ein, führe Prozedur „B_1“ aus, lese u_1 ein, führe Prozedur „B_2“ aus,. . ., lese u_m ein und akzeptiere dann. Alternativen zwischen den A-Regeln führen zu nichtdeterministischen Programmen. Im Beispiel bestimmt das erste Symbol auf rechten Regelseiten die Regelauswahl; in einem solchen Fall ergibt sich Determinismus.

Beispiel:

program recursive parser für AAP
    (ES jeweils lokale Variable für Eingabesymbol)
procedure S
    A; read (ES); if ES=# then write accept else write reject;
procedure A
    read (ES); if ES=a then B
        else if ES=$\neg$ then A
        else if ES=$\wedge$ oder $\vee$ then A; A;
        else write reject;
procedure B
    read (ES); if ES$\in \{1 \ldots 9\}$ then no-op else write reject;
($*$ Hauptprogramm $*$)
begin S end;

Aufrufe der Prozeduren (mit Angabe der Lebensdauer) über dem Beispiel-Eingabewort $\wedge \vee \wedge a1a2\neg a3 \wedge a2a3\#$:



So wie G einem rekursiven Programm entspricht, so kann man einen zugehörigen PDA als iteratives Programm (ohne rekursive Prozeduren) auffassen, in dem die Kellerinhalte die rekursiven Aufrufe nachbilden.

```
program PDA-Parser für AAP
var Keller: Folge von Buchstaben, KS: Buchstabe (für Kellerspitze)
begin Keller:=leer; S auf Keller; read (KS); A auf Keller;
    (* Wir berücksichtigen die einmalige Regelanwendung S → A#
    getrennt vor und nach der while-Schleife *)
    while Keller nicht leer do
        read (KS);
        if KS=A then read (ES);
            if ES=a then B auf Keller
                else if ES=¬ then A auf Keller
                else if ES=∧ oder ∨ then AA auf Keller
                else reject;
        if KS=B then read (ES);
            if ES∈ {1...9} then no-op else reject;
    od;
    read (ES); if ES=# then accept else reject;
end.
```

Übungen

Aufgabe (2.7): (DPDA) Geben Sie einen deterministischen PDA (mit Erklärung) an, der die Dyck-Sprache $\mathcal{D}_2$ mit Endzuständen erkennt.

Aufgabe (2.8): (DEA vs. DPDA)

1. Zeigen Sie, daß eine reguläre Sprache $L \subseteq \Sigma^* \cdot \{\#\}$ (mit Endsymbol $\# \notin \Sigma$), die von einem DEA mit n Zuständen erkannt wird, von einem DPDA mit
   (a) n Zuständen und einem Kellersymbol
   (b) einem Zustand und n Kellersymbolen

   jeweils mit leerem Keller erkannt wird.
2. Zeigen Sie, daß 1. im allgemeinen nicht gilt, wenn $L \subseteq \Sigma^*$ vorausgesetzt wird.

Aufgabe (2.9): (Auswertung aussagenlogischer Ausdrücke) Aussagenlogische Ausdrücke mit Konstanten (in Präfix-Notation) seien definiert wie aussagenlogische Ausdrücke, jedoch mit 0 und 1 an Stelle von Aussagenvariablen. Für einen solchen Ausdruck α sei $|\alpha|$ der Wahrheitswert (0 oder 1), der sich in üblicher Weise aus α durch Auswertung ergibt. Geben Sie (mit Erläuterung) einen DPDA an, der die Sprache

$$\{\alpha | \alpha \text{ ist aussagenlogischer Ausdruck mit Konstanten}, |\alpha| = 1\}$$

erkennt.

Aufgabe (2.10): (Durchschnitt mit regulären Sprachen) Sei L eine Sprache, die von einem PDA mit Endzuständen erkannt wird und R eine reguläre Sprache. Beweisen Sie, daß $L \cap R$ durch einen PDA mit Endzuständen erkannt wird.

2.5 Ableitungsbäume und Iterationssatz

Ziel dieses Abschnitts ist der Nachweis konkreter Beispiele von Sprachen, die nicht kontextfrei sind. Dazu zeigen wir das „Pumping Lemma“, das gewisse Periodizitätseigenschaften von kontextfreien Sprachen garantiert. Als Vorbereitung führen wir Ableitungsbäume ein.

Ableitungsbäume: Als Beispiel betrachten wir kontextfreie Grammatik mit der Regelmenge $S \to SbS|a$ und die Ableitungen

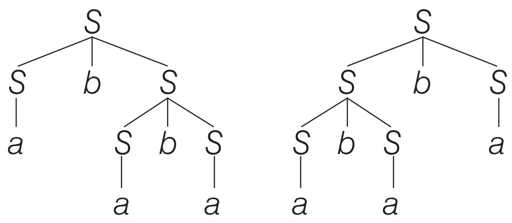
(1) $S \vdash SbS \vdash abS \vdash abSbS \vdash ababS \vdash ababa$

(2) $S \vdash SbS \vdash SbSbS \vdash SbSba \vdash Sbaba \vdash ababa$

(3) $S \vdash SbS \vdash SbSbS \vdash abSbS \vdash ababS \vdash ababa$

(4) $S \vdash SbS \vdash Sba \vdash SbSba \vdash Sbaba \vdash ababa$

(1),(2) und (3),(4) führen auf jeweils gleiche Ableitungsbäume:



Wir halten fest:

- (1) und (3) sind *Linksableitungen*, d.h. bei jedem Ableitungsschritt wird die am weitesten links stehende Variable $\in N$ ersetzt, (2) und (4) sind *Rechtsableitungen*.
- Jede Ableitung induziert einen Ableitungsbaum.
- Einem Ableitungsbaum entsprechen im allgemeinen mehrere Ableitungen.

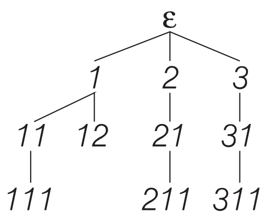
Definition:

(1) Ein unbewerteter, höchstens k-verzweigter *Baum* ist eine Teilmenge von $\{1, \ldots, k\}^*$, die

  (a) unter Präfixbildung abgeschlossen ist

  (b) jeweils für xj auch alle xi mit $1 \leq i < j$ enthält.

Beispiel: (für $k = 3$)



Sprechweisen: ε ist die *Wurzel* des Baumes, der *Knoten* xi ist *Nachfolger* von x, *Blätter* sind Knoten ohne Nachfolger, die *Front* eines Baumes ist die Folge der Blätter in lexikographischer Reihenfolge, ein *Pfad* (der Länge n) ist eine Folge $u_0, \ldots, u_n$, wobei u_0 die Wurzel des Baumes ist, u_n ist ein Blatt und u_{i+1} ist Nachfolger von u_i $(0 \leq i < n)$.

(2) Ein Σ-*bewerteter*, höchstens k-verzweigter *Baum* ist eine Abbildung

$$t : \mathrm{dom}(t) \to \Sigma,$$

wobei $\mathrm{dom}(t)$ ein unbewerteter, höchstens k-verzweigter Baum ist.

$t(x)$ ist die Beschriftung des Knotens x, $\mathrm{Yield}(t)$ ist die Folge der Beschriftungen der Knoten in der Front.

(3) Ein *Ableitungsbaum* t zur kontextfreien Grammatik $G = (N, \Sigma, P, S)$ ist ein $(N \cup \Sigma)$-bewerteter Baum mit

(i) die Wurzel ist mit S beschriftet ($t(\varepsilon) = S$)

(ii) hat x die Nachfolger $x1, \ldots, xm$, so ist $t(x) \to t(x1) \ldots t(xm)$ in P.

Bemerkung: *Sei G eine kontextfreie Grammatik, A eine Variable von G. Dann gilt: $A \vdash_G^* w \Leftrightarrow$ ex. ein Ableitungsbaum t für G, dessen Wurzel mit A beschriftet ist, (d.h. $t(\varepsilon) = A$) und für den $\mathrm{Yield}(t) = w$ gilt.*

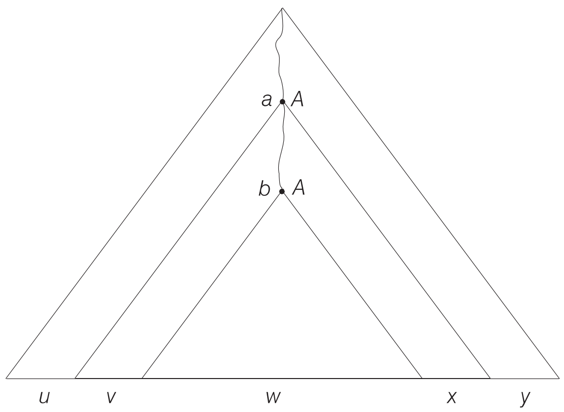
Definition: Sei $t : \mathrm{dom}(t) \to \Sigma$ ein bewerteter Baum und $x \in \mathrm{dom}(t)$. Der *Unterbaum* $t|_x$ ist gegeben durch:

- $\mathrm{dom}(t|_x) = \{y | xy \in \mathrm{dom}(t)\}$
- $(t|_x)(y) = t(xy)$

Iterationssatz: (Pumping Lemma, uvwxy-Theorem)) Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik, die keine Regel der Form $A \to B (A, B \in N)$ enthält, $|N| = m$, k die maximale Länge einer rechten Regelseite, $n := k^{m+1}$. Es gilt für alle $z \in L(G)$ mit $|z| > n$: es gibt eine Zerlegung $z = uvwxy$ mit $vx \neq \varepsilon$, $|vwx| \leq n$ und für alle $i \in \mathbb{N}_0$ gilt $uv^iwx^iy \in L(G)$.

Vorbemerkung zum Beweis: Sei t ein höchstens k-verzweigter Baum, $|\mathrm{Yield}(t)| > k^m$. Dann ex. ein Pfad der Länge $> m$ in t. Anders ausgedrückt: Haben alle Pfade in t eine Länge $\leq m$, dann ist $|\mathrm{Yield}(t)| \leq k^m$.
Der Beweis ist einfach mit Induktion über m: Für $m = 0$ ist $|\mathrm{Yield}(t)| = 1 \leq k^0$.
Für $m + 1$ gilt $|\mathrm{Yield}(t)| \leq k' \cdot k^m$ (nach I.V., mit $k' \leq k$) $\leq k \cdot k^m = k^{m+1}$.

Beweis: Sei $z \in L(G)$, $|z| > n = k^{m+1}$. Nach Vorbemerkung ex. ein Pfad der Länge $> m + 1$ im Ableitungsbaum t von z, folglich mit Wiederholung einer Variablen. Wähle eine "letzte" Wiederholung, d.h. zwei Knoten a, b mit $t(a) = t(b) = A$, so daß in $t|_a$ keine weitere Variablenwiederholung auftritt. Dann gilt $\mathrm{Yield}(t) = z = uvwxy$, wobei wir u, v, w, x, y so bestimmen, daß $\mathrm{Yield}(t|_b) = w$, $\mathrm{Yield}(t|_a) = vwx$. Dann gilt $|vwx| \leq n$ und es sind nicht v, x beide leer (keine Regeln $A \to B$ auf dem Pfad von a nach b !). Ferner $S \vdash_G^* uAy$, $A \vdash_G^* vAx$, $A \vdash_G^* w$, und es folgt $uv^iwx^iy \in L(G)$ für $i \geq 0$.



□

Folgerung: $L = \{a^ib^ic^i | i \geq 0\}$ *ist nicht kontextfrei.*

Wäre L kontextfrei, etwa erzeugt durch G kontextfrei, so könnte man n gemäß Pumping Lemma wählen und $z = a^nb^nc^n \in L$ betrachten. Nach dem Pumping Lemma existiert eine Zerlegung $z = uvwxy$ mit $vx \neq \varepsilon$, $|vwx| \leq n$ und $uv^iwx^iy \in L$ für alle $i \geq 0$. Wegen $|vwx| \leq n$ gilt $vwx \in a^*b^*$ oder $vwx \in b^*c^*$. Wir betrachten nun uwy, das (für $i = 0$) gemäß dem Pumping Lemma zu L gehört.
Im ersten Fall $vwx \in a^*b^*$ gilt wegen $vx \neq \varepsilon$, daß $|uwy|_a < n$ oder $|uwy|_b < n$ (oder beides). Andererseits $|uwy|_c = n$. Also $uwy \notin L$, Widerspruch.
Im zweiten Fall $vwx \in b^*c^*$ schließt man analog.

Übungen

Aufgabe (2.11): (Nicht-kontextfreie Sprachen) Zeigen Sie mit Hilfe des Pumping Lemmas, daß die folgenden Sprachen nicht kontextfrei sind:

1. $L_1 = \{a^i | i \text{ ist Primzahl}\}$
2. $L_2 = \{ww | w \in \{a, b\}^*\}$

Kapitel 3

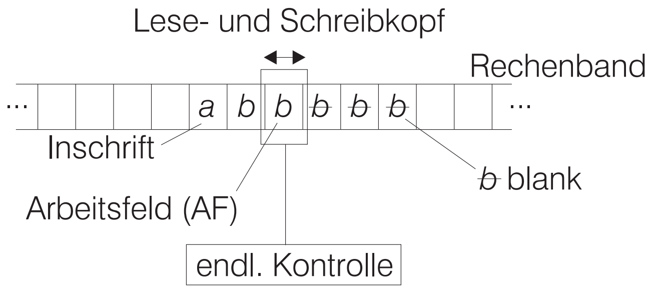
Berechenbarkeit, Komplexität

3.1 Turing-Maschinen

In einer für die Entwicklung der Informatik fundamentalen Arbeit hat Alan Turing 1936 (damals 24 Jahre alt) ein abstraktes Automatenmodell vorgeschlagen mit dem Ziel, die Durchführung beliebiger Algorithmen zur Symbolmanipulation in einem präzisen Rahmen zu erfassen. Turing analysierte hierzu die elementaren Einzelschritte, die ein Mensch bei der Realisierung eines Algorithmus (beim Rechnen auf dem Papier) durchführt. Er begründete einige wesentliche Grundannahmen: Der „Computer“ (in Turings Sinne der menschliche Rechner) hat nur eine feste endliche Anzahl interner Zustände, er arbeitet auf einem (ohne Beschränkung der Allgemeinheit) eindimensionalen Rechenpapier („Rechenband“), das in Felder geteilt ist und auf jedem Feld ein Symbol aus einem festen endlichen Zeichenvorrat aufnehmen kann. In jedem Augenblick übersieht der „Computer“ nur ein Segment gewisser fester Länge auf dem Rechenband, kann in Abhängigkeit von dessen Inschrift und seinem eigenen internen Zustand Änderungen in diesem Segment vornehmen und dann die Aufmerksamkeit auf benachbarte Felder des Rechenbandes verlegen. Auf kleinste Schritte reduziert, kann man sogar annehmen, daß das kritische Segment nur aus einem einzigen „Arbeitsfeld“ auf dem Rechenband besteht. In einem Schritt kann dann jeweils in Abhängigkeit vom internen Zustand und von der Inschrift des Arbeitsfeldes dort ein neues Symbol eingetragen werden, worauf das Arbeitsfeld dann eventuell um ein Feld nach rechts oder links verlegt wird.

Da die Rechnungen nicht an mangelndem Platz scheitern sollen, wird das Rechenband als beidseitig unendlich vorausgesetzt. Allerdings sind in jedem Augenblick nur endlich viele Felder davon beschrieben, denn anfangs ist das Band leer, abgesehen von den endlich vielen Feldern, auf denen die Eingabe steht, und in einem Schritt kann jeweils höchstens ein weiteres Feld beschrieben werden. Formal repräsentiert man die „leere Beschriftung“ eines Feldes durch ein besonderes Leerzeichen, das „blank“, welches immer zum Zeichenvorrat dazugehört. Dies führt auf das Algorithmenmodell der „Turing-Maschine“, mit folgender Struktur:

Turing-Maschine:



Drei Unterschiede zum endlichen Automaten:

- Lesen und Schreiben sind möglich
- Berechnungsrichtung $\leftrightarrow$, nicht nur $\rightarrow$
- Der Speicher (Rechenband) ist nicht beschränkt (ebenso wie bei PDA's).

Definition: Eine *Turing-Maschine* (TM) hat die Form $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \delta)$ mit endlicher Zustandsmenge Q, Eingabealphabet Σ, Arbeitsalphabet $\Gamma \supseteq \Sigma$ disjunkt zu Q, wobei für das Zeichen $\not{b}$ (*blank*) gilt $\not{b} \in \Gamma \setminus \Sigma$, Anfangszustand q_0, Transitionsfunktion $\delta : Q \times \Gamma - \rightarrow \Gamma \times \{l, r\} \times Q$ (partielle Funktion, d.h., nicht jedes Element aus $Q \times \Gamma$ besitzt einen Funktionswert).
$\delta(q, a) = (a', l/r, q')$ besagt: Im Zustand q mit a auf dem Arbeitsfeld drucke a' auf das Arbeitsfeld, bewege den Kopf ein Feld nach links/rechts und gehe auf Zustand q' über. Wir nennen dann das Quintupel $(q, a, a', l/r, q')$ auch eine *Turingzeile* von $\mathcal{A}$.
Eine *Turing-Maschinen-Konfiguration* ist ein Wort aus $\Gamma^* Q \Gamma^*$ (wobei uqv für die Bandinschrift $\ldots \not{b}\not{b}uv\not{b} \ldots$ stehe, Zustand q mit erstem Buchstaben von v auf dem Arbeitsfeld, bzw. $\not{b}$ auf dem Arbeitsfeld, falls $v = \varepsilon$).
Für $u = u_0 b, v = a v_0$ sei die *Folgekonfiguration* von uqv definiert durch:

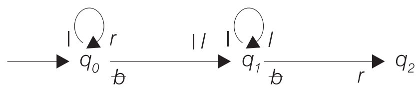
$$\begin{cases} u_0 q' b a' v_0 & \text{im Falle } \delta(q, a) = (a', l, q') \\ u_0 b a' q' v_0 & \text{im Falle } \delta(q, a) = (a', r, q') \end{cases}$$

Falls $u = \varepsilon, v = \varepsilon$, gelte dies mit $u_0 = \varepsilon, b = \not{b}$ bzw. $v_0 = \varepsilon, a = \not{b}$.
Notation: $\kappa \vdash_{\mathcal{A}} \kappa'$ für κ' Folgekonfiguration von κ,
$\kappa \vdash^*_{\mathcal{A}} \kappa'$ für ex. $n \geq 0, \kappa_0, \ldots, \kappa_n$ mit $\kappa_0 = \kappa, \kappa_n = \kappa', \kappa_i \vdash^*_{\mathcal{A}} \kappa_{i+1}$ für $i < n$.
κ heißt *Stopkonfiguration*, falls keine Folgekonfiguration von κ existiert.

Beispiel: $\mathcal{A}$ sei eine Turing-Maschine mit $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{|\}$, $\Gamma = \Sigma \cup \{\not{b}\}$, und δ sei definiert durch folgende Quintupel:

$$(q_0, |, |, r, q_0), \quad (q_0, \not{b}, |, l, q_1), \quad (q_1, |, |, l, q_1) \quad (q_1, \not{b}, \not{b}, r, q_2)$$

graphisch:



Eine Konfigurationsfolge:

$$\not{b}q_0|||\not{b}\not{b} \vdash^3 \not{b}|||q_0\not{b}\not{b} \vdash \not{b}||q_1||\not{b} \vdash^3 q_1\not{b}\not{b}||||\not{b} \vdash \not{b}q_2||||\not{b}$$

Allgemein: $q_0|^n \vdash_{\mathcal{A}}^*$ Stopkonfiguration $q_1|^{n+1}$.
Wir sagen in diesem Fall: $\mathcal{A}$ berechnet die Nachfolgerfunktion in unärer Darstellung.

Definition: Sei $f : (\Sigma^*)^n - \to \Sigma^*$, und sei $\mathcal{A}$ eine Turing-Maschine. $\mathcal{A}$ *berechnet* f, falls für alle

$$(w_1, \ldots, w_n) \in Def(f) : q_0 w_1 \not{b} w_2 \ldots w_n \not{b} \vdash_{\mathcal{A}}^* uqf(w_1, \ldots, w_n)\not{b}v$$

für Stopkonfiguration mit geeigneten u, v existiert und für alle

$$(w_1, \ldots, w_n) \in (\Sigma^*)^n \setminus Def(f)$$

$\mathcal{A}$ von $q_0 w_1 \not{b} w_2 \ldots w_n$ aus keine Stopkonfiguration erreicht.
f heißt *Turing-berechenbar*, falls eine Turing-Maschine existiert, die f berechnet.

Ein Vergleich mit endlichen Automaten:

Beispiel: Jede sequentielle Funktion $f : \Sigma^* \to \Sigma^*$ ist Turing-berechenbar.

Beweis: Ansatz: Gegeben sei eine sequentielle Maschine $\mathcal{A} = (Q, \Sigma, \Sigma, q_0, \delta, \lambda)$, $\lambda : Q \times \Sigma \to \Sigma$. Die gesuchte Turing-Maschine $\mathcal{B}$ arbeitet wie $\mathcal{A}$, überdruckt Eingabebuchstaben durch Ausgabebuchstaben und geht an den Anfang des Ausgabewortes zurück. Gilt in $\mathcal{A} : \delta_{\mathcal{A}}(q, a) = q', \lambda_{\mathcal{A}}(q, a) = a'$, so setzen wir fest $\delta_{\mathcal{B}}(q, a) = (a', r, q')$ ergänzt durch $\delta_{\mathcal{B}}(q, \not{b}) = (\not{b}, l, q_l), \delta_{\mathcal{B}}(q_l, a) = (a, l, q_l)$ für $a \in \Sigma, \delta_{\mathcal{B}}(q_l, \not{b}) = (\not{b}, r, q_s)$ mit q_s Stopzustand. Formal ist $Q_B = Q \cup \{q_l, q_s\}$. $\square$

Beispiel: Die Funktion

$$f_{prim} : \{|\}^* \to \{|\}^* \text{ mit } f_{prim}(|^n) = \begin{cases} | & \text{falls } n \text{ Primzahl} \\ \varepsilon & \text{sonst} \end{cases}$$

ist Turing-berechenbar.

Beweis: Ansatz: Wähle als Arbeitsalphabet $\Gamma = \{|, \not{b}, \$, *, \underline{|}, \underline{\not{b}}, \underline{*}\}$. Zu gegebenem $|^n$ prüfe für $i = 2, \ldots, n-1$ Teilbarkeit von n durch i (hier $n \geq 3$, Prüfung auf $n = 0, 1, 2$ erfolge vorweg). Stelle für $i = 2, \ldots, n-1$ jeweils folgende Inschriften her:

(0) unterstreiche das i-te $|$.

(1) stelle i-Block her: $\$ \ldots \$\underline{*}$ an Stelle der ersten i Zeichen $|$.

(2) Verschiebe so lange wie möglich innerhalb $|^n$ den i-Block von Beginn=Position 1 auf Beginn=Position $k \cdot i + 1$ (für $k = 1, 2, \ldots$) und teste jeweils, ob dann nach $*$ ein $\not{b}$ folgt. Dann stop, denn i würde n teilen. Falls beim Verschieben ein $\not{b}$ überdruckt wird, Übergang zum nächsten i, weiter bei (1).

Zur Tragweite des Turing-Maschinen-Modells:

f heißt *im intuitiven Sinne* berechenbar, falls es einen (eventuell umgangssprachlich beschriebenen) Algorithmus gibt, der f berechnet, also im Falle $f : (\Sigma^*)^n - \to \Sigma^*$ zu $w_1, \ldots, w_n \in Def(f)$ jeweils $f(w_1, \ldots, w_n)$ liefert und ansonsten nicht terminiert.

Churchsche These (oder: Church-Turing-These) 1936:
f im intuitiven Sinne berechenbar $\Leftrightarrow$ f Turing-berechenbar

$\Leftarrow$ ist klar
$\Rightarrow$ ist nicht beweisbar, sondern nur begründbar

Argumente dafür:

1. Turings Analyse des Rechenprozesses
2. Erfahrung mit komplexen Beispielen seit über 50 Jahren
3. andere Präzisierungen von „berechenbar" sind formal äquivalent zu Turing-Maschinen.
4. erweiterte Turing-Maschinen-Modelle sind reduzierbar auf Turing-Maschinen im eigentlichen Sinne.

