Begleittexte
zu
Informatik III

P. H. Schmitt und Ch. Pape

Winter 1996/97 und 1997/98
11. Februar 1999

Vorwort

Dieser Text entstand aus den Vorlesungen zur theoretischen Informatik, die ich in den Wintersemestern 1996/97 und 1997/98 für Informatikstudenten im dritten Semester an der Universität Karlsruhe gehalten habe.

Die verfügbaren Lehrbücher zeigen eine große Übereinstimmung was den Kern des Stoffgebietes angeht: allgemeine Berechenbarkeit, Komplexitätstheorie, Automaten und formale Sprachen. Häufig kommt noch eine Einführung in die Logik zu diesem Kanon mit hinzu. Dieser Teil fehlt hier, da in Karlsruhe dieses Thema schon im ersten Semester aufgegriffen und in einer Wahlpflichtvorlesung "Formale Systeme" im fünften Semester vertieft wird. Der aus der Analyse der Lehrbücher gewonnene Stoffkanon stimmt außerdem erstaunlich gut überein mit den Empfehlungen der ACM aus dem Jahre 1991. Ausgehend von diesem bewährten Themenkatalog bleibt bei der Gestaltung einer Vorlesung zur theoretischen Informatik jedoch noch ein großer Spielraum was die Auswahl der einzelnen Lehreinheiten und ihre Präsentation angeht. In dem vorliegenden Buch haben wir vor allem auf drei Dinge Wert gelegt:

- Wir wollen den Zusammenhang deutlich machen zwischen theoretischen Konzepten und ihren Anwendungen in der Praxis.
- Wir versuchen durch den Einsatz multimedialer Hilfsmittel abstrakte Konzepte verständlicher darzustellen.
- Wir unterstützen die Erarbeitung des Stoffes durch eine große Zahl von Übungsaufgaben, größtenteils mit Lösungen.
- Vorlesungsfolien sind vorhanden.

Wir bedanken für die aufmerksame und kritische Durchsicht der ersten Entwürfe zu diesem Buch und die Hinweise auf inkorrekte oder mißverständliche Passagen bei Steffen Bloedt, Dr. Uwe Egly, Thomas Koller, Michael Maisch, Stephan Riese.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Berechenbarkeit

Als Einstieg in die Theorie der Berechenbarkeit versuchen wir uns in eine Zeit zurückzuversetzen, in der es den Begriff einer berechenbaren Funktion noch nicht gab. Ja mehr noch, in der man sich nicht vorstellen konnte, daß es so etwas wie prinzipielle Unberechenbarkeit für jetzt und alle Zeiten überhaupt gibt. Für eine solche Rückblende eignet sich das zehnte HILBERTsche Problem besonders gut. In den darauf folgenden drei Abschnitten stellen wir drei formale Definitionen des Begriffs einer berechenbaren Funktionen vor. Die erste orientiert sich an dem Modell einer Maschine, das die Berechnung durchführt. In der zweiten Definition wird dieser Aspekt in den Hintergrund geschoben und statt dessen die Programmierung einer Berechnung betont. Der dritte Zugang schließlich ist strikt mathematisch. Es wird induktiv eine Klasse von Funktionen definiert, die auf den ersten Blick nichts mit der Idee der Berechbarkeit zu tun haben. Die drei Definitionen sind äquivalent, das heißt sie führen zur selben Klasse berechenbarer Funktionen. Darauf gehen wir in Abschnitt 1.5 näher ein. Nach der Einführung einer formalen Definition sind wir in der Lage Unberechenbarkeitresultate in einem strikten mathematischen Sinn zu beweisen. Das wird zunächst für die Urzelle aller Unberechenbarkeitsresultate, das Halteproblem, gezeigt und dann auf weitere Probleme ausgedehnt.

1.1 Das 10. HILBERTsche Problem

Das zehnte HILBERTsche Problem ist eines von insgesamt 23 Problemen, die David Hilbert in seinem Vortrag „Mathematische Probleme“ vor dem 2. Internationalen Mathematikerkongreß 1900 in Paris vorgestellt hat. Die Probleme stammen aus fast allen damals bekannten Bereichen der Mathematik und haben einen starken Einfluß auf die Forschung in unserem Jahrhundert gehabt, siehe [6].

Orginalwortlaut:

> Eine diophantische Gleichung mit irgendwelchen Unbekannten und mit ganzen rationalen Zahlenkoeffizienten sei vorgelegt: *Man soll ein Verfahren angeben, nach welchem sich mittels einer endlichen Anzahl von Operationen entscheiden läßt, ob die Gleichung in den ganzen rationalen Zahlen lösbar ist.*

Die ganzen rationalen Zahlen, von denen in diesem Problem die Rede ist, sind die ganzen Zahlen, $\mathbb{Z}$, wie wir sie kennen. Hilbert benutzt die Bezeichnung *ganze rationale Zahlen* um sie von den ganzen algebraischen Zahlen zu unterscheiden.

Um ein bißchen ein Gefühl für das zehnte HILBERTsche Problem zu bekommen, betrachten wir vier Spezialfälle. Die Lösungen der ersten beiden Fälle sind mit elementaren Mitteln zu finden, während den beiden letzten tiefliegende mathematische Untersuchungen zugrundeliegen.

Beispiel 1

$$ax + by = c$$

wobei a, b, c ganze Zahlen sind. Notwendig und hinreichend für die Lösbarkeit dieser Gleichung in den ganzen Zahlen ist die Bedingung

$$c \textit{ ist teilbar durch den } ggT \textit{ von } a \textit{ und } b$$

Beispiel 2 (Gleichungen in einer Unbekannten)
Jede rationale Lösung von

$$x^n + a_1 x^{n-1} + \ldots + a_{n-1} x + a_n = 0$$

mit ganzzahligen Koeffizienten a_i ist bereits ganzzahlig.

Beispiel 3 (Ein Resultat von Euler) *Die Gleichung*

$$x^2 + py = a(x^2 \equiv a \mod p)$$

für eine Primzahl $p > 2$ und ganzzahligem a besitzt eine ganzzahlige Lösung genau dann, wenn

$$a^{\frac{p-1}{2}} - 1 \text{ durch } p \text{ teilbar ist.}$$

Beispiel 4 (Ein Resultat von Fermat) *Die Gleichung*

$$x_1^2 + x_2^2 + x_3^2 + x_4^2 = a$$

ist für jedes $a > 0$ lösbar.

Auf diese Ausgangslage kann man verschieden reagieren. Der Optimist wird darauf hinweisen, daß schon eine ganze Reihe von Spezialfällen des Problems gelöst sind und darauf vertrauen, daß der wissenschaftliche Fortschritt schließlich auch eine komplette Lösung hervorbringen werde. Dem Pessimist würde auffallen, daß der Beweisaufwand für einige der angeführten Spezialfälle gewaltig war und sich keine einheitliche Beweisidee herauskristallisiert und eher bezweifeln, daß eine Lösung gefunden werden kann. David Hilbert gehörte gewiß nicht zu den Pessimisten, wie die folgenden Zitate belegen.

Ein Originalzitat aus Hilberts Vortrag:

> Ist dieses Axiom von der Lösbarkeit eines jeden Problems eine dem mathematischen Denken allein charakteristische Eigentümlichkeit oder ist es vielleicht ein allgemeines dem inneren Wesen unseres Verstandes anhaftendes Gesetz, daß alle Fragen, die er stellt, auch durch ihn einer Beantwortung fähig sind?
>
> ...
>
> Diese Überzeugung von der Lösbarkeit eines jeden mathematischen Problems ist uns ein kräftiger Ansporn während der Arbeit; wir hören in uns den steten Zuruf: *Da ist das Problem, suche die Lösung. Du kannst sie durch reines Nachdenken finden; denn in der Mathematik gibt es kein Ignorabimus.*

zitiert nach [3, Seiten 33–34].

Die Entwicklung hat inzwischen gezeigt, daß die optimistische Einschätzung falsch war und die Wirklichkeit noch viel schlimmer ist als die pessimistische Haltung; das 10. HILBERTschen Problem ist prinzipiell unentscheidbar. Das wurde, allerdings erst 1970, in [26] bewiesen, siehe auch [27]. Nach der Lektüre der folgenden Abschnitte in diesem Kapitel wird der Leser diese Aussage besser verstehen und den Nachweis der Unentscheidbarkeit für einfache Probleme selbst nachvollziehen können.

1.2 Turingmaschinen

Wir beginnen mit der ältesten Formalisierung des Berechenbarkeitbegriffs. Sie wurde 1936 von Alan Turing veröffentlicht und besticht auch noch heute durch ihre Einfachheit und große Plausibilität. Wer etwas mehr über das Leben und die wissenschaftlichen Leistungen von Alan Turing erfahren möchte, den verweisen wir auf die Biographie von Andrew Hodges [22].

1.2.1 Basisdefinitionen

Turings Analyse einer Berechnung Wir zitieren aus der Originalarbeit von Alan M. Turing [39].

> Computing is normally done by writing certain symbols on paper.
> ...
> I assume that the computation is carried out on one-dimensional paper, i.e., on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. ...
> The behaviour of the computer at any moment is determined by the symbols which he is observing, and his 'state of mind' at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. ...
> We will also suppose that the number of states of mind which need be taken into account is finite.
> Let us imagine the operations performed by the computer to be split up into 'simple operations' which are so elementary that it is not easy to imagine them further divided. ...
>
> (a) Changes of the symbol on one of the observed squares.
>
> (b) Changes of one of the square observed to another square within L squares of one of the previously observed squares.
>
> It may be that some of these changes necessarily involve a change of state of mind. The most general single operation must therefore be taken to be one of the following:
>
> (A) A possible change (a) of the symbol together with a possible change of state of mind.
>
> (B) A possible change (b) of observed squares, together with a possible change of state of mind.

Diese Überlegungen führten Alan Turing zu der folgenden formalen Definition:

Definition 1 (Definition einer Turingmaschine)
Eine Turingmaschine *ist durch die Angabe der folgenden Bestimmungsstücke gegeben:*

1. *Eingabealphabet* Σ *(endlich),* $B(blank) \notin \Sigma$
2. *Bandalphabet* $\Gamma \supseteq \Sigma \cup \{B\}$ *(endlich)*
3. *Zustandsmenge* Q *(endlich)*
4. *Anfangszustand* $q_0 \in Q$
5. *Übergangsfunktion (partiell)*
   $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, U\}$

Wir fassen die Bestimmungsstücke einer Turingmaschine wie üblich in dem 5-Tupel $(\Sigma, \Gamma, Q, q_0, \delta)$ zusammen. Wir funktioniert jetzt eine solche Turingmaschine? Befindet sich die Turingmaschine $T = (\Sigma, \Gamma, Q, q_0, \delta)$ zu einem Zeitpunkt in einem Zustand $q \in Q$ und steht auf dem gerade fokussierten Feld des Bandes der Buchstabe $\gamma \in \Gamma$ und ist $\delta(q, \gamma) = (q', \gamma', L)$ (insbesondere setzt diese Annahme also voraus, daß $\delta(q, \gamma)$ definiert ist), dann wechselt T in den Zustand q' über, ersetzt auf dem fokussierten Feld γ durch γ' und bewegt den Lesekopf um ein Feld nach links. Ist der letzte Eintrag im Funktionswert von δ ein R oder U, dann wird der Lesekopf um eine Feld nach rechts bewegt, bzw. er bleibt unverändert.

Beispiel 5 (Beispiel einer Turingmaschine)

1. $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, B\}$
2. $Q = \{q_0, q_1, q_2, q_3, q_f\}$, $F = \{q_f\}$
3. δ *gegeben durch*

Q	Γ	Q	Γ	
q_0	0	q_0	0	R
q_0	1	q_1	1	R
q_0	B	q_2	B	R

Q	Γ	Q	Γ	
q_1	0	q_1	0	R
q_1	1	q_0	1	R
q_1	B	q_3	B	R

Q	Γ	Q	Γ	
q_2	0	q_f	1	U
q_2	1	q_f	1	U
q_2	B	q_f	1	U

Q	Γ	Q	Γ	
q_3	0	q_f	0	U
q_3	1	q_f	0	U
q_3	B	q_f	0	U

Die Aktionen der Turingmaschine T_{par} auf dem Eingabeband $B100101B$ sind in Abb. 1.1 zu sehen.

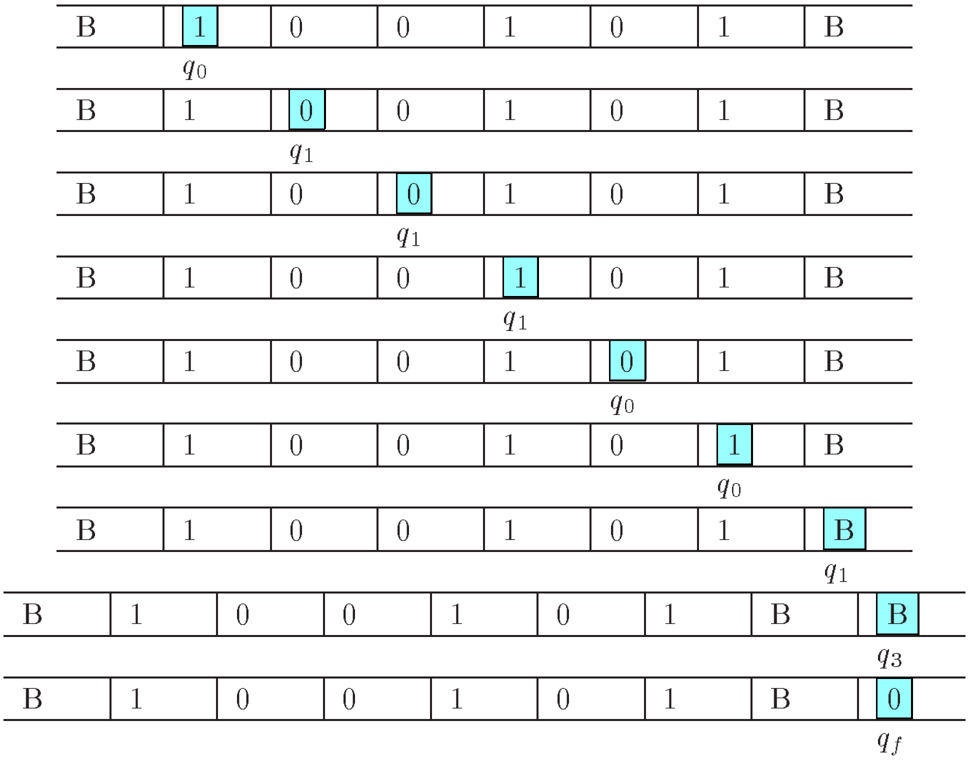


Abbildung 1.1: Eine Berechnung der Turingmaschine T_{par}

Berechnung von T_{par} in Kurznotation

$$
\begin{gathered}
\mathrm{B}q_0 100101\mathrm{B}\\
\mathrm{B}1q_1 00101\mathrm{B}\\
\mathrm{B}10q_1 0101\mathrm{B}\\
\mathrm{B}100q_1 101\mathrm{B}\\
\mathrm{B}1001q_0 01\mathrm{B}\\
\mathrm{B}10010q_0 1\mathrm{B}\\
\mathrm{B}100101q_1 \mathrm{B}\\
\mathrm{B}100101\mathrm{B}q_3 \mathrm{B}\\
\mathrm{B}100101\mathrm{B}q_f 0
\end{gathered}
$$

Um über Berechnungen von Turingmaschinen reden zu können, ist die folgende Definition hilfreich.

Definition 2 (Konfiguration)
Wörter der Form w_1qw_2 *mit* $w_i \in \Gamma^*$ *und* $q \in Q$ *heißen* Konfigurationen *oder* Augenblicksbeschreibungen *einer Turingmaschine.*

Die Funktionsweise der Turingmaschine T_{par} aus Beispiel 5 läßt sich jetzt wie folgt beschreiben: Gestartet in Bq_0wB mit $w \in \{0,1\}^*$ stoppt T_{par} in der Konfiguration $BwBq_fk$, wobei $k = 0$ falls w eine ungerade Anzahl von Einsen enthält und $k = 1$ sonst. Wir sagen, daß eine Turingmaschine *stoppt*, oder *terminiert* oder anhält, wenn sie ausgehend von einer vereinbarten Startkonfiguration eine Konfiguration $BqswB$ erreicht mit $s \in \Gamma$, $w \in \Gamma^*$, so daß $\delta(q,s)$ nicht definiert ist. Turingmaschinen modellieren allein den Berechnungsprozeß und verfügen über keinerlei Eingabe- oder Ausgaberoutinen. Die Eingabe wird zu Beginn der Berechnung in den Speicher, sprich auf das Band, beschrieben und nach dem Ende der Berechnung irgendwie auf dem Band abgelesen. Die Details einer solchen Eingabe und Ausgabe sind nicht von Bedeutung und deshalb wollen wir auch keine einheitliche Format dafür vorschreiben. Im Regelfall halten wir uns an die in der folgenden Definition festgelegte Vereinbarung.

Definition 3 (Turing-berechenbare Funktionen)
Wir nennen eine partielle Funktion $g : \Sigma^* \to \Sigma^*$ Turing-berechenbar*, wenn es eine Turingmaschine* T_g *gibt, die für jedes* $w \in \Sigma^*$*, aus der Startkonfiguration* Bq_0wB *heraus,*

- *in der Konfiguration* $BwBvq_fu$ *mit* $g(w) = v$ *terminiert, falls* $w \in \mathrm{Def}(g)$ *gilt oder*
- *nicht terminiert falls* $w \notin \mathrm{Def}(g)$ *gilt.*

Die Berechenbarkeit von Funktionen $g : \mathbb{N} \to \mathbb{N}$ *wird über dem Eingabealphabet* $\Sigma = \{|\}$ *erklärt, wobei die Zahl* n *durch eine Folge von* $n+1$ *Strichen dargestellt wird.*

Definition 4 (Turing-berechenbare Sprachen)
Eine Turingmaschine T *mit dem Eingabealphabet* Σ akzeptiert *ein Wort* $w \in \Sigma^*$*, falls* T *in der Anfangskonfiguration* Bq_0wB *gestartet schließlich terminiert.*

$$\{w \in \Sigma^* \mid T \textit{ akzeptiert } w\}$$

heißt die von T akzeptierte Sprache.

Varianten von Turingmaschinen Die bisher besprochene Basisdefinition für Turingmaschinen kann auf vielfältige Weise abgewandelt werden. Hier sind einige Beispiel dafür:

1. Turingmaschinen mit Endzustände.
   Wie aus der nachfolgenden Definition 5 hervorgeht handelt sich hier um eine geringfügige Modifikation der Basisdefinition, die hauptsächlich aus Bequemlichkeitsgründen benutzt wird oder um die Vergleichbarkeit mit anderen Maschinenmodellen herzustellen.

2. Mehrbändige Turingmaschinen.
   Diese können auf mehreren Bändern lesen und schreiben und auf jedem Band unabhängig den Schreib- und Lesekopf bewegen.

3. Mehrspurige Turingmaschine. Wie mehrbändige Turingmaschinen, die Bewegungen der Schreib- und Leseköpfe muß jedoch für alle Bänder gleich sein.

4. Das Band ist nur nach einer Seite unendlich erweiterbar. Siehe Abschnitt 1.2.2.

5. Nichtdeterministische Turingmaschinen. Siehe Abschnitt 1.2.4.

Definition 5 (Turingmaschinen mit Endzuständen)
Turingmaschinen mit Endzuständen sind wie in Definition 1 gegeben mit einem Zusatz. Es wird eine Teilmenge $F \subseteq Q$ von Zuständen ausgezeichnet. Die Elemente von F heißen Endzustände. Es wird verlangt, daß für jedes $q \in F$ und alle $\gamma \in \Gamma$ die Übergangsfunktion $\delta(q, \gamma)$ nicht definiert ist. Die Berechnung von Funktionen und die Akzeption von Sprachen für Turingmaschinen mit Endzuständen ist definiert wie im Fall ohne explizite Endzustände, wobei die Bedingungen „T hält an“ oder „T terminiert“ ersetzte werden durch „T erreicht einen Endzustand“.

1.2.2 Turingmaschinen mit einseitigem Band

In diesem Abschnitt betrachten wir stellvertretend für alle anderen eine der einfacheren Variante des Turingmaschinenmodells etwas ausführlicher.

Das modifizierte Modell nennen wir eine Turingmaschine mit einseitig unendlichem Band und benutzen zur ihrer Bezeichnung Symbole der Form $T_{1\infty}$. Die Modifikation besteht darin, daß diese Maschinen eine am weitesten links stehende Bandzelle besitzen, das Band sich also nur nach rechts hin unendlich fortsetzt. Wir wollen im folgenden stets annehmen, daß die Anfangszelle des Bandes durch ein spezielles Symbol, das nicht zum Eingabealphabet gehört, gekennzeichnet ist und diese Beschriftung durch die Aktionen von Turingmaschinen $T_{1\infty}$ nicht verändert wird. Wir benutzten dazu $\blacktriangleright$. Eine solche Kennzeichnung ist nicht unbedingt erforderlich aber für das Arbeiten mit einseitig unendlichen Turingmaschinen äußerst hilfreich. Es bleibt zu erklären, was geschehen soll, wenn eine Bewegung über die Anfangszelle hinaus nach links versucht wird. Um möglichst wenig gegenüber der Standarddefinition einer Turingmaschine ändern zu müssen, vereinbaren wir, daß in diesem Fall $T_{1\infty}$ einfach anhält. Ist in einem bestimmten Fall ein solches Verhalten unerwünscht, kann man es durch eine entsprechende Änderung des Turingmaschinenprogramms vermeiden.

Man könnte zunchst vermuten, daß durch das Fehlen der linken Hälfte des Bandes ein schwächeres Berechnungsmodell entsteht. Das ist nicht der Fall. Das modifizierte Modell gestattet nicht mehr aber auch nicht weniger Funktionen zu berechnen wir das ursprüngliche. Was diese Aussage präzise bedeutet und wie man sie streng beweist soll der Inhalt dieses Abschnitts sein.

Lemma 1
Jede Berechnung einer Turingmaschine mit zweiseitig unendlichem Band läßt sich auch auf einer Turingmaschine mit einseitig unendlichem Band durchführen.

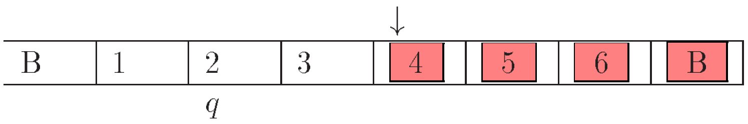
Die in Lemma 1 noch vage beschriebene Äquivalenz wollen wir konkretisieren zu der Aussage

Lemma 2
Sei $T = (\Sigma, \Gamma, Q, q_0, \delta)$ eine Turingmaschine mit zweiseitig unendlichem Band, welche die Funktion $g : \Sigma^* \rightarrow \Sigma^*$ berechnet. Dann gibt es eine Turingmaschine $T_1 = (\Sigma, \Gamma, Q_1, q_0, \delta_1)$ mit einseitig unendlichen Band, welche ebenfalls g berechnet.

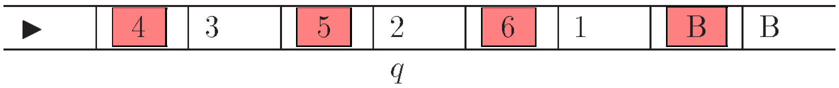
Beweis: Wird zurückgestellt bis nach dem Beweis von Lemma 3 auf Seite 11.

Natürlich gilt auch die umgekehrte Behauptung von Lemma 2: jede Funktion, die durch eine einseitig unendliche Turingmaschine berechnet wird, kann auch durch eine geeignete zweiseitig unendliche Turingmaschine berechnet werden. Aber das ist erstens intuitiv einsichtiger und auch technisch einfacher, so daß wir keine ausführliche formale Rechtfertigung dafür geben wollen.

Die Idee des ersten Beweises besteht darin ein zweiseitig unendliches Band



durch das einseitig unendliche Band



zu simulieren. Dabei ist $\downarrow$ eine Markierung für die Bandzelle, die in der Anfangskonfiguration fokussiert wurde. Formal definieren wir zu jeder Konfiguration

$$K = Bs_{-k} \dots s_{-1} \downarrow s_1 \dots qs_i \dots s_k B,$$

von T die Konfiguration

$$\mathrm{sim}(K) = \ \blacktriangleright s_1 s_{-1} \dots qs_i s_{-i} \dots s_k s_{-k} B$$

von T_1. Befindet sich der Schreib- und Lesekopf links der $\downarrow$-Markierung, also

$$K = Bs_{-k} \dots qs_{-i} \dots s_{-1} \downarrow s_1 \dots s_k B$$

so setzten wir

$$\mathrm{sim}(K) = \ \blacktriangleright s_1 s_{-1} \dots s_i q^{-1} s_{-i} \dots s_k s_{-k} B$$

Das neu einzuführende Zustandssymbol q^{-1} kodiert dabei die Information, daß in der Ausgangskonfiguration q links von $\downarrow$ auftrat. Das Symbol $\downarrow$ dient allein zur Markierung der Anfangszelle. Eine Bewegung von s_1 um einen Schritt nach links führt also zu Zelle s_{-1}. Ist die Konfiguration K bezüglich der von B verschiedenen Bandbeschriftung nicht symmetrisch mit Zentrum $\downarrow$, dann füllt man die Konfiguration durch $s_i = B$ oder $s_{-i} = B$ entsprechend auf. Die Anfangskonfiguration für die Turingmaschine T_1 zur Berechnung des Funktionswertes $g(w)$ für $w = w_1 \dots w_k$ ist jetzt

$$\mathrm{sim}(K_0) = \ \blacktriangleright q_0 w_1 B \dots w_{k-1} B w_k B$$

Ebenso ändert sich auch die Anweisung, wie der Funktionswert nach Terminierung aus der Bandinschrift abgelesen werden soll. Im einzelnen ist das schwierig zu beschreiben, aber die folgende intentionale Beschreibung vermittelt die Idee: von der fokussierten Zelle ausgehend (in der Konfiguration gekennzeichnet durch das Auftreten des Zustands q_f) liest man die Bandinschrift nach links bis zum nächsten B. Auf dem einseitigen Band heißt „lesen nach links“auf dem rechts von $\downarrow$ stehenden Teil „lesen jedes zweiten Feldes nach links“und auf dem links von $\downarrow$ stehenden Teil „lesen jedes zweiten Feldes nach rechts“.

Nach diesen einleitenden Erklärungen können wir jetzt zu gegebener zweiseitig unendlicher Turingmaschine $T = (\Sigma, \Gamma, Q, q_0, \delta)$ die einseitig unendliche Turingmaschine $T_1 = (\Sigma, \Gamma, Q_1, q_0, \delta_1)$ angeben:

$$
\begin{array}{rcl}
Q_1 & = & Q \cup \{q^{-1} \mid q \in Q\} \cup \\
 & & \{q_R \mid q \in Q\} \cup \{q_L \mid q \in Q\} \\
 & & \{q_R^{-1} \mid q \in Q\} \cup \{q_L^{-1} \mid q \in Q\}
\end{array}
$$

$$
\begin{array}{lcll}
\delta_1(q,s) & = & (q_1, s_1, U) & \text{falls} \quad \delta(q,s) = (q_1, s_1, U) \\
\delta_1(q^{-1},s) & = & (q_1^{-1}, s_1, U) & \text{falls} \quad \delta(q,s) = (q_1, s_1, U) \\
\delta_1(q,s) & = & (q_{1,R}, s_1, R) & \text{falls} \quad \delta(q,s) = (q_1, s_1, R) \\
\delta_1(q^{-1},s) & = & (q_{1,L}^{-1}, s_1, L) & \text{falls} \quad \delta(q,s) = (q_1, s_1, R) \\
\delta_1(q,s) & = & (q_{1,L}, s_1, L) & \text{falls} \quad \delta(q,s) = (q_1, s_1, L) \\
\delta_1(q^{-1},s) & = & (q_{1,R}^{-1}, s_1, R) & \text{falls} \quad \delta(q,s) = (q_1, s_1, L) \\
\delta_1(q_R,s) & = & (q, s, R) & \\
\delta_1(q_L,s) & = & (q, s, L) & \\
\delta_1(q_R^{-1},s) & = & (q^{-1}, s, R) & \\
\delta_1(q_L^{-1},s) & = & (q^{-1}, s, L) & \\
\delta_1(q_L,\blacktriangleright) & = & (q_R^{-1}, \blacktriangleright, R) & \\
\delta_1(q^{-1},\blacktriangleright) & = & (q, \blacktriangleright, R) &
\end{array}
$$

wobei stets $s \neq \blacktriangleright$ und $q, q_1 \in Q$ gilt.