Beispiel: Zweidimensionales Rechenpapier reduzierbar auf eindimensionales Band.
Illustration: Multiplikation von Dezimalzahlen entspricht

$$f : (\{0, \ldots, 9\}^*)^2 \to \{0, \ldots, 9\}^*.$$

Die Multiplikation

$$\begin{array}{r}\underline{10925 \cdot 85} \\ 87400 \\ \underline{54625} \\ 928625\end{array}$$

ist auch auf dem Turingband durchführbar. Bandinschriften einer entsprechenden Turing-Maschine:

```
     10925b85b
87400b10925b85b
    b10925b85b87400b
54625b10925b85b87400b
    b10925b85b874000b54625
       928625b874000b54625
```

mit zusätzlichen Marken (etwa Unterstreichungen, repräsentiert durch entsprechende Hilfsbuchstaben im Arbeitsalphabet).

Unwesentlicher Gebrauch der Churchschen These:
Nachweis, daß eine Funktion Turing-berechenbar ist, durch Angabe eines intuitiven Algorithmus (Churchsche These ist durch Fleiß, nämlich durch Konstruktion einer Turing-Maschine, letztlich eliminierbar).

Wesentlicher Gebrauch:
Nachweis, daß eine Funktion nicht berechenbar (im intuitiven Sinne) ist, durch Nachweis, daß sie nicht Turing-berechenbar ist.

Übungen

Aufgabe (3.1): (Turing-Maschinen und Dezimalzahlen) Geben Sie eine Turing-Maschine an, die zu jeder Dezimalzahl (Folge von Dezimalziffern) die nächste Dezimalzahl ausrechnet und danach anhält.

Aufgabe (3.2): (linksbeschränkte Turing-Maschinen) Zeigen Sie, daß es zu jeder Turing-Maschine $\mathcal{A}$ eine äquivalente Turing-Maschine $\mathcal{A}'$ (d.h. eine Turing-Maschine, die dieselbe Sprache akzeptiert) gibt, die das Rechenband links von der Anfangsbandinschrift nicht bearbeitet (Modell einer Turing-Maschine mit nach links begrenztem Rechenband).

Aufgabe (3.3): (Invertierung von Wörtern) Zu $w = a_1 \dots a_n$ sei w^r das gespiegelte Wort $a_n \dots a_1$. Sei $\Sigma = \{a, b\}$. Zeigen Sie, daß die Funktion $f : \Sigma^* \to \Sigma^*$ mit $f(w) = w^r$ Turing-berechenbar ist.

Aufgabe (3.4): (Zwei-Keller-Automaten) Ein Zwei-Keller-Automat geht aus einem PDA durch Hinzufügen eines weiteren Kellers hervor; das Kelleralphabet und das Kellerstartsymbol werde für beide Keller benutzt. Eine Transition hat also die Form: $(q, a/\varepsilon, Y, Z, \lambda, \delta, q')$; sie besagt, daß der Automat im Zustand q nach dem Lesen von a oder ε in den Zustand q' übergeht und die beiden oberen Kellerbandinschriften Y und Z durch λ bzw. δ ersetzt.
Das Akzeptieren von Wörtern wird wie bei PDA's definiert: Ein Wort w wird genau dann akzeptiert, wenn $(q_0, w, Z_0, Z_0) \vdash^* (q, \varepsilon, \varepsilon, \varepsilon)$ für ein $q \in Q$ gilt.
Zeigen Sie: Wird eine Sprache L von einer Turing-Maschine erkannt, so auch von einem Zwei-Keller-Automaten.

3.2 Entscheidbarkeit, Aufzählbarkeit, Berechenbarkeit

Entscheidbarkeit von Sprachen:

Definition: Sei $\Sigma = \{a_1, \dots, a_n\}$. Zu $L \subseteq \Sigma^*$ definiere $\chi_L : \Sigma^* \to \Sigma^*$ durch die *charakteristische Funktion* von L

$$\chi_L(w) = \begin{cases} a_1 & w \in L \\ \varepsilon & w \notin L \end{cases}$$

Beispiel: Zu $L = \{|^n \mid n \text{ Primzahl}\}$ ist $\chi_L = f_{prim}$ (s.o.).

Definition: Die Turing-Maschine $\mathcal{A}$ *entscheidet* L, falls $\mathcal{A}$ *berechnet* χ_L (d.h., $\mathcal{A}$ stoppt für jede Eingabe $w \in \Sigma^*$, und zwar mit a_1 auf dem Arbeitsfeld, falls $w \in L$, und mit $\not{b}$ auf dem Arbeitsfeld sonst).

L ist *Turing-entscheidbar*, falls eine Turing-Maschine existiert, die L entscheidet. Ein *Entscheidungsalgorithmus im intuitiven Sinne* für $L \subseteq \Sigma^*$ stoppt für jede Eingabe $w \in \Sigma^*$, und zwar mit Ausgabe *ja* für $w \in L$, *nein* sonst. L ist *entscheidbar im intuitiven Sinne*, wenn es einen Entscheidungsalgorithmus im intuitiven Sinne für L gibt. Einige derartige Algorithmen haben wir schon kennengelernt, so den CYK-Algorithmus oder den Markierungsalgorithmus zum Leerheitstest für kontextfreie Grammatiken.

Beispiel:

1. $\{|^n \mid n \text{ prim}\}$ ist (Turing-)entscheidbar.
2. Sei $G = (N, \Sigma, P, S)$ eine Typ-0-Grammatik.

   $L_1^G = \{\alpha \vdash \beta | \alpha, \beta \in (N \cup \Sigma)^*, \alpha \vdash_G \beta\}$, wobei $\vdash \notin N \cup G$, ist (Turing-)entscheidbar

   $L_2^G = \{\alpha_0 \vdash \ldots \vdash \alpha_n | n \geq 0, \alpha_i \in (N \cup \Sigma)^*, \alpha_i \vdash_G \alpha_{i+1}, i < n\}$, die Sprache der Ableitungen zu G, ist (Turing-)entscheidbar

   Zum Test $w \in L_1^G$ prüfe, ob genau ein $\vdash$ in w auftritt, prüfe dann alle Infixe des Präfixes vor $\vdash$ $(= \alpha)$ auf Anwendbarkeit einer G-Regel und in diesem Falle, ob Suffix nach $\vdash$ $(= \beta)$ aus α durch diese Regel hervorgeht.

   L_2^G-Test analog durch sukzessives Anwenden als L_1^G-Tests.

   [Hier ist die Churchsche These unwesentlich; man könnte auch direkt Turing-Maschinen angeben]

Eine kleine Modifikation macht aus Entscheidungsverfahren die sogenannten Akzeptieralgorithmen oder *Semi-Entscheidungsverfahren*: Sie terminieren genau für die Eingaben aus L (und stoppen sonst nicht). Auf formaler Ebene entspricht dies folgender Definition:

Definition: Eine Turing-Maschine $\mathcal{A}$ *akzeptiert* L $:\Leftrightarrow$ $\mathcal{A}$ stoppt genau für diejenigen Eingabewörter $w \in \Sigma^*$, die zu L gehören. L *Turing-akzeptierbar* $\Leftrightarrow$ ex. Turing-Maschine, die L akzeptiert.

Bemerkung: *L Turing-entscheidbar $\Rightarrow$ L Turing-akzeptierbar.*

Beweis: $\mathcal{A}$ entscheide L. Führe neuen Zustand q_∞ ein und ergänze für alle Paare $(q, \not{b})$, für die eine Befehlszeile in $\mathcal{A}$ fehlt, durch $q\not{b}\not{b}rq_\infty$, sowie für jedes a aus dem Arbeitsalphabet die Zeile $q_\infty aarq_\infty$. Erhalte so $\mathcal{B}$, $\mathcal{B}$ akzeptiert dann L. $\square$

Aufzählbarkeit:

Definition: Eine Turing-Maschine $\mathcal{A}$ heißt *Aufzählungs-Turing-Maschine*, falls ihre Zustandsmenge einen ausgezeichneten Zustand $q!$ enthält; und die durch $\mathcal{A}$ aufgezählte Sprache L enthalte alle Wörter $w \in \Sigma^*$ mit $\not{b}q_0\not{b} \vdash_{\mathcal{A}}^* uq!w\not{b}v$. ($\mathcal{A}$ liefert die Wörter, die jeweils bei Erreichen von $q!$ vom Arbeitsfeld bis zum nächsten $\not{b}$ stehen.)
L heißt *Turing-aufzählbar* $\Leftrightarrow$ ex. Turing-Maschine, die L aufzählt.

Beispiel:

1. $\emptyset$ ist Turing-aufzählbar: klar durch Turing-Maschine, die $q!$ nie erreicht.

2. Σ^* ist Turing-aufzählbar, mit $\Sigma = \{a_1, \ldots, a_n\}$ Benutze die *kanonische Reihenfolge* der Wörter über Σ: $\varepsilon, a_1, \ldots, a_n, a_1a_1, a_1a_2, \ldots, a_1a_n, \ldots, a_na_n, a_1a_1a_1, \ldots$ Wörter sind sukzessiv durch eine Turing-Maschine herstellbar (analog zur Herstellung der Dezimalzahlen).

3. Sei $G = (N, \Sigma, P, S)$ eine Typ-0-Grammatik. Dann ist $L(G)$ Turing-aufzählbar.

   Dies ergibt sich so: Sei $G = (N, \Sigma, P, S)$. Gesuchte Aufzählungs-Turing-Maschine $\mathcal{A}$ arbeitet wie folgt: $\mathcal{A}$ stellt über $N \cup \Sigma \cup \{\vdash\}$ alle Wörter in kanonischer Reihenfolge her, testet jeweils (gemäß Turing-Maschine zu L_2^G, s.o.), ob Ableitung vorliegt, in diesem Fall, ob vor erstem $\vdash$ nur S und nach letztem $\vdash$ ein Terminalwort steht. In diesem Fall geht sie auf dessen ersten Buchstaben in $q!$, bevor sie fortfährt..

Ein *Aufzählungsalgorithmus* im intuitiven Sinne liefert ohne Eingabe die Wörter einer Sprache sukzessiv, in irgendeiner Reihenfolge, evtl. mit Wiederholungen.

Wir formulieren und zeigen nun fundamentale Zusammenhänge zwischen diesen Begriffen. Die Beweise führen wir auf intuitiver Ebene (die Übertragung auf Turing-Maschinen ist technisch etwas mühsamer).

Satz: (über Entscheidbarkeit, Aufzählbarkeit, Berechenbarkeit)

(a) *L (Turing-) entscheidbar $\Rightarrow$ L (Turing-) akzeptierbar*

(b) *L (Turing-) akzeptierbar $\Leftrightarrow$ L (Turing-) aufzählbar*

(c) *$L \subseteq \Sigma^*$ (Turing-) entscheidbar $\Leftrightarrow$ L (Turing-) aufzählbar und $\Sigma^* \setminus L$ (Turing-) aufzählbar*

(d) *L (Turing-) aufzählbar $\Leftrightarrow$ L Definitionsbereich einer (Turing-) berechenbaren Funktion*

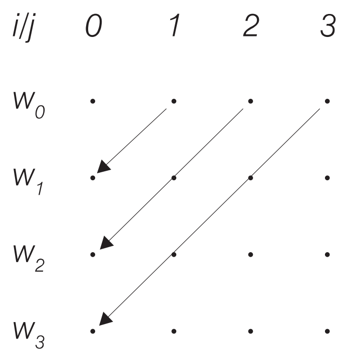
(e) *Zu $f : \Sigma^* \to \Sigma^*$ sei $G_f = \{u\#v | f(u) = v\}$ der „Graph von f“, ($\# \notin \Sigma$). G_f ist (Turing-) aufzählbar $\Leftrightarrow$ f (Turing-) berechenbar*

Beweis:

(a) schon mit obiger Bemerkung gezeigt

(b) „$\Rightarrow$“ A zähle L auf, oBdA ohne Termination. Folgender Algorithmus B stoppt dann genau für die Eingaben aus L: Zur Eingabe w starte A und prüfe dessen Ausgaben jeweils auf Gleichheit mit w. Ist dies irgendwann der Fall, terminiere B.

„$\Rightarrow$“ A akzeptiere $L \subseteq \Sigma^*$. Der gesuchte Aufzählungsalgorithmus B benutzt die Liste $w_0, w_1, w_2, \ldots$ der Wörter über Σ^* in kanonischer Reihenfolge. B bearbeite jeweils w_i j Schritte lang gemäß A und gebe w_i aus, falls A dabei w_i akzeptiert. Die Kombinationen i und j werden nach dem folgendem Diagonalschema (*Dovetailing*) abgearbeitet:



A akzeptiert (etwa w_i) $\Leftrightarrow$ ex. j: A stoppt auf w_i in j Schritten $\Leftrightarrow$ B gibt w_i aus.

(c) „$\Rightarrow$“ Aus L entscheidbar folgt L akzeptierbar wegen (a). Auch $\Sigma^* \setminus L$ ist entscheidbar (im Entscheidungsalgorithmus für L vertausche *ja*und nein). Wegen (a) ist auch $\Sigma^* \setminus L$ akzeptierbar. Wegen (b) sind wir fertig.

„$\Rightarrow$“ Gemäß (b) nehmen wir an, daß A_1, A_2 Akzeptieralgorithmen zu L, $\Sigma^* \setminus L$ seien. Gesuchter Entscheidungsalgorithmus läßt zu Eingabe w jeweils A_1, A_2 abwechselnd einen weiteren Schritt (ebenfalls zu Eingabe w) machen, bis A_1 oder A_2 stoppt. (Dies gilt irgendwann, da $w \in L$ oder $w \in (\Sigma^* \setminus L)$.) Stoppt A_1, ist die Ausgabe ja, stoppt A_2, ist die Ausgabe nein.

(d) „$\Rightarrow$“ A akzeptiere $L \subseteq \Sigma^*$. Sei f die Funktion, die A berechnet. Definitionsbereich dieser Funktion ist L.

„$\Rightarrow$“ A berechne die Funktion $f : \Sigma^* - \to \Sigma^*$, A akzeptiert dann $\mathrm{Def}(f)$. Wegen (b) ist $\mathrm{Def}(f)$ aufzählbar.

(e) vergleiche die Übungen.

□

Übungen

Aufgabe (3.5): (monotone Aufzählbarkeit) Eine Sprache L heiße monoton aufzählbar, wenn ihre Wörter durch einen Aufzählungsalgorithmus in kanonischer Reihenfolge erzeugt werden.
Zeigen Sie (auf intuitiver Ebene): Ist L monoton aufzählbar, so ist L entscheidbar.
Hinweis: Beachten Sie auch den Fall, daß die betrachtete Sprache endlich ist.

Aufgabe (3.6): (Berechenbarkeit und Aufzählbarkeit I) Der Graph einer einstelligen (partiellen) Funktion $f : \Sigma^* - \to \Sigma^*$ sei die Relation $G_f = \{(x, y) | f(x) = y\}$.
Zeigen Sie auf intuitiver Ebene: f ist berechenbar $\Leftrightarrow$ G_f ist aufzählbar.

Aufgabe (3.7): (Berechenbarkeit und Aufzählbarkeit II) Zeigen Sie: Eine nichtleere Sprache $L \subseteq \Sigma^*$ ist aufzählbar genau dann, wenn sie das Bild einer totalen berechenbaren Funktion $f : \Sigma^* \to \Sigma^*$ ist.

Aufgabe (3.8): (kontextsensitive Sprachen) Zeigen Sie, daß das Wortproblem für kontextsensitive Grammatiken entscheidbar ist.

3.3 Unentscheidbarkeit

Unser Ziel ist der Nachweis, daß es unentscheidbare Probleme gibt. Es wird sich um Probleme über Turing-Maschinen handeln. Wir fixieren dazu $\Sigma = \{a, b\}$ als Eingabealphabet von Turing-Maschinen. Zur Vorbereitung stellen wir jede Turing-Maschine über Σ durch ein Wort über Σ dar.

Zum Übergang $\mathcal{A} \mapsto \mathrm{code}(\mathcal{A}) \in \Sigma^*$: Ist das Arbeitsalphabet $\Gamma = \{a_1, a_2, a_3, \ldots, a_k\}$, die Zustandsmenge $Q = \{q_1, \ldots, q_n\}$, so werde die Turingzeile $Z = (q_i, a_j, a_{j'}, l/r, q_{i'})$ kodiert durch $\mathrm{code}(Z) = a^i b a^j b a^{j'} b (a/aa) b a^{i'} bb$. Die Turing-Maschine $\mathcal{A}$ mit den Zeilen $Z_1, \ldots, Z_l$ werde kodiert durch $\mathrm{code}(\mathcal{A}) = \mathrm{code}(Z_1) \ldots \mathrm{code}(Z_l) b$.

Bemerkung:

(a) In einem Wort $\mathrm{code}(\mathcal{A})w$ *beginnt* w *nach dem ersten Block aus 3 b's.*

(b) $C = \{\mathrm{code}(\mathcal{A}) | \mathcal{A}$ *Turing-Maschine über*$\Sigma\}$ *ist Turing-entscheidbar.*

Es sei

$$\mathrm{Univ} := \{\mathrm{code}(\mathcal{A})w \in a, b^* | \mathcal{A} \text{ akzeptiert } w\}$$

die *universelle Sprache*, die für alle Turing-Maschinen die zugehörigen Wortprobleme kodiert.

Satz: (von Turing)

(a) Univ *ist nicht Turing-entscheidbar.*

(b) Univ *ist Turing-akzeptierbar.*

Beweis:

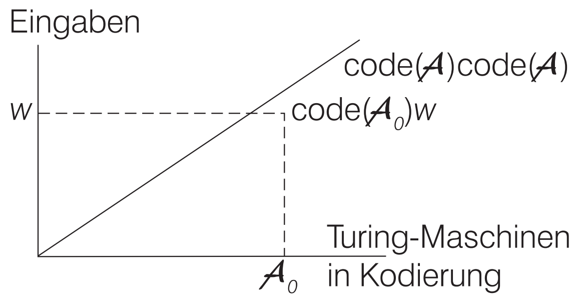
(a) Annahme: Die Turing-Maschine $\mathcal{A}_0$ entscheide Univ. Vertauschung der Ausgaben ja/nein liefert $\mathcal{A}'_0$, die $\Sigma^* \setminus$ Univ entscheidet. Betrachte die Sprache $D = \{\text{code}(\mathcal{A}) | \text{code}(\mathcal{A})\text{code}(\mathcal{A}) \notin \text{Univ}\}$. Folgende Turing-Maschine $\mathcal{A}_D$ akzeptiert dann D:

1. Gemäß der Bemerkung (a) Test der Eingabe auf Form code($\mathcal{A}$). Wenn nein, dann keine Termination, sonst:
2. Kopie von code($\mathcal{A}$) zu code($\mathcal{A}$)code($\mathcal{A}$).
3. Aufruf von $\mathcal{A}'_0$, Stop bei ja, Nichttermination bei nein.

Für alle $\mathcal{A}$ gilt dann: $\mathcal{A}_D$ akzeptiert code($\mathcal{A}$) $\Leftrightarrow$ code($\mathcal{A}$)code($\mathcal{A}$) $\notin$ Univ $\Leftrightarrow$ $\mathcal{A}$ akzeptiert code($\mathcal{A}$) nicht.

Für $\mathcal{A} = \mathcal{A}_D$ folgt ein Widerspruch, also ist (a) gezeigt.

Die Beweismethode heißt *Diagonalschluß*:



Konstruktion einer Menge (hier D) benutzt „Abänderung der Diagonalen“. Erstmals angewandt durch Cantor: Zu jeder angenommenen Abzählung der reellen Zahlen in $[0, 1]$ gibt es eine reelle Zahl, die fehlt.

Intuitive Formulierung von „Univ nicht Turing-entscheidbar“: Es gibt keinen Algorithmus, der zu einer Turing-Maschine $\mathcal{A}$ und einem Eingabewort w entscheidet, ob $\mathcal{A}$ das Wort w akzeptiert. *Das allgemeine Wortproblem für Turing-Maschinen ist also unentscheidbar.*

(b) Univ $= \{\text{code}(\mathcal{A})w | \mathcal{A} \text{ akzeptiert } w\}$ Turing-akzeptierbar. Wir geben hier die grobe Idee eines Algorithmus A, der Univ akzeptiert.

Algorithmus A:

1. Test, ob die Eingabe die Form code($\mathcal{A}$)w hat. Bei *nein* Nichttermination, sonst:
2. Simulation von $\mathcal{A}$ (in code($\mathcal{A}$) kodiert) auf Eingabe w. Termination, falls $\mathcal{A}$ schließlich stoppt, sonst keine Termination.

Bei Formulierung von A durch eine Turing-Maschine (technisch aufwendig) spricht man von einer *universellen Turing-Maschine.* Eine Universelle Turing-Maschine repräsentiert einen „Turing-Interpreter für Turing-Maschinen“ auf gegebenen Eingabeworten.

Bevor wir weitere unentscheidbare Probleme angeben, brauchen wir eine präzisere Festlegung von „Entscheidungsproblem“:

Definition: Ein *Entscheidungsproblem* ist gegeben durch ein Paar $\underline{P} = (E_P, P)$ mit einer Menge E_P von effektiv gegebenen Eingaben (repräsentierbar durch endliche Wörter) und $P \subseteq E_P$ (die Menge der Eingaben, die die Antwort *ja* verlangen).
$\underline{P}$ ist *entscheidbar*, falls ein Algorithmus existiert, der zu jeder Eingabe $x \in E_P$ terminiert mit der Antwort, ob $x \in P$ gilt oder nicht.

Beispiel: *Allgemeines Wortproblem* für Turing-Maschinen:

1. Darstellung: $\underline{W} = (E_W, W)$ mit E_W =Menge der Paare (Turing-Maschine $\mathcal{A}$ über $\{a, b\}$, Eingabewort w), W =Menge der Paare (Turing-Maschine $\mathcal{A}$ über$\{a, b\}$, Eingabewort w), so daß die Turing-Maschine $\mathcal{A}$ w akzeptiert.

2. Darstellung: E_W=Menge der Paare $(\text{code}(\mathcal{A}), w)$

   W=Menge der Paare $(\text{code}(\mathcal{A}), w)$ mit $\mathcal{A}$ akzeptiert w.

Beispiel: *Halteproblem* für Turing-Maschinen: $\underline{H} = (E_H, H)$ mit

$$\begin{aligned} E_H &= \{\mathcal{A} | \mathcal{A} \text{ ist Turing-Maschine über } \Sigma\} \\ H &= \{\mathcal{A} | \mathcal{A} \text{ ist Turing-Maschine über } \Sigma \text{, die, gestartet auf dem leeren Band, hält}\} \end{aligned}$$

Äquivalenzproblem für Turing-Maschinen: $\underline{\ddot{A}} = (E_{\ddot{A}}, \ddot{A})$ mit

$$\begin{aligned} E_{\ddot{A}} &= \{(\mathcal{A}, \mathcal{B}) | \mathcal{A}, \mathcal{B} \text{ sind Turing-Maschinen über } \Sigma\} \\ \ddot{A} &= \{(\mathcal{A}, \mathcal{B}) | \mathcal{A}, \mathcal{B} \text{ sind Turing-Maschinen über } \Sigma, L(\mathcal{A}) = L(\mathcal{B})\} \end{aligned}$$

Die Übertragung der Unentscheidbarkeit von einem Problem auf ein anderes beruht auf folgender Definition:

Definition: Seien $\underline{P} = (E_P, P)$ und $\underline{Q} = (E_Q, Q)$ Entscheidungsprobleme. $\underline{P}$ heißt *reduzierbar* auf $\underline{Q}$, formal: $\underline{P} \leq \underline{Q}$:⇔ es ex. eine totale berechenbare Funktion $f : E_P \to E_Q$ mit $\forall x \in E_P : x \in P \Leftrightarrow f(x) \in Q$.

Intuitiv bedeutet $\underline{P} \leq \underline{Q}$: $\underline{Q}$ ist mindestens so schwer wie $\underline{P}$.

Dies spiegelt sich im folgenden Lemma wieder:

Reduktionslemma: Seien $\underline{P}$ und $\underline{Q}$ Entscheidungsprobleme. Ist $\underline{P}$ unentscheidbar und $\underline{P} \leq \underline{Q}$, dann ist $\underline{Q}$ unentscheidbar.

Beweis: Gelte $\underline{P} \leq \underline{Q}$. Zeige: $\underline{Q}$ entscheidbar $\Rightarrow$ $\underline{P}$ entscheidbar.
Der Algorithmus A entscheide $\underline{Q}$. Wegen $\underline{P} \leq \underline{Q}$ gibt es eine totale berechenbare Funktion f mit $x \in P \Leftrightarrow f(x) \in Q$. A_f sei ein Algorithmus, der f berechnet. Dann entscheidet der folgende Algorithmus A' das Problem $\underline{P}$:

1. Zu $x \in E_P$ berechne mit A_f den Wert $f(x) \in E_Q$.
2. Wende A auf $f(x)$ an und gebe Antwort aus.

Dies ist korrekt wegen $x \in P \Leftrightarrow f(x) \in Q$. Also ist $\underline{P}$ entscheidbar. $\square$

Anwendung des Reduktionslemmas:

Satz: *Das Halteproblem und das Äquivalenzproblem für Turing-Maschinen sind unentscheidbar.*

Beweis: Es genügt zu zeigen (wegen des Reduktionslemmas), daß $\underline{W} \leq \underline{H}$ und $\underline{H} \leq \underline{\ddot{A}}$.

(1) $\underline{W} \leq \underline{H}$: Gesucht ist eine Funktion $f : E_W \to E_H$ total, berechenbar mit $(\mathcal{A}, w) \in W \Leftrightarrow f(\mathcal{A}, w) \in H$, d.h. $\mathcal{A}$ akzeptiert $w \Leftrightarrow f(\mathcal{A}, w)$ hält beim Start mit ε.

Konstruiere die Turing-Maschine $f(\mathcal{A}, w)$ für beliebige $(\mathcal{A}, w)$ wie folgt:

$f(\mathcal{A}, w)$ schreibt auf das leere Band w, und arbeitet dann wie $\mathcal{A}$, d.h. stoppt genau dann, wenn $\mathcal{A}$ stoppt. Mit dieser Konstruktionvorschrift ist f total und berechenbar mit

$$\begin{aligned}(\mathcal{A}, w) \in W \quad &\Leftrightarrow \quad \mathcal{A} \text{ stoppt auf } w (\text{akzeptiert } w)\\ &\Leftrightarrow \quad f(\mathcal{A}, w) \text{ stoppt auf leerem Band}\\ &\Leftrightarrow \quad f(\mathcal{A}, w) \in H.\end{aligned}$$

(2) $\underline{H} \leq \underline{\ddot{A}}$: Gesucht ist eine Funktion $f : E_H \to E_{\ddot{A}}$ total, berechenbar mit $x \in H \Leftrightarrow f(x) \in \ddot{A}$. Zu $\mathcal{A}$ ist also ein entsprechendes Paar $(\mathcal{B}_\mathcal{A}, \mathcal{C}_\mathcal{A})$ anzugeben. Konstruiere $\mathcal{B}_\mathcal{A}$ als Turing-Maschine, die

(a) ihren Input löscht, dann

(b) arbeitet wie $\mathcal{A}$, d.h., wenn $\mathcal{A}$ stoppt auf leerem Band, dann akzeptiert $\mathcal{B}_\mathcal{A}$ ihren Input.

Konstruiere $\mathcal{C}_\mathcal{A}$ durch die Vorschrift: $\mathcal{C}_\mathcal{A}$ stoppt bei jedem Input. Mit dieser Konstruktionsvorschrift ist $f : \mathcal{A} \mapsto (\mathcal{B}_\mathcal{A}, \mathcal{C}_\mathcal{A})$ total und berechenbar. Es gilt: $\mathcal{A}$ stoppt auf dem leeren Band $\Leftrightarrow \mathcal{B}_\mathcal{A}$ akzeptiert jeden Input $\Leftrightarrow \mathcal{B}_\mathcal{A}$ und $\mathcal{C}_\mathcal{A}$ sind äquivalent, d.h. $\underline{H} \leq \underline{\ddot{A}}$.

$\square$

Zum Schluß zeigen wir, daß diese Unentscheidbarkeitsresultate Beispiele eines allgemeinen Satzes sind: Alle Eigenschaften von Turing-Maschine sind unentscheidbar, sofern sie nur das Eingabe-Ausgabe-Verhalten betreffen („semantisch“ sind) und nicht-trivial sind.

Definition: Eine Turing-Maschinen-Eigenschaft E heißt *semantisch*, falls für beliebige Turing-Maschinen $\mathcal{A}, \mathcal{B}$ gilt: berechnen $\mathcal{A}$ und $\mathcal{B}$ dieselbe Funktion, so hat $\mathcal{A}$ die Eigenschaft E genau dann, wenn $\mathcal{B}$ die Eigenschaft E hat.

Beispiel: „stoppt bei leerem Wort Eingabe“ ist semantisch.
„hat ≤ 10 Befehlszeilen“ ist nicht semantisch.

Definition: Eine Turing-Maschinen-Eigenschaft heißt *nicht-trivial*, falls es Turing-Maschinen mit E und Turing-Maschinen ohne E gibt.

Satz: **(von Rice)** *Jede nicht-triviale semantische Eigenschaft E von Turing-Maschinen ist unentscheidbar.*

Beweis: Sei E eine nicht-triviale semantische Eigenschaft von Turing-Maschinen. Sei $f_\perp : \Sigma^* - \to \Sigma^*$ die Funktion, die nirgends definiert ist.

1. Fall: Die Turing-Maschinen, die $f_\perp$ berechnen, haben alle nicht die Eigenschaft E. Z.z.: $\underline{H} = (E_H, H) \leq \underline{P} = (E_H, P)$, wobei E_H alle Turing-Maschinen über Σ und P diejenigen mit der Eigenschaft E enthält. Wir benötigen eine berechenbare Funktion $g : E_H \to E_H$ mit $\mathcal{A} \in H \Leftrightarrow g(\mathcal{A}) \in P$, d.h., $\mathcal{A}$ hält auf leerem Band $\Leftrightarrow$ $g(\mathcal{A})$ hat die Eigenschaft E.
Da E nicht-trivial ist, existiert eine Turing-Maschine, die die Eigenschaft E hat und somit eine Funktion $h \neq f_\perp$ berechnet. Sei $\mathcal{A}_h$ eine solche Turing-Maschine. Konstruiere $g(\mathcal{A})$ für $\mathcal{A} \in E_H$ wie folgt: Bei Eingabe w

1. arbeitet $g(\mathcal{A})$ wie $\mathcal{A}$ auf leerem Band.
2. Falls $\mathcal{A}$ auf leerem Band stoppt, arbeitet $g(\mathcal{A})$ sodann wie $\mathcal{A}_h$ auf w.

g ist total, berechenbar, und es gilt: Falls $\mathcal{A} \in H$ (d.h. $\mathcal{A}$ stoppt auf ε), dann berechnet $g(\mathcal{A})$ die Funktion h, d.h. $g(\mathcal{A}) \in P$. Falls $\mathcal{A} \notin H$ (d.h., $\mathcal{A}$ stoppt nicht auf ε), dann hält $g(\mathcal{A})$ für alle Eingaben nicht, d.h., $g(\mathcal{A})$ berechnet $f_\perp$, d.h. $g(\mathcal{A}) \notin P$. Somit $\mathcal{A} \in H \Leftrightarrow g(\mathcal{A}) \in P$.

2. Fall analog: Die Turing-Maschinen, die $f_\perp$ berechnen, haben alle die Eigenschaft E. Hier benutzen wir das gleiche g und wählen $\mathcal{A}$ so, daß die h berechnenden Turing-Maschinen, also auch ein geeignetes $\mathcal{A}_h$, die Eigenschaft E nicht haben. $\square$

Übungen

Aufgabe (3.9): (Unentscheidbarkeit) Sei T Menge der Turing-Maschinen mit dem Eingabealphabet $\{a, b\}$. Betrachte die Entscheidungsprobleme $\underline{P_1} = (T, P_1)$ und $\underline{P_2} = (T, P_2)$ mit:

- P_1 =Menge der Turing-Maschinen, die für alle Eingaben aus $\{a, b\}^*$ stoppen
- P_2 =Menge der Turing-Maschinen, die bei Eingabe a^{17} stoppen

Zeigen Sie durch geeignete Reduktionen (ohne Rückgriff auf den Satz von Rice), daß $\underline{P_1}$ und $\underline{P_2}$ unentscheidbar sind.

Aufgabe (3.10): (Diagonalschluß) Wir betrachten Turing-Maschinen über dem Eingabealphabet $\{a\}$. Diese Turing-Maschinen seien wie in der Vorlesung durch

Wörter über $\{a, b\}$ kodiert, und $\mathcal{A}_i$ sei die in der kanonischen Reihenfolge der Kodewörter i-te Turing-Maschine, welche eine totale Funktion $f : a^* \to a^*$ berechnet. Die Funktion $F : \{a\}^* \times \{a\}^* \to \{a\}^*$ sei nun definiert durch $F(a^i, a^j)$ =Ausgabe von $\mathcal{A}_i$ bei Eingabe a^j.
Zeigen Sie durch Anwendung eines Diagonalschlusses, daß F nicht Turing-berechenbar ist.

3.4 Komplexitätsklassen

In diesem Abschnitt betrachten wir entscheidbare Probleme. (Dies ist ist der Normalfall im „Alltag des Informatikers".) Der Einfachheit halber beschränken wir uns auf Probleme der Form (Σ^*, L) mit $L \subseteq \Sigma^*$. Wir wollen die entscheidbaren Probleme nach „Schwierigkeit" unterscheiden. Die Meßlatte wird definiert durch den Zeitaufwand bis zur Entscheidung, als Funktion der Länge des Eingabewortes.

Beispiel:

- T, Menge der arithmetischen Terme über $0, 1$ mit $+, *$,
- *KFNL*, die Menge der Wörter, die kontextfreie Grammatiken mit nichtleerer Sprache darstellen,
- *SAT* die Menge der erfüllbaren aussagenlogischen Ausdrücke (aufgebaut aus Variablen xD, D Dualzahl, mit $\neg, \wedge, \vee, (,))$.

Erläuterung zu *SAT*: Ein aussagenlogischer Ausdruck β heißt erfüllbar, wenn es eine Belegung seiner Variablen durch *true*, *false* gibt, so daß β mit dieser Belegung den Wert *true* ergibt: $x0 \wedge \neg x0$ ist nicht erfüllbar, $(x0 \wedge x1) \vee \neg x1$ ist erfüllbar, z.B. mit $x0 \mapsto false$, $x1 \mapsto false$.

Bemerkung: *T, KFNL und SAT sind entscheidbar*

Beweis:

- T ist entscheidbar mit dem CYK-Algorithmus in bezug auf eine T erzeugende kontextfreie Grammatik (vergleiche Abschnitt 2.3). Zeitaufwand $O(n^3)$ für Wortlänge n.
- *KFNL* ist entscheidbar mit Markierungsalgorithmus aus Abschnitt 2.3, Zeitaufwand $O(n^2)$ bei Länge n der Darstellung der Grammatik.
- *SAT* ist entscheidbar mit „Probieralgorithmus": Auswertung eines gegebenen β mit n Variablen für alle 2^n Belegungen dieser Variablen mit true und false.

□

Den algorithmischen Zeitaufwand können wir mit dem Modell der Turing-Maschine präzise (und wie sich zeigt) sogar adäquat erfassen. Wir messen den Zeitaufwand durch die Schrittzahl von Turing-Maschinen.

Definition: Für $f : \mathbb{N} \to \mathbb{N}_+$ heiße die Turing-Maschine $\mathcal{A}$ $f(n)$-*zeitbeschränkt* [$f(n)$-*platzbeschränkt*], falls für alle w über dem Eingabealphabet stoppt $\mathcal{A}$ nach $\leq f(|w|)$ Schritten [bzw. benutzt höchstens $\leq f(|w|)$ viele Felder des Bandes].

$$\begin{array}{rcl}
\mathrm{DTIME}(f(n)) & = & \{L | L \text{ durch } f(n)\text{-zeitbeschränkte TM entscheidbar}\} \\
\mathrm{DSPACE}(f(n)) & = & \{L | L \text{ durch } f(n)\text{-platzbeschränkte TM entscheidbar}\} \\
P & := & \bigcup\limits_{p(n) \text{ Polynom von } \mathbb{N} \text{ nach } \mathbb{N}_+} \mathrm{DTIME}(p(n)), \\
 & & \text{Klasse der durch Turing-Maschinen in Polynomzeit} \\
 & & \text{entscheidbaren Sprachen.} \\
\mathrm{PSPACE} & := & \bigcup\limits_{p(n) \text{ Polynom von } \mathbb{N} \text{ nach } \mathbb{N}_+} \mathrm{DSPACE}(p(n)).
\end{array}$$

Durch Implementierung des CYK-Algorithmus und des Markierungsalgorithmus mit Turing-Maschinen erhalten wir, daß die Sprachen T und *KNFL* zu P gehören.

Arbeitshypothese von Edmonds, Cobham 1964:
Eine Sprache L ist im praktischen Sinne (oder: effizient) entscheidbar genau dann, wenn $L \in P$.

Pro-Argument 1: Der Bezug auf Turing-Maschinen ist unwesentlich: Ist RM ein anderes Standard-Rechnermodell (z.B. RAM ohne Multiplikation), so ex. ein Polynom q, so daß auf Eingaben der Länge n ein RM-Schritt durch $q(n)$ Turing-Maschinen-Schritte simuliert wird.
Somit liefert die polynomiale Laufzeitschranke $p(n)$ für RM eine ebenfalls polynomiale Schranke $p(q(n))$ für Turing-Maschinen.

Pro-Argument 2: Zwischen polynomialen und exponentiellem Rechenzeitwachstum besteht ein Qualitätssprung. Illustration bei Annahme der Rechenzeit 1μsec für Eingabegröße $n = 1$:

	10	30	60	100
n^2	0,1 msec	1 msec	3,6 msec	10 msec
n^3	1 msec	27 msec	0,2 sec	1 sec
2^n	1 sec	1,7 min	36.000 a	$4 \cdot 10^{15}$ a

Kontra-Argument 1: Die Hardware ist heute schnell genug.
Falsch: Beschleunigung um den Faktor 1000 erlaubt bei fester Rechenzeitschranke

- bei polynomialem Wachstum n^2 eine 30-fache Vergrößerung der behandelbaren Eingaben.
- bei Wachstum 2^n lediglich eine Vergrößerung um 10 Einheiten.

Kontra-Argument 2: Es gibt praktikable Algorithmen mit exponentiellem Zeitaufwand und auch unpraktikable Algorithmen mit polynomialem Zeitaufwand. Dies trifft zuweilen zu, doch ist die Edmonds-Cobham-These in der Regel angemessen.

***SAT* und das $P-NP$-Problem:** Wie ist der Status von SAT? Die Frage $SAT \in P$ ist offen. Wir skizzieren einige Teilergebnisse, die man erzielt hat.

Definition: Eine *nichtdeterministische Turing-Maschine* (NTM) $\mathcal{A}$ ist definiert wie eine Turing-Maschine, jedoch sind für $(q,a) \in Q \times \Gamma$ auch mehrere Quintupel $(q,a,\cdot,\cdot,\cdot)$ erlaubt.
$\mathcal{A}$ *akzeptiert* w :⇔ ex. Konfigurationsfolge, die $\mathcal{A}$ auf w zum Terminieren bringt.
Eine NTM $\mathcal{A}$ *akzeptiert* L :⇔ L enthält genau die durch $\mathcal{A}$ akzeptierten Wörter.

Definition: Zu $f : \mathbb{N} \to \mathbb{N}_+$ heiße die nichtdeterministische Turing-Maschine $\mathcal{A}$ $f(n)$-*zeitbeschränkt* [$f(n)$-*platzbeschränkt*], falls für alle durch $\mathcal{A}$ akzeptierten Wörter w gilt: ex. akzeptierende Konfigurationsfolge der Länge $\leq f(|w|)$ [bzw. mit $\leq f(|w|)$ benutzten Feldern].

Definition:

$$\begin{aligned} \mathrm{NTIME}(f(n)) &= \{L \mid \text{ex. } f(n)\text{-zeitbeschränkte NTM, die } L \text{ akzeptiert}\} \\ \mathrm{NSPACE}(f(n)) &= \{L \mid \text{ex. } f(n)\text{-platzbeschränkte NTM, die } L \text{ akzeptiert}\} \\ NP &:= \bigcup_{p(n)} \mathrm{NTIME}(p(n)) \end{aligned}$$

Die folgenden Beziehungen sind einfach zu verifizieren:

Bemerkung:

(a) $\mathrm{DTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n)) \subseteq \mathrm{NSPACE}(f(n))$
$\mathrm{DTIME}(f(n)) \subseteq \mathrm{NTIME}(f(n))$

(b) $P \subseteq NP \subseteq \mathrm{PSPACE}$.

Es ist offen, ob „$\subsetneq$“ gilt. Die Frage, ob $P \subsetneq NP$, ist das P-NP-Problem.

Diese Frage hängt eng mit dem Status von SAT zusammen. Zunächst stellen wir fest:

Bemerkung: SAT$\in NP$

Beweis: (Skizze) Folgende nichtdeterministische Turing-Maschine akzeptiert SAT:

1. Test, ob die Eingabe $w \in \{x, 0, 1, \wedge, \vee, \neg, (,)\}^*$ ein aussagenlogischer Ausdruck ist. (deterministisch in Polynomzeit möglich, unter Bezug auf entsprechende kontextfreie Grammatik, mit Umsetzung des CYK-Algorithmus. Wenn *nein*, Nichttermination; sonst:

2. Erzeugung einer Wahrheitswertbelegung (durch nichtdeterministisches Ersetzen jedes x in w durch *true* oder *false* und anschließenden deterministischen Test, ob gleiche Variablen gleich belegt wurden). Möglich in Polynomzeit.

3. Auswertung zum Wert *true* oder *false*, Stop bei Erreichen des Wertes *true* (wieder in Polynomzeit).

$\mathcal{A}$ akzeptiert *SAT* in Polynomzeit gemäß Definition für nichtdeterministische Turing-Maschinen. □

Wir skizzieren, daß das allgemeine Problem $P \subsetneq NP$ äquivalent zur speziellen Frage ist, ob *SAT* bereits zu P gehört oder nicht.

Hierfür sind Probleme in NP mit *SAT* zu vergleichen. Dies geschieht mit einer Anpassung der Reduktionsverfahren aus 3.3:

Definition:

(a) $f : \Sigma^* \to \Sigma^*$ heißt *polynomzeitberechenbar*, falls ex. *polynomzeitbeschränkte* Turing-Maschine, die f berechnet.

(b) Für $L, L' \subseteq \Sigma^*$ gelte, $L \leq_p L' :\Leftrightarrow$ ex. polynomzeitberechenbares $f : \Sigma^* \to \Sigma^*$ mit $w \in L \Leftrightarrow f(w) \in L'$.

(c) L_0 ist *NP-vollständig* $:\Leftrightarrow L_0 \in NP, \forall L \in NP : L \leq_p L_0$.

Ein fundamentales Ergebnis, hier ohne Beweis, ist der

Satz: (von Cook)

(a) SAT *ist NP-vollständig*

(b) L_0 *ist NP-vollständig,* $L_0 \in P \Rightarrow P = NP$.

Hieraus erhalten wir, daß *SAT*$\in P$ die Gleichung $P = NP$ impliziert. Die Umkehrung ist trivial. Also gilt: $P = NP \Leftrightarrow$ *SAT*$\in P$.

Übungen

Aufgabe (3.11): (Platzbeschränkung und Entscheidbarkeit) Sei $f : \mathbb{N} \to \mathbb{N}_+$ berechenbar und die Sprache L durch eine $f(n)$-platzbeschränkte nichtdeterministische Turing-Maschine akzeptierbar. Zeigen Sie (auf intuitiver Ebene), daß L entscheidbar ist.

Aufgabe (3.12): (KFNL) Die Sprache *KFNL* enthalte alle Wörter, die gemäß Aufgabe 2.6 eine kontextfreie Grammatik mit nichtleerer Sprache kodieren. Zeigen Sie (durch Umsetzung des entsprechenden Markierungs-Algorithmus), daß *KFNL* durch eine polynomzeitbeschränkte Turing-Maschine entschieden wird. Geben Sie hierzu typische Konfigurationen an und rechtfertigen Sie eine Abschätzung durch eine Funktion in $O(n^k)$ für geeignetes k.

Aufgabe (3.13): (Hornklauseln) Wir betrachten eine spezielle Version des Problems *SAT*, die in der Logik-Programmierung eine zentrale Rolle spielt: das Erfüllbarkeitsproblem für „Konjunktionen von Hornklauseln“. Eine Hornklausel ist eine aussagenlogische Formel der Form

$$\neg y1 \vee \ldots \vee \neg yn \vee z$$

(bzw. unter Verwendung der Implikation

$$y1 \wedge \ldots \wedge yn \rightarrow z)$$

mit aussagenlogischen Variablen $y1, \ldots, yn, z$. Es darf $n = 0$ sein, und z darf fehlen. Sei *HSAT* die Menge der erfüllbaren Konjunktionen von Hornklauseln. Zeigen Sie: *HSAT*$\in P$, d.h. *HSAT* ist in Polynomzeit entscheidbar.
Hinweis: Modifizieren Sie den Markierungsalgorithmus für das Leerheitsproblem kontextfreier Grammatiken geeignet.

Kapitel 4

WHILE-Programme

In diesem Kapitel lösen wir uns von der Inputverarbeitung Bit für Bit (Buchstabe für Buchstabe) und untersuchen globale Strukturen von Algorithmen, etwa Schleifen in Algorithmen und ihre Verschachtelungen sowie Methoden der Programmentwicklung. Als Datenbereich dient die Menge $\mathbb{N}$ der natürlichen Zahlen.
Wir betrachten zahlentheoretische Funktionen $f : \mathbb{N}^n \to \mathbb{N}$, z.B. $+, *, -$, mit der Konvention

$$m - n = \begin{cases} m - n & m \geq n \\ 0 & \text{sonst} \end{cases}$$

und div (Division ohne Rest), mod (Restbildung). Hier vereinbaren wir den Wert 0 bei Division durch 0 (wir wollen das Problem der Undefiniertheit bei den Grundfunktionen vermeiden).

4.1 Syntax und Semantik von WHILE

Syntax: Festlegung der Gestalt der Programme als Zeichenreihen.
Semantik: Festlegung der Wirkung (Bedeutung) der Programme über dem Bereich der natürlichen Zahlen.

Zur Syntax: $\Sigma_{\text{WHILE}} := \{\mathtt{a}, \ldots, \mathtt{z}, \mathtt{0}, \ldots, \mathtt{9}, \mathtt{X}, :, =, ;, >, +, *, -\}$

Ein Wort über Σ_{WHILE} heißt *WHILE-Programm*, falls es durch die folgende kontextfreie Grammatik (in BNF-Form) erzeugt wird:

```
<Programm>::=<Wertzuweisung>|<Programm>;<Programm>|
             loop <Variable> begin <Programm> end|
         (+) while <Variable> > 0 do begin <Programm> end|
             if <Variable> > 0 then begin <Programm> end
                                  else begin <Programm> end
<Wertzuweisung>::=<Variable>:=0|<Variable>:=<Variable>+1|
                  <Variable>:=<Variable>-1|
                  <Variable>:=<Variable>|
                  <Variable>:=<Variable>+<Variable>|
```

```
                  <Variable>:=<Variable>*<Variable>|
                  <Variable>:=<Variable>-<Variable>|
                  <Variable>:=<Variable> div <Variable>|
                  <Variable>:=<Variable> mod <Variable>
<Variable>::=X<pos. Zahl>
<pos. Zahl>::=<pos. Ziffer>|<pos. Zahl><Ziffer>
<Ziffer>::=0|<pos. Ziffer>
<pos. Ziffer>::=1|2|3|4|5|6|7|8|9
```

Entsteht ein WHILE-Programm ohne die Regel (+), so nennen wir es auch *LOOP-Programm.*

Beispiel:

```
X3:=X1;
while X3>0 do begin
  X2:=X2+X1;
  X3:=X3-X1
end;
X1:=X2

loop X1 begin
  X2:=X2+X1
end;
X1:=X2
```

Um eine Behauptung: *Alle WHILE-Programme haben die Eigenschaft E* nachzuweisen, führt man einen Beweis durch *Induktion über den Aufbau der WHILE-Programme.* Hierzu zeigt man:

1. Alle Wertzuweisungen haben die Eigenschaft E.
2. Haben die WHILE-Programme `P` und `Q` die Eigenschaft E, so auch folgende Programme:

```
P;Q
if Xi then do begin P end else begin Q end
loop Xi begin P end
while Xi>0 do begin P end
```

Zur **Semantik der WHILE-Programme** zunächst die Idee: Ein Programm transformiert ein Tupel $(k_1, k_2, \ldots)$ von Werten der Variablen `X1`, `X2`, ... in ein neues Tupel (bzw. liefert bei Nicht-Termination kein Tupel).

Als *Zustandsraum* wählen wir die Menge $\mathbb{N}^{\mathbb{N}_+}$ der Folgen $(k_1, k_2, \ldots)$ mit $k_i \in \mathbb{N}$.