Der Superskript -1 an einem Zustand zeigt an, daß sich die fokussierte Zelle des zweiseitig unendlichen Bandes links von der Anfangsmarkierung befindet, die Subskripte R und L dienen dazu, das jeweilige Zwischenfeld des einseitig unendlichen Bandes zu überbrücken. Mit dieser Definition von T_1 kann nun das folgende Lemma bewiesen werden:

Lemma 3 (Simulationslemma)
Seien die Turingmaschinen T und T_1 und die Übersetzung sim wie oben definiert, K_n die Konfiguration, die T nach n Schritten erreicht und K_m^1 die Konfiguration, die T_1 nach m Schritten erreicht. Dann gilt:

Für jedes $n \geq 0$ gibt es ein $m \geq n$ mit

$$\mathrm{sim}(K_n) = K_m^1.$$

Beweis: Wir beweisen den ersten Teil des Lemmas durch Induktion nach n. Für $n = 0$ können wir $m = 0$ wählen und die Gleichung $\mathrm{sim}(K_0) = K_0^1$ ist nach Vereinbarung richtig. Im Induktionsschritt von n nach $n+1$ sind eine ganze Reihe von Fällen zu untersuchen.

Wir betrachten zunächst einen einfachen Fall:

$$K_n = Bs_{-k} \ldots s_{-1} \downarrow s_1 \ldots s_{i-1} q s_i \ldots s_k B$$

und $\delta(q, s_i) = (q_1, t, R)$. Damit ist

$$K_{n+1} = Bs_{-k} \ldots s_{-1} \downarrow s_1 \ldots s_{i-1} t q_1 \ldots s_k B$$

Nach Induktionsvoraussetzung gibt es ein m mit $\mathrm{sim}(K_n) = K_m^1$. Nach Definition von *sim* ergibt das:

$$K_m^1 = \;\blacktriangleright s_1 s_{-1} \ldots q s_i s_{-i} \ldots s_k s_{-k} B$$

Nach Definition von δ_1 gilt $\delta_1(q, s_i) = (q_{1,R}, t, R)$ und somit

$$K_{m+1}^1 = \;\blacktriangleright s_1 s_{-1} \ldots s_{-i-1} t q_{1,R} s_{-i} \ldots s_k s_{-k} B$$

Aus der Tabelle für δ_1 lesen wir ab: $\delta_1(q_{1,R}, s_{-i}) = (q_1, s_{-i}, R)$, woraus

$$K_{m+2}^1 = \;\blacktriangleright s_1 s_{-1} \ldots s_{-i-1} t s_{-i} q_1 \ldots s_k s_{-k} B$$

folgt. Offensichtlich ist $\mathrm{sim}(K_{n+1}) = K_{m+2}^1$.

Ebenso einfach ist der Fall, in dem der Schreib- und Lesekopf sich nach links bewegt ohne die Anfangsmarkierung zu überqueren.

$$K_n = Bs_{-k} \ldots s_{-1} \downarrow s_1 \ldots s_{i-1} q s_i \ldots s_k B$$

und $\delta(q, s_i) = (q_1, t, L)$ und $i \geq 2$. Damit ist

$$K_{n+1} = Bs_{-k} \ldots s_{-1} \downarrow s_1 \ldots q_1 s_{i-1} t \ldots s_k B$$

Nach Induktionsvoraussetzung gibt es ein m mit $\mathrm{sim}(K_n) = K_m^1$. Nach Definition von *sim* ergibt das:

$$K_m^1 = \;\blacktriangleright s_1 s_{-1} \ldots q s_i s_{-i} \ldots s_k s_{-k} B$$

Nach Definition von δ_1 gilt $\delta_1(q, s_i) = (q_{1,L}, t, L)$ und somit

$$K_{m+1}^1 = \;\blacktriangleright s_1 s_{-1} \ldots s_{i-1} q_{1,L} s_{-i+1} t s_{-i} \ldots s_k s_{-k} B$$

Aus der Tabelle für δ_1 lesen wir ab: $\delta_1(q_{1,L}, s_{-i+1}) = (q_1, s_{-i+1}, L)$, woraus

$$K^1_{m+2} = \ \blacktriangleright s_1 s_{-1} \dots q_1 s_{i-1} s_{-i-1} t s_{-i} \dots s_{-k} B$$

folgt. Offensichtlich ist $\mathrm{sim}(K_{n+1}) = K^1_{m+2}$.
Wir betrachten noch einen weiteren interessanten Fall:

$$K_n = B s_{-k} \dots s_{-2} q s_{-1} \downarrow s_1 \dots s_k B$$

mit $\delta(q, s_{-1}) = (q_1, t, R)$. Also

$$K_{n+1} = B s_{-k} \dots s_{-2} t \downarrow q_1 s_1 \dots s_k B$$

Die Induktionsvoraussetzung liefert wieder ein m mit $\mathrm{sim}(K_n) = K^1_m$ und

$$K^1_m = \ \blacktriangleright s_1 q^{-1} s_{-1} \dots s_k s_{-k} B$$

Nach Definition von δ_1 gilt $\delta_1(q^{-1}, s_{-1}) = (q^{-1}_{1,L}, t, L)$. Somit erhalten wir

$$K^1_{m+1} = \ \blacktriangleright q^{-1}_{1,L} s_1 t \dots s_k s_{-k} B$$

Wegen $\delta_1(q^{-1}_{1,L}, s_1) = (q^{-1}_1, s_1, L)$ erhält man

$$K^1_{m+2} = \ q^{-1}_1 \blacktriangleright s_1 t \dots s_k s_{-k} B$$

Aus der Tabelle für δ_1 liest man ab $\delta_1(q^{-1}_1, \blacktriangleright) = (q_1, \blacktriangleright, R)$ woraus sich

$$K^1_{m+3} = \ \blacktriangleright q_1 s_1 t \dots s_k s_{-k} B$$

ergibt. Man sieht $\mathrm{sim}(K_{n+1}) = K^1_{m+3}$. Als vierten Fall betrachten wir noch:

$$K_n = B s_{-k} \dots s_{-2} s_{-1} \downarrow q s_1 \dots s_k B$$

mit $\delta(q, s_1) = (q_1, t, L)$. Also

$$K_{n+1} = B s_{-k} \dots q_1 s_{-1} \downarrow t s_2 \dots s_k B$$

Die Induktionsvoraussetzung liefert wieder ein m mit $\mathrm{sim}(K_n) = K^1_m$ und

$$K^1_m = \ \blacktriangleright q s_1 s_{-1} \dots s_k s_{-k} B$$

Nach Definition von δ_1 gilt $\delta_1(q, s_1) = (q_{1,L}, t, L)$. Somit erhalten wir

$$K^1_{m+1} = \ q_{1,L} \blacktriangleright t s_{-1} \dots s_k s_{-k} B$$

Wegen $\delta_1(q_{1,L}, \blacktriangleright) = (q_{1,R}^{-1}, \blacktriangleright, R)$ erhalten wir die nächste Konfiguration

$$K_{m+2}^1 = \quad \blacktriangleright q_{1,R}^{-1} t s_{-1} \ldots s_k s_{-k} B$$

Aus der Tabelle für δ_1 liest man weiter ab $\delta_1(q_{1,R}^{-1}, t) = (q^{-1}, t, R)$. Das führt zu:

$$K_{m+3}^1 = \quad \blacktriangleright t q_1^{-1} s_{-1} \ldots s_k s_{-k} B$$

und $\mathrm{sim}(K_{n+1}) = K_{m+3}^1$.

Der Beweis zeigt auch, daß man nicht generell $\mathrm{sim}(K_n) = K_{2n}^1$ behaupten kann. Man kann nur die Existenz eines m mit $\mathrm{sim}(K_n) = K_{2m}^1$ mit $2n \leq m \leq 3n$ behaupten.

■

Beweis von Lemma 2: Angenommen die zweiseitig unendliche Turingmaschine T terminiert auf der Eingabe $w = w_1 \ldots w_k$ und liefert das Ergebnis $v = v_1 \ldots v_r$. Ausgehend von der Konfiguration $K_0 = Bqw_1 \ldots w_k B$ erreicht also T die Konfiguration $K_n = XBv_1 \ldots v_r q_f uU$, so daß $\delta(q_f, u)$ nicht definiert ist. Nach Definition gibt es ein m mit $\mathrm{sim}(K_n) = K_m^1$. Nach Definition von *sim* steht auf dem fokussierten Feld der Buchstabe u und der Zustand ist q_f oder q_f^{-1}. Da $\delta(q_f, u)$ nicht definiert ist sind auch die Funktionswerte $\delta_1(q_f, u)$ und $\delta_1^{-1}(q_f, u)$ nicht definiert, d.h. T_1 terminiert auch. Die Definition von *sim* garantiert weiterhin, daß aus K_m^1 auch dasselbe Ergebnis abgelesen wird. Terminiert T mit dem Startwort w nicht, so gibt es nach dem Lemma beliebig große m, so daß K_m^1 definiert ist. Insbesondere kann kein K_m^1 eine terminierende Konfiguration sein.

■

Eine zweite Lösung

Hier zunchst die Idee: Sei die folgende Konfiguration der zweiseitig unendlichen Maschine gegeben. Dabei bezeichnet $\downarrow$ die Zelle, die zu Beginn der Berechnung im Fokus stand.

			$\downarrow$			
B	1	2	3	4	5	6	B

$$\begin{array}{|c|c|c|c|c|c|c|c|} \multicolumn{4}{c}{} & \multicolumn{1}{c}{\downarrow} \\ \hline B & 1 & 2 & 3 & 4 & 5 & 6 & B \\ \hline \multicolumn{2}{c}{} & \multicolumn{1}{c}{q} \end{array}$$

Wir *klappen* die Seite links von $\downarrow$ nach rechts um, so daß jetzt auf einigen Zellen zwei Symbole stehen, oben die Symbole, die schon vorher dort zu lesen waren und unten die Symbole aus den *umgeklappten* Zellen. Also so:

▶	$\binom{4}{3}$	$\binom{5}{2}$	$\binom{6}{1}$	B	B	B	B

$\binom{0}{q}$

Dabei können die Symbole $\binom{a}{b}$ als Zeichen eines neuen Alphabets aufgefaßt werden. Ebenso sind Zustände jetzt von der Form $\binom{0}{q}$ bzw. $\binom{q}{0}$, um festzuhalten, ob die fokussierte Zelle der zu simulierenden Maschine sich im linken bzw im rechten Teil des Bandes befindet.

Soweit die Idee. Formal würde man etwa wie folgt vorgehen. Sei eine Turingmaschine T mit zweiseitig unendlichem Band, Bandalphabet Γ, Zustandsmenge Q, Anfangszustand q_0, Finalzuständen F und Übergangsfunktion δ gegeben. Wir definieren eine Turingmaschine $T_{1\infty}$ mit einseitig unendlichem Band und

$$
\begin{array}{lcl}
\Gamma_{1\infty} & = & (\Gamma \times \Gamma) \cup \{B\} \\
Q_{1\infty} & = & (\{0\} \times Q) \cup (Q \times \{0\}) \\
\binom{q_0}{0} & = & \text{Anfangszustand} \\
F_{1\infty} & = & (\{0\} \times F) \cup (F \times \{0\}) \\
\delta_{1\infty} & & \text{wird später definiert}
\end{array}
$$

Zu jeder Konfiguration K von T definieren wir eine Konfiguration $\mathrm{sim}(K)$ von $T_{1\infty}$. Sei etwa

$$K = Bx_k \dots x_{r+1} q x_r \dots x_1 y_1 \dots y_m B$$

Die Bezeichnung wurde so gewählt, daß y_1 auf der Zelle steht, die zu Beginn der Berechnung fokussiert wurde. Dann ist

$$\mathrm{sim}(K) = \quad \blacktriangleright\, \sigma_1 \dots \sigma_r \begin{pmatrix} 0 \\ q \end{pmatrix} \sigma_{r+1} \dots \sigma_n B$$

wobei $n = max(k, m)$ und

$$\sigma_i = \begin{cases} \binom{y_i}{x_i} & \text{falls } 1 \leq i \leq k \text{ und } 1 \leq i \leq m \\ \binom{B}{x_i} & \text{falls } 1 \leq i \leq k \text{ und } m < i \\ \binom{y_i}{B} & \text{falls } k < i \text{ und } 1 \leq i \leq m \end{cases}$$

Wir sinde jetzt in der Lage die Übergangsfunktion zu beschreiben, dabei ist M ein Element der Menge $\{R, L, U\}$ der möglichen Bewegungen des Schreib- und Lesekopfes.

$$\delta_{1\infty}(\tbinom{q}{0}, \tbinom{y}{x}) \quad = \quad (\tbinom{q'}{0}, \tbinom{y'}{x}, M) \quad \text{falls } \delta(q, y) = (q', y', M).$$

$$\delta_{1\infty}(\binom{0}{q},\binom{y}{x}) \quad = \quad (\binom{0}{q'},\binom{y}{x'},M') \quad \text{falls } \delta(q,y)=(q',y',M) \text{ und}$$

$$M' = \begin{cases} R & \text{falls } M = L \\ L & \text{falls } M = R \\ U & \text{falls } M = U \end{cases}$$

$$\begin{aligned} \delta_{1\infty}(\binom{0}{q},\blacktriangleright) &= (\binom{q}{0},\blacktriangleright,R) \\ \delta_{1\infty}(\binom{q}{0},\blacktriangleright) &= (\binom{0}{q},\blacktriangleright,R) \end{aligned}$$

Ist K eine Konfiguration von T so bezeichnen wir mit $\delta(K,n)$ die Konfiguration die ausgehend von K in n Schritten erreicht wird. Also $\delta(K,0)=K$ und $\delta(K,n+1)$ ist die Nachfolgerkonfiguration von $\delta(K,n)$. Analog ist $\delta_{1\infty}(K,n)$ definiert.

Wir würden gerne für eine Anfangskonfiguration K_0 von T beweisen, daß $\mathrm{sim}(\delta(K_0,n))=\delta_{1\infty}(\mathrm{sim}(K_0),n)$ gilt, aber das ist nicht richtig. Die simulierende Maschine $T_{1\infty}$ braucht mehr Schritte als die Maschine T. Statt dessen beweist am leicht durch Induktion über n bzw m:

Lemma 4

1. *Für jedes $n>0$ gibt es ein $m\geq n$ mit*

$$\mathrm{sim}(\delta(K_0,n))=\delta_{1\infty}(\mathrm{sim}(K_0),m)$$

2. *Für jedes $m>0$ gibt es ein $n\leq m$ mit*

$$\mathrm{sim}(\delta(K_0,n))=\delta_{1\infty}(\mathrm{sim}(K_0),m)$$

Beweis: Bleibt dem Leser überlassen.

■

1.2.3 Der fleißige Biber

Dieser Abschnitt beschäftigt sich mit einem Spiel, einem intelligenten Zeitvertreib, der dennoch geeignet ist ein tieferes Verständnis für das Wesen der Berechenbarkeit zu wecken. Wir folgen hier dem Aufsatz [5] in [20].

Der Wettbewerb der *fleißigen Biber* (endl. busy beaver) wurde 1962 von Tibor Rado erfunden. Dabei geht es wieder um Turingmaschinen. Allerdings werden hier nur Turingmaschinen mit Bandalphabet $\{0,1\}$ betrachtet, die auf einem Band starten, das nur 0-en enthält. Anfangszustand ist q_1, Endzustand ist q_0

(wird nicht mitgezählt). Der Wettbewerb besteht darin zu einer festen Anzahl n von (aktiven) Zuständen eine Turingmaschine zu finden, die möglichst viele 1-en auf des Band schreibt und anhält. Wir nennen die dadurch induzierte Funktion BB(n) die Biberfunktion:

$$\mathrm{BB}(n) \quad = \quad \begin{array}{l}\text{die maximale Anzahl von 1-en, die eine terminierende}\\ \text{Turingmaschine mit } n \text{ Zuständen schreiben kann}\end{array}$$

Rados Einstiegsbeispiel, siehe Abbildung 1.2 schreibt 5 Einsen auf das leere Band und bleibt nach 21 Schritten stehen. Also gilt mindestens BB(3) ≥ 5 .

		gelesenes Zeichen 0	1
Zustand	1	1R2	1R0
	2	1L2	0R3
	3	1L3	1L1

Abbildung 1.2: Der fleißige Biber T_5^3 von T. Rado

Der frühere Rekordhalter in der Klasse mit 5 Zuständen druckte in 2 133 492 Schritten 1 915 Einsen auf ein leeres Band und wurde zuerst in [9] veröffentlicht, siehe Abb. 1.3

	0	1
1	1L2	1R3
2	1L3	1L2
3	1L4	0R5
4	1R1	1R4
5	1L0	0R1

(a) T^5_{1915}

	0	1
1	1L2	1R3
2	1L3	1L2
3	1L4	0R5
4	1R1	1R4
5	1L0	0R1

(b) T^5_{4098}

Abbildung 1.3: Beispiele für fleißige Biber mit 5 Zuständen

In [10] wird die 1989 gefundenen Maschine T^5_{4098}, siehe Abb. 1.3, vorgestellt, die BB(5) ≥ 4098 zeigt. Diese Maschine, zusammen mit der Turingmaschine mit 6 aktiven Zuständen, die 136 612 Einsen druckt bevor sie anhält, wurde zuerst in [25] der interessierten Öffentlichkeit präsentiert.

Der aktuelle Stand der Biberfunktion ist in der folgenden Tabelle zusammengefasst.

$$\begin{array}{lcr}
\mathrm{BB}(1) & = & 1 \\
\mathrm{BB}(2) & = & 4 \\
\mathrm{BB}(3) & = & 6 \\
\mathrm{BB}(4) & = & 13 \\
\mathrm{BB}(5) & \geq & 4098 \\
\mathrm{BB}(6) & \geq & 95524079
\end{array}$$

Das Spiel geht weiter; Heiner Marxen berichtet n einer Nachricht an die „newsgroup: comp.theory“von einer Turingmaschine mit 6 Zuständen, die in 8 690 333 381 690 951 Schritten 95 524 079 Einsen auf ein leeres Band schreibt. (E-mail Adresse heiner@drb.insel.de)

Wir werden im Unterabschnitt 1.6.5 noch einmal auf die Biberfunktion zurückkommen.

1.2.4 Nichtdeterministische Turingmaschinen

In diesem Unterabschnitt begegnen wir zum ersten Mal einem nichtdeterministischen Maschinenmodell. In solchen Modellen ist der Nachfolgezustand in einer Berechnungsfolge nicht eindeutig vorgeschrieben. Es wird nur eine endliche Anzahl von möglichen Fortsetzungen zur Auswahl gestellt. Ein nichtdeterministisches Maschinenmodell beschreibt daher bei einer festgelegten Anfangssituation nicht eine einzige sondern in der Regel eine Vielzahl von Berechnungsfolgen. Wozu sind solche Modelle gut? Das hängt von der betrachteten Modellklasse ab. Bei endlichen Automaten erlaubt der Nichtdeterminismus eine kürzere Repräsentation und eine einfachere Manipulation von Automaten. Im Falle der Kellerautomaten ist der Nichtdeterminismus unverzichtbar, deterministische Kellerautomaten sind echt ausdrucksschwächer. Da man nichtdeterministische Maschinenmodelle nicht direkt implementieren kann sind in diesem Fall umfangreiche Vorbereitungen erforderlich, siehe Kapitel 5. Im vorliegenden Fall nichtdeterministischer Turingmaschinen wird die Ausdrucksfähigkeit gegenüber der deterministischen Basisversion nicht erhöht, siehe Theorem 5. Dafür spielen nichtdeterministische Turingmaschinen eine große Rolle in der Komplexitätstheorie und wir sparen uns weitere Erklärungen bis wir zu diesem Thema kommen.

Hier ist ein erstes einfaches Beispiel einer nichtdeterministischen Turingmaschine.

Beispiel 6 (T_{nd0})

1. $\Sigma = \{|\}$, $\Gamma = \{|, B\}$

2. $Q = \{q_0, q_f\}$, $F = \{q_f\}$

3. δ *gegeben durch*

Q	Γ	Q	Γ	
q_0	B	q_f	$\mid$	U
q_0	$\mid$	q_0	$\mid$	R
q_0	$\mid$	q_0	$\mid$	L

Der Nichtdeterminismus wird sichtbar durch das Auftreten von zwei Zeilen in der Tabelle mit den gleichen Argumenten. Die Übergangsfunktionen δ ist also vom Typ:

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{R, L, U\})$$

Was berechnet eine nichtdeterministsche Turingmaschine? Offensichtlich macht es keinen Sinn sie zu benutzten um aus gegebenen Argumenten einen Funktionswert zu berechnen. Welchen der Werte, die in den verschiedenen Berechnungsabläufen entstehen, sollte man auch nehmen? Als Akzeptionsmodell kann sie aber sehr wohl dienen. Dabei stellen wir sehr schwache Anforderungen. Ein Eingabewort soll schon akzeptiert werden, wenn es wenigstens eine akzeptierende Berechnungsfolge gibt, unabhängig von der Existenz nichtakzeptierender oder nichtterminierender Berechnungen.

Definition 6 *Eine nichtdeterministische Turingmaschine mit dem Eingabealphabet* Σ akzeptiert *ein Wort* $w \in \Sigma^*$, *falls es eine Berechnungsfolge für* T **gibt**, *die in der Anfangskonfiguration* Bq_0wB *beginnend schließlich einen Endzustand erreicht.*

$$L(T) = \{w \in \Sigma^* \mid T \textit{ akzeptiert } w\}$$

heißt die von T akzeptierte Sprache.

Für das oben betrachtete Beispiel gilt die trivialerweise $L(T_{nd0}) = \{|\}^*$.

Beispiel 7 (T_{sat})
Wir betrachten noch ein weiteres nicht ganz so simples Beispiel. Die nichtdeterministische Turing Maschine T_{sat} *soll das Erfüllbarkeitsproblem für aussagenlogische Formeln lösen. Das sind Zeichenfolgen, die aufgebaut sind aus den logischen Operatoren* $\vee, \wedge, \neg$, *aussagenlogischen Variablen – in diesem Beispiel sollen der Einfachheit halber nur* a, b, c *vorkommen – und Klammern* $($, $)$. *Für eine vollständige Definition wende man sich z.B. an [14, Kapitel 4]. Die zu untersuchende Formel* F *wird dabei in Klauselnormalform, das heißt als Konjunktion von Disjunktionen von Literalen, auf das Eingabeband der*

Turingmaschine geschrieben. Diese startet auf dem ersten freien Feld rechts davon im Anfangszustand q_0. Nach der Berechnung hält T_{sat} in dem positiven Endzustand q_f^1 oder dem negativen q_f^0. Die Maschine ist so konstruiert, daß F erfüllbar ist genau dann, wenn es eine Berechungsfolge mit positivem Endzustand gibt.

1. $\Sigma = \Sigma_V \cup \Sigma_O$, $\Sigma_V = \{a, b, c\}$, $\Sigma_O = \{(,), \vee, \wedge, \neg\}$
2. $\Gamma = \Sigma \cup \{0, 1, B\}$
3. $Q = \{q_0, q_f^1, q_f^0, pos, neg, dis1, cl0\} \cup \{q_{ijk} \mid i, j, k \in \{0, 1\}\}$,
4. $F = \{q_f^1, q_f^0\}$
5. δ *gegeben durch*

Q	Γ	Q	Γ		
q_0	B	$\{q_{ijk} \mid i,j,k \in \{0,1\}\}$	B	L	
q_{ijk}	σ	q_{ijk}	σ	L	*für alle* $\sigma \in \Sigma_O$
q_{ijk}	a	q_{ijk}	i	L	*für alle* $i, j, k \in \{0, 1\}$
q_{ijk}	b	q_{ijk}	j	L	*für alle* $i, j, k \in \{0, 1\}$
q_{ijk}	c	q_{ijk}	k	L	*für alle* $i, j, k \in \{0, 1\}$
q_{ijk}	B	pos	B	R	

Q	Γ	Q	Γ	
pos	$)$	pos	$)$	R
pos	$\neg$	neg	$\neg$	R
pos	$\vee$	pos	$\vee$	R
pos	$\wedge$	pos	$\wedge$	R
pos	0	pos	0	R
pos	1	$dis1$	1	R

Q	Γ	Q	Γ		
neg	0	$dis1$	0	R	
neg	1	pos	1	R	
pos	$($	$cl0$	$($	R	
$dis1$	$($	pos	$($	R	
$cl0$	σ	$cl0$	σ	R	*für alle* $\sigma \neq B$
pos	B	q_f^1	B	U	
$cl0$	B	q_f^0	B	U	

Einige Erläuterungen zur Wirkungsweise von T_{sat} sind wohl angebracht. Wir weichen hier von der Standardanfangssituation ab und beginnen rechts von der Eingabe. Das spart einige Zustände bei der Behandlung des Negationszeichens. Der Tabelleneintrag für q_0 ist als Abkürzung zu lesen für

$$\delta(q_0, B) = \{(q_{ijk}, B, L) \mid i, j, k \in \{0, 1\}\}$$

Ausgehend von der in Abb. 1.4 gezeigten Anfangssituation läuft die Maschine in dem ersten Teil der Berechnung nach rechts über die Eingabeformel und ersetzt nichtdeterministisch Variablen durch 0 oder 1. Dabei sollen natürlich gleiche Variable durch den gleichen Wahrheitswert ersetzt werden. Das wird

Anfangskonfiguration:

B	(	$\neg$	a	$\vee$	b	$\vee$	c	)	$\wedge$	(	a	$\vee$	$\neg$	b	)	$\wedge$	(	a	$\vee$	b	$\neg$	c	)	B
																								q_0

Konfiguration nach dem Durchlauf nach links:

B	(	$\neg$	1	$\vee$	1	$\vee$	0	)	$\wedge$	(	1	$\vee$	$\neg$	1	)	$\wedge$	(	1	$\vee$	1	$\neg$	0	)	B
q_{110}																								

Abbildung 1.4: Beispielkonfigurationen der Turingmaschine T_{sat}

erreicht, indem zu Beginn der Berechnung nichtdeterministisch ein Tripel ijk *von Wahrheitswerten ausgewählt wird und in dem Zustand* q_{ijk} *gespeichert wird. Nehmen wir an, daß nach dem ersten Schritt der Zustand* q_{110} *eingenommen wird, dann steht nach Erreichen des Leerzeichens B am linken Ende der Eingabe die zweite Inschrift aus Abb. 1.4 auf dem Band.*

Jetzt läuft die Maschine nach links ohne die Bandinschrift zu verändern. Wir eine Disjunktion entdeckt mit dem Wahrheitswert 0*, dann wechselt die Maschine in den Zustand cl*0*, den sie nicht wieder verläßt. Ist cl*0 *nicht erreicht, dann beginnt das Einlesen der öffenden Klammer einer Disjunktion stets im Zustand pos. Wird einmal ein positives Literal gelesen, dann wechselt die Maschine in den Zustand dis*1*, den sie bis zur schließenden Klammer für die Disjunktion nicht mehr verläßt. Der Rest der Aktionen von* T_{sat} *kann jetzt einfach nachvollzogen werden.*

Theorem 5
Zu jeder nichtdeterministischen Turingmaschine T_{nd} *gibt es eine deterministische Turingmaschine* T *mit*

$$L(T_{nd}) = L(T)$$

Ist $t_{ND}(n)$ *die Zeitkomplexität von* T_{nd}*, so ist die Zeitkomplexität von* T

$$t_T(n) = O(t_{ND}^2(n))c^{t_{ND}(n)}$$

für eine Konstante $c > 1$.