Konventionen: Für $(k_1, k_2, \ldots) \in \mathbb{N}^{\mathbb{N}_+}$ sei die i-te *Projektion* $pr_i(k_1, k_2, \ldots) = k_i$. Wir betrachten Funktionen $f : \mathbb{N}^{\mathbb{N}_+} -\to \mathbb{N}^{\mathbb{N}_+}$. Allgemein definieren wir:

1. Für partielle Funktionen $f : A- \to B, g : A- \to B$ ist $f(a) = g(a)$, falls beide Seiten undefiniert sind oder beide Seiten definiert und gleich. Schreibe „$f(a) = \perp$“ statt „$f(a)$ ist undefiniert“.

2. Für $g_1, \ldots, g_m : A- \to B, f : B^m- \to C$ sei $f(g_1(a), \ldots, g_m(a))$ undefiniert, falls ein $g_i(a)$ undefiniert ist oder falls alle $g_i(a)$ definiert sind (etwa $=b_i$) und $f(b_1, \ldots, b_m)$ undefiniert ist.

3. Für $f : A- \to A$ sei $f^n : A- \to A$ definiert durch $f^0 := id_A$, $f^{n+1}_{(a)} = f(f^n(a))$.

Definition: Zu einem WHILE-Programm P wird die *semantische Funktion* $[\![P]\!] : \mathbb{N}^{\mathbb{N}_+} - \to \mathbb{N}^{\mathbb{N}_+}$ induktiv über den Aufbau der WHILE-Programme wie folgt definiert.

1. Für eine Wertzuweisung:

$$\begin{array}{rll}
P : \texttt{Xi} := \texttt{0} & \text{sei} & [\![P]\!](k_1, k_2, \ldots) = (k_1, \ldots, 0, k_{i+1}, \ldots) \\
P : \texttt{Xi} := \texttt{Xj} + \texttt{1} & \text{sei} & [\![P]\!](k_1, k_2, \ldots) = (k_1, \ldots, k_j + 1, k_{i+1}, \ldots) \\
P : \texttt{Xi} := \texttt{Xj} - \texttt{1} & \text{sei} & [\![P]\!](k_1, k_2, \ldots) = (k_1, \ldots, k_j - 1, k_{i+1}, \ldots) \\
P : \texttt{Xi} := \texttt{Xj} & \text{sei} & [\![P]\!](k_1, k_2, \ldots) = (k_1, \ldots, k_j, k_{i+1}, \ldots) \\
P : \texttt{Xi} := \texttt{Xj} + \texttt{Xk} & \text{sei} & [\![P]\!](k_1, k_2, \ldots) = (k_1, \ldots, k_j + k_k, k_{i+1}, \ldots) \\
P : \texttt{Xi} := \texttt{Xj} - \texttt{Xk} & \text{sei} & [\![P]\!](k_1, k_2, \ldots) = (k_1, \ldots, k_j - k_k, k_{i+1}, \ldots) \\
P : \texttt{Xi} := \texttt{Xj} * \texttt{Xk} & \text{sei} & [\![P]\!](k_1, k_2, \ldots) = (k_1, \ldots, k_j * k_k, k_{i+1}, \ldots)
\end{array}$$

2. Falls $P = \texttt{P}_1;\texttt{P}_2$, sei $[\![P]\!](k_1, k_2, \ldots) = [\![P_2]\!]([\![P_1]\!](k_1, k_2, \ldots))$

3. Falls $P =$ `loop Xi begin P₁ end`, sei $[\![P]\!](k_1, k_2, \ldots) = [\![P_1]\!]^{k_i}(k_1, k_2, \ldots)$

4. Falls $P =$ `while Xi>0 do begin P₁ end`, sei

$$[\![P]\!](k_1, k_2, \ldots) = \begin{cases} [\![P_1]\!]^n(k_1, k_2, \ldots) & \text{für das kleinste } n \text{ mit} \\ & pr_i([\![P_1]\!]^n(k_1, k_2, \ldots)) = 0 \\ & \text{falls solch ein } n \text{ existiert} \\ \perp & \text{sonst} \end{cases}$$

5. Falls $P =$ if $\texttt{Xi} > \texttt{0}$ then begin $\texttt{P}_1$ end else begin $\texttt{P}_2$ end, so

$$[\![P]\!](k_1, k_2, \ldots) = \begin{cases} [\![P_1]\!](k_1, k_2, \ldots) & \text{falls } k_i > 0 \\ [\![P_2]\!](k_1, k_2, \ldots) & \text{falls } k_i = 0 \end{cases}$$

Ein Eingabe-n-Tupel wird in den Variablen `X1`,...,`Xn` angenommen; die Ausgabe soll der Wert von `X1` sein. Diese Konvention zeigt sich in folgender Definition:

Definition: $in^{(n)} : \mathbb{N}^n \to \mathbb{N}^{\mathbb{N}_+}$ sei erklärt durch

$$in^{(n)}(k_1, \ldots, k_n) = (k_1, \ldots, k_n, 0, 0, \ldots).$$

Zu jedem WHILE-Programm P und zu $n \geq 0$ wird die durch P berechnete n-stellige Funktion $f_P^{(n)} : \mathbb{N}^n - \to \mathbb{N}$ definiert durch

$$f_P^{(n)}(k_1, \ldots, k_n) = pr_1([\![P]\!](in^{(n)}(k_1, \ldots, k_n))).$$

Ist $f : \mathbb{N}^n - \to \mathbb{N}$, P WHILE-Programm, sagen wir: P *berechnet* f, falls $f = f_P^{(n)}$. f heißt *WHILE-berechenbar* (bzw. *LOOP-berechenbar*), falls ex. ein WHILE- (bzw. LOOP-) Programm, das f berechnet.

Beispiel:

$$\begin{array}{ll} \texttt{Q}: & \left.\begin{array}{l} \texttt{loop X1 begin} \\ \quad \texttt{X2} := \texttt{X2} + \texttt{X1} \quad \} \, \texttt{P}_0 \\ \texttt{end;} \end{array}\right\} \texttt{P}_1 \\ & \texttt{X1} := \texttt{X2} \quad \} \, \texttt{P}_2 \end{array}$$

Wir erhalten:

$$\begin{aligned} [\![Q]\!](k, 0, \ldots) &= [\![P_2]\!]([\![P_1]\!](k, 0, \ldots)) \\ &= [\![P_2]\!]([\![P_0]\!]^k(k, 0, \ldots)) \\ &= [\![P_2]\!](k, k^2, 0, \ldots) \\ &= (k^2, k^2, 0, \ldots) \end{aligned}$$

Zur vorletzten Gleichung: $[\![P]\!]^i(k, 0, \ldots) = (k, i \cdot k, 0, \ldots)$ zeigen wir durch Induktion über i, setze dann $i = k$.
$i = 0$ ist klar
$i + 1$:

$$\begin{aligned} [\![P_0]\!]^{i+1}(k, 0, \ldots) &= [\![P_0]\!]([\![P_0]\!]^i(k, 0, \ldots)) \\ &= [\![P_0]\!](k, i \cdot k, 0, \ldots)[\text{nach I.V.}] \\ &= (k, i \cdot k + k, 0, \ldots) \\ &= (k, (i+1) \cdot k, 0, \ldots) \end{aligned}$$

Also $f_Q^{(1)}(k) = k^2$.

Beispiel:

$$\begin{array}{ll} \texttt{P}: & \texttt{while X1} > \texttt{0 do begin} \\ & \quad \left.\begin{array}{l} \texttt{loop X2 begin} \\ \quad \texttt{X1} := \texttt{X1} - \texttt{1} \quad \} \, \texttt{P}_1 \\ \texttt{end} \end{array}\right\} \texttt{P}_0 \\ & \texttt{end} \end{array}$$

Behauptung über P_0: $[\![P_0]\!](k_1, k_2, \ldots) = (k_1 - k_2, k_2, \ldots)$.
Hierzu zeigt man zunächst $[\![P_1]\!]^i(k_1, k_2, \ldots) = (k_1 - i, k_2, \ldots)$ per Induktion über i. Dann

$$[\![P_0]\!](k_1, k_2, \ldots) = [\![P_1]\!]^{k_2}(k_1, k_2, \ldots) = (k_1 - k_2, k_2, \ldots).$$

Also

$$[\![P_0]\!]^n(k_1, k_2, \ldots) = (k_1 - n \cdot k_2, k_2, \ldots)$$

mit Induktion über n: Somit

$$[\![P]\!](k_1, k_2, \ldots) = \begin{cases} (k_1 - n \cdot k_2, k_2, \ldots) & \text{für das kleinste } n \text{ mit } k_1 - n \cdot k_2 = 0, \\ & \text{falls solches } n \text{ existiert} \\ \perp & \text{sonst} \end{cases}$$

Also

$$[\![P]\!](k_1,k_2,k_3,\ldots) = \begin{cases} (0,k_2,k_3,\ldots) & k_2 > 0 \\ (0,0,k_3,\ldots = & k_2 = 0, k_1 = 0 \\ \perp & k_2 = 0, k_1 = 0 \end{cases}$$

Ferner:

$$\begin{aligned} f_P^{(0)} &= 0 \\ f_P^{(1)}(k) &= \begin{cases} 0 & k = 0 \\ \perp & k > 0 \end{cases} \\ f_P^{(2)}(k_1,k_2) &= \begin{cases} 0 & k_2 > 0, k_2 = 0 \wedge k_1 = 0 \\ \perp & \text{sonst} \end{cases} \end{aligned}$$

Abschließend notieren wir eine spezielle Eigenschaft von LOOP-Programmen und zeigen, wie man `loop` durch `while` ersetzen kann.

Bemerkung: *Jede LOOP-berechenbare Funktion ist total*

Beweis: Zeige, für jedes LOOP-Programm P ist $[\![P]\!]$ total, induktiv über den Aufbau der LOOP-Programme.
I.A.: P Wertzuweisung, $[\![P]\!]$ total nach Definition.
I.S.: Sei $\mathtt{P} : \mathtt{P_1};\mathtt{P_2}$ bzw. $\mathtt{P}$: if $\mathtt{Xi} > \mathtt{0}$ then $\mathtt{P_1}$ else $\mathtt{P_2}$ bzw. $\mathtt{P}$: `loop Xi begin` $\mathtt{P_1}$ `end` mit den LOOP-Programmen $\mathtt{P_1}, \mathtt{P_2}$. I.V.: $[\![P_1]\!], [\![P_2]\!]$ total.
Dann ist in allen Fällen nach Definition von $[\![P]\!]$ auch $[\![P]\!]$ total. $\square$

Folgerung: *Es gibt WHILE-berechenbare nicht LOOP-berechenbare Funktion (etwa nichttotale Funktionen, z.B. $f_P^{(1)}$).*

Definition: $i(P)$:=maximaler Index der in P vorkommenden Variablen.

Lemma: *Ist P ein LOOP-Programm und gilt $m \geq i(P)$, dann ex. ein WHILE-Programm $\underline{P}$ ohne* `loop`*, so daß* $[\![P]\!](k_1,k_2,\ldots,k_m,0,\ldots) = [\![\underline{P}]\!](k_1,k_2,\ldots,k_m,0,\ldots)$.

Beweis: induktiv über den Aufbau der LOOP-Programme. Für P Wertzuweisung setze $\underline{P} = P$. Seien $\mathtt{P} = \mathtt{P_1};\mathtt{P_2}$ und $m \geq i(P)$ gegeben; dann $m \geq i(P_1), m \geq i(P_2)$. I.V. liefert $\underline{P}_1, \underline{P}_2$. Setze $\underline{\mathtt{P}} = \underline{\mathtt{P}_1;\mathtt{P}_2}$. Dann gilt die Induktionsbehauptung:

$$\begin{aligned} [\![\underline{P}]\!](k_1,\ldots,k_m,0,\ldots) &= [\![\underline{P}_2]\!]([\![\underline{P}_1]\!](k_1,\ldots,k_m,0,\ldots)) \\ &= [\![\underline{P}_2]\!]([\![P_1]\!](k_1,\ldots,k_m,0,\ldots)) \\ &= [\![P_2]\!]([\![P_1]\!](k_1,\ldots,k_m,0,\ldots) \\ &= [\![P]\!](k_1,\ldots,k_m,0,\ldots). \end{aligned}$$

$\mathtt{P}$: `if Xi` $>$ `0 then` $\mathtt{P_1}$ `else` $\mathtt{P_2}$ analog zu $\mathtt{P_1};\mathtt{P_2}$.
$\mathtt{P}$: `loop Xi begin` $\mathtt{P_1}$ `end`, $m \geq i(P)$ gegeben. Dann ist $m \geq i(P_1)$. Wähle $\underline{P}_1$ gemäß I.V. Setze $j := \max m, i(\underline{P}_1) + 1$ und

$$\underline{\mathtt{P}} : \mathtt{Xj := Xi;\ while\ Xj > 0\ do\ begin\ }\underline{\mathtt{P_1}}\mathtt{;\ Xj := Xj - 1\ end}$$

Dann hat $\underline{\mathtt{P}}$ kein `loop`, und $[\![\underline{P}]\!](k_1,\ldots,k_m,0,\ldots) = [\![P]\!](k_1,\ldots,k_m,0,\ldots)$. □

Satz: *Jede LOOP-berechenbare Funktion ist auch durch ein WHILE-Programm ohne loop berechenbar.*

Beweis: Sei $f : \mathbb{N}^n \to \mathbb{N}$ LOOP-berechenbar, etwa $f = f_P^{(n)}$ mit LOOP-Programm P. Setze $m := \max n, i(P)$. Dann ist $m \geq i(P)$ und gemäß Lemma existiert $\underline{P}$ mit: $(*)[\![P]\!](k_1,k_2,\ldots,k_m,0,\ldots) = [\![\underline{P}]\!](k_1,k_2,\ldots,k_m,0,\ldots)$.
Somit

$$\begin{aligned}
f_{\underline{P}}{}^{(n)}(k_1,\ldots,k_n) &= pr_1([\![\underline{P}]\!](in^{(n)}(k_1,\ldots,k_n)))\\
&= pr_1([\![\underline{P}]\!](k_1,\ldots,k_n,0,\ldots))\\
&= pr_1([\![P]\!](k_1,\ldots,k_n,0,\ldots)) \quad \text{(wegen (*) und } m \geq n)\\
&= f_P^{(n)}(k_1,\ldots,k_n) = f(k_1,\ldots,k_n).
\end{aligned}$$

$\underline{P}$ ohne loop berechnet also f. □

Übungen
Aufgabe (4.1): (Reduktion der WHILE-Programme auf WHILE_0-Programme) Die Sprache der WHILE_0-Programme entstehe aus WHILE durch Streichen aller Wertzuweisungen außer den Anweisungen `Xi:=Xi+1, Xi:=Xi-1`. Eine durch ein WHILE_0-Programm berechenbare Funktion heißt WHILE_0-berechenbar.
Zeigen Sie durch Angabe geeigneter Programmänderungen, daß eine n-stellige Funktion f, die durch ein WHILE-Programm berechnet wird, auch WHILE_0-berechenbar ist.

Aufgabe (4.2): (Semantik eines LOOP-Programms) Zeigen Sie durch Rückgriff auf die Semantik der WHILE-Programme für das Programm P:

```
loop X1 begin
  X1:=X1-1
end;
loop X2 begin
  loop X3 begin
    X1:=X1+1
  end
end
```

daß gilt $[\![P]\!](k_1,k_2,k_3,\ldots) = (k_2 \cdot k_3, k_2, k_3, \ldots)$.

4.2 Vergleich zu Turing-Maschinen und GOTO-Programmen

Wir identifizieren $f : \mathbb{N}^n - \to \mathbb{N}$ mit der Funktion $f' : (\{|\}^*)^n - \to \{|\}^*$, gegeben durch $f'(|^{i_1},\ldots,|^{i_n}) = |^{f(i_1,\ldots,i_n)}$. In diesem Sinne berechnet eine Turing-Maschine über $\Sigma = \{|\}$ eine Funktion $f : \mathbb{N}^n - \to \mathbb{N}$, und wir können WHILE-Programme mit Turing-Maschinen vergleichen. Als Zwischenetappe sind „GOTO-Programme“ von Nutzen. Sie bilden eine Kern-„Maschinensprache“ mit Sprungbefehlen.

Definition:

(a) Syntax: Ein *GOTO-Programm* hat die Form $P : 1\alpha_1; \ldots; m\alpha_m$ mit $\alpha_m =$ `stop`, wobei für $1 \leq i < m$: $\alpha_i = \texttt{Xj} := \texttt{Xj} + \texttt{1}$ oder $\texttt{Xj} := \texttt{Xj} - \texttt{1}$ oder $\texttt{if Xj} = \texttt{0 goto l}$ mit $1 \leq l \leq m$ und j, l jeweils positive natürliche Zahlen in Dezimaldarstellung.

(b) Semantik: Eine *GOTO-Programm-Konfiguration* (zu P wie oben) hat die Form $(l, k_1, k_2, \ldots)$ mit $1 \leq l < m$. Die Folgekonfiguration hierzu ist:

$$\begin{array}{rll} (l+1, k_1, \ldots, k_j + 1, k_{j+1}, \ldots), & \text{falls} & \alpha_l = \texttt{Xj} := \texttt{Xj} + \texttt{1} \\ (l+1, k_1, \ldots, k_j - 1, k_{j+1}, \ldots), & \text{falls} & \alpha_l = \texttt{Xj} := \texttt{Xj} - \texttt{1} \\ (l', k_1, k_2, \ldots), & \text{falls} & \alpha_l = \texttt{if Xj} = \texttt{0 goto l}' \text{ und } k_j = 0 \\ (l+1, k_1, k_2, \ldots), & \text{falls} & \alpha_l = \texttt{if Xj} = \texttt{0 goto l}' \text{ und } k_j \neq 0 \end{array}$$

(c) P *berechnet* die n-stellige Funktion $f : \mathbb{N}^n - \to \mathbb{N}$, falls gilt: P erreicht von der Konfiguration $(1, k_1, \ldots, k_n, 0, \ldots)$ schließlich eine Stopkonfiguration $(m, k, \ldots)$ mit $k = f(k_1, \ldots, k_n)$ falls $f(k_1, \ldots, k_n)$ definiert ist, und stoppt andernfalls nicht.

Drei Beispielprogramme:

```
P1  1 if X1 = 0 goto 4;      P2  1 if X1 = 0 goto 1;      Pn  1 stop
    2 X1 := X1 - 1;              2 stop;
    3 if X2 = 0 goto 1;
    4 stop;
```

Andere Bezeichnungen: URM-Programme („universelle Register-Maschine“), RAM-Programme („Random-Access-Maschine“)

$f_P^{(n)}$ stehe für die durch P berechnete Funktion.

Wir sagen: f ist *GOTO-berechenbar* $\Leftrightarrow$ ex. GOTO-Programm P, das f berechnet, d.h. falls $f = f_P^{(n)}$.

Beispiel:

$$\begin{array}{rcl} f_{P_1}^{(1)}(k) & = & 0 \\ f_{P_1}^{(2)}(k,l) & = & \begin{cases} 0 & l = 0 \\ 0 & k = 0, l \neq 0 \\ k-1 & k \neq 0, l \neq 0 \end{cases} \\ f_{P_2}^{(1)}(k) & = & \begin{cases} \perp & k = 0 \\ k & k > 0 \end{cases} \\ f_{P_3}^{(1)}(k) & = & k \end{array}$$

Der Äquivalenzbeweis zwischen WHILE-Programmen und Turing-Maschinen erfolgt durch folgende Kette von Schritten:

$$\text{WHILE}\to\text{WHILE}_0\to\text{GOTO}\to\text{TM}\to\text{WHILE}$$

Hierbei seien WHILE_0-Programme WHILE-Programme nur mit Wertzuweisungen $\mathtt{Xj} := \mathtt{Xj} + \mathtt{1}$, $\mathtt{Xj} := \mathtt{Xj} - \mathtt{1}$ und ohne Vorkommen von `loop`.

Satz: *Jede WHILE-berechenbare Funktion ist WHILE_0-berechenbar.*

Beweis: Übungsaufgabe 4.1 zusammen mit dem Satz über Elimination von `loop`.
□

Satz: *Jede WHILE_0-berechenbare Funktion ist GOTO-berechenbar.*

Beweis: Zeige durch Induktion über den Aufbau der WHILE_0-Programme:
Zu einem WHILE_0-Programm P und $m \geq i(P)$ kann man ein GOTO-Programm P' konstruieren, so daß für alle $(k_1,\ldots,k_m,0,\ldots)$: $[\![P]\!](k_1,\ldots,k_m,0,\ldots)$ definiert und gleich $(l_1,\ldots,l_m,0,\ldots)$ ist $\Leftrightarrow$ P' stoppt angesetzt auf $(k_1,\ldots,k_m,0,\ldots)$ und liefert $(l_1,\ldots,l_m,0,\ldots)$, sowie $[\![P]\!](k_1,\ldots,k_m,0,\ldots)$ undefiniert $\Leftrightarrow$ P' stoppt auf $(k_1,\ldots,k_m,0,\ldots)$.
Dann sind wir fertig: Wird $f : \mathbb{N}^n - \to \mathbb{N}$ durch P berechnet, dann wähle $m := \max\{n, i(P)\}$, also $m \geq i(P)$ und finde P' wie oben. Es gilt $f_P^{(n)} = f_{P'}^{(n)}$. Weil $f = f_P^{(n)}$, wird f durch P' berechnet.
I.A.: Für `P: Xi:=Xi+1`, setze `P': 1 Xi:=Xi+1; 2 stop`. Für „-“ analog.
I.S.: Für $\mathtt{P} : \mathtt{P_1};\mathtt{P_2}$ wähle P_1', P_2' gemäß I.V. P_1' habe l Zeilen. Setze $P' := P_1'$ ohne `stop`, gefolgt von P_2 mit um $l-1$ erhöhten Zeilennummern.
Für $\mathtt{P} : \mathtt{if\ Xi} > \mathtt{0\ then\ P_1\ else\ P_2}$ liefert die I.V.P_1', P_2' mit l_1, l_2 Zeilen. Setze P':

1	$\mathtt{if\ Xi} = \mathtt{0\ goto\ l_1} + \mathtt{2}$;
2	$\mathtt{P}_1'$ (ohne `stop` mit um 1 erhöhten Zeilennummern);
$\mathtt{l_1} + \mathtt{1}$	$\mathtt{if\ X(m} + \mathtt{1)} = \mathtt{0\ goto\ l_1} + \mathtt{l_2} + \mathtt{1}$;
$\mathtt{l_1} + \mathtt{2}$	$\mathtt{P}_2'$ (ohne `stop` mit um $l_1 + 1$ erh. Zeilennummern);
$\mathtt{l_1} + \mathtt{l_2} + \mathtt{1}$	`stop`

Für $\mathtt{P} : \mathtt{while\ Xi} > \mathtt{0\ do\ P_1}$, wähle P_1' gemäß I.V. mit l_1 Zeilen. Setze P':

1	$\mathtt{if\ Xi} = \mathtt{0\ goto\ l_1} + \mathtt{2}$;
2	$\mathtt{P}_1'$ (ohne `stop` mit um 1 erhöhten Zeilennummern);
$\mathtt{l_1} + \mathtt{1}$	$\mathtt{if\ X(m} + \mathtt{1)} = \mathtt{0\ goto\ 1}$;
$\mathtt{l_1} + \mathtt{2}$	`stop`

□

Satz: *Jede GOTO-berechenbare Funktion ist Turing-berechenbar.*

Beweis: Das GOTO-Programm P berechne $f : \mathbb{N}^n - \rightarrow \mathbb{N}$, es sei $m := \max\{n, i(P)\}$.
Idee: P-Konfiguration $(k, r_1, \ldots, r_m, 0, \ldots)$ entspricht der Turing-Maschinen-Konfiguration $q_{(k)}\#|^{r_1}\$\ldots\$|^{r_m}\$$, wobei $q_{(k)}$ ein spezieller Zustand zum Index k sei. Hat P l Zeilen, so habe die Turing-Maschine unter anderem die Zustände $q_{(1)}, \ldots, q_{(l)}$.
Konstruiere Turing-Maschine $\mathcal{A}$ zu P und m durch Turing-Zeilenblöcke $B_0, \ldots, B_l$, wobei die erste Zeile in B_i $(i > 0)$ mit $q_{(i)}$ beginne. Für $k = 1, \ldots, l-1$ soll B_k folgendes leisten: Führt P von der Konfiguration

$$(k, r_1, \ldots, r_m, 0, \ldots) \text{ zu } (k', r'_1, \ldots, r'_m, 0, \ldots),$$

so soll durch B_k die Turing-Maschinen-Konfiguration

$$q_{(k)}\#|^{r_1}\$\ldots\$|^{r_m}\$ \text{ in } q_{(k')}\#|^{r_1}\$\ldots\$|^{r_m}\$$$

übergehen und umgekehrt.
Arbeitsweise von B_i (wir verzichten auf Programmierung durch Turingzeilen):

(a) falls in P Zeile i: $\mathtt{Xj} := \mathtt{Xj} + \mathtt{1}$: Gehe auf das j-te \$, verschiebe die Inschrift von j-tem \$ bis m-tem \$ um ein Feld nach rechts. Fülle die Lücke durch | und gehe zurück zu # und in den Zustand $q_{(i+1)}$.

(b) falls in P Zeile i: $\mathtt{Xj} := \mathtt{Xj} - \mathtt{1}$: Gehe auf das Feld vor dem j-ten \$. Falls dort \$ oder #, gehe zurück zu # in Zustand $q_{(i+1)}$; sonst verschiebe die Inschrift von j-tem \$ bis zum m-ten \$ um ein Feld nach links, dann zurück zu # in $q_{(i+1)}$.

(c) falls in P Zeile i: `if Xj = 0 goto k`. Gehe auf das Feld vor dem j-ten \$. Falls dort \$ oder #, zurück zu # in $q_{(k)}$, andernfalls zurück zu # in $q_{(i+1)}$.

B_0 überführe Konfiguration $q_0|^{r_1}b\!\!\!/\ldots b\!\!\!/|^{r_m}b\!\!\!/$ in Konfiguration $q_{(1)}\#|^{r_1}\$\ldots\$|^{r_m}\$$, B_l überführe $q_{(l)}\#|^{r_1}\$\ldots\$|^{r_m}\$$ in $q_s|^{r_1}b\!\!\!/$ mit einem Stopzustand q_s. □

Die Hauptschwierigkeit des Äquivalenzsatzes liegt in der folgenden Behauptung:

Satz: *Jede Turing-berechenbare Funktion ist WHILE-berechenbar.*

Beweis: Idee: Sei $\mathcal{A}$ Turing-Maschine mit der Zustandsmenge $Q = \{q_1, \ldots, q_m, q_0\}$ und Arbeitsalphabet $\Gamma = \{a_0, a_1, \ldots, a_{n-1}\}$ mit $a_0 = b\!\!\!/$, $a_1 = |$; es sei $\{|\}$ das Eingabealphabet und q_1 Startzustand und q_0 Stopzustand.
OBdA gelte: $\forall i \in \{1, \ldots, m\}$ $\forall j \in \{0, \ldots, n-1\}$ ex. eine Zeile $q_i a_j \ldots$. Eine Turing-Maschinen-Konfiguration κ $\ldots a_0 a_0 a_{i_k} \ldots a_{i_0} q_r a_{j_0} \ldots a_{j_l} a_0 a_0 \ldots$ wird kodiert durch das Zahlentripel Z_k, L_k, R_k mit

$$Z_k = r, L_k = i_0 + i_1 \cdot n + \ldots + i_k \cdot n^k, R_k = j_0 + j_1 \cdot n + \ldots + j_l \cdot n^l.$$

Beachte: Wegen der Eindeutigkeit der n-adischen Zahlendarstellung sind die i_s, j_s durch L_k, R_k eindeutig bestimmt.
Bsp: Im Fall $\Gamma = \{a_0, a_1\} = \{b\!\!\!/, |\}$, $n = 2$, $\kappa = \ldots b\!\!\!/b\!\!\!/|||b\!\!\!/|q_{17}|b\!\!\!/||b\!\!\!/b\!\!\!/$ ergibt sich

$$Z_k = 17, L_k = 29, R_k = 13.$$

Im gesuchten WHILE-Programm werden die Variablen `X1`, `X2`, `X3` die Werte Z_k, L_k, R_k aufnehmen. Berechnet $\mathcal{A}$ etwa $f : \mathbb{N}^n - \rightarrow \mathbb{N}$, so hat das gesuchte WHILE-Programm $P_{\mathcal{A}}$ folgende Grobstruktur (3 Blöcke):

1. Überführung der Variablenwerte $k_1, \ldots, k_n$ (in `X1`,…,`Xn`) in die Kodierung der entsprechenden $\mathcal{A}$-Anfangskonfiguration $q_1|^{k_1}b\!\!\!/ \ldots b\!\!\!/|^{k_n}b\!\!\!/$, gebracht nach `X1`, `X2`, `X3`.

2. TM-Simulation:

   while $X1 > 0$ Fall q_i mit $i > 0$, d.h. Turing-Maschine macht weiter

   falls X1=i und X3 mod n=j

   {TM-Zeile $q_i a_j \ldots$ zu simulieren}

   passe Werte von `X1`,`X2`,`X3` gemäß Turing-Zeile $q_i a_j \ldots$ an.

3. Extrahiere aus `X3` (Kodierung der rechten Bandinschrift) den Ausgabewert und bringe diesen nach `X1`.

Insgesamt entsteht ein „WHILE-Interpreter für Turingmaschinen“. Präzisierung zu 1. (hier für 2-stellige Funktion):
Halbformal:

(a) $X3 := 1 + 1 \cdot n + \ldots + 1 \cdot n^{X1-1} + 0 \cdot n^{X1} + 1 \cdot n^{X1+1} + \ldots + 1 \cdot n^{X1+X2}$

Beachte: Dies kodiert die Buchstabenfolge $|^{X1}\, b\!\!\!/ \,|^{X2}$.

(b) $X2 := 0; X1 := 1$

Zu (a) geben wir – wieder für eine zweistellige Funktion, d.h. für Eingabevariablen `X1`,`X2` – noch eine genaue Formulierung: Benutze `X4` für den Wert n, `X5` für n-Potenzen, `X6` für Teilsummen.

`X4:=0;`
`X4:=X4+1; ... ; X4:=X4+1;`(n-mal)
`X5:=1; X6:=0;`
(∗ Aufbau von $1 + \ldots + n^{X1-1}$ ∗)
`loop X1 begin X6:=X6+X5; X5:=X5*X4 end;`
(∗ Aufbau von $1 + \ldots + n^{X1-1} + n^{X1+1} + \ldots + n^{X1+X2}$ ∗)
`loop X2 begin X5:=X5*X4; X6:=X6+X5 end;`
`X3:=X6; X5:=0; X6:=0` □

Zur Simulation der Turing-Maschinen-Schritte:
Illustration eines Linksschritts (der Einfachheit halber für den Fall $\Gamma = \{a_0.,,,.a_9\} = \{0, \ldots, 9\}$, also $n = 10$): Auf die Turing-Maschinen-Konfiguration $\kappa = 1085q_{17}7702$ werde die TM-Zeile $(q_{17}, 7, 6, l, q_{18})$ angewandt.

Der Kode R (in üblicher Dezimaldarstellung) verändert sich so:

$$2077 \xrightarrow{-7} 2070 \xrightarrow{+6} 2076 \xrightarrow{*10} 20760 \xrightarrow{+5} 20765$$

1. Löschen des Arbeitsfeld-Buchstaben

2. Drucken des neuen Buchstaben

3. Platz für Buchstaben gemäß Linksschritt
4. Besetzung des Platzes mit letztem Buchstaben der linken Inschrift

Der Kode L der linken Bandinschrift geht über in L div 10. Es ergeben sich die Kodes $L' = 108$, $R' = 20765$ der neuen Turing-Maschinen-Konfiguration $\kappa' : 108q_{18}56702$.

Allgemein ($\Gamma = \{a_0, \ldots, a_n\}$) induziert ein Linksschritt $q_i a_j a_{j'} l q_{i'}$ die folgenden Wertzuweisungen (in Kurznotation):

```
X3:=(X3-j+j')*n+(X2 mod n);
X2:=X2 div n;
X1=i';
```

Analog induziert ein Turing-Maschinen-Rechtsschritt $q_i a_j a_{j'} r q_{i'}$ die Wertzuweisungen

```
X2:=X2*n+j';
X3:=X3 div n;
X1:=i';
```

Folgendes WHILE-Programm (in Kurznotation) simuliert also die sukzessive Schrittsimulation (beachte, daß n in `X4` gespeichert wird):

`while` $\mathtt{X1} > 0$ `do`
  `begin` Liste von Anweisungen und zwar
  für jede Turingzeile $q_i a_j a_{j'} l q_{i''}$
  `if` $\mathtt{X1} = \mathtt{i}$ `and` $\mathtt{X3}$ `mod` $\mathtt{X4} = \mathtt{j}$ `then`
    `begin` $\mathtt{X3} := (\mathtt{X3} - \mathtt{j} + \mathtt{j}') * \mathtt{X4} + (\mathtt{X2}\ \mathtt{mod}\ \mathtt{X4})$;
      $\mathtt{X2} := \mathtt{X2}\ \mathtt{div}\ \mathtt{X4}$; $\mathtt{X1} := \mathtt{i}'$
    `end`
  für jede Turingzeile $q_i a_j a_{j'} r q_{i''}$
  `if` $\mathtt{X1} = \mathtt{i}$ `and` $\mathtt{X3}$ `mod` $\mathtt{X4} = \mathtt{j}$ `then`
    `begin` $\mathtt{X2} := \mathtt{X2} * \mathtt{X4} + \mathtt{j}'$;
      $\mathtt{X3} := \mathtt{X3}\ \mathtt{div}\ \mathtt{X4}$; $\mathtt{X1} := \mathtt{i}'$
    `end`
  `end`

Hierbei steht z.B.

- `if` $<$ `Bedingung` $>$ `then` $<$ `Anweisung` $>$ für
  `if` $<$ `Bedingung` $>$ `then` $<$ `Anweisung` $>$ `else` $\mathtt{X1} := \mathtt{X1}$
- `if` $\mathtt{X1} = \mathtt{i}$ `and` $\mathtt{X3}$ `mod` $\mathtt{X4} = \mathtt{j}$ `then` $<$ `Anweisung` $>$ für
  $\mathtt{X5} := 1 - [(\mathtt{X1} - \mathtt{i}) + (\mathtt{i} - \mathtt{X1}) + ((\mathtt{X3}\ \mathtt{mod}\ \mathtt{X4}) - \mathtt{j}) + (\mathtt{j} - (\mathtt{X3}\ \mathtt{mod}\ \mathtt{X4}))]$;
  `if` $\mathtt{X5} > 0$ `then begin` $\mathtt{X5} := 0$; $<$ `Anweisung` $>$

Nachzutragen sind noch:

(a) LOOP-Programme für div, mod

(b) Punkt 3: Extraktion des Ergebniswerts der Turing-Maschinen-Stopkonfiguration

zu (a)

$$x \text{ div } y := \begin{cases} \left\lfloor \frac{x}{y} \right\rfloor & y > 0 \\ 0 & y = 0 \end{cases}$$

$$x \text{ mod } y := \begin{cases} \text{Rest von } x \text{ bei Division durch } y & y > 0 \\ 0 & y = 0 \end{cases}$$

Idee: Stelle für $i = 0, 1, 2, \ldots$ die Werte $x' = x - (i+1) \cdot y$ und $r = x - i \cdot y$ her. Verwende den Zähler Z für die i-Werte, solange $x' > 0$ bzw. $x' = 0$ und $r = y$. In diesem Fall ist x mod $y = r$ bzw. $= 0$.

Für Variablen `X,Y,Z` und ein neues `X'` simuliert folgendes WHILE-Programm die Anweisungen `Z:=X div Y; R:=X mod Y`:

```
if Y=0 then begin Z:=0; R:=0 end else
 X':=X; R:=X; Z:=0; {R=X-Z*Y}
 loop X begin
   X':=X'-Y; {X'=X-(Z+1)*Y}
   if X'>0 or R=Y then begin
     Z:=Z+1; R:=R-Y
   end {else: Z=X div Y und R=X mod Y}
 end
```

zu (b) Wir betrachten nur den Fall des Einfabealphabets $\{|\}$. Benutze den Zähler `X1` für die Anzahl der Divisionen von `X3` durch `X4` (Wert n) mit Rest 1.

```
X1:=0;
loop X3 begin
  if X3 mod X4=1 then X1:=X1+1 else X3:=0;
  X3:=X3 div X4
end
```

Übungen

Aufgabe (4.3): (Test auf positive Zahlen) GOTO'-Programme seien wie GOTO-Programme definiert, jedoch mit der Sprunganweisung `if Xj>0 goto l` und der entsprechend abgeänderten Semantik. Zeigen Sie: Eine Funktion $f : \mathbb{N}^n - \rightarrow \mathbb{N}$ ist GOTO-berechenbar, genau dann wenn sie GOTO'-berechenbar ist.

Aufgabe (4.4): („Minimalität" der Sprache der GOTO-Programme) Zeigen Sie, daß man in GOTO-Programmen nicht auf einen der drei Anweisungstypen verzichten kann, ohne die Berechenbarkeit gewisser Funktionen zu verlieren. Seien GOTO$_1$-Programme GOTO-Programme ohne Inkrement, GOTO$_2$-Programme GOTO-Programme ohne Dekrement und GOTO$_3$-Programme GOTO-Programme ohne Sprunganweisung. Geben Sie für $i = 1, \ldots, 3$ jeweils eine Funktion f_i an, die GOTO-berechenbar, jedoch nicht GOTO$_i$-berechenbar ist.

Aufgabe (4.5): (Semantik der GOTO-Programme) Bestimmen Sie die Funktionen $f_P^{(1)}, f_P^{(2)}, f_P^{(3)}$ und $f_P^{(4)}$ für das folgende Programm P:

```
1 X1:=X1+1
2 if X2=0 goto 6
3 X2:=X2-1
4 X1:=X1+1
5 if X4=0 goto 2
6 if X3=0 goto 10
7 if X1=0 goto 10
8 X1:=X1-1
9 if X4=0 goto 7
10 stop
```

4.3 Zur Stärke von WHILE und LOOP

Rückblick: Der in 4.2 konstruierte WHILE-Interpreter für Turing-Maschinen zur Berechnung einer n-stelligen Funktion hat die Gestalt

1. LOOP-Program mit `X1`,...,`Xn` und ≤ 5 Hilfsvariablen

2. `while X1>0 do` LOOP-Programm zur Turing-Maschinen-Simulation mit `X1`, `X2`, `X3` und ≤ 5 Hilfsvariablen

3. LOOP-Programm zur Herstellung des Ergebnisswertes in `X1` ebenfalls mit ≤ 5 Hilfsvariablen

Durch den Transfer WHILE$\to$WHILE$_0$$\to$GOTO$\to$Turing-Maschine$\to$WHILE folgt der

WHILE-Normalformensatz: Ist $f : \mathbb{N}^n - \to \mathbb{N}$ WHILE-berechenbar oder GOTO-berechenbar, so auch durch ein WHILE-Programm mit nur

(a) einer `while`-Anweisung

(b) einer festen Schachtelungstiefe der `loop`-Anweisungen

(c) höchstens mit den Variablen `X1`,...,`Xn` und ≤ 5 Hilfsvariablen.

(Bei der Berechnung einstelliger Funktionen kann man mit mehr technischem Aufwand die Anzahl der Variablen auf 2 reduzieren.)

Die `while`-Anweisung unter (a) ist nicht verzichtbar; sie ist zur Berechnung nicht-totaler Funktionen nötig. Kann man auf sie bei der Berechnung totaler Funktionen verzichten? Wir zeigen, daß LOOP-Programme nicht ausreichen. Der Beweis geht (in anderer Terminologie) auf Ackermann 1928, zurück.

Definition: Die *Ackermann-Funktion* $\alpha : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ ist definiert durch

(1) $\alpha(0, y) = y + 1$

(2) $\alpha(x + 1, 0) = \alpha(x, 1)$

(3) $\alpha(x + 1, y + 1) = \alpha(x, \alpha(x + 1, y))$

Beispiel: zur Auswertung gemäß (1),(2),(3):

$$\begin{aligned}
\alpha(1,3) &= \alpha(0, \alpha(1,2)) \\
&= \alpha(0, \alpha(0, \alpha(1,1))) \\
&= \alpha(0, \alpha(0, \alpha(0, \alpha(1,0)))) \\
&= \alpha(0, \alpha(0, \alpha(0, \alpha(0,1)))) \\
&= \alpha(0, \alpha(0, \alpha(0,2))) \\
&= \alpha(0, \alpha(0,3)) \\
&= \alpha(0,4) \\
&= 5
\end{aligned}$$

Wir zeigen mit dem nächsten Lemma, daß α total und berechenbar ist:

Lemma: *Für alle x, y führt die Anwendung von (1),(2),(3) nach endlich vielen Schritten auf eine Zahl, kurz: „$\alpha(x, y)$ ist definiert“.*

Beweis: durch Induktion über x (im Induktionsschritt mit Induktion über y):
$x = 0$: Zu zeigen: $\forall y : \alpha(0, y)$ ist definiert. Dies gilt nach (1)
I.V. $\forall y : \alpha(x, y)$ ist definiert, I.B. $\forall y : \alpha(x + 1, y)$ ist definiert zeige I.B. induktiv über y:
$y = 0 : \alpha(x + 1, 0) = \alpha(x, 1)$, nach I.V. definiert. Nach i.v. sei $\alpha(x + 1, y)$ definiert.
i.b.: $\alpha(x+1, y+1)$ ist definiert. $\alpha(x+1, y+1) = \alpha(x, \alpha(x+1, y))$ ist definiert nach i.v. und I.V. □

Wir zeigen nun, daß α stärker als jede LOOP-berechenbare Funktion wächst. Die Auswertungen für kleine x deuten dies schon an:

Bemerkung: *Es gilt:*

$\alpha(0, y) > y$	$\alpha(1, y) > y + 1$	$\alpha(2, y) > 2y$
$\alpha(3, y) > 2^y$	$\alpha(4, y) > 2^{2^{\cdot^{\cdot^{2}}}}\Big\}{y-mal}$	$\alpha(5, 4) > 10^{10.000}$

Satz von Ackermann: Die Ackermann-Funktion α ist nicht LOOP-berechenbar.

Als Vorbereitung für den Satz von Ackermann dient folgendes **Abschätzungslemma:**

(a) $\alpha(x,y) > y$

(b) $\alpha(x,y+1) > \alpha(x,y)$ (Monotonie im 2. Argument)

(c) $\alpha(x+1,y) \geq \alpha(x,y+1)$

(d) $\alpha(x+1,y) > \alpha(x,y)$ (Monotonie im 1. Argument)

(e) $\alpha(x+2,y) > \alpha(x,2y)$

Beweis:

(a) Induktion über x: $\forall y : \alpha(x,y) > y$

$x = 0$: $\alpha(0,y) = y+1 > y$, I.V. $\forall y : \alpha(x,y) > y$, I.B. $\forall y : \alpha(x+1,y) > y$.

Beweis durch Induktion über y:

$y = 0$: $\alpha(x+1,0) = \alpha(x,1) > 1$ (wegen (a)) > 0, i.v. $\alpha(x+1,y) > y$, i.b. $\alpha(x+1,y+1) > y+1$, $\alpha(x+1,y+1) = \alpha(x,\alpha(x+1,y)) > \alpha(x+1,y) \geq y+1$ nach I.V. und i.v.

(b) siehe Übungen

(c) siehe Übungen

(d) $\alpha(x+1,y) \geq \alpha(x,y+1) > \alpha(x,y)$ mit (c),(b).

(e) Induktion über y:

$y = 0$: $\alpha(x+1,0) > \alpha(x,0) = \alpha(x,2\cdot 0)$ (mit (d))

$y+1$: $\alpha(x+1,y+1) = \alpha(x+1,\alpha(x+2,y)) > \alpha(x+1,\alpha(x,2y)) \geq \alpha(x,\alpha(x,2y)+1) \geq \alpha(x,2y+2) = \alpha(x,2(y+1))$ (Beim ersten $<$ geht I.V. und (b) ein, dann (c), dann (a) und (b).)

□

Notationen:

1. Für ein LOOP-Programm P, $m \geq i(P)$, fassen wir $[\![P]\!]$ als Funktion $[\![P]\!] : \mathbb{N}^m \to \mathbb{N}^m$ auf. Schreibe $[\![P]\!](\bar{x})$ für entsprechenden Wert in $\mathbb{N}^m$.

2. Zu $\bar{x} = (x_1,\ldots,x_m)$ sei $\max\bar{x} = \max x_1,\ldots,x_m$

Ackermann-Lemma: Sei $m \geq 1$. Zu jedem LOOP-Programm P mit $i(P) \leq m$ ex. $k_P \in \mathbb{N}$ mit $\max([\![P]\!](\bar{x})) < \alpha(k_P, \max\bar{x})$.

Beweis des Satzes von Ackermann mit dem Lemma:
Annahme: α sei durch das LOOP-Programm P berechnet, etwa mit Variablen `X1`,...,`Xm`. Wähle k_P zu P gemäß dem Lemma. Dann:

$$\alpha(x,y) = f_P^{(2)}(x,y) \leq \max[\![P]\!](x,y,0,\ldots,0) < \alpha(k_P, \max x,y,0,\ldots,0).$$

Für $x = y = k_P$ folgt ein Widerspruch.

Beweis des Ackermann-Lemmas:
induktiv über den Aufbau der LOOP-Programme P: Wir arbeiten oBdA ohne $*$-Wertzuweisung. (Ersatz von `Xi:=Xj*Xk` möglich durch `Xi:=0; loop Xk begin Xi:=Xi+Xj end`.) Sei P eine Wertzuweisung. Wähle $k_P := 3$. Beachte: Ist $\bar{x}$ gegeben, so ist

$$\left.\begin{array}{r} x_i \pm x_j \\ x_i \pm 1 \end{array}\right\} \leq 2\max\bar{x} + 1.$$

Also ist (unter Verwendung des Abschätzungslemmas):

$$\max [\![P]\!](\bar{x}) \leq 2\max\bar{x} + 1 < \alpha(0, 2\max\bar{x} + 1) \leq \alpha(1, 2\max\bar{x}) < \alpha(3, \max\bar{x}).$$

$\mathtt{P} = \mathtt{P_1;P_2}$: I.V. liefert k_1, k_2 mit $\max([\![P_1]\!](\bar{x})) < \alpha(k_1, \max\bar{x})$. Wir setzen $k := \max(k_1, k_2), k_P := k + 2$. Dann:

$$\begin{array}{rcl} \max [\![P_1; P_2]\!](\bar{x}) & = & \max [\![P_2]\!]([\![P_1]\!](\bar{x})) \\ & < & \alpha(k_2, \max [\![P_1]\!](\bar{x})) \\ & < & \alpha(k_2, \alpha(k_1, \max\bar{x})) \\ & < & \alpha(k, \alpha(k+1, \max\bar{x})) \\ & = & \alpha(k+1, \max\bar{x} + 1) \\ & \leq & \alpha(k+2, \max\bar{x}) \\ & = & \alpha(k_P, \max\bar{x}) \end{array}$$

$\mathtt{P} = \mathtt{if\ Xi} > \mathtt{0\ then\ P_1\ else\ P_2}$ I.V. liefert k_1, k_2, wie oben; setze $k_P = \max(k_1, k_2)$.

$$\max [\![P]\!](\bar{x}) = \left\{ \begin{array}{l} \text{für } x_i > 0 : \max [\![P_1]\!](\bar{x}) < \alpha(\alpha(k_P, \max\bar{x}) \\ \text{für } x_i = 0 : \max [\![P_2]\!](\bar{x}) < \alpha(\alpha(k_P, \max\bar{x}) \end{array} \right.$$

$\mathtt{P} = \mathtt{loop\ Xi\ begin\ P_1\ end}$. I.V. liefert k_1 mit $\max [\![P_1]\!](\bar{x}) < \alpha(k_1, \max\bar{x})$. Wir setzen $k_P = k_1 + 3$ und zeigen die Hilfsbehauptung

$$\forall y : \max [\![P_1]\!]^y(\bar{x}) < \alpha(k_1 + 1, \max\bar{x} + y).$$

Beweis induktiv: $y = 0$: $\max [\![P_1]\!]^0(\bar{x}) = \max\bar{x} < \alpha(k_1, \max\bar{x} + 0)$
$y + 1$:

$$\begin{array}{rcl} \max [\![P_1]\!]^{y+1}(\bar{x}) & = & \max [\![P_1]\!]([\![P_1]\!]^y(\bar{x})) \\ & < & \alpha(k_1, \max [\![P_1]\!]^y(\bar{x})) \text{ (nach i.v.)} \\ & < & \alpha(k_1, \alpha(k_1 + 1, \max\bar{x} + y)) \text{ (nach I.V.)} \\ & \leq & \alpha(k_1 + 1, \max\bar{x} + (y+1)) \text{ (nach (3))}. \end{array}$$

Beweis der Induktionsbehauptung:

$$\begin{array}{rcl} \max [\![P]\!](\bar{x}) & = & \max [\![P_1]\!]^{x_i}(\bar{x}) \\ & < & \alpha(k_1 + 1, \max\bar{x} + x_i) \text{ (nach Hilfsbehauptung)} \\ & \leq & \alpha(k_1 + 1, 2\max\bar{x}) \\ & < & \alpha(k_1 + 3, \max\bar{x}) \\ & = & \alpha(k_P, \max\bar{x}). \end{array}$$

Zum Schluß geben wir eine Funktion an, die noch wesentlich schneller wächst als die Ackermann-Funktion: nämlich eine *Funktion, die stärker wächst als jede (streng monotone) berechenbare Funktion.* Es handelt sich um die „busy beaver"-Funktion.