Beweis: siehe [15, Satz 13.13, Seite 30]

■

1.2.5 Ein persönlicher Kommentar

Turingmaschinen sind schon häufig totgesagt worden. Sie seien nicht mehr zeitgemäß, sie seien viel zu unrealistisch, sagen ihre Gegner, und die Informatik hätte inzwischen bessere Modelle für prinzipielle Berechenbarkeit einwickelt. Wir werden im nächsten Abschnitt ein solches, mehr von der Informatik geprägtes Modell, kennenlernen. Jeder Leser kann dann für sich selbst entscheiden, was er für eingänger hält. Für mich ist Turings im Jahr 1936 veröffentlichte Analyse der Berechenbarkeit immer noch fasziniernd vor allem durch ihre Einfachheit und Geradlinierkeit.

Es gibt aber noch andere Gründe für das Weiterleben der Turingmaschine: Die Idee ist so einfach, daß Forscher immer wieder auf sie zurückkommen. Das tat z.B. Yuri Gurevich Anfang der achtziger Jahre. Er setzte bei dem unangenehmsten Punkt des Turingschen Modells an, bei der einschränkten Datenstruktur. Diese besteht aus einem unendlichen Wort, eben der Bandinschrift, mit endlich vielen nichttrivialen Buchstaben. Die in komplizierteren Problemen auftretenden Datentypen müssen zumeist mühselig in Wörter kodiert werden. Wir werden beim Beweis der Unentscheidbarkeit des Postschen Korrespondenzproblems in Lemma 21 im Detail sehen, wie das vor sich geht. Muß das so sein? Das wichtigste an dem Turingmaschinenmodell sind die Berechnungsfolgen, also Folgen von Wörtern und aktuellen Zuständen. Yuri Gurevichs Idee war nun, die Wörter zu ersetzten durch beliebige endliche mathematische Strukturen. Präziser gesagt, handelt es sich dabei um Algebren, das sind Strukturen, in denen die Gleichheit die einzige Relation ist. Deswegen hießen diese Modelle zuerst *dynamische Algebren* (engl. evolving algebras). Heute sind sie unter dem Namen *abstrakte Zustandsmaschinen* (engl. abstact state machines) bekannt. Die Elementaroperationen der Maschine bestehen jetzt nicht mehr aus Schreiben, Lesen und Navigieren auf einem Band sondern aus Operationen, die Algebren verändern. Dazu gehört zum Beispiel das Hinzufügen eines neuen Elements zum Grundbereich der Algebra, oder das Weglassen eines alten, oder das Abändern einer Funktion an einer Argumentstelle. Diese Überlegungen führten mit der Zeit zu einem neuen Modell für die Semantik von Programmiersprachen und letzten Endes zu einer neuen allgemein einsetzbaren Spezifikationsmethode.

Die erste Veröffentlichung zu diesem Thema [16] unter dem programmatischen Titel *Reconsidering Turing's Thesis* ist heute vielleicht nicht mehr so leicht erhältlich. Eine umfassende Literaturübersicht zu *abstract state machines* findet man auf der ASM Homepage[1]

[1] http://www.eecs.umich.edu/gasm/

1.3 While-Sprachen

Eine weitere Möglichkeit, den Begriff einer berechenbaren Funktion einzuführen, der sich besonders in der Informatik großer Beliebtheit erfreut, ist die Angabe einer möglichst einfachen Programmiersprache $\mathcal{P}$. Eine Funktion f wird dann $\mathcal{P}$-berechenbar genannt, wenn es ein Programm in der Sprache $\mathcal{P}$ gibt, das f berechnet. Wir werden als ein Beispiel die *while*-Programme und eine Variante davon betrachten. In der Tradition der Berechenbarkeitstheorie wurden ausschließlich Funktionen über den natürlichen Zahlen $\mathbb{N}$ betrachtet. Waren andere Datenstrukturen erforderlich oder hilfreich, so konnte man sie durch Kodierungen in $\mathbb{N}$ realisieren. Um den dabei unerläßlichen Kodierungsaufwand einzusparen ist es in jüngster Zeit üblich geworden, *while*-Programme gleich über verschiedenen Datenstukturen einzuführen, siehe z.B. [11]. Wir werde sowohl Programme über Wörtern als auch über den natürlichen Zahlen betrachten.

1.3.1 *for*-Programme

Wenn wir uns die Frage stellen, welche Konstrukte einer Programmiersprache uns unverzichtbar erscheinen, so fällt uns als erstes die Zuweisung eines Wertes an eine Programmvariable ein. Ohne sie kann man sich schwerlich vorstellen, daß ein Programm überhaupt etwas berechnet. Die nächste Festlegung, die zu treffen ist, betrifft die Konstrukte, die auf der rechten Seite einer Zuweisung auftreten dürfen. Das hängt in erster Linie von den Datenstrukturen ab, die wir zulassen wollen. In dem ersten Ansatz eine minimale Programmiersprache zu erfassen, beschränken wir uns einzig und allein auf die Datenstruktur der natürlichen Zahlen. Als einziges weiteres Programmkonstrukt neben dem Zuweisungsbefehl verfügen die *for*-Programme über die *for*-Schleife.

Eine Erweiterung auf die etwas flexiblere *while*-Schleifen wird im nächsten Unterabschnitt eingeführt werden.

Definition 7 (*for*-Programme über den natürlichen Zahlen)

Algorithmus	::=	**begin** *Anweisungen* **end**
Anweisungen	::=	*Anweisung* |
		Anweisungen ';' *Anweisung*
Anweisung	::=	**leer** | *Zuweisung* | *Spezial*
Zuweisung	::=	*Var* ':=' *Ausdruck*
Zuweisung	::=	**0** | *Var* | **succ** *Var*
		| **pred** *Var*
Var	::=	*beliebiger Bezeichner*
Spezial	::=	**for** *Var* **loop** *Anweisungen* **end**

Dabei bedeutet: **pred** n = Vorgänger von n, **pred** 0 = 0, **succ** n = n+1, **for** V **loop** p **end** bewirkt die m-fache Ausführung des Schleifenrumpfes p, wenn m der Wert von V beim Eintritt in die *for*-Schleife ist. Hier sind zwei einfache Beispiele für *for*-Programme:

Beispiel 8

```
Eingabe: x,y
Ausgabe: z

begin
    z := y;
    for x loop
        z := succ z
    end
end
```

```
Eingabe: x,y
Ausgabe: u
begin
    u := 0;
    for x loop
        for y loop
            u:= succ u
        end
    end
end
```

Die *for*-Programme enthalten ebenso wie die später betrachteten *while*-Programme keine Anweisungen für Eingabe und Ausgabe. Wie diese beiden Operationen realisiert sind spielt für prinzipielle Betrachtungen zur Berechenbarkeit keine Rolle, aber irgendwie müssen sie schon geregelt sein. Deswegen die zusätzlichen Angaben in den obigen Beispielen. Die Berechnung, die von einem *for*-Programm durchgeführt wird, ist leicht zu verstehen, wenn man die folgenden Erklärungen kennt:

- **pred** n = Vorgänger von n, **pred** 0 = 0
- **succ** n = n+1
- **for** Var **loop** p **end** bewirkt die m-fache Ausführung des Schleifenrumpfes p, wobei m der Wert der Variablen Var beim Eintritt in die

Programmschleife ist. Spätere Änderungen des Wertes von Var haben keinen Einfluß auf m.

Eine formale Definition der Semantik von *for*- und *while*-Programmen ist natürlich möglich, ist aber für eine elementare Einführung zu aufwendig und entbehrlich, siehe z.B. [11, Abschnitt 2.2].

Offensichtlich berechnet die Programme aus Beispiel 8 die Addition $z = x + y$ bzw. die Multiplikation $u = x * y$. Da der Befehlssatz für *for*-Programm sehr klein ist, werden schon einfache Programme unübersichtlich. Man behilft sich mit der Einführung von Makros.

Beispiel 9

Eingabe: x,y
Ausgabe: u
*u := x * y:*
begin
    u := 0;
    for *x* **loop**
        u := u + y
    end
end

Hier wird die Zuweisung " u := u + y" als eine Makroanweisung benutzt. Die Semantik von Makros wird durch textuelle Ersetzung festgelegt. Abbildung 1.5 zeigt einige weitere Makros. Man wird sich fragen, ob *for*-

1. $Z := X + Y$
2. $Z := X \dot{-} Y$
3. $Z := X * Y$
4. $Z := X \textbf{ div } Y$
5. $Z := X \textbf{ mod } Y$
6. $Z := X ** Y$ (Potenz)
7. $Z := 2^X$
8. $Z := log_2(X)$

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

$n = (n \textbf{ div } m) * m + (n \textbf{ mod } m)$
mit $(n \textbf{ mod } m) < m$

$log_2(n) =$ die größte ganze Zahl m mit $2^m \leq n$

Abbildung 1.5: Einige häufig benutzte Makros

Programme nicht vielleicht doch zu einfach sind. Sie enthalten z.B. keine Verzweigungsbefehle. Diese können jedoch durch Zuweisungen und *for*-Schleifen simuliert werden. So kann z.B.

if u = 0 **then** A **end**

ersetzt werden durch:

```
begin
    h := 1;
    for u loop h:=0 end
    for h loop A end
end
```

Die *for*-Programme haben eine Eigenschaft, von der die Programmierer in realistischen Programmiersprachen nur träumen können. Man kann überhaupt keine Programme hinschreiben, die nicht terminieren.

Theorem 6
Ist $f : \mathbb{N}^k \to \mathbb{N}$ eine for-berechenbare Funktion, dann ist f eine totale Funktion. Mit anderen Worten ein for-Programm terminiert für jede Eingabe.

Beweis: Einfache Induktion über die Länge von *for*-Programmen.

∎

Diese Theorem zeigt aber zugleich, daß nicht jede Turing-berechenbare f Funktion durch ein *for*-Programm berechnet werden kann, denn f kann auch eine partielle Funktion sein. Es wird sich allerdings später (siehe Theorem 12) herausstellen, daß die *for*-berechenbaren Funktionen eine wichtige und umfangreiche Teilklasse der berechenbaren Funktionen bilden.

1.3.2 *while*-Programme

In *while*-Programmen ist zusätzlich die Benutzung von *while*-Schleifen erlaubt. Zur Bequemlichkeit des Lesers wiederholen wir die ganze Definition,

Definition 8 (*while*-Programme über den natürlichen Zahlen)

Algorithmus	::=	**begin** *Anweisungen* **end**
Anweisungen	::=	*Anweisung* |
		Anweisungen ';' Anweisung
Anweisung	::=	*leer* | *Zuweisung* | *Spezial*
Zuweisung	::=	*Var ':=' Ausdruck*
Ausdruck	::=	*0* | *Var* | **succ** *Var*
		| **pred** *Var*
Var	::=	*beliebiger Bezeichner*
Spezial	::=	**while** *Var* **loop**
		Anweisungen **end**

wobei
while v **loop** p **end**
bewirkt, daß die Anweisungen p so lange ausgeführt werden bis v = 0 gilt.

In dieser Definition sind *for*-Schleifen nicht als primitives Konstrukt vorgesehen. Man sieht jedoch leicht, daß sie als Makros definierbar sind.

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt *while*-berechenbar, wenn es ein *while*-Programm $\mathcal{P}$ gibt, das f berechnet. Die Deklaration der Eingabe- und Ausgabevariablen erfolgt wieder als zusätzlicher Hinweis außerhalb der formalen Definition von *while*-Programmen. Für *while*-Programme benutzten wir Makros wie für *for*-Programme. Jetzt spielt auch noch eine andere Form von Makros, die sogenannten Testmakros, eine Rolle. Sie erlauben es komplexere Abbruchbedingungen für *while*-Schleifen zu benutzen. Ein **Test** ist eine Boolesche Kombination von Aussagen der Form

$$X \neq Y, X = Y, X < Y, X \leq Y, n < X, n < m$$

Ist $\mathcal{C}$ ein Test, so kann

while $\mathcal{C}$ **loop** Anweisungen **end**

durch ein **while**-Programm simuliert werden. So kann man etwa

while X = Y **loop** Anweisungen **end**

unter Ausnutzung von Makrozuweisungen ersetzen durch

```
V := (X - Y) + (Y - X)
while V loop
        P ; V := (X - Y) + (Y - X)
end
```

Wir kommen auf die in der Einleitung zu diesem Abschnitt angesprochene Frage der Datentypen zurück. Wir nehmen uns vor, Wörter anstelle natrlicher Zahlen benutzen zu wollen. Wir bezeichnen mit ε das leere Wort. Welche Operationen auf Wörtern treten dabei an die Stelle der Nachfolger- und Vorgängerfunktion? Die Antwort kann man aus der folgenden Funktion ablesen.

Definition 9 (*while*-**Programme über Wörtern**)
Gegeben sei ein endliches Alphabet Γ

Algorithmus	::=	**begin** *Anweisungen* **end**
Anweisungen	::=	*Anweisung* |
		Anweisungen ';' *Anweisung*
Anweisung	::=	*leer* | *Zuweisung* | *Spezial*
Zuweisung	::=	*Var* ':=' *Ausdruck*
Ausdruck	::=	ε | *Var* | $\gamma \in \Gamma$
		| **append** *Var Var*
		| **tail** *Var* | **head** *Var*
Var	::=	*beliebiger Bezeichner*
Spezial	::=	**while** $Var \neq \varepsilon$ **loop**
		Anweisungen **end** |
		while $Var \neq \gamma$ **loop**
		Anweisungen **end** *für alle* $\gamma \in \Gamma$

wobei **while** w **loop** p **end** *bewirkt, daß die Anweisungen* p *so lange ausgeführt werden bis* $w = \varepsilon$ *gilt. Außerdem gilt*

$$\begin{array}{lcl} \textbf{append}\ w_1 \quad w_2 & = & w_1 w_2 \\ \textbf{tail}\ x_1 \ldots x_k & = & x_2 \ldots x_k, \\ \textbf{tail}\ \varepsilon & = & \varepsilon \\ \textbf{head}\ x_1 \ldots x_k & = & x_1, \\ \textbf{head}\ \varepsilon & = & \varepsilon \end{array}$$

Hier ist ein Beispiel für ein *while*-Programm über Wörtern.

Beispiel 10 (P_{inv})

Eingabe: X
Ausgabe: Y
begin
    $Y := \varepsilon$;
    while X **loop**
        $U :=$ **head** X

$Y :=$ **append** U Y
$X :=$ **tail** X
end
end

P_{inv} berechnet die Funktion $f_{\text{inv}} : \Gamma^* \to \Gamma^*$ mit $f_{\text{inv}}(x_1 \ldots x_k) = x_k \ldots x_1$.

1.3.3 Ein persönlicher Kommentar

Läßt sich die in diesem Abschnitt vorgetragene Abstraktion realer Programmiersprachen noch weiter fortsetzen? Wir haben uns bisher sehr eng an den üblichen Programmkonstrukten orientiert und nicht die Frage gestellt, ob sie sich in noch einfachere Elementarbefehle zerlegen lassen. Erstaunlicherweise ist das tatsächlich möglich. Durchgeführt wurde diese Analyse von David Harel, der heute, das heißt zum Zeitpunkt, an dem diese Zeilen geschrieben werden, für eine ganz andere Leistung bekannt geworden ist, nämlich für die Entwicklung und Einführung der Spezifikationsmethode *state charts*, die inzwischen ein Bestandteil von UML geworden ist. Ich will hier über die Arbeit [17] referieren. Es handelt sich dabei um eine überarbeitete Version der Doktorarbeit von D. Harel aus dem Jahr 1978. Leichter zugänglich ist [18]. In diesen Veröffentlichungen geht es um die Beschreibung eine Logik, genannt *dynamic logic* oder *dynamische Logik* (nicht zu verwechseln mit den dynamischen Algebren aus Unterabschnitt 1.2.5). Dynamische Logik dient dem Argumentieren über Eigenschaften von Programmen und Programmtransformationen. Diesen Aspekt blenden wir in dieser Kurzbeschreibung aus. Zur dynamischen Logik gehört aber auch die Beschreibung von Programmen, die ja schließlich präzise beschrieben werden müssen, bevor man Aussagen über sie beweisen oder falsifizieren kann. In der dynamischen Logik wird von Anfang an ein sehr allgemeiner Standpunkt eingenommen, spezialisieren kann man zu einem späteren Zeitpunkt ja immer noch. Ein Programm ist dabei eine Relation zwischen einem Anfangszustand und möglicherweise mehreren Endzuständen. Damit ist auch die Modellierung nichtdeterministischer Programme mit eingeschlossen. Einen Zustand kann man sich dabei als eine Belegung aller auftretenden Programmvariablen mit Werten vorstellen. Sind α und β Programme und ist F eine Formel, so sind die folgenden Programmkonstrukte erlaubt:

$(\alpha; \beta)$	Hintereinanderausführung
$(\alpha \cup \beta)$	nichtdeterministische Auswahl
(α^*)	Iteration
$(F?)$	Test

Ist S die Menge aller möglichen Zustände, so werden α und β interpretiert als Relationen zwischen Elementen von S. Bezeichnen wir die Interpretationsfunktion mit ρ können wir das als $\rho(\alpha) \subseteq S^2$ und $\rho(\beta) \subseteq S^2$ schreiben. Die Bedeutung der vier Programmkonstrukte kann jetzt präzise definiert werden:

$$
\begin{array}{lcl}
\rho(\alpha;\beta) & = & \{(s,t) \mid \text{ es gibt ein } u \text{ mit } (s,u) \in \rho(\alpha) \text{ und } (u,t) \in \rho(\beta)\} \\
\rho(\alpha \cup \beta) & = & \rho(\alpha) \cup \rho(\beta) \\
\rho(\alpha^*) & = & \{(s,t) \mid \text{ es gibt ein } k \text{ und es gibt } s_0, \ldots, s_k \text{ mit } s_0 = s, \\
& & s_k = t \text{ und } (s_{i-1}, s_i) \in \rho(\alpha) \text{ für alle } 1 \leq i \leq k\} \\
\rho(F?) & = & \{(s,s) \mid F \text{ ist wahr in } s\}
\end{array}
$$

Bisher haben wir Regeln kennengelernt um aus gegeben Programmen neue zu kombinieren. Die einzige zusätzliche Regel die notwendig ist, damit einmal ein Anfang gemacht werden kann, ist die Zuweisungsregel $x := t$ mit der erwarteten formalen Semantik

$$\rho(x := t) \quad = \quad \{(w_1, w_2) \mid w_2(y) = w_1(y) \text{ für } x \neq y \text{ und } w_2(x) = \text{wert}(t)\}$$

Die höheren Programmkonstrukte, die wir in den vorangegangenen Abschnitten behandelt haben, lassen sich folgendermaßen in der dynamischen Logik ausdrücken:

$$
\begin{array}{lll}
\textbf{if } A \textbf{ then } \alpha \textbf{ else } \beta & \text{für} & (A?;\alpha) \cup (\neg A?;\beta) \\
\textbf{while } A \textbf{ do } \alpha & \text{für} & (A?;\alpha)^*;\neg A? \\
\textbf{repeat } \alpha \textbf{ until } A & \text{für} & \alpha;(\neg A?;\alpha)^*
\end{array}
$$

Das ist sicherlich wenig benutzerfreundlich, aber das ist auch nicht das Ziel der dynamischen Logik. Wir zitieren aus [18]

> ... the idea is to study the mathematical properties of programs and their behaviour and the tools for reasoning about them, with the objective of singling out and gaining a better understanding of those concepts and methods which are fundamental to the reasoning process.

1.4 Rekursive Funktionen

Wir kommen zur dritten der angekündigten Möglichkeiten den Begriff der berechenbaren Funktion zu definieren. Bei der ersten Möglichkeit stand ein Maschinenmodell, eben die Turingmaschine, im Zentrum der Definition. Bei der Definition *while*-berechenbarer Funktionen tauchte zumindest implizit ein Maschinenmodell auf, nämlich die gedachte Maschine, welche die Kommandos der *while*-Programme ausführt. Die Definition allgemein-rekursiver Funktionen kommt ohne Anlehnung an eine Maschinenmodell aus.

1.4.1 Primitiv-rekursive Funktionen

Wie schon bei der Einführung der *while*-Programme betrachten wir auch hier zunächst eine Teilklasse der Funktionen, die wir schließlich definieren werden.

Definition 10 (Primitiv rekursive Funktionen)

1. *Die folgenden Funktionen sind primitiv rekursiv*
   (a) *die Zahl* 0 *(0-stellige Funktion)*
   (b) *die konstante Funktion* $N(n) \equiv 0$
   (c) *die Nachfolgerfunktion* $\mathrm{succ}(n) = n + 1$
   (d) *alle Projektionsfunktionen* $U_i^n(a_1, \ldots, a_n) = a_i$
2. *ist* $h : \mathbb{N}^k \to \mathbb{N}$ *primitiv rekursiv und sind* $g_1, \ldots, g_k$ *primitiv rekursiv, wobei* $g_i : \mathbb{N}^n \to \mathbb{N}$, *dann ist auch*
   $$h(g_1(a_1, \ldots, a_n), \ldots, g_k(a_1, \ldots, a_n))$$
   primitiv rekursiv.
3. *sind* $g : \mathbb{N}^k \to \mathbb{N}$ *und* $h : \mathbb{N}^{k+2} \to \mathbb{N}$ *rimitiv rekursiv, dann auch die Funktion* f*:*
   $$\begin{aligned} f(0, \vec{a}) &= g(\vec{a}) \\ f(n+1, \vec{a}) &= h(f(n, \vec{a}), n, \vec{a}) \end{aligned}$$

Dabei haben wir die Notation $\vec{a} = a_1, \ldots, a_k$ *benutzt.*

In den folgenden Beispielen halten wir uns strikt an die in der Definition vorgeschriebene syntaktische Form.

Beispiel 11

1. $$\begin{aligned} \mathrm{add}(0, a) &= U_1^1(a) \\ \mathrm{add}(n+1, a) &= \mathrm{succ}(U_1^3(\mathrm{add}(n, a), n, a)) \end{aligned}$$
2. $$\begin{aligned} \mathrm{mult}(0, a) &= N(a) \\ \mathrm{mult}(n+1, a) &= \mathrm{add}(U_1^3(\mathrm{mult}(n, a), n, a), U_3^3(\mathrm{mult}(n, a), n, a)) \end{aligned}$$
3. $$\begin{aligned} \mathrm{factorial}(0) &= \mathrm{succ}(0) = 1 \\ \mathrm{factorial}(n+1) &= h(\mathrm{factorial}(n), n) \end{aligned}$$
   mit $h(x, y) = \mathrm{mult}(x, \mathrm{succ}(y))$

4. $$\begin{array}{lcl} \text{pred}(0) & = & 0 \\ \text{pred}(n+1) & = & n = U_2^2(\text{pred}(n), n) \end{array}$$

5. *Sind $f, g : \mathbb{N} \to \mathbb{N}$ und $Q : \mathbb{N} \to \{0,1\}$ primitiv rekursive Funktionen, dann ist auch r primitiv rekursiv:*

   $$r(n) = \begin{cases} f(n) & \textit{falls } Q(n) = 1 \\ g(n) & \textit{falls } Q(n) = 0 \end{cases}$$

   $$r(n) = Q(n) * f(n) + (1 \dot{-} Q(n)) * g(n)$$

Zur besseren Vergleichbarkeit mit der syntaktischen Form der Definition haben wir in diesen Beispielen die Präfixnotation benutzt. Im folgenden schreiben wir wieder $+$ und $*$ für add und mult.

Fast alle zahlentheoretische Funktionen, die einem einfallen, sind primitiv rekursiv. Man muß schon zu einer ungewöhnlichen Konstruktion greifen um eine nicht primitiv rekursive Funktion zu erhalten.

Beispiel 12 (Die Ackermann-Funktion)

$$\begin{array}{lcl} A(0,y) & = & y+1 \\ A(x,0) & = & A(x-1,1) \\ A(x,y) & = & A(x-1, A(x, y-1)) \end{array}$$

Man beachte, daß die Funktion A nicht nach dem Schema der primitiven Rekursion definiert ist.

Lemma 7 (Einfache Eigenschaften der Ackermann Funktion)

1. $y < A(x,y)$
2. $A(x,y) < A(x, y+1)$
3. $A(x,y) < A(x+1, y)$
4. $A(x, y+1) \leq A(x+1, y)$
5. $\forall c_1 \dots c_r \in \mathbb{N} \quad \exists c \in \mathbb{N} \quad \forall y \in \mathbb{N} \quad : A(c_1, y) + \dots A(c_r, y) \leq A(c, y)$

Beweis: Einfaches Nachrechnen. Wer nicht klar kommt kann nachschauen in [21, §13].

Lemma 8
Zu jeder primitiv rekursiven Funktion $g(x_1, \dots, x_n)$ gibt es eine Konstante $c \in \mathbb{N}$, so daß für alle $x_1, \dots, x_n$ gilt

$$g(x_1, \dots, x_n) < A(c, x_1 + \dots + x_n)$$

Beweis: Der Beweis erfolgt unter Ausnutzung der Eigenschaften des vorangegangenen Lemmas durch Induktion über die Länge der Definition einer primitiv rekursiven Funktion. Er ist elementar, enthält aber einige nichttriviale Klipppen. Eine ausführliche Darstellung findet man wieder in [21, §13].

Lemma 9

Die Ackermann Funktion ist berechenbar, aber nicht primitiv rekursiv.

Beweis: Die Berechenbarkeit von $A(x, y)$ ist intuitiv offensichtlich. Für weitere Hinweise wird auf Theorem 12 verwiesen.

Wäre $A(x, y)$ primitiv rekursiv, dann auch $g(x) = A(x, x)$. Nach Lemma 8 gibt es eine Konstante c mit $g(x) < A(c, x)$ für alle x. Speziell für $x = c$ ergibt sich der Widerspruch $A(c, c) = g(c) < A(c, c)$.

∎

Etwas vereinfacht gesagt, ist die Ackermann Funktion nicht primitiv rekursiv, weil sie schneller wächst als alle primitiv rekursiven Funktionen. Einen Eindruck von dem extremen Wachstum der Ackermann Funktion vermitteln schon die ersten Stufen in der Ackermann-Hierarchie:

$$\begin{array}{lcll}
A(1, y) & = & y + 2 & \\
A(2, y) & = & 2y + 3 & x > 0 \\
A(3, y) & = & 2^{y+3} - 3 & x > 0, y > 0 \\
A(4, y) & = & \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{y+3 \text{ mal}} -3 &
\end{array}$$

Man sieht leicht, daß jede primitiv rekursive Funktion total ist, d.h. überall definiert ist. Schon aus diesem Grund kann diese Klasse nicht alle berechenbaren Funktionen ausschöpfen. Was fehlt noch? Darauf gibt die nächste Definition eine Antwort.