Definition: Ein *Biber* ist eine Turing-Maschine über $\Sigma = \Gamma = \{b\!\!\!/, |\}$, die, auf das leere Band angesetzt, stoppt. Konvention: *q a b stop* ist eine erlaubte Turingzeile.
Eine Turing-Maschine heißt *fleißiger Biber* (*busy beaver*), wenn sie Biber ist und unter den Bibern gleicher Zustandszahl bei Termination die maximale Strichzahl auf dem Band liefert. Die busy beaver-Funktion BB ist definiert durch $BB(x)$=Anzahl der gelieferten Striche eines fleißigen Bibers mit x Zuständen.

Bemerkung: *Es gilt $BB(x) < BB(x+1)$, denn ist $\mathcal{A}$ ein fleißiger Biber mit x Zuständen, so druckt folgende Turing-Maschine mit einem Zustand mehr als $\mathcal{A}$ mehr als $BB(x)$ Striche: Arbeite wie $\mathcal{A}$ und gehe bei „stop" in den Zusatzzustand, nach rechts bis zum ersten $b\!\!\!/$, dort drucke $|$ und stoppe.*

Satz: (Rado) *Ist $f : \mathbb{N} \to \mathbb{N}$ eine (Turing-) berechenbare Funktion mit $f(x) < f(x+1)$, so gilt $f(x) < BB(x)$ für hinreichend großes x.*

Beweis: Sei f wie oben gegeben. Betrachte $g(x) := f(2x+2)$. Mit f ist auch g Turing-berechenbar, etwa durch $\mathcal{A}$ mit k Zuständen.
Zu $x \geq 0$ betrachte die Turing-Maschine $\mathcal{B}_x$, die auf dem leeren Band zunächst $|^x$ druckt und dann arbeitet wie $\mathcal{A}$.
Arbeitsweise von $\mathcal{B}_x : q_0 \to |q_1 \to \ldots \to |^{x-1}q_{x-1}$, dann Drucken von $|$ und Rückkehr vor die Striche mit q. $\mathcal{B}_x$ hat $x+1+k$ Zustände.
Für $x \geq k$ gilt: $f(2x+1) < f(2x+2) = g(x) \leq$Anzahl der Striche von $\mathcal{B}_x$ bei Termination angesetzt auf das leere Band$\leq BB(x+1+k) \leq BB(2x+1) < BB(2x+2)$.
Für $y \geq 2k+1$ gilt also $f(y) < BB(y)$ (für gerade y mit dem letzten „$<$", für ungerade y mit dem ersten „$<$" der obigen Abschätzung). $\square$

Folgerung: *BB ist nicht berechenbar.*

Einige Werte:

$$\begin{array}{rcl}
BB(1) & = & 2 \\
BB(2) & = & 4 \\
BB(3) & = & 6 \\
BB(4) & = & 13 \\
BB(5) & \geq & 1915\ [\geq 63 \cdot 10^{12} \text{ Turing-Maschinen}] \\
B(12) & > & 6 \cdot 4096^{\left.4096^{\cdot^{\cdot^{4096}}}\right\}^{4}_{164 mal}}
\end{array}$$

Übungen

Aufgabe (4.6): (Ackermann-Funktion) Geben Sie ein WHILE-Programm an, das die Ackermann-Funktion berechnet.

Aufgabe (4.7): (Abschätzungen zur Ackermann-Funktion): Beweisen Sie die die folgenden Behauptungen des Abschätzungslemmas:
(b) $\alpha(x, y+1) > \alpha(x, y)$
(c) $\alpha(x+1, y) \geq \alpha(x, y+1)$

Aufgabe (4.8): Zeigen Sie, daß $\alpha(3, y) > 2^y$ ist.

4.4 Programmkorrektheit und Hoare-Kalkül

Wir nutzen nun einen entscheidenden Vorteil der WHILE-Programmiersprache aus: Man kann dem Aufbau der Programme folgend deren Korrektheit zeigen. Darüber hinaus ergibt sich so eine mächtige Methodik der systematischen Programmentwicklung. Dies geschieht durch Eintragen von „Zusicherungen“ (Kommentaren) an geeigneten Stellen im Programm. Ein einführendes Beispiel betrifft ein Programm zur Berechnung des ggT (nach der Idee des euklidischen Algorithmus):

```
P_ggT :  X1 > 0  ∧  X2 > 0
         Y := X1;  Z := X2;
         Y > 0  ∧  Z > 0  ∧  ggT(X1, X2) = ggT(Y, Z)
         while Y ≠ Z do begin
           if Y < Z then Z := Z − Y else Y := Y − Z
           Y > 0  ∧  Z > 0  ∧  ggT(X1, X2) = ggT(Y, Z)
         end
         Y = Z,  ggT(X1, X2) = ggT(Y, Z)
         Y = ggT(X1, X2)
```

Basis des Korrektheitsbeweises ist eine elementare (rein zahlentheoretische) Bemerkung über die Funktion ggT:

ggT-Lemma:

(a) $\mathrm{ggT}(y, y) = y$ für $y > 0$

(b) für $y > z > 0 : \mathrm{ggT}(y, z) = \mathrm{ggT}(y - z, z)$

(c) für $z > y > 0 : \mathrm{ggT}(y, z) = \mathrm{ggT}(y, z - y)$

Hiermit erweist sich, daß die Zusicherungen im ggT-Programm „zutreffen“ und die Schlußzusicherung (Nachbedingung) erfüllt ist. Getrennt weisen wir die *Termination* nach:

Die Laufzeitschranke $t : \mathbb{N}^2 \to \mathbb{N}$ sei definiert durch $t(y, z) = \max(y, z)$. Es gilt:

1. $t(y, z) \geq 0$ für $y, z \in \mathbb{N}$
2. $y \neq z \Rightarrow t(y, z)$ wird im while-Schleifenkörper echt vermindert

Somit ist eine nichtabbrechende Folge von Ausführungen des Schleifenkörpers ausgeschlossen. Also terminiert P_{ggT}.
In der Programmkorrektheit spielen drei Sprachen eine Rolle:

1. *Programmiersprache*

2. *Zusicherungssprache*

3. *Korrektheitssprache*

Wir legen folgende Sprachen fest:

zu 1. Wir betrachten sogenannte *while-Programme*, die aufgebaut seien aus den WHILE-Wertzuweisungen mit der Verkettung (;), if-then-else, if-then, while-do, wobei die Bedingungen nach `if`, `while` die Form t_1 *op* t_2 haben mit $op \in \{=, \neq, <, >, \geq, \leq\}$; hierbei seien die Terme t_1, t_2 aus Variablen `X1`,`X2`,..., den Konstanten 0,1 mit $+, -, *$ aufgebaut.

Der Lesbarkeit halber lassen wir Variablen $Y, Y', Z, Z', \ldots$ zu. In Definitionen und Sätzen ziehen wir uns auf `X1`,`X2`,... zurück.

$\mathcal{P}_n$ sei die Menge der `while`-Programme mit höchstens den Variablen `X1`,...,`Xn`.

zu 2. Die Zusicherungssprache ist die *Sprache der Prädikatenlogik erster Stufe* mit folgender Signatur:

- Gleichheit $=$
- Konstanten $0, 1, 2, \ldots$ für die natürlichen Zahlen
- Funktionssymbole $+, -, *, \text{ggT}, \text{kgV}, !$,etc.
- Relationssymbole $\neq, <, >, \geq, \leq, even, odd$,etc.

jeweils mit ihrer natürlichen Interpretation über $\mathbb{N}$ (formal in der Struktur $\mathfrak{N} = (\mathbb{N}, 0^{\mathbb{N}}, 1^{\mathbb{N}}, \ldots, +^{\mathbb{N}}, -^{\mathbb{N}}, \ldots))$

Terme seien aus den Konstanten und Variablen `X1`,`X2`,... mit Funktionssymbolen aufgebaut. *Formeln* der Zusicherungssprache entstehen aus atomaren Formeln (Gleichungen $t_1 = t_2$, Relationsausdrücken $R\ t_1, \ldots, t_n$ mit Termen $t_1, \ldots, t_n$) mit $\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \forall, \exists$.

Formeln geben wir durch $\varphi, \psi, \chi, \ldots$ wieder. $\varphi(\mathtt{X1}, \ldots, \mathtt{Xn})$ bedeutet, daß höchstens `X1`,...,`Xn` in φ frei vorkommen. In üblicher Weise wird die Gültigkeitsbeziehung $(\mathfrak{N}, k_1, \ldots, k_n) \models \varphi(\mathtt{X1}, \ldots, \mathtt{Xn})$ definiert.

zu 3. Die Korrektheitssprache besteht aus sogenannten *Hoare-Formeln* der Form $\{\varphi\}P\{\psi\}$ mit Zusicherungen φ und ψ und dem while-Programm P.

Dabei ist φ die *Vorbedingung* und ψ die *Nachbedingung.*

$$\{\varphi(\mathtt{X1}, \ldots, \mathtt{Xn})\}P\{(\psi(\mathtt{X1}, \ldots, \mathtt{Xn})\}$$

deutet an, daß $\mathtt{X1}, \ldots, \mathtt{Xn}$ höchstens frei in φ, ψ sind und $P \in \mathcal{P}_n$.

Semantik der Hoare-Formeln: Eine Hoare-Formel $\{\varphi(\mathtt{X1},\ldots,\mathtt{Xn})\}P\{(\psi(\mathtt{X1},\ldots,\mathtt{Xn})\}$ gilt (oder trifft zu), falls folgendes zutrifft: Für $\bar{k}=(k_1,\ldots,k_n),\bar{l}=(l_1,\ldots,l_n)$: Wenn $(\mathfrak{N},k_1,\ldots,k_n)\models\varphi(\mathtt{X1},\ldots,\mathtt{Xn})$ und $[\![P]\!](k_1,\ldots,k_n)=(l_1,\ldots,l_n)$, dann gilt $(\mathfrak{N},l_1,\ldots,l_n)\models\psi(\mathtt{X1},\ldots,\mathtt{Xn})$.

Kurz: „Gilt φ vor Ausführung von P und terminiert P, so gilt ψ nach Ausführung von P.“

Beispiel:
$\{\mathtt{X1}>0\wedge\mathtt{X2}>0\}\mathrm{P}_{\mathrm{ggT}}\{\mathtt{y}=\mathrm{ggT}(\mathtt{X1},\mathtt{X2})\}$ gilt
$\{\mathtt{X1}=7\}\mathtt{X1}:=\mathtt{X1}+1\{\mathtt{X1}=8\}$ gilt
$\{\mathtt{X1}\leq 7\}\mathtt{X1}:=\mathtt{X1}+1\{\mathtt{X1}=8\}$ gilt nicht
$\{\mathtt{X1}=7\}\mathtt{X1}:=\mathtt{X1}+1\{\mathtt{X1}\leq 8\}$ gilt
$\{\mathtt{X1}<\mathtt{X1}\}\mathtt{X1}:=\mathtt{X1}+1\{\mathtt{X1}=8\}$ gilt
$\{\mathtt{X1}=1\}\mathtt{while\ X1}>0\ \mathtt{do\ X1}:=\mathtt{X1}+1\{\mathtt{X1}=0\}$ gilt

Hoare-Formeln behaupten die sogenannte *partielle Korrektheit*, nämlich unter der Annahme der Termination. *Totale Korrektheit* behauptet Termination (neben der Gültigkeit der Nachbedingung zu gegebener Vorbedingung).

Der Nachweis der Korrektheit eines Programms bedeutet: Beweis einer Hoare-Formel $\{\varphi\}P\{\psi\}$, gegebenfalls mit gesondertem Terminationsbeweis. Dies sind die Aufgaben der *Programmverifikation*.

Die Beweis-Methode folgt dem Programmaufbau: Führe den Nachweis passender Hoare-Formeln für <u>Teil</u>programme Q von P, ausgehend von den Wertzuweisungen bis hin zum Gesamtprogramm P.

Definition: Ein while-Programm P heißt *korrekt kommentiert*, wenn es durch Vorbedingung, Nachbedingung und weitere Zusicherungen so ergänzt ist, daß:

1. Jede darin auftretende Hoare-Formel $\{\varphi\}Q\{\psi\}$ gilt, wenn Q Teilprogramm von P und φ bzw. ψ Zusicherungen direkt davor bzw. danach sind.

2. Treten $\varphi(\mathtt{X1},\ldots,\mathtt{Xn}),\psi(\mathtt{X1},\ldots,\mathtt{Xn})$ direkt hintereinander auf, so gilt $\mathfrak{N}\models\forall\mathtt{X1},\ldots,\mathtt{Xn}(\varphi(\mathtt{X1},\ldots,\mathtt{Xn})\rightarrow\psi(\mathtt{X1},\ldots,\mathtt{Xn}))$.

Beispiel: P_{ggT} ist korrekt kommentiert.

Wir können die Verifikationsaufgabe auch auf rein syntaktische Regeln stützen. Ein solcher formaler Aufbau der Korrektheitsbeweise ist möglich mit den sogenannten *Hoare-Regeln*. Eine Hoare-Regel hat die Form

$$\frac{H_1,\ldots,H_k}{H}$$

mit Hoare-Formel H und Hoare-Formeln bzw. Zusicherungen $H_1,\ldots,H_k$ $(k\geq 0)$. Die Regel heißt *korrekt*, falls sie von gültigen (Hoare-) Formeln immer zu einer gültigen Hoare-Formel führt. (Eine Formel $\varphi(\mathtt{X1}, \ldots, \mathtt{Xn})$ der Zusicherungssprache heiße gültig, falls $\mathfrak{N} \models \forall \mathtt{X1}, \ldots, \mathtt{Xn} : \varphi(\mathtt{X1}, \ldots, \mathtt{Xn}))$

Beispiel:

$$\frac{\{\varphi\}P_1\{\psi\}, \{\psi\}P_2\{\chi\}}{\{\varphi\}P_1; P_2\{\chi\}} \quad \text{Kompositionsregel}$$

Korrektheit: Betrachte $\varphi = \varphi(\mathtt{X1}, \ldots, \mathtt{Xn}), \psi = \psi(\mathtt{X1}, \ldots, \mathtt{Xn}), P_1, P_2 \in \mathcal{P}_n$.
Es gelte: (1) $\forall \bar{k}, \bar{l}((\mathfrak{N}, \bar{k}) \models \varphi(\mathtt{X1}, \ldots, \mathtt{Xn}), [\![P_1]\!](\bar{k}) = \bar{l} \Rightarrow (\mathfrak{N}, \bar{l}) \models \psi(\mathtt{X1}, \ldots, \mathtt{Xn}))$ und (2) $\forall \bar{k}, \bar{l}((\mathfrak{N}, \bar{k}) \models \psi(\mathtt{X1}, \ldots, \mathtt{Xn}), [\![P_2]\!](\bar{k}) = \bar{l} \Rightarrow (\mathfrak{N}, \bar{l}) \models \chi(\mathtt{X1}, \ldots, \mathtt{Xn}))$.
Zeige: $\forall \bar{k}, \bar{l}((\mathfrak{N}, \bar{k}) \models \varphi(\mathtt{X1}, \ldots, \mathtt{Xn}), [\![P_1; P_2]\!](\bar{k}) = \bar{l} \Rightarrow (\mathfrak{N}, \bar{l}) \models \chi(\mathtt{X1}, \ldots, \mathtt{Xn}))$
Angenommen, die Voraussetzung gilt. Dann gilt $[\![P_1]\!](\bar{k}) = \bar{k}'$ für geeignetes $\bar{k}'$ mit $[\![P_2]\!](\bar{k}') = \bar{l}$. Nach (1) gilt $(\mathfrak{N}, \bar{k}') \models \psi(\mathtt{X1}, \ldots, \mathtt{Xn})$, nach (2) gilt folglich (mit $\bar{k}'$ für $\bar{k}$): $(\mathfrak{N}, \bar{l}) \models \chi(\mathtt{X1}, \ldots, \mathtt{Xn})$.

Die Regeln des Hoare-Kalküls:

Wertzuweisungsregel: $\dfrac{\varphi \rightarrow \psi(x/t)}{\{\varphi\}\mathtt{X := t}\{\psi\}}$

Kompositionsregel: $\dfrac{\{\varphi\}P_1\{\psi\}, \{\psi\}P_2\{\chi\}}{\{\varphi\}P_1; P_2\{\chi\}}$

if-then-else-Regel: $\dfrac{\{\varphi \wedge \beta\}P_1\{\psi\}, \{\varphi \wedge \neg\beta\}P_2\{\psi\}}{\{\varphi\}\mathtt{if}\ \beta\ \mathtt{then}\ \mathtt{P}_1\ \mathtt{else}\ \mathtt{P}_2\{\psi\}}$

if-then-Regel: $\dfrac{\{\varphi \wedge \beta\}P_1\{\psi\}, \varphi \wedge \neg\beta \rightarrow \psi}{\{\varphi\}\mathtt{if}\ \beta\ \mathtt{then}\ \mathtt{P}_1\{\psi\}}$

while-Regel: $\dfrac{\{\varphi \wedge \beta\}P\{\varphi\}}{\{\varphi\}\mathtt{while}\ \beta\ \mathtt{do}\ \mathtt{P}\{\varphi \wedge \neg\beta\}}$

Konsequenzregeln: $\dfrac{\{\varphi\}P\{\psi\}, \psi \rightarrow \chi}{\{\varphi\}P\{\chi\}} \quad \dfrac{\varphi \rightarrow \psi, \{\psi\}P\{\chi\}}{\{\varphi\}P\{\chi\}}$

Der Korrektheitsnachweis ist in allen Fällen einfach (wie im obigen Beispiel). Wie kann man den Terminationsbeweis, der jeweils hinzukommen muß, formal abstützen? Die Standardmethode benutzt den Begriff der Wohlordnung:

Definition: Eine lineare Ordnung $(A, <)$ heißt *Wohlordnung*, wenn es keine absteigende Kette unendlicher Länge in A gibt.

Beispiel:

(a) $(\mathbb{N}, <)$ ist Wohlordnung

(b) $(\mathbb{N}^2, <)$ mit $(i, j) < (i', j') :\Leftrightarrow i < i'$ oder $(i = i' undj < j')$ ist Wohlordnung

(c) $(\mathbb{Z}, <)$ ist keine Wohlordnung

Terminationslemma für while-Schleifen:

(1) $P \in \mathcal{P}_n$ terminiere für alle $\bar{k} \in \mathbb{N}^n$ mit $(\mathfrak{N}, \bar{k}) \models (\varphi \wedge \beta)(\bar{x})$ und

(2) es gebe eine Wohlordnung $(A, <)$ und eine Funktion $t : \mathbb{N}^n \to A$, die sogenannte *Laufzeitschranke*, mit $\{\varphi \wedge \beta \wedge t(\bar{k}) = a\}P\{\varphi \wedge t([\![P]\!](\bar{k})) < a\}$.

Dann gilt: `while` β `do P` terminiert für alle $\bar{k} \in \mathbb{N}^n$ mit $(\mathfrak{N}, \bar{k}) \models \varphi(\bar{x})$.

Beweis: Es gelte $(\mathfrak{N}, \bar{k}) \models \varphi(\bar{x})$ für ein $\bar{k} \in \mathbb{N}^n$. Falls $(\mathfrak{N}, \bar{k}) \models \neg\beta(\bar{x})$, dann terminiert `while` β `do P` sofort. Sei also $(\mathfrak{N}, \bar{k}) \models \beta(\bar{x})$. Annahme: [`while` β `do P`]$(\bar{k})$ ist undefiniert. Dann ist wegen (1) für alle $i \geq 0$ $[\![P]\!]^i(\bar{k})$ definiert, d.h. $[\![P]\!]^i(\bar{k}) = \bar{k_i}$ für geeignetes $\bar{k_i}$ und $(\mathfrak{N}, \bar{k_i}) \models (\varphi \wedge \beta)(\bar{x})$. Wegen (2) gibt es eine unendlich absteigende Kette $t(\bar{k}) = t(\bar{k_1}) > t(\bar{k_2}) > \ldots$, im Widerspruch zur Annahme, daß $(A, <)$ Wohlordnung ist. □

Aus der `while`-Regel und dem Terminationslemma folgt, wie man `while`-Schleifen verifizieren oder auch aufbauen kann, d.h. geeignete Zwischenzusicherungen finden kann. Basis ist folgendes *Schema der korrekten Kommentierung einer while-Schleife:*

Vorbedingung	$\{\varphi_0\}$
	Initialisierung
	$\{\varphi_1\}$
Invariante	$\{\varphi\}$
[*Laufzeitschranke* $t : \mathbb{N}^k \to \mathbb{N}$]	
	`while` β `do begin`
	$\{\varphi \wedge \beta\}$
	P_0
	$\{\varphi\}$
	`end`
	$\{\varphi \wedge \neg\beta\}$
Nachbedingung	$\{\psi\}$

Check-Liste zum Schema für die while-Schleife (mit Bedingung β und Schleifenkörper P_0):

(1) Zeige, daß die Invariante φ vor Erreichen der `while`-Schleife erfüllt ist.

(2) Zeige, daß $\{\varphi \wedge \beta\}P_0\{\varphi\}$ gilt.

(3) Zeige, daß $\varphi \wedge \neg\beta \to \psi$ gilt.

(4) Zeige, daß im Falle $\varphi \wedge \beta$ der t-Wert durch P_0 echt verkleinert wird.

Beispiel: Ein unübliches Multiplikationsprogramm

Vorbedingung	$\{X1 \geq 0 \wedge X2 \geq 0\}$
Initialisierung	`U:=X1; V:=X2; W:=0;`
Invariante	$\{V \geq 0 \wedge W + U * V = X1 * X2\}$
	$[t = V]$
	`while V>0 do begin`
	$\{W + U * V = X1 * X2 \wedge V \geq 0 \wedge V > 0\}$

```
        if odd(V) then begin V:=V-1; W:=W+U end
            else begin V:=V div 2; U:=U+U end
    end
```

$\{V = 0 \wedge V \geq 0 \wedge W + UV = X1 * X2\}$

Nachbedingung $\{W = X1 * X2\}$

Übungen

Aufgabe (4.9): Entwickeln Sie nach dem Schema der Vorlesung ein korrekt kommentiertes und terminierendes while-Programm, das folgender Spezifikation genügt: $\{X1 \geq X2\}P\{Z = (X1 - X2)^{X1}\}$

Aufgabe (4.10): Für die Funktion *bin*, gegeben durch $bin(N, K) = \binom{n}{k}$, gilt $bin(N, K) = (bin(N, K + 1) * (K + 1))$ div $(N - K)$, falls $0 \leq K \leq N$. Ergänzen Sie folgendes `while`-Programm (hier mit einer div-Anweisung) zu einem korrekt kommentierten Programm gemäß Vorlesung.

```
{0 ≤ K ≤ N}
X := N;  Y := 1;  B := 1;
while X ≠ K do begin
     B := (B * X) div Y;
     X := X − 1;
     Y := Y + 1
end
{B = bin(N, K)}
```

4.5 Programmentwicklung

Der methodische Rahmen bei der Programmentwicklung weist Analogien zur Mathematik auf:

Mathematik	Programmierung
Problem	Spezifikation
Satz	Programm
Beweis	Verifikation (z.B. im Hoare-Kalkül)

So wie zu jedem Satz ein Beweis gehört, so auf zu jedem Programm ein Korrektheitsbeweis. Wie in der Mathematik sind auch in der Informatik Beweismethoden der Schlüssel für die Programmentwicklung (entsprechend der Formulierung von Sätzen). Folgende Teilschritte bei der Programmentwicklung wollen wir untersuchen:

$$\overbrace{Spezifikation \rightarrow Zusicherungen \rightarrow Invari}^{\text{(a)}}\underbrace{anten \rightarrow Schleifenverifikation}_{\text{(b)}}$$

Zunächst zum Übergang (b) von einer Invarianten zur entsprechenden while-Schleife:

Spezifikation: $\{u \geq 0\} Q \{s = \sum_{i=0}^{n} i^2\}$
Invariante φ: $0 \leq j \leq n \wedge s = \sum_{i=0}^{j} i^2$

Methode zum Aufbau einer while-Schleife:

(1) Initialisiere so, daß die Invariante φ gilt, und finde eine Laufzeitschranke.

(2) Finde booleschen Ausdruck β, so daß $\varphi \wedge \neg\beta$ die Nachbedingung der `while`-Schleife ist (oder impliziert).

(3) Entwickele den Schleifenkörper P, so daß P den Laufzeitschrankenwert verkleinert und die Invariante (wieder) herstellt.

Anwendung im Beispiel: $\{n \geq 0\}$
`j:=0; s:=0;` (∗ Initialisierung ∗)
$\left\{0 \leq j \leq n \wedge s = \sum_{i=0}^{j} i^2\right\}$
$\{t = n - j\}$ Laufzeitschranke
`while j` $\neq$ `n do begin`
$\left\{0 \leq j \leq n \wedge s = \sum_{i=0}^{j} i^2 \wedge j \neq n\right\}$
    `j:=j+1;` (∗ Verkleinerung von t ∗)
    `s:=s+j*j;` (∗ Wiederherstellung der Invarianten ∗)
`end`
$\left\{0 \leq j \leq n \wedge s = \sum_{i=0}^{j} i^2 \wedge j = n\right\}$
$\{s = \sum_{i=0}^{n} i^2\}$

Nun zum entscheidenden Übergang (a) *von einer Nachbedingung zur Invarianten*: Die Invariante kann man durch geeignete Abschwächungen der Nachbedingung gewinnen. Wir klären zunächst den Begriff „Abschwächung“.

Definition: Sei $\varphi(\texttt{X1}, \ldots, \texttt{Xn})$ eine Formel erster Stufe mit den Variablen $\texttt{X1}, \ldots, \texttt{Xn}$.

- $R\varphi = \{\bar{k} \in \mathbb{N}^n \,|\, (\mathfrak{N}, \bar{k}) \models \varphi(\bar{x})\}$ ist die zu φ gehörende Relation (*arithmetische Relation* zu φ)
- Für $R_1, R_2 \subseteq \mathbb{N}^n$ heiße R_2 *stärker* als R_1, falls $R_2 \subseteq R_1$
- Für φ_1, φ_2 Formeln erster Stufe heißt φ_2 *stärker* als φ_1, falls $R\varphi_2$ ist stärker als $R\varphi_1$.

Bemerkung:

$$\varphi_2 \textit{ ist stärker als } \varphi_1 \Leftrightarrow \forall \bar{k} \in \mathbb{N}^n : (\mathfrak{N}, \bar{k}) \models \varphi_2(\bar{x}) \Rightarrow (\mathfrak{N}, \bar{k}) \models \varphi_1(\bar{x})$$

Also: „ist stärker als“ entspricht „impliziert“.

Beispiel:

$$\left.\begin{array}{r} \varphi_1 : x\text{ist durch 5 teilbar} \\ \varphi_2 : x\text{ist durch 15 teilbar} \end{array}\right\} \Rightarrow \varphi_2 \text{ ist stärker als } \varphi_1$$

Sind P und die Nachbedingung gegeben, müssen Vorbedingungen mindestens so stark wie die „schwächste Vorbedingung“ sein.

Definition: Sei $P \in \mathcal{P}_n, \psi$ Formel erster Stufe. Die Relation

$$wp(P, \psi) = \{\bar{k} | \text{wenn } [\![P]\!](\bar{k}) = \bar{l}, \text{ dann } (\mathfrak{N}, \bar{l}) \models \psi(\bar{x})\}$$

heißt *schwächste Vorbedingung* zu P und ψ (*weakest precondition*).
($wp(P, \psi)$ ist eine abstrakte Relation, zunächst nicht definiert durch eine Formel der Zusicherungssprache!)

Bemerkung: *(trivial)* $\{\varphi\}P\{\psi\}$ *ist genau dann gültig, wenn* $R\varphi$ *stärker ist als* $wp(P, \psi)$.

Daher kann man die Programmverifikation auch auf den wp-Operator gründen.

Nun zur Strategie für das Auffinden von Zwischenzusicherungen (insbesonere von Invarianten) durch geeignete Abschwächung der Nachbedingungen von `while`-Schleifen.

Methoden:

(A) Weglassen eines Konjunktionsgliedes oder Hinzufügen eines Disjunktionsgliedes.

(B) Ersatz von Konstanten durch Variablen mit geeignetem Wertebereich oder Aufspalten von zwei Vorkommen einer Variablen in zwei Variablen. Z.B. Übergang von
$$s = \sum_{i=0}^{n} i^2 \text{ zu } 0 \leq j \leq n \wedge s = \sum_{i=0}^{j} i^2.$$

(C) Vergrößerung des Wertebereichs einer Variablen, z.B. Übergang von $1 \leq i \leq 10$ zu $0 \leq i \leq 20$.

Wir diskutieren Beispiele zu (A) und (B).

zu (1) $\{x \geq 0\}P\{y = \lfloor\sqrt{x}\rfloor\}$ sei die Spezifikation.

Formaler: $\{x \geq 0\}P\{0 \leq y^2 \leq x < (y+1)^2\}$

Mögliche Abschwächung: Streichen des Konjunktionsgliedes $x < (y+1)^2$.

Dies ergibt $\varphi : 0 \leq y^2 \leq x$ als Kandidat für die Invariante. Natürliche Ansätze für Initialisierung und Laufzeitschranke sind:

1. Initialisierung: `y:=0`
2. Laufzeitfunktion: $t := x - y$

3. Suche β, so daß $\varphi : 0 \leq y^2 \leq x \wedge \neg\beta$ impliziert $x < (y+1)^2$. Also setze $\beta : (y+1)^2 \leq x$

Damit ist das `while`-Programm fertig:

$\{x \geq 0\}$
`y:=0;`
$\{0 \leq y^2 \leq x\}$
$[t = x - y]$
`while` $(y+1)^2 \leq x$ `do begin`
    $\{0 \leq y^2 \leq x \wedge (y+1)^2 \leq x\}$
    `y:=y+1`
    $\{0 \leq y^2 \leq x\}$
`end`
$\{0 \leq y^2 \leq x < (y+1)^2\}$

Der Schleifenkörper P_0 wurde so bestimmt, daß die Laufzeitschranke erniedrigt wird und die Invariante am Ende wieder gilt.

zu (2) Einführung einer neuen Variablen (mit Angabe eines Wertebereichs) z.B. durch Übergang von Konstanten zu Variablen oder Aufspalten zweier Vorkommen einer Variablen in zwei Variablen.

Spezifikation: $\{x \geq 0\}P\{y^2 \leq x < (y+1)^2\}$

Wähle neue Variable z für $y+1$, Wertebereich $y < z \leq x+1$. Dann wird z das nächste Quadrat nach x, also höchstens $z^2 \leq (x+1)^2$. Ansatz für Wertebereich von z: $z \leq x+1$.

Invariante: $y^2 \leq x < z^2 \wedge 0 \leq y < z \leq x+1$
Initialisierung: `y:=0; z:=x+1;`
Laufzeitschranke: $t = z - y$

Invariante$\wedge\neg\beta$ soll die Nachbedingung liefern. Ansatz: $\neg\beta : z = y+1$, also $\beta : z \neq y+1$

Damit ergibt sich bereits das (korrekt kommentierte) Programm:

$\{x \geq 0\}$
`y:=0; z:=x+1;`
$\{y^2 \leq x < z^2 \wedge 0 \leq y < z \leq x+1\}$
$[t = z - y]$
`while` $z \neq y+1$ `do begin`
    `d:=(y+z) div 2;`
    `if d*d≤x then y:=d else z:=d`
`end`
$\{y^2 \leq x < (y+1)^2\}$

Historische Stationen:
A. Turing 1950: „Checking a large routine“
R. Floyd 1966: „Assigning meanings to programs“Zusicherungen an Kontrollpunkten in GOTO-Programmen (Flußdiagramme)

C.A.R. Hoare 1969: „An axiomatic basis for computer programming“
Dijkstra u.a. in den 70er Jahren: Programmiermethodik
D. Gries: „The Science of Programming.“ (Lehrbuch).

Zur Tragweite des Hoare-Kalküls:

Definition: Eine Hoare-Formel heißt *herleitbar* relativ zur Arithmetik, wenn man sie ausgehend von geltenden Formeln der Zusicherungssprache (Arithmetik erster Stufe) mit den Hoare-Regeln herleiten kann.

Adäquatheitssatz für den Hoare-Kalkül: (ohne Beweis)
Für die Zusicherungssprache der Arithmetik erster Stufe und für `while`-Programme gilt: Eine Hoare-Formel $\{\varphi\}P\{\psi\}$ gilt genau dann, wenn sie relativ zur Arithmetik im Hoare-Kalkül herleitbar ist.

Wir sprechen bei „$\Leftarrow$“ von der *Korrektheit des Hoare-Kalküls*, bei „$\Rightarrow$“ von der *Vollständigkeit des Hoare-Kalküls.*

In die formalen Beweise mit dem Hoare-Kalkül gehen geltende Formeln der Arithmetik als unbewiesene Voraussetzungen ein (Wertzuweisungs- und Konsequenz-Regeln).
Kann man genau die geltenden Aussagen der Arithmetik ihrerseits durch einen Kalkül herleiten?
Die negative Antwort folgt aus dem „Gödelschen Unvollständigkeitssatz“; wir geben einen Beweis unter Benutzung des Hoare-Kalküls und des Halteproblems.

Satz: *Die Menge der geltenden Hoare-Formeln ist nicht aufzählbar.*

Beweis: Ist $P \in \mathcal{P}_m$, so stellen wir zunächst fest:

$$\{X1 = 0 \wedge \ldots \wedge Xm = 0\}P\{0 = 1\} \text{ gilt} \Leftrightarrow \text{für Eingabe 0 stoppt } P \text{ nicht.}$$

Nun nehmen wir an, die Menge der geltenden Hoare-Formeln sei aufzählbar. Dann ist auch die Menge der Hoare-Formeln der obigen Form, somit die Menge der `while`-Programme P, die bei Eingabe 0 nicht stoppen, aufzählbar. Da die Menge der P, die bei Eingabe 0 stoppen, ebenfalls aufzählbar ist (einfache Übung), ist entscheidbar, ob ein gegebenes P auf Eingabe 0 stoppt. Dies steht im Widerspruch zur Unentscheidbarkeit des Halteproblems. $\square$

Gödelscher Unvollständigkeitssatz:
Es gibt keinen Beweiskalkül, der genau die geltenden Formeln der Arithmetik erster Stufe herzuleiten gestattet.

Beweis: Wir nehmen an, es gäbe einen Beweiskalkül dafür. Dieser zusammen mit dem Hoare-Kalkül ergäbe nach dem Adäquatheitssatz einen Gesamtkalkül zur Herleitung genau der geltenden Hoare-Formeln. Dies liefert einen Aufzählungsalgorithmus für die geltenden Hoare-Formeln, im Widerspruch zum vorangehenden Satz. □

Übungen

Aufgabe (4.11): Bestimmen Sie für das `while`-Programm

```
{X1 ≥ 0 ∧ X2 ≥ 0}
X:=X1; Y:=X2; Z:=0;
while X≠ 0 do {φ } begin
  T:=0;
  while Y≠ T do {ψ } begin
    Z:=Z+1;
    T:=T+1
  end;
  X:=X-1
end
{Z = f(X1,X2)}
```

geeignete Schleifeninvarianten φ und ψ (mit Begründung) und geben Sie eine passende Funktion f in der letzten Zusicherung an. Zeigen Sie außerdem, daß das Programm terminiert.

Kapitel 5

Rekursive Programme

In diesem Abschnitt führen wir mit den „rekursiven Programmen“ eine Kernsprache der funktionalen Programmierung ein, so wie die WHILE-Programme eine Kernsprache des imperativen Stils bilden.

5.1 Definition, operationale Semantik, primitive Rekursion

Grundidee ist: Berechnungen sind nicht Transformationen von Speicherinhalten, sondern Auswertungen von Termen.

Einige **Beispiele** rekursiver Programme:

(1) $F(x) = \text{if } x = 0 \text{ then } 1 \text{ else } x * F(x-1)$

(2) $F(x) = \text{if } x = 0 \text{ then } 1 \text{ else } F(x+1)$

(3) $F(x) = \text{if } x > 100 \text{ then } x - 10 \text{ else } F(F(x+11))$

(4) $F(x,y) = \text{if } x = y \text{ then } x \text{ else if } x < y \text{ then } F(x, y-x) \text{ else } F(x-y, y)$

(5) $F(x,y) = \text{if } x = 0 \text{ then } y+1 \text{ else}$
$\quad\text{if } y = 0 \text{ then } F(x-1, 1) \text{ else } F(x-1, F(x, y-1))$

Man kann diese Gleichungen als Auswertungs- oder Reduktionsvorschriften (von links nach rechts) lesen. Wir geben jeweils eine Beispielauswertung:

(1) $F(4) \to 4 \cdot F(3) \to 12 \cdot F(2) \to 24 \cdot F(1) \to 24 \cdot F(0) \to 24$

(2) $F(4) \to F(5) \to F(6) \to \ldots$

(3) $F(99) \to F(F(110)) \to F(100) \to F(F(111)) \to F(101) \to 91$

(4) $F(21,35) \to F(21,14) \to F(7,14) \to F(7,7) \to 7$

(5) $F(1,3) \to F(0, F(1,2)) \to \ldots \to 5$

Wir sehen:

(1) definiert die Fakultätsfunktion

(2) die (bis auf $x = 0$ mit Wert 1) überall undefinierte Funktion

(4) den ggT

(5) die Ackermann-Funktion

Auf das Programm (3) kommen wir am Ende des Kapitels zurück.

In (5) ist das if-then-else geschachtelt. Wir erreichen eine Entschachtelung des if-then-else durch Einführung neuer Funktionssymbole:
(4) steht dann als Abkürzung für:

$$\begin{aligned}&F(x,y) = \text{if } x = y \text{ then } x \text{ else } F_1(x,y)\\&F_1(x,y) = \text{if } x < y \text{ then } F(x,y-x) \text{ else } F(x-y,y)\end{aligned}$$

Eine solche Folge von Gleichungen ist die allgemeine Form eines rekursiven Programms:

Definition: Ein *rekursives Programm* P hat die Form

$$\begin{array}{rcl} F_1\underbrace{(x_1,\ldots,x_{n_1})}_{\bar{x}_1} & = & \mathit{if}\ b_1\ \mathit{then}\ s_1(F_1\ldots F_m,\bar{x}_1)\ \mathit{else}\ t_1(F_1\ldots F_m,\bar{x}_1)\\ & \vdots & \\ F_m\underbrace{(x_1,\ldots,x_{n_m})}_{\bar{x}_m} & = & \mathit{if}\ b_m\ \mathit{then}\ s_m(F_1\ldots F_m,\bar{x}_m)\ \mathit{else}\ t_m(F_1\ldots F_m,\bar{x}_m)\end{array}$$

mit Funktionsvariablen F_i (jeweils der Stelligkeit n_i) und Variablen $x_1, x_2, \ldots$ für natürliche Zahlen; die b_i seien Bedingungen wie in `while`-Programmen, und die s_i, t_i seien Terme, die aus den $\bar{x}_i$, Konstanten für natürliche Zahlen mit $+, -, *$ und den Symbolen F_i aufgebaut sind.
Kurznotation: Wir lassen auch Gleichungen $F_i(\bar{x}_i) = s(\bar{F}, \bar{x}_i)$ zu, als Kürzel für if $0 = 0$ then $s(\bar{F}, \bar{x}_i)$ else $s(\bar{F}, \bar{x}_i)$.

Definition: Für ein Programm $P : F_i(\bar{x}_i) = \tau_i(\bar{F}, \bar{x}_i), (i = 1, \ldots, m)$ und für $\bar{k} = (k_1, \ldots k_{m_1})$ ist die zugehörige *Berechnung* als Termfolge $\sigma_0, \sigma_1, \ldots$ wie folgt definiert:
$\sigma_0 = F_1(k_1, \ldots, k_{m_1})$
σ_{i+1}=Auswertungsergebnis nach Substitution aller am weitesten innen stehenden Funktionsvariablen F_j in σ_i jeweils durch τ_j von P, falls σ_i noch ein F_j enthält; sonst terminiert die Berechnung mit σ_i.
(Wir identifizieren hier die Zahlen k_i mit den entsprechenden Konstanten.)

Beispiel:

$$\underbrace{\mathtt{F(4)}}_{\sigma_0} \to \mathtt{if\ 4{=}0\ then\ 1\ else\ 4 \cdot F(4-1)} \to \underbrace{\mathtt{4 \cdot F(3)}}_{\sigma_1}$$

Der Übergang von σ_i nach σ_{i+1} zerfällt also in zwei Etappen: die Einsetzung (Expansion) und die möglichst weitgehende Auswertung (Reduktion).

Definition: (*operationale Semantik* der rekursiven Programme) Die durch ein rekursives Programm P wie oben berechnete Funktion $f_P^{op} : \mathbb{N}^{n_1} - \to \mathbb{N}$ wird definiert durch

$$f_P^{op}(k_1, \ldots, k_{n_1}) = \begin{cases} k & \text{falls die Berechnung zu } P, \bar{k} \text{ terminiert mit } k \\ \perp & \text{sonst} \end{cases}$$

Beispiel: $f_{P_1}^{op}(k) = k!, \quad f_{P_4}^{op}(k, l) = \mathrm{ggT}(k, l)$

Dieser Formalismus der rekursiven Programme ist gleichwertig zur Programmiersprache der WHILE-Programme, also universell:

Satz: *Eine Funktion f ist durch ein rekursives Programm berechenbar $\Leftrightarrow$ f ist WHILE- oder GOTO-berechenbar.*

Beweis: „$\Rightarrow$“ klar mit einer unwesentlichen Anwendung der Churchschen These (intuitiv berechenbare Funktionen sind WHILE-berechenbar).

„$\Leftarrow$“ Als Ausgangspunkt betrachten wir GOTO-Programm P. Die Idee klärt ein Beispiel:

```
1 if X1=0 goto 4;
2 X1=X1-1;
3 if X2=0 goto 1;
4 stop
```

$f_P^{(1)}$ wird berechnet durch folgendes rekursive Programm Q:

$$\begin{aligned} F(x) &= G_1(x, 0) \\ G_1(x_1, x_2) &= \text{if } x_1 = 0 \text{ then } G_4(x_1, x_2) \text{ else } G_2(x_1, x_2) \\ G_2(x_1, x_2) &= G_3(x_1 - 1, x_2) \\ G_3(x_1, x_2) &= \text{if } x_2 = 0 \text{ then } G_1(x_1, x_2) \text{ else } G_4(x_1, x_2) \\ G_4(x_1, x_2) &= x_1 \end{aligned}$$

Die Korrespondenz zwischen P und Q sehen wir anhand der P-Konfigurationsfolge:

$$(1; 2, 0) \to (2; 2, 0) \to (3; 1, 0) \to (1; 1, 0) \to \ldots \to (4; 0, 0)$$

Die entsprechende Q-Auswertung lautet:

$$F(2) \to G_1(2, 0) \to G_2(2, 0) \to G_3(1, 0) \to \ldots \to G_4(0, 0) \to 0$$

Die Konfiguration $(i; j_1, j_2)$ entspricht also dem Term $G_i(j_1, j_2)$ Allgemeiner Fall: Zu P: $1\alpha_1, \ldots, k\alpha_k$, etwa mit `X1`,...,`Xm`, so daß P berechnet $f : \mathbb{N}^n \to \mathbb{N}$, oBdA $m \geq n$, bilde rekursives Programm Q wie folgt:

$F(x_1,\ldots,x_n) = G_1(x_1,\ldots,x_n,0,\ldots,0)$, sowie für $i = 1,\ldots,k$: $G_i(x_1,\ldots,x_m) = \tau_i$, wobei für $i = 1,\ldots,k-1$:

$$\tau_i = \begin{cases} G_{i+1}(x_1,\ldots,x_j+1,\ldots,x_m) & \alpha_i = \texttt{Xj := Xj + 1;} \\ G_{i+1}(x_1,\ldots,x_j-1,\ldots,x_m) & \alpha_i = \texttt{Xj := Xj - 1;} \\ \text{if } x_j = 0 \text{ then } G_l(x_1,\ldots,x_m) \text{ else } G_{i+1}(x_1,\ldots,x_m) & \alpha_i = \texttt{if Xj=0 goto l;} \end{cases}$$

und $\tau_k = x_1$.
Dann folgt mit einer einfachen Induktion über die Schrittzahl r: Von der Konfiguration $(1; k_1,\ldots,k_n,0,\ldots,0)$ erreicht P nach r Schritten die Konfiguration $(l; l_1,\ldots,l_m)$ $\Leftrightarrow$ Q liefert ausgehend von $F(k_1,\ldots,k_n)$ nach $r{+}1$ Schritten den Term $G_l(l_1,\ldots,l_m)$.
Damit ergibt sich (nach Definition von τ_k) die Behauptung. $\square$

Eine besondere Form der rekursiven Programme entspricht dem Schema, mit dem häufig zahlentheoretische Funktionen definiert werden, z.B. die Multiplikation durch $F(x,y) = \text{if } y = 0 \text{ then } 0 \text{ else } F(x,y{-}1){+}x$. Diese Form der „primitiven Rekursion“ ist Ausgangspunkt der sogenannten primitiv rekursiven Programme.

Definition: *Primitiv rekursive Programme* sind induktiv definiert durch folgende Bedingungen:

(1) $F(x_1,\ldots x_n) = x_i + 1$,
$F(x_1,\ldots,x_n) = x_i$,
$F(x_1,\ldots,x_n) = k$
sind primitiv rekursiv.

(2) aus einem primitiv rekursiven Programm P entsteht jeweils ein neues primitiv rekursives Programm durch Voranstellen einer der folgenden Gleichungen:

(a) $F(x_1,\ldots x_n) = x_i + 1$ oder x_i oder k (wie bei (1)).

(b) $F(x_1,\ldots,x_n) = G_0(G_1(x_1,\ldots,x_n),\ldots,G_m(x_1,\ldots,x_n))$

(c) $F(x_1,\ldots,x_n,y) = \text{if } y = 0 \text{ then } G(x_1,\ldots,x_n)$
$\text{else } H(y-1,x_1,\ldots,x_n,F(x_1,\ldots,x_n,y-1))$

wobei $G_0,\ldots,G_m,G,H$ von passender Stellenzahl seien und in P vorkommen. Im 2. Fall spricht man von *Einsetzung*, im 3. Fall von *primitiver Rekursion.*

f heißt primitiv rekursiv, falls $f = f_P^{op}$ für ein primitiv rekursives Programm P.

Beispiel: Die Potenzfunktion ist primitiv rekursiv. Ein primitiv rekursives Programm konstruiert man ausgehend von drei Zeilen für die konstanten Funktionen 0,1 und für eine dreistellige Nachfolgerfunktion (siehe Zeilen [4],[5],[6], gebildet gemäß Teil (1) und (2a) der obigen Definition). Nun werden sukzessiv drei Zeilen gemäß (2c) der Definition vorangestellt; sie legen die Summe $H_1(z,x,y) = x + y$ (Zeile [3]), das Produkt $H(z,x,y) = x \cdot y$ (Zeile [2]) und schließlich die Potenz (Zeile [1]) fest.

[1] $F(x,y) = \text{if } y = 0 \text{ then } G_1(x) \text{ else } H(y-1,x,F(x,y-1))$

[2] $H(z,x,y) = \text{if } y = 0 \text{ then } G_0(x) \text{ else } H_1(y-1,x,H(x,y-1))$

[3] $H_1(z, x, y) = \text{if } y = 0 \text{ then } x \text{ else } H_2(y-1, x, H_1(x, y-1))$

[4] $G_0(x) = 0$

[5] $G_1(x) = 1$

[6] $H_2(z, x, y) = y + 1$

Aus der Form der primitiv rekursirven Programme geht hervor, daß ihre Auswertung immer terminiert. Ohne Beweis notieren wir folgenden Äquivalenzsatz:

Satz: $f : \mathbb{N}^n \to \mathbb{N}$ *ist primitiv rekursiv* $\Leftrightarrow$ f *ist LOOP-berechenbar*

5.2 Fixpunktsemantik

Wir ordnen nun einem rekursiven Programm

$$\begin{aligned} P : F_1(\bar{x}_1) &= \tau_1(\bar{F}, \bar{x}_1) \\ &\vdots \\ F_m(\bar{x}_m) &= \tau_m(\bar{F}, \bar{x}_m) \end{aligned}$$

eine Semantik zu, die nicht auf einen Auswertungsprozeß zurückgreift. Schlüssel hierzu ist eine Halbordnung auf Argumenten und Funktionen (durch die Beziehung $\sqsubseteq$, mit der intuitiven Bedeutung „ist höchstens so definiert wie").