Definition 11 (Allgemein rekursive Funktionen)

1. *Alle primitiv rekursiven Funktionen sind rekursiv.*
2. *Sind* $h : \mathbb{N}^k \to \mathbb{N}$ *und* $g_1, \ldots, g_k$ *rekursiv, wobei* $g_i : \mathbb{N}^n \to \mathbb{N}$*, dann auch*
$$h(g_1(a_1, \ldots, a_n), \ldots, g_k(a_1, \ldots, a_n)).$$
3. *Sind* $g : \mathbb{N}^k \to \mathbb{N}$ *und* $h : \mathbb{N}^{k+2} \to \mathbb{N}$ *rekursiv, dann auch* f*:*
$$\begin{array}{rcl}
f(0, \vec{a}) & = & g(\vec{a}) \\
f(n+1, \vec{a}) & = & h(f(n, \vec{a}), n, \vec{a})
\end{array}$$

4. *Ist $g(x_1, \ldots, x_n)$ rekursiv, dann auch $(\mu g)(\vec{x})$.*

$$(\mu g)(x_2, \ldots, x_n) = \begin{cases} \textit{das kleinste } n \textit{ mit} & \textit{falls ein} \\ g(n, \vec{x}) = 0 \textit{ und} & \textit{solches } n \\ \forall i < n \; g(i, \vec{x}) \downarrow & \textit{existiert} \\ & \\ \uparrow & \textit{sonst} \end{cases}$$

In dieser Definition haben wir schon eine Notation benutzt, die wir im folgenden noch häufiger brauchen werden.

Definition 12
Für eine Funktion $f : D \to W$ und $d \in D$ schreiben wir

- *$f(d)\downarrow$ falls f an der Stelle d definiert ist.*
- *$f(d)\uparrow$ falls f an der Stelle d undefiniert ist.*

Eine Funktion, die für alle Stellen ihres Argumentbereiches definiert ist heißt eine totale *Funktion. Eine Funktion, die für manche Argumente undefiniert sein kann, heißt eine* partielle *Funktion. Einen Extremfall stellt die überall undefinierte Funktion dar, die wir mi λ bezeichnen werden.*

1.4.2 Rekursive Mengen

Wir haben bisher den Begriff der Berechenbarkeit nur auf Funktionen angewendet. Er kann aber auch auf Mengen ausgedehnt werden. Intuitiv nennen wir eine Menge M berechenbar, wenn für jedes Element in endlich vielen Schritten entschieden werden kann, ob es ein Element von M ist oder nicht. In der folgenden formalen Definition wird der Begriff einer berechenbaren Menge auf den einer berechenbaren Funktion zurückgeführt. Für Mengen gibt es zusätzlich eine sehr haufig auftretende Eigenschaft, die zwischen Entscheidbarkeit und Unentscheidbarkeit liegt und deswegen manchmal Semi-Entscheidbarkeit genannt wird. Für solche semi-entscheidbaren Mengen, oder wie wir sagen werden rekursiv aufzählbare Mengen, gibt es ein Aufzählungsverfahren. Wenn eine Element a in M vorkommt, so kommt es nach endlich vielen Schritten in der Aufzählung vor und wir wissen positiv, daß $a \in M$ gilt. Dagegen müßte man unendlich lange warten um die Aussage $a \notin M$ sicher zu wissen.

Definition 13
Eine Teilmenge $S \subseteq \mathbb{N}$ heißt:

- rekursiv*, falls die* charakteristische Funktion χ_S *von* S *berechenbar ist. Dabei ist* $\chi_S : \mathbb{N} \to \mathbb{N}$ *definiert durch*

  $\chi_S(n) = \begin{cases} 1 & \textit{falls } n \in S \\ 0 & \textit{falls } n \notin S \end{cases}$

- rekursiv aufzählbar*, falls es eine berechenbare Funktion* $f : \mathbb{N} \to \mathbb{N}$ *gibt mit*

$$S = Def(f)$$

Die folgenden beiden Lemmata fassen zusammen, was wir über rekursive und rekursiv-aufzählbare Mengen wissen müssen.

Lemma 10
Die folgenden Eigenschaften einer Menge $S \subseteq \mathbb{N}$ *sind äquivalent:*

1. *es gibt eine berechenbare totale Funktion* f*, so daß* $S = \emptyset$ *oder* $S = Bild(f)$*.*

2. S *ist rekursiv aufzählbar.*

Beweis: Um die Implikation von 1. nach 2. zu zeigen, können wir $S \neq \emptyset$ annehmen. Also gilt $S = \mathrm{Bild}(f)$ für eine berechenbare totale Funktion f. Eine rekursive Funktion g mit $S = Def(g)$ kann dann durch das folgende einfache *while*-Programm berechnet werden, wobei wir wegen der Berechenbarkeit von f das Makro $X := f(Y)$ benutzen dürfen.

```
Eingabe: X
Ausgabe: X
begin
    Z := 1 ; N := 0;
    while Z loop
        U := f(N);
        if X = U then Z := 0 else N := N+1
    end
end
```

Ist der Eingabewert n ein Element der Menge S, dann findet dieses Programm einen Wert m für die Variable N mit $n = f(m)$, da f überall definiert ist.

Der Beweis der umgekehrten Implikation ist komplizierter und wir beschreiben den notwendigen Algorithmus auf einer etwas abstrakteren Ebene. Gegeben ist eine berechenbare Funktion g mit $S = Def(g)$. Wir nehmen wieder an, daß S nicht die leere Menge ist und wählen ein beliebiges Element $c \in S$.

Wir benutzen eine Hilfsfunktion H. Der Wert $H(n)$ ist eine Liste (in geeigneter Weise durch eine natürliche Zahl kodiert). Die Berechnung von $H(n)$ startet damit, daß die ersten $n + 1$ Schritte in den Berechnungen für $g(0), g(1) \dots g(n)$ ausgeführt werden. $H(n)$ ist dann die Liste aller i, so daß die Berechnung von $g(i)$ in höchstens $n + 1$ Schritten terminiert. Es könnte zum Beispiel der Funktionswert $g(5)$ undefiniert sein, aber $g(6)$ definiert sein. Die angegebene Konstruktion gewährleistet, daß für hinreichend großes n die Zahl 6 in der Liste $H(n)$ vorkommt. Der Funktionswert $f(m)$ ist jetzt die kleinste Zahl aus $H(m)$, die noch nicht in der Menge der bisherigen Wert $\{f(0), f(1), \dots, f(m-1)\}$ vorkommt, falls es eine solche Zahl gibt. Anderenfalls wird $f(m) = c$ gesetzt. Man sieht leicht ein, daß f eine totale Funktion ist mit $S = Bild(f)$ und intuitiv ist auch einsichtig, daß f berechenbar ist.

∎

Lemma 11

1. *Jede rekursive Menge ist auch rekursiv aufzählbar.*
2. *Es gibt Mengen die rekursiv aufzählbar aber nicht rekursiv sind.*
3. *Eine Menge S ist genau dann rekursiv, wenn S und $(\mathbb{N} \setminus S)$ rekursiv aufzählbar sind.*

Beweis: Zu 1. Sei S eine rekursive Menge. Wir können wieder annehmen, daß S nicht die leere Menge ist, etwa $c \in S$. Eine Funktion f mit $Bild(f) = S$ wird dann durch das folgende *while*-Programm berechnet.

Eingabe: X
Ausgabe: Y
begin
  $U := \chi_S(X)$;
  if $U = 0$ **then** $Y :=$ c **else** $Y := X$
end

Zu 2. In Theorem 14 werden wir eine solche Menge kennenlernen.

Zu 3. Gelte $Bild(f_1) = S$ und $Bild(f_2) = \mathbb{N} \setminus S$ für totale berechenbare Funktionen f_i. Wir zählen in einer *while*-Schleife die Folge

$$f_1(0), f_2(0), f_1(1), f_2(1) \ldots$$

auf. Falls für ein gegebenes n eine Zahl i existiert mit $f_1(i) = n$, dann setzen wir $f(n) = 1$, falls i existiert mit $f_2(i) = n$, setzen wir $f(n) = 0$. Wegen der vorausgesetzten Eigenschaften von f_1 und f_2 führt diese Anweisung nicht zu Konflikten und die Berechnung von $f(n)$ terminiert. Offensichtlich ist f berechenbar und $f = \chi_S$.

■

1.5 Die Churchsche These

Wir haben bisher drei Möglichkeiten kennengelernt den Begriff der Berechenbarkeit einzuführen: Turingmaschinen, *while*-Programme und allgemeinrekursive Funktionen. Wie der Leser schon vermutet haben wird, liefern alle drei Definitionen dieselbe Klasse berechenbarer Funktionen. Die Erkenntnis, daß jede Formalisierung des intuitiven Begriffes der Berechenbarkeit, nicht nur die drei hier genannten, zur selben Klasse berechenbarer Funktionen führt, wird zu Ehren von Alonso Church die Churchsche These genannt. Manche Autoren schließen auch Turing in die Referenz mit ein und sprechen von der Church-Turing-These. Beweisen in einem mathematischen Sinne läßt sich die Churchsche These natürlich nicht, da sie ja einen intuitiven Begriff mit einem formalisierten vergleicht. Die Äquivalenz verschiedener Formalisierung ist dagegen für alle bisher bekannten Versionen nachgewiesen worden.

Als Service für den interessierten Leser geben wir noch Literaturhinweise auf einige andere Berechenbarkeitsmodelle

- λ-definierbare Funktionen, eingeführt 1936 von A. Church, siehe [30, §20.9].
- Registermaschinen, eingeführt von Sheperdson und Sturgis, 1963, siehe z.B. [11], [40, Abschnitt 2.1] oder [12, Kapitel X, §2]
- Postsche kanonische Systeme, eingeführt von Emil Post, 1943, siehe [28, Chapter 12].
- Markov-Algorithmen, 1951 von A.A. Markov eingeführt.

Erst die Churchsche These verleiht den in Abschnitt 1.6 präsentierten Unentscheidbarkeitsresultaten ihre volle Bedeutung. Um diese Aussage zu erklären betrachten wir ein anderes „Unmöglichkeitsresultat", z.B. das Resultat, daß ein reguläres 7-Eck nicht mit Zirkel und Lineal konstruiert werden kann. Dieses Ergebnis ist relativ, hängt ab von der genannten Einschränkung. Läßt man sie fallen, und erlaubt z.B. das Newtonsche Näherungsverfahren als Lösungmethode, so wird das 7-Eck konstruierbar. Die Unentscheidbarkeitsresultate sind dagegen absolut.

1.5.1 Äquivalenzbeweise

Äquivalenzbeweise für die drei bisher betrachteten Definitionen berechenbarer Funktionen werden in fast allen Einführungen in die Theorie der Berechenbarkeit gebracht. Wir verzichten deshalb darauf sie alle hier noch einmal zu reproduzieren und beschränken uns auf zwei Teilresultate.

Theorem 12 *Die folgenden Aussagen über eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ sind äquivalent:*

1. *f ist Turingmaschinen-berechenbar.*
2. *f ist durch ein while-Programm berechenbar.*
3. *f ist allgemein rekursiv.*

Außerdem ist f genau dann primitiv-rekursiv, wenn f durch ein for-Programm berechenbar ist.

Die Äquivalenz von 1 und 3 wird in [30, §17] und [15, Satz 13.27, Seite 45] ausführlich bewiesen. Die Gleichwertigkeit von 1 und 2 wird in [24, Theorem 5, Chapter 9] formuliert und bewiesen. Als eine weitere Quelle dient [15, Satz 13.9, Seite 26 ff]. Schließlich kann man einen Äquivalenzbeweis für $2 \leftrightarrow 3$ in [24, Theorem 26, Chapter 9] finden. Die Äquivalenz der eingschränkten Klassen der primitiv-rekursiven Funktionen und der durch ein *for*-Prgramm berechenbaren findet man in [24, Theorem 21, Chapter 9].

Beweis zu 1⇒ 2 Wir geben eine Beweisskizze. Für diesen Zweck ist es bequem mit Turingmaschinen mit Endzuständen zu arbeiten. Gegeben sei eine Turingmaschine T mit Bandalphabet Γ, Zustandmenge Q und Übergangsfunktion δ. Wir konstruieren ein *while*-Programm $P = P_1; P_2; P_3$ über dem Alphabet $\Gamma_s = \Gamma \cup Q$ mit

- P_1 Initialisierung
- P_2 Hauptprogramm
- P_3 liest Ergebnis aus

Wir benutzen hier *while*-Programme über Γ_s-Wörtern mit

- Eingabevariable X, Ausgabe Variable Y
- Programmvariablen A, Z, E.

Idee: Zu jedem Zeitpunkt der Simulation wird AZE die Konfiguration von T sein, mit

- A = Anfangsstück,
- Z = Zustand,
- E = Endstück

P_1 initialisiert die Programmvariablen, so daß sie die Anfangskonfiguration der Turingmaschine kodieren. Die Eingabevariable X enthält den nichtleeren Teil der Inschrift des Bandes.

begin
  A := 'B';
  Z := q_0;
  E := X;
  E := **append** E 'B'
end

P_3 liest aus der TM Konfiguration

$$BwBvq_fB$$

das Ergebnis v in die Ausgabevariable Y. d.h. aus der Eingabe

$$A = BwBv, Z = q_f, E = B$$

berechnet P_3 das Ergebnis $Y = v$. Den Hauptteil der Arbeit trägt das *while*-Programm P_2

```
begin
    while Z ∉ F loop
    ...
    if Z = q_i ∧ first(E) = γ_j then P_{i,j}
    ...
    end
end
```

Der Schleifenrumpf enthält für jedes Paar q_i, γ_j aus $Q \times \Gamma$ einen **if** Befehl der angegebenen Form. Es bleibt $P_{i,j}$ zu erklären. Ist etwa $\delta(q_i, \gamma_j) = (q', \gamma', R)$ dann:

```
P_{i,j}:
begin
    A := append A γ';
    Z := q';
    E := tail E;
    if E = ε then
        E := append E 'B'
    end;
end
```

Man überzuegt sich leicht, daß nach Ausführung des Programms $P_{i,j}$ die Programmvariablen A, X, E den Zustand er Turingmaschine nach Ausführung des Übergangs $\delta(q_i, \gamma_j) = (q', \gamma', R)$ bezeichnen.

Für Transitionen der Form $\delta(q_i, \gamma_j) = (q', \gamma', L)$ oder $\delta(q_i, \gamma_j) = (q', \gamma', U)$ ist das Programm $P_{i,j}$ entsprechend zu modifizieren.

∎

Beweis zu 3⇒ 2 Dem aufmerksamen Leser wird vielleicht aufgefallen sein, daß bei der Einführung der *while*-Sprachen die vertraute Möglichkeit zur rekursiven Programmierung nicht vorkam. Der folgende Beweis zeigt, daß jede rekursive definierte Funktion, zumindest im Prinzip, durch ein iteratives Programm berechnet werden kann.

Sei f eine allgemein-rekursive Funktion. Wir konstruieren durch Induktion nach der Komplexität von f ein *while*-Programm, das f berechnet. Für die primitiv rekursiven Anfangsfunktionen ist das einfach. Auch das Einsetzungsschema stellt keine Schwierigkeit dar. Wir betrachten nur die beiden kritischen Induktionsschritte: primitive Rekursion und den μ-Operator.

primitive Rekursion Sei f definiert durch: $f(0, \vec{a}) = g(\vec{a})$ und $f(n+1, \vec{a}) = h(f(n, \vec{a}), n, \vec{a})$ Nach Induktionsvoraussetzung dürfen wir annehmen, das für die Funktionen g und h schon *while*-Programme gefunden wurden. Mit anderen Worten, wir können die Makros $Y := g(\vec{X})$ und $Y := h(U, W, Z)$ benutzen.

Eingabe: N,$\vec{X}$
Ausgabe: Z
begin
    Z := $g(\vec{X})$; IP := 0;
    for N **loop**
        Z := $h(Z, IP, \vec{X})$; IP := **succ** IP
    end
end

μ**-Operator:** Sei f definiert durch $f(\vec{X}) = (\mu g)(\vec{X})$. Nach Induktionsvoraussetzung dürfen wir das Makro $Y := g(Y, \vec{X})$ benutzen.

Eingabe: $\vec{X}$
Ausgabe: Z
begin
    Z := 0 ; U := $g(Z, \vec{X})$;
    while U **loop**
        Z := **succ** Z ; U := $g(Z, \vec{X})$
    end
end

Eine *while*-Schleife wird nur zur Simulation des μ-Operators gebraucht in allen anderen Fällen genügen *for*-Schleifen. Ist f also eine primitiv rekursive Funktion,so kann sie durch ein *for*-Programm berechnet werden.

∎

1.6 Unentscheidbare Probleme

1.6.1 Das Halteproblem

Das Halteproblem ist die Urzelle aller unentscheidbaren Probleme. Die Unentscheidbarkeit des Halteproblems wird direkt bewiesen, was noch nicht einmal besonders aufwendig oder schwierig ist. Der Nachweis der Unentscheidbarkeit für die übrigen unentscheidbaren Probleme geschieht indirekt, durch Reduktion auf ein bereits als unentscheidbar bekanntes Problem. Das Halteproblem kann für Turingmaschinen, für *while*-Programme oder jede andere Formalisierung berechenbarer Funktionen formuliert werden. Wir betrachten zunächst das Halteproblem für *while*-Programme über Wörtern aus einem festen Vokabular Γ und kommen danach auf *while*-Programme über $\mathbb{N}$ zurück. Ist P ein *while*-Programm mit einer Eingabe- und einer Ausgabevariablen, so wollen wir die durch P berechnete partielle Funktion von Γ^* nach Γ^* mit f_P bezeichnen. Das allgemeine *Halteproblem* besteht nun darin einen Algorithmus zu finden, der für jedes *while*-Programm P und jedes Wort $w \in \Gamma^*$ entscheidet, ob $f_P(w)$ definiert ist oder nicht. Nach der Churchschen These kann jeder Algorithmus durch eine *while*-Programm beschrieben werden. Das Halteproblem fragt also nach der Existenz eines *while*-Programms, nennen wir es Halt, über einem Alphabet Δ, so daß für jedes *while*-Programm P und jedes Wort $w \in \Gamma^*$gilt:

$$f_{\mathrm{Halt}}(P, w) = \begin{cases} \gamma & \text{falls } f_P(w) \downarrow \\ \varepsilon & \text{sonst} \end{cases}$$

Dabei ist γ ein beliebiges Element aus Γ. Außerdem setzen wir voraus, daß die Alphabete Γ und Δ alle Zeichen, die in *while*-Programmen erlaubt sind, enthalten. Dann ist P aufgefaßt als ein Wort über Γ oder Δ eine legale Eingabe für Halt und für P selbst. Das spezielle Halteproblem fragt nach der Existenz eine Programms halt, so daß für jedes *while*-Programm P gilt

$$f_{\mathrm{halt}}(P) = \begin{cases} \gamma & \text{falls } f_P(P) \downarrow \\ \varepsilon & \text{sonst} \end{cases}$$

Theorem 13
Das Halteproblem, sowohl das spezielle als auch das allgemeine, für while-Programme über Wörtern ist unentscheidbar.

Beweis : Offensichtlich genügt es die Nichtexistenz von halt nachzuweisen. Wir führen einen Widerspruchsbeweis und nehmen an, das halt mit der angegebenen Eigenschaft existiert. Wir definieren eine neues *while*-Programm c über Wörtern aus Δ mit Eingabevariable x und Ausgabevariable z:

```
begin
    y := halt(x);
    z := γ
    if y = ε
        then leer
        else while z ≠ ε
            loop
            z := append z z
            end
    end
end
```

Man rechnet leicht nach, daß für das *while*-Programm c gilt:

$$
\begin{array}{lll}
f_c(c) = \gamma & \text{gdw} & f_{\text{halt}}(c) = \varepsilon \\
 & \text{gdw} & f_c(c) \text{ terminiert nicht} \\
 & & \\
f_c(c) \uparrow & \text{gdw} & f_{\text{halt}}(c) = \gamma \\
 & \text{gdw} & f_c(c) \text{ terminiert}
\end{array}
$$

Das ist ein offensichtlicher Widerspruch. Die Annahme, daß ein Programm halt mit den angegebenen Eigenschaften existiert ist zu verwerfen.

■

Der vorangegangene Beweis macht Gebrauch von der Tatsache, daß wir *while*-Programme über Wörtern betrachten und deswegen eine Situation herstellen können, in der ein Programm P sich selbst als Eingabe akzeptiert. Das ist aber kein wesentlicher Teil des Arguments. Um das einzusehen betrachten wir noch die Formulierung des Halteproblems für *while*-Programme über $\mathbb{N}$. Das wesentliche Hilfsmittel dazu sind die Gödelnumerierungen.

Definition 14
Sei $\Gamma = \{s_1, \ldots, s_k\}$ ein endliches Alphabet. Eine Gödelnumerierung *ist eine injektive Funktion $g : \Sigma^* \to \mathbb{N}$. Mit anderen Worten, g ordnet jedem Wort über Γ in eindeutiger Weise eine natürliche Zahl zu.*

Beispiel 13
Sei

$$p_1, p_2, \ldots, p_n, \ldots$$

die Folge aller Primzahlen (mit $p_1 = 2$). Die Funktion g werde definiert durch:

$$g(\varepsilon) = 1 \ , \ g(x_1 \ldots x_n) = p_1^{i_1} \ldots p_n^{i_n}$$

wobei $x_j = s_{i_j}$. *Wegen der Eindeutigkeit der Primzahlzerlegung ist* g *eine Gödelnumerierung. Für* $\Gamma = \{0,1\} = \{s_1, s_2\}$ *und* $w = x_1x_2x_3 = 101 = s_2s_1s_2$ *gilt*

$$g(w) = p_1^2 p_2^1 p_3^2 = 2^2 \cdot 3^1 \cdot 5^2 = 300$$

Diese Kodierung kann man jetzt noch weiter treiben und jedem *while*-Programm P, aufgefaßt als Wort über einem endlichen Alphabet, seine Gödelnummer index(P) zuordnen. Ist der Index e gegeben, so ist ein *while*-Programm P_e mit index(P_e) $= e$ eindeutig bestimmt.

Dasselbe geht auch mit Turingmaschinen. Jeder Turingmaschine TM, aufgefaßt als Wort über einem endlichen Alphabet, wird eine Gödelnummer index(TM) zugeordnet. Ist der Index e gegeben, so ist eine Turingmaschine TM_e mit index(TM_e) $= e$ eindeutig bestimmt.

Wir geben einen zweiten Beweis für die Unterscheidbarkeit des Halteproblems, jetzt für *while*-Programme über den natürlichen Zahlen unter Ausnutzung einer Gödelisierung.

Theorem 14
Das Halteproblem, sowohl das spezielle als auch das allgemeine, für while-Programme über $\mathbb{N}$ *ist unentscheidbar.*

Beweis : Angenommen es gäbe ein *while*-Programm halt über $\mathbb{N}$ mit der Eingabevariable x und der Ausgabevariablen y, so daß für jedes *while*-Programm P mit index(P) $= e$ gilt:

halt mit Eingabe $x = e$ hält an

mit $y = 1$ falls $P_e(e)$ terminiert
mit $y = 0$ falls $P_e(e)$ nicht terminiert

Analog zu dem Programm c über Wörtern schreiben wir mit dem Programm halt als Makro das Programm confuse.

confuse:
begin
    y := halt(x); z := 1;
    if y = 0
        then leer
        else while $z \neq 0$
            loop
            z := **succ** z
            end

end
end

Der Rest der Argumentation verläuft wie im oben betrachteten Fall für Programme über Wörtern. Sei e die Gödelnummer des Programms confuse.

$$
\begin{array}{lll}
\text{confuse}(e) = 1 & \text{gdw} & \text{halt}(e) = 0 \\
 & \text{gdw} & P_e(e)\uparrow \\
 & \text{gdw} & \text{confuse}(e) \text{ terminiert nicht} \\
\text{confuse}(e)\uparrow & \text{gdw} & \text{halt}(e) = 1 \\
 & \text{gdw} & P_e(e) \text{ terminiert} \\
 & \text{gdw} & \text{confuse}(e) \text{ terminiert}
\end{array}
$$

Wir haben dabei die Abkürzung: „$P(a)\uparrow$ für P terminiert nicht bei Eingabe a“ benutzt.

1.6.2 Universelle Turingmaschinen

Die Konstruktion einer universellen Turingmaschine war schon Alan Turing selbst bekannt. Gemeint ist damit eine Turingmaschine $\mathbf{U}$, so daß für jede Turingmaschine T mit $\text{index}(T) = e$ und jedes Eingabewort w, die Maschine $\mathbf{U}$ mit Eingabe w und e die Maschine T mit Eingabe w simuliert. Analog gibt es ein universelles *while*-Programm $\text{Uprog}(x_1, \ldots, x_j, y)$ für jede feste Anzahl j von Variablen. In der Notation des nachfolgenden Theorems gilt $\text{Uprog}(x_1, \ldots, x_j, y) = \Phi^j(x_1, \ldots, x_j, y)$. Unabhängig von einem bestimmten Berechnungsmodell allein in Termini berechenbarer Funktionen formuliert läßt sich derselbe Sachverhalt wie folgt ausdrücken.

Theorem 15 (Aufzählungstheorem)
Für jedes $j > 0$ gibt es eine berechenbare Funktion $\Phi^j(x_1, \ldots, x_j, y)$, so daß für jede berechenbare Funktion $f(x_1, \ldots, x_j)$ mit $\text{index}(f) = e$ für alle $a_1, \ldots, a_j \in \mathbb{N}^j$ gilt

$$\Phi^j(a_1, \ldots, a_j, e) = f(a_1, \ldots, a_j)$$

Beweis: Nach allem was wir bisher gelernt haben spielt es keine Rolle welches Berechenbarkeitsmodell wir zugrundelegen. Wir entscheiden uns für *while*-Programme und nehmen der Einfachheit halber noch $j = 1$ an. Wir müssen also ein *while*-Programm Uprog finden, das sich bei Eingabe eines Wortes w und eines Programms P, so verhält, wie P sich bei der Eingabe w verhalten hätte. Wir müssen mit anderen Worten einen Interpreter für *while*-Programme schreiben in der *while*-Sprache selbst. Es ist zumindest plausibel, das das möglich ist. Für einen detailierten Beweis sind dann doch eine Reihe technischer Schwierigkeiten zu überwinden, siehe z.B. [15, Satz 13.15, Seite 34]. Man schaue sich auch Aufgabe 1.8.12 an.

∎

Manchmal schreiben wir auch $\phi_e(a_1, \dots , a_j)$ anstelle von $\Phi^j(a_1, \dots , a_j, e)$.

Unser Interesse an den universellen berechenbaren Funktionen Φ^j liegt darin, daß wir sie im nächsten Abschnitt als Hilfsmittel brauchen.

Das in der angegebenen Literaturreferenz konstruierte universelle *while*-Programm ist zwar einfach zu überschauen, aber umfangreich. Dasselbe trifft auf universelle Turingmaschine zu. Wenn man es jedoch darauf anlegt kann man eine universelle Turingmaschine konstruieren, die mit 4 Zuständen und einen Alphabet mit 7 Buchstaben auskommt, siehe [20]. Die Vermutung, daß keine universelle Turingmaschine mit 2 Zuständen und zwei Buchstaben existieren kann, ist bis heute unbewiesen.

1.6.3 Die Reduktionsmethode

Die Unentscheidbarkeit des Halteproblems ist nur der unvermeidliche Einstieg in das Gebiet. Wirklich interessant wird es, wenn man die Unentscheidbarkeit relevanter Probleme zeigen kann, z.B. der Frage nach der Eindeutigkeit von kontextfreier Grammatiken, siehe Theorem 69. Dieses Problem ist unentscheidbar. Aber von dem Beweis der Unentscheidbarkeit des Halteproblems dahin liegt ein weiter Weg auf dem wir jetzt ein kleines Stückchen weitergehen wollen. Abgesehen von Anfangsfällen wie dem Halteproblem, die mit Hilfe eines Diagonalisierungsarguments bewiesen werden, wird der Nachweis der Unentscheidbarkeit in allen anderen Fällen durch die *Reduktionsmethode* geliefert. Die Reduktionsmethode erlaubt von der bekannten Unentscheidbarkeit eines Problems P_1 auf die, zunächst nur vermutete, Unentscheidbarkeit eines anderen Problems P_2 zu schließen.

P_1		P_2
unentscheidbar		?
Instanz i	$\rightarrow$	Instanz $g(i)$
Antwort auf i	$=$	Antwort auf $g(i)$

Dabei wird vorausgesetzt, daß die Transformation g berechenbar ist. Man argumentiert jetzt: Wäre P_2 entscheidbar, dann wäre auch P_1 entscheidbar. Denn um die Antwort auf P_1 für die Instanz i zu erhalten, berechnet man $g(i)$ und wendet das nach Voraussetzung bekannte Entscheidungsverfahren für P_2 auf die Instanz $g(i)$ an. **Widerspruch.**

Die Probleme P_i können von recht allgemeiner Form sein. Wir geben eine formale Definition, für den Fall, daß das zu lösende Problem darin besteht zu entscheiden, ob ein Wort Element einer Sprache ist. Dieser Definition werden wir in abgewandelter Form späterwiederbegegnen.

Definition 15

Eine Sprache L_1 heißt reduzierbar *auf eine Sprache L_2, in Zeichen:*

$$L_1 \leq L_2,$$

wenn es eine berechenbare Funktion $f : \Sigma^* \to \Sigma^*$ gibt mit

$$w \in L_1 \textit{ gdw } f(w) \in L_2.$$

Wir betrachten in diesem Abschnitt Anwendungen der Reduktionsmethode auf Funktionen $f : \mathbb{N} \to \mathbb{N}$ und beginnen mit einem einfachen Beispiel. Wir wissen, daß die Funktion halt nicht berechenbar ist und wollen zeigen, daß auch die Funktion total nicht berechenbar ist:

$$\text{total}(e) = \begin{cases} 1 & \text{falls } \phi_e \text{ eine totale Fkt. ist} \\ 0 & \text{sonst} \end{cases}$$

Wir beginnen mit der Definition der Transformationsfunktion g. Zu gegebenem Index e sei $g(e)$ der Index des *while*-Programms:

Eingabe: X
Ausgabe: Y
begin
    UProg(e,e);
    Y := 0
end

wobei UProg das universelle *while*-Programm für einstellige *while*-Programme ist. Es gilt offensichtlich

$$\Phi_e(e)\downarrow \quad \Leftrightarrow \quad \text{total}(g(e)) = 1$$

mit anderen Worten:

$$\text{halt}(x) \quad = \quad \text{total}(g(x)) \qquad \text{für alle } x \in \mathbb{N}$$

Wir wollen noch einen Moment darüber nachdenken, wieso g berechenbar ist. Nach der Churchschen These ist das einleuchtend. Wenn wir den Programmtext für ein universelles *while*-Programm in das obige Programm einfügen, dann ist es nur noch eine rein syntaktische Manipulation den Index des so entstehenden Programms auszurechnen. Als infomales Argument ist das völlig in Ordnung. Es ist jedoch instruktiv einen strikt formalen Beweis anzuschauen. Dazu betrachten wir die Funktion $h(x, y)$

$$h(x,y) = \begin{cases} 0 & \text{falls } \phi_x(x)\downarrow \\ \uparrow & \text{sonst} \end{cases}$$

Diese Funktion ist sicherlich berechenbar, etwa $h(x, y) = null(\Phi^1(x, x))$, wobei $null$ die einstellige Funktion mit konstantem Wert 0 ist. Sei etwa i ein Index für h, dann kann man etwas umständlich auch $h(x, y) = \Phi^2(x, y, i)$ schreiben. Die gesuchte Funktion $g(e)$ ist offensichtlich der Index der Spezialisierung von x zu e also $\Phi^2(e, y, i)$. Das nachfolgende s-m-n-Theorem sagt gerade aus, daß der Index einer Spezialisierung berechnet werden kann aus dem Index der allgemeineren Funktion und den Werten, die substituiert werden. Manche Autoren benutzen die etwas informativere Bezeichnung „Parametrisierungstheorem“dafür.

Theorem 16 (Das s-m-n-Theorem)
Es gibt totale, berechenbare Funktionen $s^m_n : \mathbb{N}^{m+1} \to \mathbb{N}$, so daß

$$\phi_{s^m_n(y_1,\ldots,y_m,e)}(z_1, \ldots, z_n) = \phi_e(y_1, \ldots, y_m, z_1, \ldots, z_n)$$

Beweis: siehe etwa [24, Seiten 80–83].

∎

Mit der Reduktionsmethode kann auch die folgende Liste von Funktionen als unberechenbar nachgeweisen werden:

Theorem 17
Die folgenden Funktionen sind unberechenbar:

1. $\mathrm{const}(e) = \begin{cases} 1 & \textit{falls } \phi_e \textit{ eine konstante Fkt. ist} \\ 0 & \textit{sonst} \end{cases}$

2. $\mathrm{halt}(i, j) = \begin{cases} 1 & \textit{falls } \phi_i(j) \textit{ definiert ist} \\ 0 & \textit{sonst} \end{cases}$

3. $\mathrm{id}(i) = \begin{cases} 1 & \textit{falls } \phi_i(j) = j \textit{ für alle } j \\ 0 & \textit{sonst} \end{cases}$

4. $eqv(i, j) = \begin{cases} 1 & \textit{falls } \phi_i = \phi_j \\ 0 & \textit{sonst} \end{cases}$

5. $\mathrm{infbild}(i) = \begin{cases} 1 & \textit{falls } \mathrm{Bild}(\phi_i) \textit{ unendlich ist} \\ 0 & \textit{sonst} \end{cases}$

6. $\mathrm{infdef}(i) = \begin{cases} 1 & \textit{falls } \mathrm{Def}(\phi_i) \textit{ unendlich ist} \\ 0 & \textit{sonst} \end{cases}$

Wir benutzen dabei die Notation: $\mathrm{Def}(\phi) = \{n \in \mathbb{N} \mid \phi(n) \downarrow\}$ *und* $\mathrm{Bild}(\phi) = \{m \in \mathbb{N} \mid \textit{ es gibt } n \textit{ mit } m = \phi(n)\}$

Beweis: Die Beweise sind sich alle sehr ähnlich und einfach. Wir betrachten die letzte Aussage. Zu gegebenem $e \in \mathbb{N}$ sei $g(e)$ der Index des Programms:

Eingabe: X
Ausgabe: Y
begin
    UProg(e,e);
    Y := X
end

Falls UProg(e,e) nicht terminiert berechnet dieses Programm die nirgends definierte Funktion λ, anderenfalls die überall definierte Identitätsfunktion. Es gilt also

$$\mathrm{halt}(x) \quad = \quad \mathrm{infdef}(g(x)) \qquad \text{für alle } x \in \mathbb{N}$$

Die übrigen Teile des Theorems eignen sich gut als einfache Übungsaufgaben.

∎

Die in diesem Theorem genannten Resultate sind alle Spezialfälle eines einzigen Resultats. Zur Vorbereitung darauf betrachten wir die Aufgabenstellung aus einer geringfügig anderen Perspektive, aus einer mengenorientierten Perspektive. Sei, um an die erste Aussage des Theorems anzuknüpfen, $C = \{e \in \mathbb{N} \mid \phi_e \text{ ist eine konstante Fkt.}\}$. Dann ist die Funktion const aus dem Theorem die charakteristische Funktion für die Menge C und die Nichtberechenbarkeit von const ist nach Definition 13 gleichbedeutend mit der Nichtrekursivität von C. Außerdem hat die Menge C noch eine besondere Eigenschaft: ob eine Zahl e zu C gehört hängt nicht so sehr von der Zahl selbst ab, sondern von der Funktion, für welche e ein Index ist.

Theorem 18 (Satz von Rice)
Sei $S \subseteq \mathbb{N}$ eine Menge mit der Eigenschaft, daß für jedes $i \in S$ und jedes $j \in \mathbb{N}$ mit $\phi_i = \phi_j$ auch $j \in S$ gilt. Dann ist S rekursiv genau dann, wenn $S = \emptyset$ oder $S = \mathbb{N}$ gilt.

Beweis: Sei $S \neq \emptyset$,$S \neq \mathbb{N}$ und $i \in S$. Sei λ die Funktion, die nirgends definiert ist und j ein Index für λ, i.e. $\lambda(x) = \phi_j(x)$. Wir können o.B.d.A. annehmen, daß $j \notin S$ gilt. (Dazu braucht man die Voraussetzungen $S \neq \emptyset$ und $S \neq \mathbb{N}$.) Sei f eine berechenbare Funktion mit

$$\phi_{f(n)}(x) = \begin{cases} \phi_i(x) & \text{falls } \Phi(n,n)\downarrow \\ \lambda(x) & \text{sonst} \end{cases}$$

Dann gilt

$$f(n) \in S \text{ gdw } \Phi(n,n)\downarrow$$

im Widerspruch zur Unentscheidbarkeit des Halteproblems.

∎

Wir beenden diesen Unterabschnitt einem der wichtigsten elementaren Sätze der Rekursionstheorie. Bisher haben wir hauptsächlich gezeigt, daß gewisse Funktionen nicht berechenbar sind. Das Rekursionstheorem macht eine Aussage über die Existenz berechenbarer Funktionen mit vorgegebenen Eigenschaften. Gibt es ein berechenbare Funktion g, so daß für alle $n \in \mathbb{N}$ gilt $g(n) = \text{index}(g)$? mit anderen Worten gilt $\phi_e(x) = e$? Das Rekursionstheorem gibt eine Antwort.

Theorem 19 (Rekursionstheorem)
Sei $f : \mathbb{N} \to \mathbb{N}$ eine totale berechenbare Funktion. Dann gibt es für jedes $j > 0$ eine Zahl $n \in \mathbb{N}$ mit der Eigenschaft:

$$\Phi^j(x_1, \ldots, x_j, n) = \Phi^j(x_1, \ldots, x_j, f(n))$$

oder anders notiert

$$\phi_n = \phi_{f(n)}$$

Beweis: O.B.d.A. $j = 1$. Es gibt eine totale berechenbare Funktion g, die jedem $i \in \mathbb{N}$ den Index des folgenden Programms zuordnet:

$P_{g(i)} =$

```
Eingabe: X1
Ausgabe: X1
begin
    X2 := Uprog(i,i);
    X1 := Uprog(X1,X2)
end
```

Wir überlegen uns, wie die von $P_{g(i)}$ berechnete Funktion $\phi_{g(i)}$ aussieht. Bei Eingabe k wird zunächst $\phi_i(i)$ berechnet, unabhängig von der Eingabe. Falls

$\phi_i(i)$ nicht definiert ist, ist auch $\phi_{g(i)}(k)$ nicht definiert. Anderenfalls wird $\phi_{\phi_i(i)}(k)$ berechnet und als Ergebnis ausgegeben. Zusammenfassend:

$$\phi_{g(i)}(k) = \begin{cases} \phi_{\phi_i(i)}(k) & \text{falls } \phi_i(i)\downarrow \\ \uparrow & \text{sonst} \end{cases}$$

Sei m der Index der Funktion $f \circ g$, i.e. $\phi_m = f \circ g$. Dann ist $n = g(m)$ der gesuchte „Fixpunkt", wie die folgende simple Rechnung zeigt:

$$\begin{array}{rcll} \phi_n(k) & = & \phi_{g(m)}(k) & \text{Definition von } n \\ & = & \phi_{\phi_m(m)}(k) & \text{Definition von } g, \text{ wobei man beachte, daß} \\ & & & \phi_m \text{ total ist} \\ & = & \phi_{f \circ g(m)}(k) & \text{Wahl von } m \\ & = & \phi_{f(n)}(k) & \text{Definition von } n \end{array}$$

■

Kommen wir zurück zu der vor dem Rekursionstheorem gestellten Frage. Sei f die Funktion, die jeder Zahl $n \in \mathbb{N}$ den Index des Programms:

```
Eingabe: X
Ausgabe: Z
begin
    Z := n
end
```

zuordnet. Mit anderen Worten heißt das, daß für alle x gilt $\Phi(x, f(n)) = n$. Nach dem Rekursionstheorem gibt es einen Index e mit $\Phi(x, e) = \Phi(x, f(e))$. Zusammengenommen ergibt sich also für alle x, wie gewünscht, $\phi_e(x) = e$.

1.6.4 Das Postsche Korrespondenzproblem

In diesem Abschnitt wollen wir die Unentscheidbarkeit eines komplizierteren, kombinatorischen Problems zeigen. Die Bedeutung dieses Problems liegt vor allem darin, daß es immer noch recht einfach zu verstehen ist und trotzdem häufig als Ausgangspunkt für Reduktionsbeweise benutzt werden kann. Ein Beispiel dafür werden wir in Theorem 69 kennen lernen.

Definition 16
Eine Instanz des Postschen Korrespondenzproblems, kurz PKP, besteht aus einer endlichen Folge

$$K = \langle (x_1, y_1), \ldots (x_k, y_k) \rangle$$

von Wortpaaren $x_i, y_i \in \Sigma^+$ über einem Alphabet Σ. Eine Lösung des Postschen Korrespondenzproblems ist eine Folge $\langle i_1, \dots i_n \rangle$ von Indizes mit $n > 0$, so daß

$$x_{i_1} x_{i_2} \dots x_{i_n} = y_{i_1} y_{i_2} \dots y_{i_n}$$

gilt.

Beispiel 14

$$K: \langle \begin{array}{ll} (1 & 111) \\ (10111 & 10) \\ (10 & 0) \end{array} \rangle$$

Die Lösungsfolge $\langle 2, 1, 1, 3 \rangle$ führt zu der Gleichung

$$101111110 = 101111110$$

$$101111110 = 101111110$$

Beispiel 15

Das System

$$K_1: \langle \begin{array}{ll} (10 & 101) \\ (011 & 11) \\ (101 & 011) \end{array} \rangle$$

besitzt keine Lösung. Als erstes Paar kommt nur $(10, 101)$ in Frage. Die einzige Fortsetzung von hier aus ist $(101, 1010)$ mit Hilfe des dritten Paares. Für jede weitere Fortsetzung kommt immer nur das dritte Paar in Frage und die rechte Seite bleibt immer um einen Buchstaben länger als die linke.

Theorem 20

Das Postsche Korrespondenzproblem ist unentscheidbar.

Beweis: Um die Unentscheidbarkeit des Postschen Korrespondenzproblems zu zeigen, machen wir Gebrauch davon, daß die Unentscheidbarkeit des Halteproblems für Turingmaschinen schon bekannt ist. Das geschieht in zwei Schritten. Zunächst wird in Lemma 21 das Halteproblem auf das modifizierte Postsche Korrespondenzproblem, MPKP, reduziert und danach in Lemma 22 MPKP auf PKP zurückgeführt.

∎

Definition 17

Die Modifikation des Postschen Korrespondenzproblems, in der gefordert wird, daß eine Lösung $i_1, \dots, i_k$ zusätzlich die Bedingungen

1. $i_1 = 1$,

2. $i_j \neq 1$ *für* $0 < j \leq k$

erfüllt, nennen wir das **M***odifizierte* **P***ostsche* **K***orrespondenz***P***roblem, kurz MPKP*

Lemma 21
Zu jeder Turingmaschine $T = \langle \Gamma, Q, F, \delta \rangle$ *und jeder Anfangskonfiguration* w *von* T *gibt es eine Instanz des MPKP ,* $Post_T$*, so daß gilt:*

$$T \text{ hält auf } w$$
$$\text{gdw}$$
$$Post_T \text{ besitzt eine Lösung}$$

Beweis: Das Alphabet für $Post_T$ ist $\Gamma_{post} = \Gamma \cup \{\#\} \cup Q$. Das PKP $Post_T$ besteht aus der Menge der Paare (x, y) wie sie in der folgenden Tabelle aufgeführt sind. Dabei sind a, b Elemente aus Γ.

x	y	
$\#$	$\#q_0w\#$	Anfang
	Kopierregeln	
d	d	$d \in \Gamma \cup \{\#\}$
	Überführungsregeln	
qa	$q'c$	$\delta(q, a) = (q', c, U)$
qa	cq'	$\delta(q, a) = (q', c, R)$
bqa	$q'bc$	$\delta(q, a) = (q', c, L)$
$\#qa$	$\#q'Bc$	$\delta(q, a) = (q', c, L)$
$q\#$	$q'c\#$	$\delta(q, B) = (q', c, U)$
$q\#$	$cq'\#$	$\delta(q, B) = (q', c, R)$
$bq\#$	$q'bc\#$	$\delta(q, B) = (q', c, L)$
$\#q\#$	$\#q'Bc$	$\delta(q, B) = (q', c, L)$
	Löschregeln	
qa	q	$q \in F$
$aq\#$	$q\#$	$q \in F$
$q\#\#$	$\#$	$q \in F$

Zunächst soll versucht werden die Grundidee des Beweises zu erklären. Die Berechnung der Turingmaschine soll simuliert werden durch die Suche nach einer Lösung des angegebenen modifizierten Postschen Problems. Dabei wird versucht durch Anfügen von Wortpaaren aus der obigen Tabelle zwei gleiche Wörter zu erzeugen. Der Anfang ist in der modifizierten Form des Problems vorschrieben. Das obere Wort besteht dabei allein aus dem Trennzeichen #, das untere Wort enthält zwischen zwei Trennzeichen eingeschlossen die Anfangskonfiguration von T. Diese Situation: zwischen den letzten beiden Trennzeichen des unteren Wortes steht die aktuelle Konfiguration von T, das obere Wort hinkt um eine Konfiguration hinterher, wird solange beibehalten bis ein Endzustand auftritt. Dann treten die Löschregeln in Aktion und gestatten das obere und das untere Wort aneinander anzugleichen. Soweit die grobe Idee. Das spezielle hier betrachtete MPKP ist deterministisch, d.h. wenn überhaupt eine Lösung existiert, dann gibt es nur eine. (Dabei sehen wir von Wiederholungen ab, wenn $i_1, \dots, i_k$ eine Lösung ist, dann natürlich auch $i_1, \dots, i_k, i_1, \dots, i_k$.)

Anstelle eines ausführlichen Nachweises, daß die angegebene Definition von $Post_T$ die gewünschte Eigenschaft besitzt, rechnen wir anhand der Turingmaschine T_{par} aus Beispiel 5 und dem Eingabewort 100101 detailiert vor, wie die Lösung zu $Post_{T_{par}}$ gefunden werden kann.

$\#$ Anfang
$\#q_0 100101\#$

Hier gibt es nur eine Fortsetzungsmöglichkeit:

$\#q_0 1$ $\delta(q_0, 1) = (q_1, 1, R)$
$\#q_0 100101\#1q_1$

Auch hier gibt es nur eine Fortsetzungsmöglichkeit:

$\#q_0 10$ Kopierregel
$\#q_0 100101\#1q_1 00$

Wir schreiben hier und auch im folgenden nicht alle Anwendungen der Kopierregel auf und springen gleich zum Abschluß der aktuellen Konfiguration.

$\#q_0 100101\#$ Kopierregel
$\#q_0 100101\#1q_1 00101\#$

Nach Anwendung einer weiteren Kopierregel ist das nächste Zeichen, das im unteren Wort aber nicht im oberen auftritt ein Zustandssymbol. Dafür gibt es keine Kopierregel. Es muß jetzt die zweite Aktion von T_{par} simuliert werden. Wir springen jetzt in der Simulation ein ganzes Stück nach vorne bis T_{par} das ganze Eingabewort durchlaufen hat. Man beachte, daß wir jetzt die beiden Wörter nicht mehr in voller Länge sondern nur noch die relevanten rechten Enden notieren. Der letzte Schritt vor der Simulation des nächsten

Zustandsübergangs nachdem die Turingmaschine das rechte Ende des Eingabewortes erreicht hat:

$\ldots\#100101$ Kopierregel
$\ldots\#100101q_1\#100101$

und jetzt der Übergang selbst:

$\ldots\#100101q_1\#$ $\delta(q_1, B) = (q_3, B, R)$
$\ldots\#100101q_1\#100101Bq_3\#$

Nach einer weiteren Runde von Kopierregeln erscheint zum ersten Mal der Endzustand:

$\ldots\#100101Bq_3\#$ $\delta(q_3, B) = (q_f, 0, U)$
$\ldots\#100101Bq_3\#100101Bq_f0\#$

Zunächst geschieht noch nichts Neues, es folgt eine weitere Runde von Kopierregeln bis zu:

$\ldots\#100101B$
$\ldots\#100101Bq_f0\#100101B$

Jetzt wird zum ersten Mal eine Löschregel anwendbar:

$\ldots\#100101Bq_f0$
$\ldots\#100101Bq_f0\#100101Bq_f$

Die Differenz der beiden Wortlängen ist um einen Buchstaben kleiner geworden. Es folgen wieder Anwendungen der Kopierregeln von

$\ldots\#100101Bq_f0\#$
$\ldots\#100101Bq_f0\#100101Bq_f\#$

bis

$\ldots\#10010101$
$\ldots\#1001010Bq_f\#100101$

Jetzt kann die zweite Form der Löschregeln angewendet werden:

$\ldots\#10010101Bq_f\#$
$\ldots\#1001010Bq_f\#10010q_f\#$

Durch Wiederholung dieser Prozedur erreicht man schließlich

$\ldots\#1q_f\#$
$\ldots\#1q_f\#q_f\#$

Woraus sich durch Anwendung der letzten Löschregel

$\ldots\#1q_f\#q_f\#\#$
$\ldots\#1q_f\#q_f\#\#$

ergibt.

■

Lemma 22

Das Postsche Korrespondenzproblem kann auf das modifizierte Postsche Korrespondenzproblem zurückgeführt werden.

Beweis: Ziel dieses Lemmas ist nachzuweisen, daß aus der Entscheidbarkeit von PKP auch die Entscheidbarkeit von MPKP folgen würde. Aus Lemma 21 wissen wir, daß MPKP unentscheidbar ist, also werden wir auch die Unentscheidbarkeit des uneingeschränkten PKP als Konsequenz erhalten. Nun zum Beweis selbst. Für

$$K = \{(x_1, y_1), \ldots, (x_k, y_k)\}$$

definieren wir

$$f(K) = \{(x'_1, y'_1), \ldots, (x'_k, y'_k), (\#\#, \#)\},$$

so daß K eine Lösung mit $i_1 = 1$ und $i_j \neq 1$ für $j > 1$ besitzt, genau dann wenn $f(K)$ irgendeine Lösung besitzt. Für $x_i = a^i_1 \ldots a^i_{n_1}$, $y_i = b^i_1 \ldots b^i_{m_i}$ ist:

für $i = 1$
$x'_1 = \#a^1_1\# \ldots \#a^1_{n_1}$ $\quad y'_1 = \#b^1_1\# \ldots \#b^1_{m_1}\#$
für $2 \leq i \leq k$
$x'_i = \#a^i_1\# \ldots \#a^i_{n_i}$ $\quad y'_i = b^i_1\# \ldots \#b^i_{m_i}\#$
zur Vereinheitlichung der Notation setzen wir
$x'_{k+1} = \#\#$ $\quad y'_{k+1} = \#$

Ist $i_1, \ldots, i_n$ eine Lösung von K mit $i_1 = 1$ und $i_j \neq 1$ für alle $2 \leq j \leq n$, dann ist $i_1, \ldots, i_n, k+1$ eine Lösung für $f(K)$.

Sei jetzt umgekehrt $i_1, \ldots, i_n$ eine Lösung von $f(K)$. Aus den ersten Buchstaben der Paare in $f(K)$ sieht man, daß nur $i_1 = 1$ oder $i_1 = k+1$ möglich ist. Letzter kann aber nicht eintreten, da das Paar $(\#\#, \#)$ nicht sinnvoll fortgesetzt werden kann. Im ncahsten Schritt ist wieder nur dasselbe Paar $(\#\#, \#)$ angelegt werden, was offensichtlich nie zu zwei gleichen Wörtern führt. Also muß schon einmal $i_1 = 1$ gelten. Das zweite Wort des Paares enthält immer in # Zeichen mehr als das erste, bis das Paar $(\#\#, \#)$ angelegt wird. An dieser Stelle können wir aber aufhören, das erste und zweite Wort müssen gleich sein, auch wenn noch weitere Indizes folgen. Mit anderen Worten, wir können annehmen, daß $i_n = k+1$ gilt und für alle $1 \leq j < n$ gilt $i_j \neq k+1$. Dann ist aber $i_1, \ldots, i_{n-1}$ eine Lösung für K, welche den geforderten Einschränkungen genügt.

∎

1.6.5 Unentscheidbarkeit der Beasy Beaver Funktion

In Abschnitt 1.2.3 haben wir spezielle Turingmaschinen — fleißige Biber — betrachtet, mit deren Hilfe wir eine Funktion BB konstruierten. Ziel dieses Abschnitts ist es die Unentscheidbarkeit dieser Funktion zu zeigen. Dieser Beweis ist durchsichtiger und mit weniger technischen Details belastet, wenn man ihn nicht für Turingmaschinen sondern für *while*-Programme durchführt. Dazu müssen wir das Analogon zur Biberfunktion für *while*-Programme über den natürlichen Zahlen definieren. Als erstes benötigen wir ein Maß für die Größe von *while*-Programmen:

Definition 18 *Die* Länge $|P|$ *eines while-Programms P über natürlichen Zahlen ist rekursiv wie folgt definiert:*

$$\begin{array}{lcrcl} |X := Y| & = & |X := 0| & = & 1 \\ |X := \mathbf{succ}\ Y| & = & |X := \mathbf{pred}\ Y| & = & 1 \\ |\mathbf{begin}\ \ C \mathbf{end}\ | & = & 2 + |C| & & \\ |\mathbf{while}\ \ X \mathbf{loop}\ \ C \mathbf{end}\ | & = & 1 + |C| & & \end{array}$$

Dabei sind X, Y Variablen und C ist ein while-Programm (ohne begin-end).

Beispiel 16 *Das kleinst mögliche while-Programm hat die Länge 2 und berechnet die Identität:*

| *Eingabe X; Ausgabe X;* **begin** **end** $| = 2$

Jedes nicht triviale while-Programm hat die Länge 3, z.B.:

| **begin** *X :=* **succ** *X;* **end** $| = 3$

Für die üblichen Makros und $i, j \in \mathbb{N}$ kann man zeigen, daß $|X := i| = i + 1$, $|Z := X * Y| = 9$ und $|Z := i * j| = 11 + i + j$ gilt.