$\mathbb{N}_\perp := \mathbb{N} \cup \{\perp\}$. $\perp$ steht für „undefiniert".
Für $x, y \in \mathbb{N}_\perp$ gelte $x \sqsubseteq y$, falls $x = \perp$ (und $y \in \mathbb{N}_\perp$ beliebig) oder $x \in \mathbb{N}$ und $x = y$.
$(x_1, \ldots, x_n) \sqsubseteq (y_1, \ldots, y_n) :\Leftrightarrow x_i \sqsubseteq y_i$ für $i = 1, \ldots, n$.

Beispiel: $(n = 3) : (\perp, \perp, \perp) \sqsubseteq (1, 1, \perp) \sqsubseteq (1, 1, 5)$, aber nicht $(1, 1, \perp) \sqsubseteq (2, 1, \perp)$.

Definition: $f : \mathbb{N}_\perp^n \to \mathbb{N}$ heißt *monoton*, falls

$$(x_1, \ldots, x_n) \sqsubseteq (y_1, \ldots, y_n) \Rightarrow f(x_1, \ldots, x_n) \sqsubseteq f(y_1, \ldots, y_n).$$

Zu $f_0 : \mathbb{N}^n \to \mathbb{N}_\perp$ sei die *strikte Erweiterung* $f : \mathbb{N}_\perp^n \to \mathbb{N}_\perp$ definiert durch:

$$f(\bar{x}) = \begin{cases} f_0(\bar{x}) & \bar{x} \in \mathbb{N}^n \\ \perp & \bar{x} \in \mathbb{N}_\perp^n \setminus \mathbb{N}^n \end{cases}$$

(Nichtdefiniertheit von Argumenten führt zu Nichtdefiniertheit des Wertes.)

Bemerkung: *Strikte Erweiterungen sind monoton.*

$[\mathbb{N}_\perp^n \to \mathbb{N}_\perp]$ sei die Menge der monotonen Funktionen von $\mathbb{N}_\perp^n$ nach $\mathbb{N}_\perp$.

Bemerkung: *Sei P ein rekursives Programm wie oben und $\tau_j(F_1, \ldots, F_m, \bar{x}_j)$ die rechte Seite der j-ten Gleichung ($j = 1, \ldots, m$). Dann definiert τ_j ein Funktional*

$$|\tau_j| : [\mathbb{N}_\perp^{n_1} \to \mathbb{N}_\perp] \times \ldots \times [\mathbb{N}_\perp^{n_m} \to \mathbb{N}_\perp] \to [\mathbb{N}_\perp^{n_j} \to \mathbb{N}_\perp],$$

gegeben durch: $|\tau_j|(f_1, \ldots, f_m)$ ist die Funktion mit

$$|\tau_j|(f_1, \ldots, f_m)(\bar{k}) = \tau_j(f_1, \ldots, f_m, \bar{k}).$$

Entsprechend definiert $\bar{\tau} = (\tau_1, \ldots, \tau_m)$, das Tupel der rechten Seiten von P, ein Funktional

$$|\tau| : [\mathbb{N}_\perp^{n_1} \to \mathbb{N}_\perp] \times \ldots \times [\mathbb{N}_\perp^{n_m} \to \mathbb{N}_\perp] \to [\mathbb{N}_\perp^{n_1} \to \mathbb{N}_\perp] \times \ldots \times [\mathbb{N}_\perp^{n_m} \to \mathbb{N}_\perp],$$

wobei

$$|\tau|(f_1, \ldots, f_m) = (|\tau_1|(f_1, \ldots, f_m), \ldots, |\tau_m|(f_1, \ldots, f_m)).$$

Beispiel: Sei P das Programm $F_1(x, y) = \text{if } x < y \text{ then } F_1(x, y - x) \text{ else } F_2(x, y)$
Wir wollen τ_1 anwenden auf das Paar (f_1, f_2) mit
f_1=die strikte Erweiterung von $+$ auf $\mathbb{N}_\perp^2$,
f_2=die strikte Erweiterung von $*$ auf $\mathbb{N}_\perp^2$.
Es gilt $|\tau_1|(f_1, f_2)$=if $x < y$ then $x + (y - x)$ else $x * y$, also

$$|\tau_1|(f_1, f_2)(k, l) = \begin{cases} l & k < l \\ k * l & k \geq l \\ \perp & k = \perp \text{ oder } l = \perp \end{cases}$$

Die nicht-operationale Semantik für ein rekursives Programm P beruht auf folgender Idee: P formuliert Bedingungen an ein Funktionentupel $(f_1, \ldots, f_m)$, nämlich, daß es ein Fixpunkt des zugehörigen Operators $|\bar{\tau}|$ sein soll. Wir werden für jedes beliebige Programm P einen kanonischen Fixpunkt garantieren und als Bedeutung von P ansehen (Fixpunkt-Semantik).

Definition: $(f_1, \ldots, f_m)$ heißt *Fixpunkt* von $|\bar{\tau}|$, falls $|\bar{\tau}|(f_1, \ldots, f_m) = (f_1, \ldots, f_m)$.

Beispiel: Im letzten Beispiel ist (f_1, f_2) kein Fixpunkt von $|(\tau_1)|$

Beispiel: Sei P: $F(x) = \text{if } x = 0 \text{ then } 1 \text{ else } x \cdot F(x - 1) = \tau(F, x)$. Also hat $|\bar{\tau}|$ die Form $|\bar{\tau}| : [\mathbb{N}_\perp \to \mathbb{N}_\perp] \to [\mathbb{N}_\perp \to \mathbb{N}_\perp]$ Sei $f \in [\mathbb{N}_\perp \to \mathbb{N}_\perp]$ gegeben durch

$$f(x) = \begin{cases} 2^x & x \in \mathbb{N} \\ \perp & x = \perp \end{cases}$$

Es gilt:

$$|\bar{\tau}|(f) = g \text{ mit } g(x) = \begin{cases} 1 & x = 0 \\ x \cdot 2^{x-1} & x > 0 \\ \perp & x = \perp \end{cases}$$

Es ist $f \neq g$, d.h. f ist kein Fixpunkt von $|\bar{\tau}|$. Sei

$$f(x) = \begin{cases} x! & x \in \mathbb{N} \\ \perp & x = \perp \end{cases}, |\bar{\tau}|(f) = g \text{ mit } g(x) = \begin{cases} 1 & x = 0 \\ x \cdot (x-1)! & x > 0 \\ \perp & x = \perp \end{cases},$$

f ist Fixpunkt von $|\bar{\tau}|$.

Bemerkung: *Es gilt für ein rekursives Programm P wie oben:*

$$\begin{aligned} (f_1, \ldots, f_m) \text{ „erfüllt } P\text{“} &\Leftrightarrow f_1(\bar{x}_1) = \tau_1(\bar{f}, \bar{x}_1), \ldots, f_m(\bar{x}_m) = \tau_m(\bar{f}, \bar{x}_m) \\ &\Leftrightarrow (f_1, \ldots, f_m) \text{ ist Fixpunkt von } |\bar{\tau}|. \end{aligned}$$

Wir bauen unseren Fixpunkt auf von der überall undefinierten Funktion aus. Es sei: $\perp^{(n)} : \mathbb{N}_\perp^n \to \mathbb{N}_\perp$ definiert durch $\perp^{(n)}(x_1, \ldots, x_n) = \perp$.

Bemerkung: $\perp^{(n)}$ *ist monoton.*

Definition:

(1) Für $f, g \in [\mathbb{N}_\perp^n \to \mathbb{N}]$ gelte $f \sqsubseteq g :\Leftrightarrow \forall \bar{x} \in \mathbb{N}_\perp^n : f(\bar{x}) \sqsubseteq g(\bar{x})$

(2) Für $f_i, g_i \in [\mathbb{N}_\perp^n i \to \mathbb{N}_\perp]$, $1 \leq i \leq m$ gelte

$$(f_1, \ldots, f_m) \sqsubseteq (g_1, \ldots, g_m) :\Leftrightarrow \forall i \in 1, \ldots, m : f_i \sqsubseteq g_i.$$

Bemerkung: $\perp^{(n)} \sqsubseteq f$ *für alle* $f \in [\mathbb{N}_\perp^n \to \mathbb{N}_\perp]$

Definition: Für alle $\forall i \in \mathbb{N}$ sei $f_i \in [\mathbb{N}_\perp^n \to \mathbb{N}_\perp]$ und es gelte $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots$. Dann sei die Funktion $\text{lub}(f_i)$ definiert durch

$$\text{lub}(f_i)(\bar{x} = \begin{cases} k & \text{ex. } i_0 \text{ mit } f_{i_0}(\bar{x}) = k \\ \perp & \forall i \in \mathbb{N} : f_i(\bar{x}) = \perp \end{cases}$$

Sie heißt *least upper bound* der Kette (f_i). Analog ist für eine Folge $\bar{f}_i$ von Funktionentupeln mit $\bar{f}_0 \sqsubseteq \bar{f}_1 \sqsubseteq \bar{f}_2 \sqsubseteq \ldots$ die Funktion $\text{lub}\bar{f}_i$ definiert.

Bemerkung: *(Rechtfertigung der Bezeichnung). Es gilt:*

(1) $\forall j \in \mathbb{N} : f_j \sqsubseteq \text{lub}(f_i)$ *(d.h.* $\text{lub}(f_i)$ *ist eine obere Schranke).*

(2) Falls ein g ex. mit $\forall j : f_j \sqsubseteq g$, dann $\text{lub}(f_i) \sqsubseteq g$.

Beweis:

(1) Falls $f_j(\bar{x}) = \perp$, dann ist $f_j(\bar{x}) \sqsubseteq \text{lub}(f_i)(\bar{x})$. Falls $f_j(\bar{x}) = k$, so ist nach Definition von $\text{lub}(f_i)$ und nach Definition von $\sqsubseteq$: $f_j(\bar{x}) \sqsubseteq \text{lub}(f_i)(\bar{x})$ (weil $\text{lub}(f_i)(\bar{x}) = k$).

(2) Es gelte für alle j: $f_j \sqsubseteq g$; sei $\bar{x} \in \mathbb{N}_\perp^n$ beliebig.
Falls $\mathrm{lub}(f_i)(\bar{x}) = \perp$, so $\forall j : f_j(\bar{x}) = \perp$, also $\mathrm{lub}(f_i)(\bar{x}) \sqsubseteq g(\bar{x})$.
Falls $\mathrm{lub}(f_i)(\bar{x}) = k$, so $\exists i_0 : f_{i_0}(\bar{x}) = k$, also $g(\bar{x}) = k$, d.h. $\mathrm{lub}(f_i)(\bar{x}) \sqsubseteq g(\bar{x})$,
Insgesamt $\mathrm{lub}(f_i) \sqsubseteq g$.

□

Fixpunktsatz: Sei P ein rekursives Programm. Dann hat $|\bar{\tau}|$ bezüglich der Halbordnung $\sqsubseteq$ einen kleinsten Fixpunkt, nämlich $(f_1, \ldots, f_m) = \mathrm{lub}(\bar{f}_i)$ mit

$$\bar{f}_i = |\bar{\tau}|^i(\perp^{n_1}, \ldots, \perp^{n_m}).$$

Vor dem Beweis ein Beispiel zur Berechnung der Funktionen $|\bar{\tau}|^i(\perp^{n_1}, \ldots, \perp^{n_m})$: Wir betrachten P: $F(x) = \text{if } x = 0 \text{ then } 1 \text{ else } x \cdot F(x-1)$, also

$$\tau = \tau_1 = \text{ if } x = 0 \text{ then } 1 \text{ else } x \cdot F(x-1).$$

Wir berechnen $|\tau^i|$ für $i = 0, 1, 2, \ldots, :$,
$f_0 = |\tau|^0(\perp^{(1)}) = \perp^{(1)}, \quad f_1 = |\tau|^1(\perp^{(1)}), \quad f_2 = |\tau|^2(\perp)$.

$$f_1(x) = \begin{cases} 1 & x = 0 \\ \perp & \text{sonst} \end{cases} \qquad f_2(x) = \begin{cases} 1 & x = 0 \\ 1 & x = 1 \\ \perp & \text{sonst} \end{cases}$$

$$f_3(x) = \begin{cases} 1 & x = 0 \\ 1 & x = 1 \\ 2 & x = 2 \\ \perp & \text{sonst} \end{cases} \qquad f_4(x) = \begin{cases} 1 & x = 0 \\ 1 & x = 1 \\ 2 & x = 2 \\ 6 & x = 3 \\ \perp & \text{sonst} \end{cases}$$

per Induktion: $f_i(x) = \begin{cases} x! & x < i \\ \perp & \text{sonst} \end{cases}$, somit $\mathrm{lub} f_i = x!$.
Also approximiert die Kette $f_0 \sqsubseteq f_1 \sqsubseteq \ldots$ die Fakultätsfunktion, die Fixpunkt von P ist. Dieses Schema wird auch im allgemeinen Fall angewandt.

Nun zum Beweis des Fixpunktsatzes. Wir erweitern zuerst den Begriff der Monotonie von Funktionen auf Funktionale.

Definition:

(1) Ein Funktional $|\bar{\tau}|$ heißt *monoton* $:\Leftrightarrow$ aus $(f_1, \ldots, f_m) \sqsubseteq (g_1, \ldots, g_m)$ folgt $|\bar{\tau}|(f_1, \ldots, f_m) \sqsubseteq |\bar{\tau}|(g_1, \ldots, g_m)$.

(2) $|\bar{\tau}|$ heißt *stetig* $:\Leftrightarrow$ aus $\bar{f}_0 \sqsubseteq \bar{f}_1 \sqsubseteq \bar{f}_2 \sqsubseteq \ldots$ folgt $|\bar{\tau}|(\mathrm{lub}(\bar{f})_i) = \mathrm{lub}(|\bar{\tau}|(f_i))$.

Lemma: *Sei $|\bar{\tau}|$ das Funktional eines rekursiven Programms, dann ist $|\bar{\tau}|$ monoton und stetig.*

Beweis: Induktion über den Aufbau des Funktionals $|\bar{\tau}|$. □

Beweis des Fixpunktsatzes:

(1) Die Funktionentupel $|\bar{\tau}|^i(\bot^{(n_1)}, \ldots, \bot^{(n_m)})$ bilden eine aufsteigende Kette. Induktion über i:

$i = 1$: $(\bot^{(n_1)}, \ldots, \bot^{(n_m)}) \sqsubseteq |\bar{\tau}|^1(\bot^{(n_1)}, \ldots, \bot^{(n_m)})$, da für alle f gilt: $\bot^{(n)} \sqsubseteq f \in [\mathbb{N}_\bot^n \to \mathbb{N}_\bot]$.

$i + 1$: I.B. $|\bar{\tau}|^i(\bot^{(n_1)}, \ldots, \bot^{(n_m)}) \sqsubseteq |\bar{\tau}|^{i+1}(\bot^{(n_1)}, \ldots, \bot^{(n_m)})$.

I.V. $|\bar{\tau}|^{i-1}(\bot^{(n_1)}, \ldots, \bot^{(n_m)}) \sqsubseteq |\bar{\tau}|^i(\bot^{(n_1)}, \ldots, \bot^{(n_m)})$. Aus der Monotonie von $|\bar{\tau}|$ folgt die Behauptung.

(2) Sei $\bar{f}_i = |\bar{\tau}|^i(\bot^{(n_1)}, \ldots, \bot^{(n_m)})$. Gemäß (1) können wir $\text{lub}(\bar{f}_i)$ bilden. Wir zeigen, daß $\text{lub}(\bar{f}_i)$ Fixpunkt von $|\bar{\tau}|$ ist.

$$\begin{aligned}
|\bar{\tau}|(\text{lub}(\bar{f}_i)) &= |\bar{\tau}|(\text{lub}(|\bar{\tau}|^i(\bot^{(n_1)}, \ldots, \bot^{(n_m)}))) \\
&= \text{lub}(|\bar{\tau}|^{i+1}(\bot^{(n_1)}, \ldots, \bot^{(n_m)})) \text{ wegen } |\bar{\tau}| \text{ stetig} \\
&= \text{lub}(|\bar{\tau}|^i(\bot^{(n_1)}, \ldots, \bot^{(n_m)})) \text{ da } |\bar{\tau}|^i(\bot^{(n_1)}, \ldots, \bot^{(n_m)}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{aufsteigende Kette} \\
&= \text{lub}(f_i)
\end{aligned}$$

(3) Nun zeigen wir, daß $\text{lub}(f_i)$ <u>kleinster</u> Fixpunkt ist, d.h., daß für jeden Fixpunkt $\bar{g}$ gilt: $\text{lub}(f_i) \sqsubseteq \bar{g}$. Sei also $|\bar{\tau}|(\bar{g}) = \bar{g}$. Es genügt z.z., daß $\forall i$: $|\bar{\tau}|^i(\bot^{(n_1)}, \ldots, \bot^{(n_m)}) \sqsubseteq \bar{g}$.

$i = 0$: $(\bot^{(n_1)}, \ldots, \bot^{(n_m)}) \sqsubseteq \bar{g}$

$i + 1$: $|\bar{\tau}|^{i+1}(\bot^{(n_1)}, \ldots, \bot^{(n_m)}) = |\bar{\tau}|(|\bar{\tau}|^i(\bot^{(n_1)}, \ldots, \bot^{(n_m)})) \sqsubseteq |\bar{\tau}|(\bar{g}) \sqsubseteq \bar{g}$. Somit $\text{lub}\bar{f}_i \sqsubseteq \bar{g}$.

Diese Argumentation läßt sich ganz allgemein durchführen:

Definition: Eine Halbordnung $(D, \sqsubseteq)$ heißt *cpo* (complete partial order), falls jede Kette $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$ eine kleinste obere Schranke $d = \text{lub}(d_0, d_1, \ldots)$ hat und falls ein $\sqsubseteq$-minimales Element $\bot_D$ in D existiert.

Beispiel: $[\mathbb{N}_\bot^{n_1} \to \mathbb{N}_\bot] \times \ldots \times [\mathbb{N}_\bot^{n_m} \to \mathbb{N}_\bot]$ ist cpo mit $\sqsubseteq$ wie zuvor definiert.

Fixpunktsatz für cpo's: Sei $f : D \to D$ monoton und stetig. Dann hat f einen kleinsten Fixpunkt, nämlich $\text{lub}(f_i(\bot_D))$.

Der Beweis erfolgt analog zu den Schritten (1),(2),(3) wie oben.

Der Fixpunktsatz erlaubt es, einem rekursiven Programm P eine Funktion f_P^{fix} als „Bedeutung“ zuzuordnen, ohne auf Auswertungsschritte zurückzugreifen. Man setzt hierzu fest:

Definition: (*Fixpunktsemantik*) Für ein rekursives Programm P: $F_i(\bar{x}_i) = \tau_i(\bar{F}, \bar{x}_i)$ $(i = 1, \ldots, m)$ sei f_P^{fix} die erste Komponente f_1 des kleinsten Fixpunkts $(f_1, \ldots, f_m)$ des Funktionals $|\bar{\tau}|$.

Eine mühsame Argumentation zeigt, daß diese Fixpunktsemantik die gleichen Funktionen liefert wie die operationale Semantik:

Satz: *Für ein rekursives Programm P gilt $f_P^{fix} = f_P^{op}$.*

Für den Beweis sei auf das Lehrbuch von Loeckx, Sieber verwiesen.

Die Fixpunktsemantik erlaubt es in manchen Fällen, Aussagen über Funktionen, die durch rekursive Programme berechnet werden, direkter nachzuweisen als dies mit der operationalen Semantik möglich ist. Wir geben eine Illustration anhand des Beispielprogramms

$$(3)\ P\colon F(x) = \text{if } x > 100 \text{ then } x - 10 \text{ else } F(F(x+11))$$

Wir wollen eine partielle Korrektheitsbehauptung nachweisen:

Beispiel: Ist f_P^{fix} total, so gilt $f_P^{fix} = f_0$ mit $f_0(x) = \text{if } x > 100 \text{ then } x - 10 \text{ else } 91$.

Wir prüfen nach, daß f_0 ein Fixpunkt des Funktionals τ zum Programm (3) ist:

$$\begin{aligned}
|\tau|(f_0(x)) &= \text{if } x > 100 \text{ then } x - 10 \text{ else } f_0(\quad \underbrace{f_0(x-11)}_{\text{if } x>89 \text{ then } x+1 \text{ else } 91} \quad) \\
&= \text{if } x > 100 \text{ then } x - 10 \text{ else if } [\ldots] > 100 \text{ then } [\ldots] - 10 \text{ else } 91 \\
&= \text{if } x > 100 \text{ then } x - 10 \text{ else} \\
&\ \ (\text{im Fall } x = 100) \qquad \text{if } 101 \text{ then } 101 - 10 \text{ else } 91 \\
&\ \ (\text{im Fall } 89 < x < 100) \qquad \text{if } x + 1 > 100 \text{ then } x + 1 - 10 \text{ else } 91 \\
&\ \ (\text{im Fall } x \leq 89) \qquad \text{if } 91 > 100 \text{ then } 91 - 10 \text{ else } 91 \\
&= \text{if } x > 100 \text{ then } x - 10 \text{ else } 91 \\
&= f_0(x)
\end{aligned}$$

Nach dem Beweis des Fixpunktsatzes (Punkt (3)) muß also $f_P^{fix} \sqsubseteq f_0$ gelten. Falls, wie angenommen, f_P^{fix} total ist, folgt also $f_P^{fix} = f_0$.

Weitere Beispiele und Beweisprinzipien, die auf dem Fixpunktsatz beruhen, finden sich im Lehrbuch von Manna.

Literaturhinweise

1. Einführende Bücher

- W.S. Brainerd, L.H. Landweber, Theory of Computation, John Wiley and Sons, New York 1974
- M.D. Davis, E.J. Weyuker, Computability, Complexity and Languages, Academic Press, New York 1983
- E. Engeler, P. Läuchli, Berechnungstheorie für Informatiker, B.G. Teubner, Stuttgart 1988
- D. Harel, Algorithmics - The Spirit of Computing, Addison-Wesley, Reading, Mass. 1987
- H.R. Lewis, C. Papadimitriou, Elements of the Theory of Computation, Prentice-Hall, Englewood Cliffs, N.J. 1981
- J. Loeckx, K. Sieber, The Foundations of Program Verification, Wiley-Teubner, New York 1984
- Z. Manna, Mathematical Theory of Computation, McGraw-Hill, New York 1974
- R. McNaughton, Elementary Computability, Formal Languages and Automata, Prentice-Hall, Englewood Cliffs, N.J. 1982
- H.R. Nielson, F. Nielson, Semantics with Applications, Wiley, New York 1992
- U. Schöning, Theoretische Informatik - kurzgefaßt, BI-Wissenschaftsverlag, Mannheim 1992
- D. Wätjen, Theoretische Informatik, Oldenbourg-Verlag, München 1994
- K.W. Wagner, Theoretische Informatik, Springer-Verlag, Berlin 1994
- D. Wood, Theory of Computation, Harper & Row, New York 1987

2. Weiterführende Standardwerke

- E. Börger, Berechenbarkeit, Komplexität, Logik, Vieweg, Braunschweig 1985
- M.R. Garey, D.S. Johnson, Computers and Intractability, W.H. Freeman, New York 1979
- D. Gries, The Science of Programming, Springer, New York 1981
- M.A. Harrison, Introduction to Formal Language Theory, Addison-Wesley, Reading, Mass. 1978

- M. Hennessy, The Semantics of Programming Languages. An Elementary Introduction using Structural Operational Semantics, Wiley 1990
- J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, Reading, Mass. 1979; deutsche Ausgabe: Einführung in Automatentheorie, Formale Sprachen und Komplexitätstheorie, Addison-Wesley 1988
- J. v. Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vols. I and II, Elsevier Publ. Comp., Amsterdam.
- G. Winskel, The Formal Semantics of Programming Languages, MIT Press 1993

Index