Die Länge eines *while*-Programmes ist ein zu der Anzahl der Zustände einer Turingmaschine analoges Maß. Fleißige Biber (über *while*-Programmen) sind *while*-Programme P, die einstellige Funktionen berechnen und eine natürliche Zahl als Ergebnis zurückliefern. Dabei interessiert uns nur der Funktionswert $f_P(0)$ von P an der Stelle 0 (Analog zum leeren Band als Anfangskonfiguration). Die Biberfunktion ist dann wie folgt definiert (beachte, daß der kleinste Biber mindestens Länge 2 hat):

Definition 19 (Biberfunktion)
$\mathrm{BB}(n) = \max(\{f_P(0) : P$ *ist ein while-Programm mit* $|P| \leq n\})$

Theorem 23
Die Biber Funktion $\mathrm{BB}(n)$ *ist nicht berechenbar.*

Beweis: Wir führen einen indirekten Beweis und nehmen an, daß BB berechenbar ist. Es gibt also ein *while*-Programm, das BB berechnet. Wir bezeichnen dieses *while*-Programm ebenfalls mit BB. Die Länge von BB sei k. BB ist sicherlich nicht identisch mit einem der trivialen *while*-Programme aus Beispiel 16. Wir können deswegen $k = |BB| \geq 4$ annehmen.

Wir zielen auf einen Widerspruch ab und betrachten dazu folgendes *while*-Programm P:

```
Eingabe X;
Ausgabe Y;
begin
  N := 10*k;
  Y := BB(N);
  Y := succ  Y;
end
```

Offensichtlich gilt $f_P(0) = BB(10 * k) + 1$

Wir können die Länge von P mit $|P| = 2k + 24 \leq 2k + 8 * 4 \leq 10k$ grob abschätzen und erhalten dann die widersprüchliche Ungleichung $\mathrm{BB}(10*k) \geq f_P(0) = \mathrm{BB}(10 * k) + 1$.

∎

1.6.6 Wortprobleme

Definition 20
Ein Semi-Thue-System T *ist durch eine Menge von Ersetzungsregeln* $l \to r$ *gegeben, wobei* l, r *Wörter über einem Alphabet* Σ *sind.*

Für $u, w \in \Sigma^*$ *heißt* w ableitbar *aus* u *in* T, *wenn es eine Ersetzungsregel* $l \to r$ *in* T *gibt, so daß* l *ein Teilwort von* u *ist, etwas* $u = u_1 l u_2$ *und* $w = u_1 r u_2$ *gilt. Wir schreiben dafür*

$$u \Rightarrow w$$

$\Rightarrow^*$ *ist der transitive Abschluß von* $\Rightarrow$ *und* $\Rightarrow^+$ *ist der transitive Abschluß von* $\Rightarrow$, *wobei der Fall, daß* w *aus* u *durch* 0 *Regelanwendungen hervorgeht ausgeschlossen ist.*

T *heißt eine* $Thue - System$, *wenn für jede Regel* $l \to r$ *in* T *auch die umgekehrte Regel* $r \to l$ *in* T *liegt.*

Beispiel 17 (Beispiel eines Semi-Thue-Systems) *T sei gegeben durch die Ersetzungsregeln über* $\Sigma = \{|, *\}$*:*

$$\begin{array}{ccc} | & \rightarrow & * \\ | & \rightarrow & || \end{array}$$

Beispiel einer Ableitung in T:

$|$
Regel 2
$||$
$||$
Regel 2
$|||$
$|||$
Regel 1
$|*|$

Definition 21
Das allgemeine Wortproblem *für Semi-Thue-Systeme ist das Problem für beliebige Semi-Thue-Systeme S und zwei beliebige Wörter* $x, y \in \Sigma^*$ *über dem Alphabet von S zu entscheiden, ob*

$$x \Rightarrow_S^* y$$

gilt.

Das spezielle Wortproblem *fragt nach der Ableitbarkeit von*

$$x \Rightarrow_S^* y$$

für beliebige $x, y \in \Sigma^*$ *aber für ein festes Semi-Thue-System S.*

Theorem 24

1. *Das allgemeine Wortproblem für Semi-Thue-Systeme ist unentscheidbar.*

2. *Es gibt Semi-Thue-Systeme S, so daß das spezielle Wortproblem für S unentscheidbar ist.*

In der Gruppentheorie werden konkrete Gruppen häufig durch die Angabe von *erzeugenden Elementen* und *Relationen* beschrieben. Dazu gibt man eine Menge

$$E = \{g_1, \ldots, g_n\}$$

von erzeugenden Elementen und eine Menge von Gleichungen

$$R = \{w_1 = u_1, \ldots, w_m = u_m\}$$

zwischen Gruppenwörtern, die aus diesen Elementen aufgebaut sind, an. Mit $G(E, R)$ bezeichnen wir die durch E und R eindeutig bestimmte Gruppe.

Beispiel 18

$$E = \{s, t\} \quad , \quad R = \{s^5 = 1, t^2 = 1, tst = s^{-1}\}$$

definiert die Diedergruppe D_5. Dabei ist t die Rotation und s eine Zentralachsenspiegelung des regulären 5-Ecks.

Definition 22
Das allgemeine Wortproblem *für Gruppen ist das Problem für beliebige Repräsentationen (E, R) und zwei beliebige Gruppenwörter x, y zu entscheiden, ob in $G(E, R)$*

$$x = y$$

gilt.

Das spezielle Wortproblem *fragt, ob*

$$x = y$$

für beliebige x, y aber für eine feste Gruppe $G = G(E, R)$ gilt.

Theorem 25

1. *Das allgemeine Wortproblem für Gruppen ist unentscheidbar.*
2. *Das allgemeine Wortproblem für Halbgruppen ist unentscheidbar.*
3. *Das allgemeine Wortproblem für Monoide ist unentscheidbar.*
4. *Das allgemeine Wortproblem für freie Gruppen (d.h. $R = \emptyset$) ist lösbar.*
5. *Das allgemeine Wortproblem für abelsche Gruppen ist lösbar.*
6. *Es gibt eine Halbgruppe $G = (E, R)$ mit unentscheidbarem speziellen Wortproblem.*

Beispiel 19 $E = \{a, b, c, d, e, f\}$

$$\begin{array}{llll}
ac = ca & e & = efe \\
ad = da & feca & = ecf \\
ae = ea & fedb & = edf \\
bc = cb & fce & = acef \\
bd = db & fde & = bdef \\
be = eb & &
\end{array}$$

(Dana Scott, 1956). Zitiert nach [30, Seite 203].

1.7 Lehrbücher

1. [24]
   Didaktisch gut aufbereitet. Als Berechenbarkeitsmodell werden schwerpunktmäßig *while*-Programme behandelt. Turingmaschinen und μ-rekursive Funktionen werden aber auch sorgfältig eingeführt. Ebenso werden die Anfänge der Rekursionstheorie (Rekursionstheorem, rek. und rek.aufz.Mengen) behandelt. Manchmal ist der Text allerdings ein bißchen schlampig.

2. [37]
   In dieser umfassenden Einführung in die theoretische Informatik gibt es auch ein Kapitel über Berechenbarkeitstheorie. Als Basis werden primitiv rekursive und μ-rekursive Funktionen benutzt. Der Stoff wird verständlich erklärt, verlangt aber vom Leser die Bereitschaft, sich in die konsequent eingesetzte konzise, formale Notation einzuarbeiten. Enthält gute Übungsaufgaben, mit Lösungshinweisen.

3. [23]
   Umfangreiches Lehrbuch. Verständlich aber knapp geschrieben. Umfangreiche Übungsaufgaben mit Angabe des Schwierigkeitsgrades.

4. [22]
   Lesenswerte Biographie Alan Turings, die auch versucht die wissenschaftlichen Leistungen von Turing, in den durch die angestrebte Allgemeinverständlichkeit gesetzten Grenzen, zu erklären.

5. [40]
   Für den Berechenbarkeitsteil entwickelt der Autor ein eigenes Programmiersprachenmodell, das zwar den *while*-Programmen ähnelt, aber doch

verschieden ist. Weitere Berechenbarkeitsmodelle sind Turingmaschinen, partiell-rekursive Funktionen, Registermaschinen. Übungsaufgaben, ohne Lösungen. Betont formale Darstellungsweise.

6. [10]
   Behandelt in kurzen thematisch abgerundeten, anschaulich aufbereiteten Kapiteln fast alle Themen, die wir hier behandeln. Wendet sich allerdings mehr an Schüler der 13. Klasse als an Universitätsstudenten.

7. [41]
   Kein Lehrbuch, aber ein interessanter neuer Ansatz. Es wird versucht die wesentlichen Ideen und Motivationen zu vermitteln, ohne dabei allzu sehr in technische Details einzusteigen. Gedacht als Ergänzung zu Lehrbuch und Vorlesung. Enthält keine Aufgaben, aber Testfragen mit Antworten.

8. [30]
   Führt wesentlich weiter in die Theorie der Berechenbarkeit ein, als in der Vorlesung beabsichtigt. Typisch mathematisches Lehrbuch. Ohne Übungsaufgaben.

9. [11]
   Weiterführende Literatur zum Thema Berechenbarkeit und Komplexität. Sehr klar und eingängig geschrieben. Betrachtet konsequent Turingmaschinen über endlichen Bäumen. Einige ungewöhnliche Fragen findet man hier in elementarer Darstellung: was bedeutet das "speed-up" Theorem für die praktische Programmierung?, wo tritt die aus dem Beweis der Unentscheidbarkeit des Halteproblems bekannte Selbstanwendung in Compiler-Compilern auf? Gibt es immer optimale Algorithmen? Mit Übungsaufgaben, ohne Lösungen.

1.8 Übungsaufgaben

Übungsaufgabe 1.8.1

Gegeben sei das Eingabealphabet $\Sigma = \{0, 1\}$ und das leere Zeichen B. Geben Sie eine deterministischen Turingmaschine an, die ein Wort aus Σ^* so um eine Stelle nach links verschiebt, daß

- die Turingmaschine im Endzustand auf den Anfang des verschobenen Wortes zeigt und

- außer dem Wort nur leere Zeichen (B) auf dem Band stehen.

Wenn die Turingmaschine in der Anfangskonfiguration

...	B	0	1	0	B	B	...
		$\uparrow$					

gestartet wird, dann soll sie schließlich mit folgender Konfiguration anhalten:

...	0	1	0	B	B	B	...
	$\uparrow$						

Übungsaufgabe 1.8.2

Betrachten Sie die folgende Turingmaschine: $\Sigma = \{0,1\}$, $\Gamma = \{B\} \cup \Sigma$, $Q = \{q_0, q_1, q_f, r_0, r_1, l_0, l_1, c_0, c_1, d_0, d_1, t\}$, $F = \{q_f\}$ und δ gegeben durch (wobei $i = 0,1$ ist):

Q	Γ	Q	Γ	
q_0	0	r_0	B	R
q_0	1	r_1	B	R
q_0	B	q_f	B	U

Q	Γ	Q	Γ	
r_i	0	r_i	0	R
r_i	1	r_i	1	R
r_i	B	c_i	B	L

Q	Γ	Q	Γ	
c_i	i	q_1	B	L
c_i	$1-i$	t	$1-i$	R
c_i	B	q_f	B	U

Q	Γ	Q	Γ	
q_1	0	l_0	B	L
q_1	1	l_1	B	L
q_1	B	q_f	B	U

Q	Γ	Q	Γ	
l_i	0	l_i	0	L
l_i	1	l_i	1	L
l_i	B	d_i	B	R

Q	Γ	Q	Γ	
d_i	i	q_0	B	R
d_i	$1-i$	t	$1-i$	R
d_i	B	q_f	B	U

Q	Γ	Q	Γ	
t	0	t	0	R
t	1	t	1	R
t	B	t	B	R

(a) Geben Sie die Konfigurationen an, die die Turingmaschine bei Eingabe von 010 durchläuft.

(b) Was bewirkt der Zustand t?

(c) Welche Sprache akzeptiert die Turingmaschine?

Übungsaufgabe 1.8.3

Wieviele fleißige Biber gibt es in Abhängigkeit der Anzahl n von Zuständen (exklusive dem Endzustand)? Das heißt, wieviele deterministischen Turingmaschinen über dem Bandalphabet $\Gamma = \{0,1\}$ mit n Zuständen existieren?

Übungsaufgabe 1.8.4

In dieser Aufgabe sollen Sie Makros für die gebräuchlichen logische Operationen NOT, OR und AND angeben. Benutzen Sie dabei ausschließlich die elementaren Operationen der jeweiligen Makrosprache und verwenden Sie 1 für den Wahrheitswert *wahr* sowie 0 für *falsch*. Gehen Sie davon aus, daß die Makros korrekt verwendet werden, d.h. x und y sind 0 oder 1.

Schreiben Sie jeweils for- und while-Makros für:

(a) z := NOT x, welches z die Negation von x zuweist.

(b) z := x OR y, welches z die Disjunktion von x und y zuweist.

(c) z := x AND y, welches z die Konjunktion von x und y zuweist.

Übungsaufgabe 1.8.5

Konstruieren Sie eine deterministische Turingmaschine mit zweiseitig unendlichem Band über dem Bandalphabet $\{0, 1, B\}$, die genau alle Wörter $w \in \{0, 1\}^*$ mit gerader Anzahl Nullen und gerade Anzahl Einsen akzeptiert.

Die Wörter 00 und 10011010 sollen zum Beispiel akzeptiert werden, nicht aber 01 und 00110.

Übungsaufgabe 1.8.6

Betrachten Sie vollständig geklammerte aussagenlogische Formeln mit den üblichen Boolschen Operatoren $\wedge$ (Konjunktion), $\vee$ (Disjunktion) und $\neg$ (Negation) über den Wahrheitswerten 1 (wahr), 0 (falsch) und n verschiedene aussagenlogischen Variablen $x_1, \ldots, x_n$, wie zum Beispiel $(0 \wedge (\neg((x_1 \wedge x_2) \vee ((\neg x_1) \wedge 1))))$.

Gegeben sei eine nichtdeterministische Turingmaschine T_1 mit Startzustand q^0 und einzigem Endzustand q^e über dem Bandalphabet $\Gamma = \{0, 1, \wedge, \vee, \neg, (,), B\}$, die zu jeder variablenfreien aussagenlogischen Formel auf dem Eingabeband dessen Wahrheitswert bestimmt und ihn auf das ansonsten leere Band schreibt, d.h. die Eingabe wird komplett gelöscht. Bei syntaktisch nicht korrekten Formeln hält T_1 nicht. Zum Beispiel hält T_1 bei Anfangskonfiguration $Bq^0(1 \vee 0)B$ mit Bq^e1B, bei Anfangskonfiguration $Bq^0(1\vee)B$ hält sie nicht.

(a) Konstruieren Sie eine nichtdeterministische Turingmaschine T mit Startzustand q_0 und Endzustand q_e über dem Bandalphabet

$$\Gamma \cup \{x_1, \ldots, x_n\},$$

die bei jeder aussagenlogischer Formel die Variablen durch eine Belegung mit Nullen und Einsen ersetzt und anhält. Der Schreib-/Lesekopf soll am Ende über dem ersten Zeichen der Formel stehen. Zum Beispiel soll T bei Anfangskonfiguration $Bq_0(x_1 \vee (\neg x_1))B$ in $Bq_e(1 \vee (\neg 1))B$ übergehen können.

(b) Konstruieren Sie eine nichtdeterministische Turingmaschine T', die genau alle erfüllbaren aussagenlogischen Formeln akzeptiert.

Übungsaufgabe 1.8.7

Das Simulationslemma beschreibt, wie Turingmaschinen mit zweiseitig unendlichem Band auf Turingmaschinen mit einseitig unendlichem Band simuliert werden können. Der Beweis ist konstruktiv und liefert zu jeder Turingmaschine T_1 mit zweiseitig unendlichem Band b_1 eine äquivalente Turingmaschine T_2 mit einseitig unendlichem Band b_2.

Geben Sie möglichst genaue obere Schranken an für

(a) die Anzahl von Zustände von T_2 in Abhängigkeit von der Anzahl n von Zustände von T_1 und

(b) die Anzahl von Schritten, die T_2 bei Eingabe b_2 durchläuft, in Abhängigkeit von der Anzahl m von Schritten, die T_1 bei Eingabe von b_1 durchläuft.

Übungsaufgabe 1.8.8

Betrachten Sie die while-Sprache über Wörter (im folgenden als W1 bezeichnet), wobei $\Gamma = \gamma_1, \ldots, \gamma_n$ gilt.

Wir führen folgende Modifikation durch (im folgenden als W2 bezeichnet): Statt **append** w_1 w_2 gibt es für jedes Zeichen γ ein Ausdruck $\mathbf{append}_\gamma$ w_1 der γ and w_1 anhängt.

(a) Zeigen Sie: Jede W2 berechenbare Funktion ist auch W1 berechenbar.

(b) Schreiben Sie ein W1-Makro **if** $x = \gamma$ **then** A **else** B **end**, welches A ausführt, falls $x = \gamma$ mit $\gamma \in \Gamma$ gilt und B sonst.

(c) Zeigen Sie, daß jede W1 berechenbare Funktion W2 berechenbar ist.

Übungsaufgabe 1.8.9

Eine anderes Grundkonzept für eine möglichst rudimentäre abstrakte Programmiersprache, die trotzdem erlaubt alle berechenbaren Funktionen zu programmieren, sind Sprungbefehle. Darauf baut die folgende Sprache auf.

Definition 23 (*goto*-**Programme**)

Algorithmus	::=	**begin** *Anweisungen* **end**
Anweisungen	::=	*[Marke ':'] Anweisung* |
		Anweisungen ';'
		[Marke ':'] Anweisung
Anweisung	::=	*leer* | *Zuweisung* | *Spezial*
Zuweisung	::=	*Var ':=' Ausdruck*
Ausdruck	::=	*0* | *Var* | **succ** *Var*
		| **pred** *Var*
Var	::=	*beliebiger Bezeichner*
Marke	::=	*beliebiger Bezeichner*
		$\neq$ *Var*
Spezial	::=	**goto** *Marke* | **halt**|
		if *Var '=' '0'* **then**
		goto *Marke* **end**

Zeigen Sie, daß jedes *while*-Programm kann durch ein *goto*-Programm simuliert werden.

Die Umgekehrung, jedes *goto*-Programm kann durch ein *while*-Programm simuliert werden, gilt auch, siehe z.B. [15].

Übungsaufgabe 1.8.10

Betrachten Sie folgende 2-dimensionale Variante einer deterministischen Turingmaschine mit einseitig unendlichem Band. Das „Band“ ist nach rechts sowie nach unten unendlich und nach links sowie oben beschränkt:

x				...
				...
				...
⋮	⋮	⋮	⋮	

Es gibt einen Schreib-/Lesekopf, der sich am Anfang über der linken oberen Zelle befindet (x). Neben den Kopfbewegungen R,L und U gibt es zwei weitere: der Kopf bewegt sich nach oben (O), der Kopf bewegt sich nach unten (D). Die 2-dimensionale Turingmaschine hält, wenn sie in einen Endzustand übergeht oder sich über das linke bzw. obere Ende des Bandes hinausbewegt. Ein Wort w wird von einer Turingmaschine mit 2-dimensionalem Band akzeptiert, wenn das initiale Band w (irgendwie) enthält und die Turingmaschine ausgehend von der Startkonfiguration irgendwann anhält.

(a) Geben Sie eine Transformationsvorschrift f an, mit der das zwei-dimensionale Band bijektiv auf ein einseitig unendliches 1-dimensionales Band abgebildet wird.

(b) Zeigen Sie (informell), daß jede von einer Turingmaschine mit 2-dimensionalem Band akzeptierte Sprachen auch von einer Turingmaschine mit einseitig unendlichem Band akzeptiert wird. Dazu benötigen Sie die Transformation f. Welche wichtige Eigenschaft muß f besitzen?

Übungsaufgabe 1.8.11

Betrachten Sie die Beweisskizze zu der Behauptung, daß jede von einer deterministischen Turingmaschine berechnete Funktion auch von einem while-Programm über Wörtern berechnet werden kann.

(a) Geben Sie das fehlende Programmfragment P_3 an.

(b) Geben Sie zu Transitionen der Form $\delta(q_i, \gamma_j) = (q', \gamma', L)$ schematisch das entsprechende Programmfragment $P_{i,j}$ an.

(c) Zeigen Sie, daß die Menge aller Sprachen, die von nichtdeterministischen Turingmaschinen mit Endzuständen akzeptiert werden, auch von *while*-Programmen über Wörtern akzeptiert werden. Verwenden und Modifizeiren Sie dazu die Programmfragmente P_1 und P_2.

Übungsaufgabe 1.8.12

Die Existenz einer universellen Funktion $\Phi^1(x, y)$ für alle einstelligen berechenbaren Funktionen scheint, oberflächlich betrachtet, die Tür zu öffen zu einem Diagonalargument, das die Nichtexistenz einer solchen Funktion zeigt. Das Argument könnte so lauten: Wenn $\Phi^1(x, y)$ berechenbar ist, dann ist auch die Funktion $g(x) = \Phi^1(x, x) + 1$ berechenbar. Der Index von g sei e. Dann ergibt sich der Widerspruch

$$\Phi^1(e, e) = g(e) = \Phi^1(e, e) + 1$$

Decken Sie auf, wo der Trugschluß liegt.

Übungsaufgabe 1.8.13

(a) Zeigen Sie mit Hilfe des Rekursionstheorems, daß es ein *while*-Programm gibt, das anhält und 1 ausgibt, wenn es seinen Index als Eingabe erhält, und ansonsten nie anhält.

(b) Schreiben Sie ein Programm, das sich selbst als Ausgabe produziert (Zeilenumbrüche und Leerzeichen können dabei ignoriert werden). Benutzen Sie dazu folgende Pseudo-Sprache:

Programm	::=	Initialisierung Ausgabe
Initialisierung	::=	Bezeichner '=' Zeichenkette ';'
Ausgabe	::=	'print(' Zeichenkette { Bezeichnerliste }');'
Bezeichnerliste	::=	Bezeichner | Bezeichnerliste ',' Bezeichner
Bezeichner	::=	a | ... | z

Eine Zeichenkette ist eine beliebige Folge von alphanumerischen Zeichen (auch ").

Die Ausgabeanweisung print ist ähnlich zu printf in der Programmiersprache C definiert: Das erste Argument ist eine Zeichenkette, welche das Ausgabeformat bestimmt. Zu jedem %s, das im ersten Argument vorkommt, muß als zusätzliches Argument ein (vorher initialisierter) Bezeichner angeben werden. Der Inhalt dieses Bezeichners wird an der entsprechenden Stelle der ersten Zeichenkette ausgegeben.

Ein mögliches Programm ist:

```
a = "42";
b = "deep "thought"";
print("%s: %s",b,a);
```

Ausgabe ist:

```
deep "thought": 42
```

(c) Schreiben Sie ein C-Programm (oder ein Programm in einer anderen Programmiersprache), welches seinen eigenen Programmtext ausgibt.

Übungsaufgabe 1.8.14

Läßt man in der Definition des μ-Operators, die Berücksichtigung undefinierter Funktionswerte außer acht, so erhält man den folgenden μ_0-Operator:

$$(\mu_0 x_1 g)(x_2, \ldots, x_n) = \begin{cases} \text{das kleinste } n \text{ mit} & \text{falls ein solches } n \text{existiert} \\ g(n, \vec{x}) = 0 & \\ \uparrow & \text{sonst} \end{cases}$$

Zeigen Sie, daß es eine rekursive Funktion $g(x, y)$ gibt, so daß $\mu_0 x(g(x, y))$ nicht berechenbar ist.

Übungsaufgabe 1.8.15

Betrachten Sie eine Variante von while-Programmen über natürlichen Zahlen, bei denen das **while** x **loop** A **end** Konstrukt durch **repeat** A **until** x ausgetauscht wird. Die Semantik der *repeat*-Schleife sei wie folgt: Zuerst wird der Rumpf ausgeführt. Ist x danach Null, so wird die Schleife verlassen, ansonsten wird wieder der Rumpf ausgeführt und die Bedingung getestet, usw.

Analog zu *while*-berechenbaren Funktionen sprechen wir bei diesen *repeat*-Programmen von *repeat*-berechenbaren Funktionen.

(a) Zeigen Sie, daß jede *repeat*-berechenbare Funktion *while*-berechenbar ist.

(b) Schreiben Sie *repeat*-Makros $Z := X + Y$ und $Z := X * Y$ für die Addition und die Multiplikation.

(c) Schreiben Sie eine *repeat*-Makro $Z := X > 0$, welches Z die 0 zuweist, falls X=0 gilt, und welches Z die 1 zuweist, falls X>0 gilt.

(d) Zeigen Sie, daß ein Konstrukt **if** $\;\mathrm{X} \neq 0\;$ **then** A **end** (mit wohlbekannter Semantik) in ein entsprechendes *repeat*-Programmfragment transformiert werden kann.

Hinweis: Tranformieren Sie A in ein Programmfragment A', in welchem ein „Schalter“ Y (Variable mit Wert 0 oder 1) in geeigneter Weise Einfluß auf die Berechnungen nimmt.

(e) Zeigen Sie, daß jede *while*-berechenbare Funktion *repeat*-berechenbar ist.

Übungsaufgabe 1.8.16

(a) Zeigen Sie, daß die folgende Instanz des Postschen Korrespondenzproblems lösbar ist:

$$\langle (aa, a), (b, aa), (a, aab) \rangle$$

(b) Zeigen Sie, daß die folgende Instanz des modifizierten Postschen Korrespondenzproblems keine Lösung besitzt.

$$\langle (01, 0), (0, 101), (101, 0) \rangle$$

Übungsaufgabe 1.8.17

(a) Betrachten Sie folgende eingeschränkte Version des Postschen Korrespondenzproblems, bei der die Wortpaare aus einer *geraden* Anzahl von Zeichen bestehen:

Definition 24 (Das gerade Postsche Korrespondenzproblem) *Eine Instanz des* geraden Postschen Korrespondenzproblems*, kurz EPKP, besteht aus einer endlichen Folge* $K = \langle(x_1, y_1), \dots (x_k, y_k)\rangle$ *von Wortpaaren* $x_i, y_i \in \Sigma^*, |x_i| = 2k', |y_i| = 2l, k', l \in \mathbb{N}$ *über einem Alphabet* Σ.

Eine Lösung des EPKPs ist eine Folge $\langle i_1, \dots i_n\rangle$ *von Indizes mit* $n > 0$, *so daß* $x_{i_1}x_{i_2}\dots x_{i_n} = y_{i_1}y_{i_2}\dots y_{i_n}$ *gilt.*

Zeigen Sie, daß folgende Instanz des EPKPs keine Lösung besitzt:

$$\langle(aaaa, aa), (aa, ab), (aa, abab), (aaaa, abab)\rangle$$

(b) Zeigen Sie mit Hilfe eines Reduktionsbeweises, daß das EPKP unentscheidbar ist.

(c) Bei folgender Einschränkung des Postschen Korrespondenzproblems ist verlangt, daß eine Lösung mit dem ersten Wortpaar *endet*:

Definition 25 (Umgekehrtes MPKP) *Eine Instanz des* umgekehrten modifizierten Postschen Korrespondenzproblems*, kurz UMPKP, besteht aus einer endlichen Folge*

$$K = \langle(x_1, y_1), \dots (x_k, y_k)\rangle$$

von Wortpaaren $x_i, y_i \in \Sigma^*$ *über einem Alphabet* Σ. *Eine Lösung des UMPKPs ist eine Folge* $\langle i_1, \dots i_n, \rangle$ *von Indizes mit* $n > 0$, *so daß* $x_{i_1}\dots x_{i_n} = y_{i_1}\dots y_{i_n}$ und $i_n = 1$ *gilt.*

Zeigen Sie, daß die folgende Instanz des UMPKP eine Lösung besitzt.

$$\langle(1, 01), (0, 1), (011, 0), (1, 00)\rangle$$

(d) Zeigen Sie mit Hilfe eines Reduktionsbeweises, daß das UMPKP unentscheidbar ist.

Übungsaufgabe 1.8.18

(a) Betrachten Sie den Beweis zur Unentscheidbarkeit des MPKPs und des PKPs.

  (i) Wieso kann die im Beweis zur Unentscheidbarkeit des MPKPs angegebene Transformation nicht direkt zur Reduktion des Halteproblems auf das PKP verwendet werden?

(ii) Wie müßte man die Transformation ändern, um damit das Halteproblem auf das PKP zu reduzieren?

(b) Zeigen Sie, daß das allgemeine Wortproblem für Semi-Thue-Systeme unentscheidbar ist.

Hinweis: Der Beweis ähnelt dem Beweis zur Unentscheidbarkeit des MPKPs.

Übungsaufgabe 1.8.19

(a) $f : \mathbb{N}^n \to \mathbb{N}$ und $g : \mathbb{N}^k \to \mathbb{N}$ seien primitiv rekursiv. Zeigen Sie unter ausschließlicher Verwendung der Definition primitiv rekursiver Funktionen, daß für jedes $i = 1, \ldots, n$ auch

$$f(x_1, \ldots, x_{i-1}, g(x_i, \ldots, x_{i+k-1}), x_{i+k}, \ldots, x_{n+k-1})$$

primitiv rekursiv ist.

(b) Zeigen Sie, daß folgende Funktion $f : \mathbb{N}^2 \to \mathbb{N}$ primitiv rekursiv ist.

$$m(x, y) = \begin{cases} x^2, & \text{falls } x \geq y \\ y^2, & \text{falls } x < y \end{cases}$$

Übungsaufgabe 1.8.20

Zeigen Sie, daß die folgenden Funktionen primitiv rekursiv sind. Sie dürfen dabei Gebrauch machen von den Ergebnissen aus Beispiel 11.

(a)
$$\begin{array}{lcll} \mathrm{add}(x, y) & = & 0 & \text{falls } x < y \\ \mathrm{add}(x, y) & = & x - y & \text{falls } y \leq x \end{array}$$

(b)
$$\begin{array}{lcll} \mathrm{add}(x, y) & = & x - y & \text{falls } y < x \\ \mathrm{add}(x, y) & = & y - x & \text{falls } x \leq y \end{array}$$

Übungsaufgabe 1.8.21

In welchem Jahr wurde die erste Arbeit über TURING Maschinen veröffentlicht? Bitte markieren/unterstreichen Sie die richtigen Antworten!

- 1902 ○
- 1936 ○
- 1950 ○

Übungsaufgabe 1.8.22

Welche der folgenden Aussagen sind richtig und welche sind falsch? Bitte kreuzen Sie die richtigen Kästchen an!

Für jede TURING Maschine muß die Übergangsfunktionan an mindestens einer Stelle undefiniert sein.	◯ Ja	◯ Nein
Ist für eine TURING Maschine für eine Eingabewort die komplette Folge aller Konfigurationen bekannt, dann kann man daraus die Übergangsfunktion rekonstruieren	◯ Ja	◯ Nein

Übungsaufgabe 1.8.23

Welche der folgenden Behauptungen über nichtdeterministische Turing Maschinen sind wahr? Bitte kreuzen Sie die zutreffenden Kästchen an!

Sie haben mehrere Bänder	◯ Ja	◯ Nein
Der Funktionswert der Übergangsfunktion ist immer eine Menge	◯ Ja	◯ Nein
Sie muß nicht terminieren	◯ Ja	◯ Nein
Man kann nicht vorhersagen welche Menge sie akzeptiert	◯ Ja	◯ Nein

Übungsaufgabe 1.8.24

Welche der folgenden Fragmente von *while*-Programmen sind syntaktisch korrekt über den natürlichen Zahlen? In dieser Aufgabe sind keine Makros zugelassen. Bitte kreuzen Sie die zutreffenden Kästchen an!

while X **loop** X := 2 **end**	◯ Ja	◯ Nein
while X **loop** **while** X **loop** X := X **end** **end**	◯ Ja	◯ Nein
while =/= **succ** **succ** 0 **loop** X := **pred** X **end**	◯ Ja	◯ Nein
X:= Y;Y := X	◯ Ja	◯ Nein
X := **succ** **succ** Y	◯ Ja	◯ Nein
while X **loop** Y := pred Y **end**	◯ Ja	◯ Nein

1.9 Lösungen zu den Aufgaben

Lösung zu Aufgabe 1.8.1

$Q = \{q_0, s_0, s_1, s_B\}, F = \{q_f\}$. δ sei gegeben durch (wobei $i = 0, 1, B$ ist):

Q	Γ	Q	Γ	
q_0	0	q_0	0	R
q_0	1	q_0	1	R
q_0	B	s_B	B	L

Q	Γ	Q	Γ	
s_i	0	s_0	i	L
s_i	1	s_1	i	L
s_i	B	q_f	i	U

Lösung zu Aufgabe 1.8.2

(a) $B_{q_0}010B$
$BB_{r_0}10B$
$BB1_{r_0}0B$
$BB10_{r_0}B$
$BB1_{c_0}0B$
$BB_{q_1}1BB$
$B_{l_1}BBBB$
$BB_{d_1}BBB$
$BB_{q_f}BBB$

(b) Wenn die TM im Zustand t ist, dann kann sie keinen Endzustand mehr erreichen, d.h. die Eingabe wird nicht akzeptiert.

(c) Die TM erkennt genau alle „Palindrome“ von Σ^*.

Lösung zu Aufgabe 1.8.3

Zu jedem Zustand $q \neq q_0$ und Buchstaben $\gamma \in \Gamma$ gibt es $(n+1) * |\Gamma| * |\{R, L, U\}| = 6 * (n+1)$ mögliche Zustandsübergänge. Diese alle miteinander kombiniert ergeben $(6n+6)^{2n}$ Möglichkeiten, d.h. soviele verschiedene fleißige Biber mit n Zuständen gibt es.

Die genaue Anzahl von Bibern (also ganz bestimmten deterministischen Turingmaschinen) ist gleich der Anzahl verschiedener Zusandsübergangsfunktionen $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, U\}$. Zu gegebenen n ist der Definitionsbereich all dieser Funktionen identisch. Sie unterscheiden sich aber im Wertebereich. Betrachte dazu folgende Tabelle:

Q	Γ	Q	Γ	
q_1	0	$\{q_0, \dots, q_n\}$	$\{0,1\}$	$\{R, L, U\}$
q_1	1			
q_2	0			
q_2	1			
q_3	0			
q_3	1			
q_4	0			
q_4	1			
$\vdots$				
q_n	0			
q_n	1			

Die linke Hälfte der Tabelle stellt den Definitionsbereich von δ dar. Er enthält $2n$ Einträge. Diese sind fest und ändern sich nicht egal welches δ man betrachtet.

Die rechte Seite entspricht dem Wertebereich und kann variiert werden. In jeder Zeile gibt es $(n+1) \cdot 2 \cdot 3$ mögliche Einträge. Bei $n = 1$ gibt es also $(6(1+1))^2$ verschiedene Zustandsübergangsfunktionen (alle Variationen der ersten Zeile mit allen der zweiten kombiniert). Bei $n = 2$ erhöht sich die Anzahl auf $(6(2+1)^{2 \cdot 2}$ usw. Also insgesamt $(6(n+1))^{2n}$ Kombinationen.

Unter diesen Kombinationen sind natürlich viele Biber, die die gleiche Funktion berechnen oder nie anhalten. Es sind aber trotzdem verschiedene Turingmaschinen (also Biber), da deren Zustandsübergangsfunktionen alle verschieden sind.

Lösung zu Aufgabe 1.8.4

(a) Hier ein *for*-Makro:

```
z := NOT x:
begin
   z := 1;
   for  x loop
      z := 0
   end
end
```

(b) Hier ein *while*-Makro:

```
z := x OR y:
begin
   z := x;
   u := y;
   while  u loop
      z := 1;
      u := 0
   end
end
```

(c) Hier ein goto-Makro:

```
z := x AND y:
begin
   z := 0;
   if  x=0 then goto  M end
   if  y=0 then goto  M end
   z := 1;
   M:
end
```

Lösung zu Aufgabe 1.8.5

Die grundlegende Idee ist folgende: Die Eingabe wird von links nach rechts durchlaufen. Dabei merkt man sich wie viele Nullen und Einsen modulo 2 das bisher überlaufene Wort enthält. Diese Information wird in die Zustände der Turingmaschine kodiert:

q_{00}	Anfangszustand, gerade Anzahl Nullen und gerade Anzahl Einsen
q_{10}	ungerade Anzahl Nullen, gerade Anzahl Einsen
q_{01}	gerade Anzahl Nullen, ungerade Anzahl Einsen
q_{11}	ungerade Anzahl Nullen, ungerade Anzahl Einsen

Wenn am Ende der Eingabe Zustand q_{00} erreicht ist, dann geht die Turingmaschine in den Endzustand q_e über und ansonsten läuft sie weiter nach rechts über das Band und hält nie.

Q	Γ	Q	Γ	
q_{00}	0	q_{10}	0	R
q_{00}	1	q_{01}	1	R
q_{00}	B	q_e	B	R
q_{10}	0	q_{00}	0	R
q_{10}	1	q_{11}	1	R
q_{10}	B	q_{10}	B	R
q_{01}	0	q_{11}	0	R
q_{01}	1	q_{00}	1	R
q_{01}	B	q_{01}	B	R
q_{11}	0	q_{01}	0	R
q_{11}	1	q_{10}	1	R
q_{11}	B	q_{11}	B	R

Lösung zu Aufgabe 1.8.6

(a) Wir konstruieren eine TM, die einmal von links nach rechts über die Eingabe läuft und die Variablen nichtdeterministisch durch Nullen und Einsen ersetzt. Danach läuft sie wieder zum Anfang der Eingabe zurück. Diese TM hält immer, insbesondere falls die Belegung die Formel erfüllt. Die Schwierigkeit besteht darin, sich zu merken, daß eine Variable x_i schon durch eine Null oder eine Eins ersetzt wurde. Wir kodieren wir diese Informationen in die Zustände.

Dazu betrachten wir Folgen f der Länge n über den Zahlen 0 und 1. $f(i)$ bezeichne das i-te Element der Folgen (erste Element hat Index 1). Also z.B. $f = (0, 1, 0, 1, 1)$ und $f(2) = 1$.

Zu allen solchen Folgen f definieren wir einen Zustand q_f mit folgenden Zustandsübergängen, wobei $i = 1, \ldots, n$ gilt:

Q	Γ	Q	Γ	
q_f	x_i	q_f	$f(i)$	R
q_f	0	q_f	0	R
q_f	1	q_f	1	R
q_f	B	q_l	B	L
q_f	)	q_f	)	R
q_f	(	q_f	(	R
q_f	$\vee$	q_f	$\vee$	R
q_f	$\wedge$	q_f	$\wedge$	R
q_f	$\neg$	q_f	$\neg$	R

Q	Γ	Q	Γ	
q_l	0	q_l	0	L
q_l	1	q_l	1	L
q_l	)	q_l	)	L
q_l	(	q_l	(	L
q_l	$\vee$	q_l	$\vee$	L
q_l	$\wedge$	q_l	$\wedge$	L
q_l	$\neg$	q_l	$\neg$	L
q_l	B	q_e	B	R

Dieser Teil der Turingmaschine ist noch deterministisch. Ausgehend vom Startzustand q_0 erlauben wir eine nichtdeterministische Auswahl einer Variablenbelegung, indem für alle obigen Folgen f und Terminalzeichen γ folgende Übergänge definiert sind:

Q	Γ	Q	Γ	
q_0	γ	q_f	γ	U

Es gibt also $|\Gamma|2^n$ viele mögliche Zustandsübergänge vom Startzustand aus.

(b) Man benutze T, um nichtdeterministisch alle Variablen durch eine Belegung zu ersetzen. Danach benutzte man T_1, um nachzuprüfen, ob diese Belegung die Eingabeformel erfüllt. Wenn nicht, dann geht man von q^e in einen Zustand über, so daß die Turingmaschine nicht anhält.

Lösung zu Aufgabe 1.8.7

Wenn man sich die einzelnen Simulationsschritte im Beweis und die transformierte TM ansieht, bekommt man folgende oberen Schranken:

(a) Sei Q_2 die Menge Zustände der simulierten Maschine. Die Anzahl Zustände $|q_1|$ der simulierenden TM ist

$$|Q_1| = |Q_2 \cup \{q^{-1}, q_R, q_L, q_R^{-1}, q_L^{-1} | q \in Q_2\}| = n + 5n = 6n \ .$$

(b) $3m$, denn jede Schritt der zweiseitig unendlichen TM wird durch maximal 3 Schritte der einseitig unendlichen TM simuliert.

Lösung zu Aufgabe 1.8.8

(a) Wir transformieren jedes W2 *while*-Programm in ein äquivalentes W1 *while*-Programm, indem jedes Vorkommen von $\mathbf{append}_\gamma\ w$ durch

$z := \gamma;$
append $w\ z$

ersetzt wird, wobei z eine neue Variable ist.

(b) **if** $x = \gamma$ **then** A **else** B **end**:
begin
$y := \gamma_1$;
$z := x$;
while $z \neq \gamma$ **loop**
$y := \epsilon$;
$z := \gamma$;
B
end
while $y \neq \epsilon$ **loop**
A
$y := \epsilon$;
end
end

Dabei sei $\gamma_1 \neq \epsilon$ ein beliebiges Zeichen aus Γ und y, z neue Variablen.

(c) Wir ersetzen jedes Vorkommen von $x :=$ **append** u v durch folgendes Konstrukt. Dabei machen wir von einem Makro $x :=$ **reverse** y Gebrauch, welches x die Umkehrung von y zuweist.

$x := u$;
$y :=$ **reverse** v;
while y **loop**
$h :=$ **head** y;
$y :=$ **tail** y;
if $h = \gamma_1$ **then** $x :=$ **append**$_{\gamma_1}$ x **end**
$\vdots$
if $h = \gamma_n$ **then** $x :=$ **append**$_{\gamma_n}$ x **end**
end

Lösung zu Aufgabe 1.8.9

Jedes Programmstück der Form

while v **loop** A **end**;
nächste Anweisung

kann ersetzt werden durch

```
begin
W:  if v = 0 then goto F end;
    A;
    goto W;
F: nächste Anweisung
end
```

Lösung zu Aufgabe 1.8.10

(a) Um die Funktion f anzugeben ordnen wir jeder Zelle der 2-dimensionalen Turingmaschine wie folgt ein 2-Tupel $(x, y) \in \mathbb{N}^2$ zu:

y\ x	1	2	3	4	...
1	1	3	6	...	
2	2	5	9	...	
3	4	8	13	...	
⋮	⋮	⋮	⋮		

Die Zellen der einseitig unendlichen Turingmaschine seien von links nach rechts beginnend mit 1 durchnumeriert. Eine Zelle $\langle x, y\rangle$ der 2-dimensionalen Turingmaschine bilden wir dann auf die Zelle $f(x, y) = 0.5((x+y)^2 - x - 3y + 2)$ der Turingmaschine mit einseitig unendlichem Band ab.

f läßt sich wie folgt berechnen: Aus dem Schema erkennt man

$$f(x, y) = f(1, x + y - 1) + x - 1$$

und

$$\begin{aligned} f(1, y) &= f(1, y-1) + y - 1 \\ &= f(1, y-2) + y - 2 + y - 1 \\ &= \ldots \\ &= f(1,1) + 1 + 2 + \ldots + y - 2 + y - 1 \\ &= 1 + \tfrac{(y-1)y}{2} \end{aligned}$$

Letzteres eingesetzt in obiges $f(x, y)$ ergibt die geschlossene Formel.

Aus dem obigen Schema „erkennt“ man, daß f bijektiv ist.

(b) Wir müssen jede Turingmaschine T_2 mit 2-dimensionalen Band in eine Turingmaschine T_1 mit einseitig unendlichem Band transformieren, so daß gilt:

T_2 akzeptiert ein Wort w gdw T_1 akzeptiert w.

Dazu müssen wir erst klar stellen, was es bedeutet, daß T_2 ein Wort akzeptiert. Wir benutzen dazu die Funktion f. Ist w ein Eingabewort, dann wird es gemäß der durch f definierten Reihenfolge auf das zwei-dimensionale Band geschrieben. T_2 akzeptiert w, wenn T_2 bei Start mit diesem Eingabeband hält.

$\Gamma = \Sigma_1 \cup \{B\}$ sei das Bandalphabet von T_2. Das zwei-dimensionale Band wird gemäß der Abbildungsvorschrift von f auf das ein-dimensionale Band abgebildet. Jede Zelle des ein-dimensionalen Bandes entspricht also genau einer Zelle des zwei-dimensionalen Bandes.

Ein Schritt von T_2 wird durch eine Vielzahl Schritten von T_1 simuliert. Die Anzahl dieser Schritte ist im Gegensatz zur Simulation der Turingmaschine mit zweiseitig unendlichem Band auf der Turingmaschine mit einseitig unendlichem Band nicht konstant beschränkt, sondern hängt von der Position x, y auf dem zwei-dimensionalen Band ab. Wir müssen also die Schrittweite für jede Position auf dem ein-dimensionalen Band berechnen. Dazu muß f berechenbar sein. Zu jeder Richtung läßt sich die Schrittweite in Abhängigkeit von x, y wie folgt berechnen:

Richtung		Schrittweite
U	$f(x,y) - f(x,y)$	$= 0$
O	$f(x,y) - f(x,y-1)$	$= -x + 1 - y + 1$
R	$f(x,y) - f(x+1,y)$	$= x + y$
L	$f(x,y) - f(x-1,y)$	$= -x - y + 3$
D	$f(x,y) - f(x,y+1)$	$= x - 1 + y$

Um diese Berechnungen durchzuführen, müssen wir uns von der aktuellen Position die x und y Koordinaten der entsprechenden Zelle des zwei-dimensionalen Bandes merken. Wir simulieren dazu ein zweites Band unterhalb des ersten, indem wir in jeder Zelle Paare von Symbolen (a, b) schreiben, so daß a dem Inhalt der Zelle der zwei-dimensionalen Turingmaschine entspricht und b für Berechnungen zu Verfügung steht. Befindet sich der Schreib-/Lesekopf oberhalb einer Zelle (a, b), so befinden sich in den unteren Einträge der Zellen (nach rechts hin) das Paar (x, y) kodiert als Binärzahlen jeweils getrennt und eingerahmt durch eine Trennzeichen # ($a_i, b_j \in \{0, 1\}$, „oberes“ Band nicht angegeben):

		...			...				...				...
B	B	...	#	a_n	...	a_0	#	b_m	...	b_0	#	B	...

Das initiale Band wird also auf folgendes ein-dimensionales Band abgebildet:

#	1	#	1	#	B	B

Das Bandalphabet der simulierenden Turingmaschine TM_1 ist also $\Gamma \times \{0, 1, B, \#\}$ Simuliert wird wie folgt (beispielhaft anhand von $\delta(q, a) = (r, b, R)$) und der Konfiguration:

		...	a	...		...		...			...
B	B	...	#	x	#	y	#	$x + y$	#	B	...

(i) a wird auf dem „oberen“ Band durch b ersetzt und geht in einen „Berechnungsmodus“ über. Der Folgezustand r wird ebenfalls in diesem Modus gespeichert.

		...	b	...		...		...			...
B	B	...	#	x	#	y	#	$x + y$	#	B	...

(ii) Gemäß des zu simulierenden Zustandsübergang wird die Schrittweite aus den Koordinaten berechnet (einfache Additionen und Subtraktionen) und rechts von den derzeitigen Koordinaten auf das Band mit anschließendem Trennsymbol geschrieben. Das obere Band wird bei all diesen Aktionen nicht verändert.

		...	a	...		...		...			...
B	B	...	#	x	#	y	#	$x + y$	#	B	...

(iii) Dann wird das untere Band um die Schrittweite nach rechts verschoben. Der Wert $x+y$ dient dabei als Zähler, er soll anschließend entfernt werden. Desweiteren sollen die Koordinaten entsprechend geändert werden. Entsteht dabei eine Zahl kleiner als 1, so hält die Turingmaschine, da T_2 über die Begrenzung gelaufen ist.

	...		...	a	...		...					...
B	...	B	...	#	$x + 1$	#	y	#	B	B	B	...

(iv) Von dem Berechnungsmodus aus geht T_1 in den Folgezustand r über.

Die anderen Zustandsübergänge werden analog behandelt.

Lösung zu Aufgabe 1.8.11

(a) P_3 liest aus der Konfiguration $BwBvq_fB$ der Turingmaschine das Ergebnis v heraus, d.h. aus der Eingabe $A = BwBv, (Z = q_f, E = B)$ berechnet P_3 das Ergebnis $Y = v$:

```
Eingabe A
Ausgabe Y
begin
    Y := tail A;
    U := head Y;
    while U≠ 'B' loop
        Y := tail Y;
        U := head Y;
    end
    Y := tail Y;
end
```

(b) Wir benutzen (neben einigen anderen Makros) das *while*-Programm zum Umkehren von Wörtern P_{inv} (als Makro y := reverse(x) verwendet):

```
begin
    A := reverse(A);
    X := head(A);
    A := tail(A);
    if A=ε then
        A := 'B';
    end
    A := reverse(A);
    Z := q';
    H := γ';
    E := tail E;
    E := append H E;
    E := append X E;
end
```

(c) Das folgende *while*-Programm benutzt Breitensuche, um die nichtdeterministische Turingmaschine zu simulieren. Es berechnet mit dieser Suchstrategie alle erreichbaren Konfigurationen bis ein Endzustand erreicht ist. Die Konfigurationen werden in einer Variablen K gespeichert.

Als Trennsymbol wird dabei $\# \notin \Gamma$ verwendet. Desweiteren verwenden wir die Programmfragmente P_1 (unverändert) und P_2 (modifiziert) aus dem Beweis zu Theorem 12. P_3 können wir nicht gebrauchen, da kein Funktionsergebnis zurückgeliefert wird.

Zuerst die an P_2 vorgenommene Modifikation: Da ein Übergang $\delta(q_i, \gamma_j)$ der nichtdeterministischen Turingmaschine eine Menge von mögliche Aktionen beschreibt, muß P_2 alle möglichen Konfigurationen berechnen. Das Ergebnis soll eine von $P'_{i,j}$ gelieferte Folge T' (ein Wort) all dieser durch # voneinander getrennten Konfigurationen sein. Nach dieser Berechnung wird T' an T angehängt. Dann entfernt man von T die erste Konfiguration und schreibt sie in die Variablen A, Z und E (dieser Teil ist hier nicht weiter ausgeführt).

P_2:

begin
    while Z $\notin$ F **loop**
        A' := A; Z' := Z; E' := E;
        ...
        if Z=$q_i \wedge$ first(E)=γ_j **then**
            $P'_{i,j}$
            hole nächste Konfiguration A Z E aus T
        end
        ...
    end
end

Das modifizierte $P'_{i,j}$ ist eine Folge der ursprünglichen $P_{i,j}$. Für jede mögliche Transition in $\delta(q_i, \gamma_j)$ muß ein entsprechendes $P^k_{i,j}$ vorkommen $(1 \leq k \leq |\delta(q_i, \gamma_j)| =: n)$:

$P'_{i,j}$:

begin
    $P^1_{i,j}$
    T := append T '#' A Z E;
    A := A'; Z := Z'; E := E;
    $\vdots$
    $P^n_{i,j}$
    T := append T '#' A Z E;
end

Das simulierende *while*-Programm sieht dann wie folgt aus:

begin
P_1
T := ϵ;
P_2
end

Lösung zu Aufgabe 1.8.12

Die Gleichungen $\Phi^1(e,e) = g(e) = \Phi^1(e,e) + 1$ sind nur dann ein Widerspruch, wenn $\Phi^1(e,e)$ definiert ist. Anderenfalls gilt $\Phi^1(e,e) = \Phi^1(e,e) + 1$, da beide Funktionswerte undefiniert sind.

Lösung zu Aufgabe 1.8.13

(a) Definiere Programmumschreibefunktion $f : \mathbb{N} \to \mathbb{N}$ so, daß $f(i)$ der Index des folgenden *while*-Programmes ist (d.h. jedes Programm mit Index i wird zu folgendem Programm umgeschrieben):

Eingabe: x
Ausgabe: y
begin
if $x =$ i **then**
$y := 1$
else
$z := 1$; **while** z **loop** $z := 1$ **end**
end
end

f ist total (klar) und berechenbar, da für jedes i die Gödelnummer für obiges Programm ausgerechnet werden kann.

Nach dem Rekursionstheorem gibt es eine natürliche Zahl m, so daß $\phi_m = \phi_{f(m)}$ ist. Nach Definition von f bedeutet dies:

$$\phi_m(x) = \begin{cases} 1 & \text{, für } x = m \\ \uparrow & \text{, sonst} \end{cases}$$

(b) Folgendes Programm macht das gewünschte:

s = "s = "%s"; print("%s",s,s);";
print("s = "%s"; print("%s",s,s);",s,s);

Man beachte, daß der Inhalt der Zeichenkette dem ersten Argument von print entspricht.

(c) Bei der folgenden Lösung sind noch einige unnötigen Zeilenumbrüche vorhanden:

```
#include<stdio.h>
main(){
char*a="#include<stdio.h>%cmain()
{char*a=%c%s%c,q='%c',b='%c%c';
printf(a,'%cn',q,a,q,q,b,b,b,b,'%cn');}%c",
q='"',
b='\\';
printf(a,'\n',q,a,q,q,b,b,b,b,'\n');}
```

Hier noch eine Lösung in Lisp:

((lambda (x) (list x (list (quote quote) x))) (quote (lambda (x) (list x (list (quote quote) x)))))

Lösung zu Aufgabe 1.8.14

Wir setzen:

$$g(x,y) = \begin{cases} \uparrow & \text{falls } \phi_y(y)\uparrow \text{ und } x = 0 \\ 0 & \text{falls } \phi_y(y)\downarrow \text{ und } x = 0 \\ 0 & \text{falls } x > 0 \end{cases}$$

Offensichtlich ist $g(x, y)$ berechenbar. Man sieht aber leicht, daß

$$\mu_0 x g(x,y) = \begin{cases} 1 & \text{falls } \phi_y(y)\uparrow \\ 0 & \text{falls } \phi_y(y)\downarrow \end{cases} \quad \text{gilt.}$$

Das heißt, $\mu_0 x g(x, y) = \text{halt}(y)$ und damit ist $\mu_0 x$ nicht berechenbar.

Lösung zu Aufgabe 1.8.15

(a) Es genügt zu zeigen, daß jede *repeat*-Schleife **repeat** A **until** x durch ein entsprechendes *while*-Makro simuliert werden kann:

```
repeat A until x:
begin
   A;
   while x loop
      A;
   end
end
```

(b)

```
Z := X+Y:
begin
   Z := X;
   U := succ Y;
   repeat
      Z := succ Z;
      U := pred U;
   until U;
   Z := pred Z;
end
```

```
Z := X*Y:
begin
   Z := 0;
   U := succ Y;
   repeat
      Z := Z+X;
      U := pred U;
   until U
   U := X;
   repeat
      Z := pred Z;
      U := pred U;
   until U;
end
```

(c) Z := X>0;
begin
  Z := **succ** X;
  U := X;
  repeat
    U := **pred** U;
    Z := **pred** Z;
  until U;
end

Für X=0 wird das gewünschte berechnet (ausprobieren). Bei X>0 werden die Anweisungen im Rumpf X mal ausgeführt, d.h. von Z wird X abgezogen. Da Z Anfangs X+1 war, ist am Ende Z=1.

(d) $X_1, \ldots, X_n$ seien alle Variablen, die in A vorkommen. Wir führen neue Variablen $X_i', X_i'', (i = 1, \ldots, n), Y, Y'$ und Y'' ein. Ersetze in A alle Vorkommen von X_i durch X_i' und Vorkommen von „**until** X_i" durch „$X_i'' := X_i' * Y$; **until** X_i''". B sei das resultierende Programmfragment. Dann läßt sich das Programmfragment **if** X $\neq$ 0 **then** A **end** durch folgendes Programmfragment A' simulieren:

$Y := X > 0;$
$Y' = 2; Y'' = Y;$ **repeat** $Y' :=$ **pred** $Y'; Y'' :=$ **pred** $Y'';$ **until** $Y'';$
$X_1' := X_1; \ldots\ X_n' := X_n;$
$B;$
$X_1 := X_1' * Y + X_1 * Y''; \ldots ; X_n := X_n' * Y + X_n * Y'';$

Die Anweisung $X := A * B + C * D$ ist folgendes Makro

$X := A * B + C * D$:
begin
  X := A*B;
  Y := C*D;
  X := X+Y;
end

Falls zuanfangs $X = 0$ gilt, dann ist $Y = 0$ und $Y' = 1$. Die Modifikation der *repeat*-Schleifen garantiert in diesem Fall die Terminierung von B. Die Ergebnisse der Berechnung werden verworfen $(X_i' * Y + X_i * Y' = X_i)$.

Falls $X > 0$ gilt, ist $Y = 1$ und $Y' = 0$. B verhält sich in diesem Fall genauso wie A. Das Ergebnis der Berechnung ist in X'_i zu finden und wird den ursprünglichen Variablen zugewiesen ($X'_i * Y + X_i * Y' = X'_i$).

(e) Wir müssen eine *while*-Schleife **while** X **loop** A **end** in ein *repeat*-Programmfragment transformieren, welches die gleichen Berechnungen durchführt. Das wesentliche Problem dabei ist, daß A eventuell nicht terminiert. Dies können wir mit dem **if**-„Makro“ elegant umgehen:

```
if X ≠ 0 then
  repeat
    A;
  until X
end
```

Lösung zu Aufgabe 1.8.16

(a) Die kleinste Lösung besteht aus 5 Wortpaaren. Eine Lösung ist zum Beispiel die Folge $\langle 1, 3, 2, 1, 1 \rangle$ mit folgendem Lösungswort:

aaabaaaa
aaabaaaa

(b) Es gibt mehrere verschiedene Argumentationen:

(i) Jede Lösung des MPKP muß mit dem erstem Wortpaar beginnen: Dadurch entsteht eine überschüssige Eins am Ende der linken Seite des so gebildeten Wortpaars $01, 0$ (*). Nur das zweite Wortpaar paßt an diese partielle Lösung. Danach entsteht auf der rechten Seite eine überschüssige Eins: $010, 0101$. Analog paßt nun nur das zweite Wortpaar: $010101, 01010$. Danach besteht die gleiche Situation wie zuvor (*). Es kann also keine Lösung konstruiert werden.

(ii) Jede Lösung eines (modifizierten) PKPs muß mit einem Wortpaar enden, bei welchem das linke und rechte Wort auf dem gleichen Buchstaben enden oder bei dem das linke oder rechte Wort das leere Wort ist. Beides ist hier nicht der Fall. Also hat die Instanz des MPKPs keine Lösung.

Lösung zu Aufgabe 1.8.17

(a) Mit dem erstem Wortpaar alleine kann offenbar keine Lösung erzeugt werden. Da bei jedem anderen Wortpaar ein b im rechten Wort vorkommt, aber niemals im linken Teil eines Wortpaares enthalten ist, kann bei Verwendung von Wortpaar 2–4 niemals eine Lösunge konstruiert werden.

(b) Wir reduzieren das Postsche Korrespondenzproblem auf das gerade Postsche Korrespondenzproblem mit einer Funktion δ, die jeden Buchstabe eines Wortpaares einer Instanz I verdoppelt. Die resultierende Instanz $\delta(I)$ besitzt also nur Wortpaare mit gerader Anzahl von Buchstaben. δ ist offenbar berechenbar.

Ist eine Instanz I des PKPs lösbar, dann auch $\delta(I)$, da korrespondierende Buchstaben in 2 gleiche korrespondierende Buchstaben abgebildet werden. Ist $\delta(I)$ lösbar, dann bekommt man durch „Raustreichen" jedes zweiten Buchstabens eine Lösung für I.

Also gilt: I lösbar genau dann, wenn $\delta(I)$ lösbar ist.

Falls das EPKP entscheidbar ist, dann kann damit auch das PKP enschieden werden, im Widerspruch zu dessen Unentscheidbarkeit. Also ist auch das EPKP nicht entscheidbar.

(c) Die Lösung $\langle 3, 2, 2, 4, 2, 1\rangle$ ist deterministisch:

$$\begin{array}{c} 01100101 \\ 01100101 \end{array}$$

(d) Wir reduzieren das modifizierte Postsche Korrespondenzproblem auf das umgekehrte modifizierte Postsche Korrespondenzproblem, indem jede Instanz $\langle(x_1, y_1), \ldots, (x_n, y_n)\rangle$ des MPKPs in eine Instanz $\langle(x_1^r, y_1^r), \ldots, (x_n^r, y_n^r)\rangle$ transformiert wird. Dabei sei x^r die Umkehrung des Worts x. Diese Funktion ist in (linearer) Zeit berechenbar. Die Funktion ist lösungserhaltend, da für eine Lösung $x_{i_1}x_{i_2} \ldots x_{i_n} = y_{i_1}y_{i_2} \ldots y_{i_n}$ mit $i_1 = 1$ des MPKPs gilt:

$$\begin{array}{rcl} (x_1 x_{i_2} \ldots x_{i_n})^r & = & (y_1 y_{i_2} \ldots y_{i_n})^r \\ & \text{gdw} & \\ x_{i_n}^r \ldots x_{i_2}^r \ldots x_1^r & = & (y_{i_n} \ldots y_{i_2} \ldots y_1^r \end{array}$$

Lösung zu Aufgabe 1.8.18

(a) (i) Sei POST die Transformation aus dem Beweis. TM sei eine Turingmaschine, bei Eingabe von w nicht hält. Da POST(TM) Kopierregeln für B enthält, hätte man mit dem zugehörigen Wortpaar $\langle B, B\rangle$ sofort eine Lösung des PKPs POST(TM).

(ii) Sei f die Transformation aus dem Reduktionsbeweis von MPKP auf PKP. Da die Technik der Reduktion transitiv ist, ist die Transformation POST $\circ f$, die Turingmaschinen in PKPs abbildet, eine Reduktion des Halteproblems auf das PKP.[2]

(b) Wir müssen eine Transformation SEMI finden, die jede Turingmaschine TM mit Eingabe w in ein Semi-Thue-System und zwei Wörtern u, v transformieren, so daß die TM bei Eingabe w genau dann hält, wenn $u \Rightarrow^*_{\text{SEMI(TM)}} v$ gilt.

Wie beim Beweis zur Unentscheidbarkeit des MPKPs wollen wir mit dem Semi-Thue-System das Verhalten der Turingmaschine simulieren. Wir benutzen als Darstellung der Turingmaschine im Semi-Thue-System die übliche Kurznotation für Konfigurationen, wobei wir zusätzlich das linke und rechte Ende mit einem $\# \notin \Gamma$ markieren. Jeder Zustandsübergang wird durch eine geeignete Ersetzungsregel kodiert.

Zu einer Turingmaschine TM mit Bandalphabet Γ, Zuständen Q, Endzuständen $F \subseteq Q$ und Zustandsübergangsfunktion δ sei SEMI(TM) das Semi-Thue-System mit $\Sigma = \Gamma \cup Q$ und folgenden Ersetzungsregeln:

Ersetzungsregel	Zustandsübergänge $q, r \in Q, a, b, x \in \Gamma$
$qa \to rb$	$\delta(q,a) = (r,b,U)$
$q\# \to rb\#$	$\delta(q,B) = (r,b,U)$
$qa \to br$	$\delta(q,a) = (r,b,R)$
$q\# \to rb\#$	$\delta(q,B) = (r,b,R)$
$xqa \to rxb$	$\delta(q,a) = (r,b,L)$
$\#qa \to \#rBb$	$\delta(q,B) = (r,b,L)$
$xq\# \to rxb\#$	$\delta(q,B) = (r,b,L)$
Löschregeln für $q_f \in F, a \in \Gamma$	
$aq_f \to q_f$	
$q_fa \to q_f$	
$\#q_f\# \to \epsilon$	

[2] Genau genommen muß man bei einer der Transformationen ein anderes Trennzeichen als # benutzen. Daß dieses aber einfach ausgetauscht werden kann ist klar.

Es ist leicht zu sehen, daß eine Turingmaschine TM mit Startzustand q_0 auf Eingabe w hält genau dann, wenn für $u = \#Bq_0wB\#$ und $v = \epsilon$

$$u \Rightarrow^*_{\mathrm{SEMI(TM)}} v \text{ gilt.}$$

Wenn man diese Behauptung beweisen wollte, müßte man einen Induktionsbeweis über die Länge der Konfigurationsfolgen führen. Das allgemeine Wortproblem ist also unentscheidbar, da ansonsten mit obiger Transformation das Halteproblem entschieden werden kann.

Lösung zu Aufgabe 1.8.19

(a) Nach Punkt 1d von Definition 10 sind die Projektionsfunktionen

$$U_j^{n+k-1}, j = 1, \ldots, n+k-1$$

primitiv rekursiv, dann ist nach Punkt 2 die Funktion

$$g'(x_1, \ldots, x_{n+k-1})(\vec{x}) = g(U_i^{n+k-1}(\vec{x}), \ldots, U_{i+k-1}^{n+k-1}(\vec{x}))$$

und auch

$$f(U_1^{n+k-1}(\vec{x}), \ldots, U_{i-1}^{n+k-1}(\vec{x}), g'(\vec{x}), U_{j+k}^{n+k-1}(\vec{x}), \ldots, U_{n+k-1}^{n+k-1}(\vec{x}))$$

primitiv rekursiv, wobei $\vec{x} = x_1, \ldots, x_{n+k-1}$ gilt.

(b) Es gilt $m(x,y) = (x \dot{-} y)x + (y \dot{-} x)y + xy$, wobei $x \dot{-} y = \max(x-y, 0)$ ist. Wie benötigen also die Subtraktion ($\mathrm{sub}(x,y) = x \dot{-} y$), Multiplikation ($\mathrm{mult}(x,y)$), eine Funktion $h'(x,y) = (x \dot{-} y)x$ und müssen sie entsprechend der Definition primitiv rekursiver Funktionen zusammensetzen. *pred* und *add* sind schon aus Beispiel 11 bekannt.

$\mathrm{sub}'(0,y)$	$= U_1^1(y)$
$\mathrm{sub}'(n+1,y)$	$= \mathrm{pred}(U_1^3(\mathrm{sub}'(n,y), n, y))$
$\mathrm{sub}(x,y)$	$= \mathrm{sub}'(U_2^2(x,y), U_1^2(x,y))$
$\mathrm{mult}(0,y)$	$= 0$
$\mathrm{mult}(n+1,y)$	$= \mathrm{add}(U_3^3(\mathrm{mult}(n,y), n, y), U_1^3(\mathrm{mult}(n,y), n, y))$
$h'(x,y)$	$= \mathrm{mult}(\mathrm{sub}(x,y), U_1^2(x,y))$
$m(x,y)$	$= \mathrm{add}(\mathrm{add}(h'(x,y), h'(U_2^2(x,y), U_1^2(x,y))), \mathrm{mult}(x,y))$

Lösung zu Aufgabe 1.8.20

(a) $$\begin{array}{lcl} \mathrm{add}(x,0) & = & U_1^1(x) \\ \mathrm{add}(x,y+1) & = & \mathrm{pred}(U_1^3(\mathrm{add}(x,y),y,x)) \end{array}$$

(b) $\mathrm{add}(x,y) = diff(x,y) + diff(y,x)$.

Lösung zu Aufgabe 1.8.21

In welchem Jahr wurde die erste Arbeit über TURING Maschinen veröffentlicht? Bitte markieren/unterstreichen Sie die richtigen Antworten!

1902	◯
1936	√
1950	◯

Lösung zu Aufgabe 1.8.22

Welche der folgenden Aussagen sind richtig und welche sind falsch? Bitte kreuzen Sie die richtigen Kästchen an!

Für jede TURING Maschine muß die Übergangsfunktionan an mindestens einer Stelle undefiniert sein.	◯ Ja	√ Nein
Ist für eine TURING Maschine für eine Eingabewort die komplette Folge aller Konfigurationen bekannt, dann kann man daraus die Übergangsfunktion rekonstruieren	◯ Ja	√ Nein

Lösung zu Aufgabe 1.8.23

Welche der folgenden Behauptungen über nichtdeterministische Turing Maschinen sind wahr? Bitte kreuzen Sie die zutreffenden Kästchen an!

Sie haben mehrere Bänder	◯ Ja	√ Nein
Der Funktionswert der Übergangsfunktion ist immer eine Menge	√ Ja	◯ Nein
Sie muß nicht terminieren	√ Ja	◯ Nein
Man kann nicht vorhersagen welche Menge sie akzeptiert	◯ Ja	√ Nein

Lösung zu Aufgabe 1.8.24

Welche der folgenden Fragmente von *while*-Programmen sind syntaktisch korrekt über den natürlichen Zahlen? In dieser Aufgabe sind keine Makros zugelassen. Bitte kreuzen Sie die zutreffenden Kästchen an!

while X **loop** X := 2 **end**	◯ Ja	√ Nein
while X **loop** **while** X **loop** X := X **end** **end**	√ Ja	◯ Nein
while =/= **succ** **succ** 0 **loop** X := **pred** X **end**	◯ Ja	√ Nein
X:= Y;Y := X	√ Ja	◯ Nein
X := **succ** **succ** Y	◯ Ja	√ Nein
while X **loop** Y := pred Y **end**	√ Ja	◯ Nein

Kapitel 2

Komplexität

2.1 Einleitung

Die frühste Referenz, die man in der einschlägigen Literatur zu Komplexitätsbetrachtungen für Algorithmen findet. ist:

> G.Lamé
> Note sur la limite du nombre des divisions dans la récherche du plus grand commun diviseur entre deux nombres entiers.
> (Bemerkung über eine Schranke für die Anzahl der Divisionen bei der Suche nach dem größten gemeinsamen Teiler zweier ganzer Zahlen.)
> Comptes Rendues des Séances de l'Academie des Sciences, Paris, 19, pp. 867-870, 1884.

Zitiert nach [31].
Die heutige Komplexitätstheorie beschäftigt sich allerdings nicht mit der Bestimmung des Zeit- oder Platzbedarfs eines Algorithmus, sondern will die Rechenzeit- und Speicherplatzressourcen eines Problems bestimmen. Bevor wir das genauer erklären, wollen wir zuerst abgrenzen, welche Probleme betrachtet werden und wie sie dargestellt werden sollen. In dem Buch [13] werden mehr als 300 Probleme aufgelistet, die in der Komplexitätstheorie eine wichtigen Rolle spielen. Die Beschreibung der einzelnen Probleme erfolgt nach einem einheitlichen Schema.

Beispiel 20 (Probleme)

ERREICHBARKEIT

Instanz:	*Ein ungerichteter Graph* (G, R) *und zwei Knoten* V_1, V_2
Frage:	*Gibt es einen Pfad von* V_1 *nach* V_2*?*

SAT

Instanz:	*Eine aussagenlogische Formel* F
Frage:	*Ist* F *erfüllbar?*

3SAT

Instanz:	*Eine konjunktive Normalform* F *mit höchstens 3 Literalen pro Klausel*
Frage:	*Ist* F *erfüllbar?*

HAMILTONSCHE KREISE

Instanz:	*Ein ungerichteter Graph* (G, R), $n \in \mathbb{N}$
Frage:	*Gibt es einen Kreis durch alle* n *Knoten in* (G, R)*?*

CLIQUE

Instanz:	*Ein ungerichteter Graph* (G, R), $n \in \mathbb{N}$
Frage:	*Gibt es eine Clique mit* n *Knoten in* (G, R)*?*

Eine Clique ist ein vollständiger Teilgraph von (G, R), *d.h. eine Teilmenge* $G_0 \subseteq G$, *so daß für alle* $V_1, V_2 \in G_0$ *gilt* $R(V_0, V_1)$.

Wie die Eingabe eines Problems kodiert ist, ist damit noch nicht festgelegt. Betrachten wir das Beispiel eines ungerichteten Graphen, so bieten sich z.B. die in Abbildung 2.1 gezeigten Kodierungsmöglichkeiten an (aus: [13, Seite 10]).

Codierung	Wort	Länge
Knotenliste,Kantenliste	V[1]V[2]V[3]V[4](V[1]V[2])(V[1]V[3])	36
Nachbarlisten	(V[2]V[3])(V[1])(V[1])()	24
Adjazenzmatrix	0110/1000/1000/0000	19

Abbildung 2.1: Darstellungen eines ungerichteten Graphen

Für die Zwecke der Komplexitätstheorie wird ein Problem durch eine Sprache L, d.h. eine Menge von Wörtern über einem fixierten Alphabet Γ dargestellt. Eine Instanz eines Problems ist ein Wort $w \in \Gamma^*$ und die Frage, die es zu lösen gilt ist $w \in L$?. Das Erreichbarkeitsproblem wird dann durch die Sprache

$$L_{er} = \{G\#V_1\#V_2 \mid V_2 \text{ ist in } G \text{ von } V_1 \text{ aus erreichbar}\}$$

Das ist eine Normierung aber keine Einschränkung der Allgemeinheit.

Beispiel 21 (Komplexität eines Algorithmus)

EINGABE: (G, R),V_1,V_2

S := { *1* }
while $S \neq \emptyset$ **loop**
    Wähle $i \in S$
    $S := S \setminus \{i\}$
    for all j **with** iRj **do**
        if $unmarked(j)$ **then** $S := S \cup \{j\}$
        if $unmarked(j)$ **then** $mark(j)$
    end
end
if $marked(2)$ **then** *output "Yes"*

Komplexität: $\mathcal{O}(n^2)$ *für* $n = card(G)$.

2.2 Die Klasse P

Läßt man die Beschreibungsmittel für einen Algorithmus offen, so ist nicht klar welche Einzelschritte gezählt werden sollen. Ein Vergleich des Aufwands verschiedener Algorithmen ist dann schwierig. Deshalb wird auf ein festes Berechnungsmodell normiert.

Definition 26 (Komplexität eines Algorithmus)
Sei T eine Turingmaschine mit Eingabealphabet Σ, f eine Funktion von $\mathbb{N}$ in $\mathbb{N}$. Für jedes $w \in \Sigma^*$ sei $t(w)$ die Anzahl der Schritte die T bei Eingabe w läuft, $s(w)$ die Anzahl der Zellen des Bandes, die während der Berechnung besucht werden.

T besitzt die Zeitkomplexität f, wenn es ein $n \in \mathbb{N}$ gibt, so daß für alle $w \in \Sigma^*$ mit $|w| \geq n$ gilt $t(w) < f(n)$.

T besitzt die Speicherkomplexität f, wenn es ein $n \in \mathbb{N}$ gibt, so daß für alle $w \in \Sigma^*$ mit $|w| \geq n$ gilt $s(w) < f(n)$.

Analog zur Turing-Komplexität kann man Zeit- und Speicherkomplexität auch für **while**-Programme, für Registermaschinen und andere Modelle der Berechenbarkeit definieren. In der Definition der Turing-Komplexität kann man z.B. die Anzahl der Bänder und das Bandalphabet variieren. Diese Variationen verändern natürlich auch die Komplexität eines Algorithmus. So kann man z.B. eine Registermaschine, die mit Zeitkomplexität $f(n)$ arbeitet, durch eine Turingmaschine mit Zeitkomplexität $f^3(n)$ simulieren. Erstaunlicherweise sind jedoch einige Komplexitätsklassen stabil gegenüber Änderungen des Berechnungsmodells. Alle Änderungen sind nämlich höchstens polynomiell. Deshalb sind die folgenden Klassen von besonderer Bedeutung:

Definition 27
Eine Sprache L liegt in der Komplexitätsklasse **P** *(oder genauer* **PTIME***), wenn es ein $k \in \mathbb{N}$ gibt mit $L \in TIME(n^k)$, d.h.*

$$P = PTIME = \bigcup_{k>0} TIME(n^k)$$

Analog:

$$PSPACE = \bigcup_{k>0} SPACE(n^k)$$

Beispiel 22 (Probleme in P)

KANTENÜBERDECKUNG

Instanz:	*Ein Graph (G, R) und $k \in \mathbb{N}$*
Frage:	*Gibt es eine Menge von Kanten $R' \subseteq R$ mit $card(R') \leq k$, so daß jede Ecke Endpunkt einer Kante in R' ist?*

2SAT

Instanz:	*Eine konjunktive Normalform F mit höchstens 2 Literalen pro Klausel*
Frage:	*Ist F erfüllbar?*

LINEARE DIOPHANTISCHE GLEICHUNGEN

Instanz:	$a, b, c \in \mathbb{N}$
Frage:	*Gibt es ein $x, y \in \mathbb{N}$ mit $ax + by = c$*

EULERSCHE KREISE

Instanz:	*Ein Graph (G, R) mit $card(R) = m$*
Frage:	*Gibt es ein Aufzählung $k_1, \ldots k_m$ aller Kanten, sodaß alle Paare $\{k_i, k_{i+1}\}$ und $\{k_1, k_m\}$ einen Endpunkt gemeinsam haben?*

Theorem 26
Das 2SAT Problem liegt in P.

Beweis: Zu jeder Menge $\mathcal{C}$ von Klauseln mit höchstens zwei Literalen konstruieren wir einen gerichteten Graphen $G(\mathcal{C})$. Die Knotenmenge von $G(\mathcal{C})$ ist die Menge L aller in $\mathcal{C}$ auftretenden Literale. O.B.d.A. enthalte $\mathcal{C}$ keine Einer-Klauseln. $\mathcal{C}_I$ sei die Menge aller Implikationen $\neg L_1 \rightarrow L_2$ und $\neg L_2 \rightarrow L_1$ für jede Klausel $L_1 \vee L_2$ in $\mathcal{C}$. Offensichtlich sind $\mathcal{C}$ und $\mathcal{C}_I$ logisch äquivalent. Eine Kante von L_1 nach L_2 tritt in $G(\mathcal{C})$ genau dann auf, wenn die Klausel $L_1 \rightarrow L_2$ in $\mathcal{C}_I$ vorkommt. Beispiele für $G(\mathcal{C})$ sind in den Abbildungen 2.2 und 2.3zu sehen. Man überzeugt sich leicht, daß gilt:

$\mathcal{C}$ ist unerfüllbar genau dann, wenn es ein Literal L gibt, so daß in $G(\mathcal{C})$ ein Weg von L nach $\neg L$ und ein Weg von $\neg L$ nach L existiert.

■

$$
\begin{aligned}
\mathcal{C} &= \{(L_1 \vee \neg L_2), (L_1 \vee L_2), (\neg L_1 \vee L_2)\} \\
\mathcal{C}_{\mathcal{I}} &= \{\neg L_1 \rightarrow \neg L_2, L_2 \rightarrow L_1, \neg L_1 \rightarrow L_2, \\
&\quad \neg L_2 \rightarrow L_1, L_1 \rightarrow L_2, \neg L_2 \rightarrow \neg L_1\}
\end{aligned}
$$

$$
\begin{tikzcd}
L_1 \arrow[d, leftrightarrow] & \neg L_2 \arrow[l] \arrow[d, leftrightarrow] \\
L_2 & \neg L_1 \arrow[l]
\end{tikzcd}
$$

Abbildung 2.2: Ein erfüllbares 2-SAT-Problem

2.3 Die Klasse NP

Definition 28
Eine Sprache L besitzt die nichtdeterministische Zeitkomplexität $f(n)$,

in Symbolen: $L \in NTIME(f(n))$,

.

$$\mathcal{C} = \{(L_1 \vee \neg L_2), (L_1 \vee L_2), (\neg L_1 \vee L_2), (\neg L_1 \vee \neg L_2)\}$$

$$\begin{tikzcd}
L_1 \arrow[rrr, leftrightarrow] \arrow[d, leftrightarrow] & & & \neg L_2 \arrow[d, leftrightarrow] \\
L_2 \arrow[rrr, leftrightarrow] & & & \neg L_1
\end{tikzcd}$$

Abbildung 2.3: Ein unerfüllbares 2-SAT-Problem

wenn es eine nichtdeterministische Turingmaschine T (siehe Abschnitt 1.2.4) gibt, die L akzeptiert und für jede Eingabe w die minimale Berechnungsfolge von T die Länge $\mathcal{O}(f(|w|))$ hat. Analog wird definiert:

$$L \in NSPACE(f(n)),$$

$$NP = NPTIME = \bigcup_{k>0} NTIME(n^k)$$

$$NPSPACE = \bigcup_{k>0} NSPACE(n^k)$$

Es herrscht weitgehend Übereinstimmung darüber Algorithmen, deren Komplexität in **P** liegt, **effizient** oder *machbar* (engl. *feasible*) zu nennen. Algorithmen, die nicht in **P** liegen, betrachtet man dagegen als **ineffizient**, als nicht machbar.

Wir stellen einige Resultate über Inklusionsbeziehungen zwischen Komplexitätsklassen zusammen.

Theorem 27

1. $TIME(f(n)) \subseteq NTIME(f(n))$

2. $SPACE(f(n)) \subseteq NSPACE(f(n))$
3. $NTIME(f(n)) \subseteq TIME(c^{f(n)})$
4. $NTIME(f(n)) \subseteq SPACE(f(n))$
5. $NSPACE(f(n)) \subseteq TIME(c^{f(n)})$
6. $P \subseteq NP \subseteq PSPACE$
7. $PSPACE = NPSPACE$

Beweis:

zu 1 und 2 einfach.

zu 3 folgt aus Satz 5.

zu 4 einfach.

zu 5 [11, Theorem 23.4.1, Seite 345].

zu 6 Spezialfall $f(n) = n^k$ zu 1 und 4.

zu 7 [15, Satz 14.21, Seite 74 ff]. Dieses Resultat ist unter der Bezeichnung „Satz von Savitch"bekannt.

∎

2.4 NP-Vollständigkeit

Definition 29
Eine Sprache L_1 heißt polynomiell reduzierbar *auf eine Sprache L_2, in Zeichen:*

$$L_1 \leq_p L_2,$$

wenn es eine in polynomieller Zeit berechenbare Funktion $f : \Sigma^* \to \Sigma^*$ gibt mit

$$w \in L_1 \text{ gdw } f(w) \in L_2.$$

Als unmittelbare Folgerung aus der Definition ergibt sich die Transitivität von $\leq_p$, d.h. aus $L_2 \in P$ und $L_1 \leq_p L_2$ folgt $L_1 \in P$

Definition 30
Eine Sprache $L \in NP$, so daß für jede Sprache $L' \in NP$ gilt $L' \leq_p L$ heißt NP-vollständig.

Wird $L \in NP$ nicht vorausgesetzt, so heißt L eine NP-harte *Sprache.*

Beispiel 23 (NP-vollständige Probleme)

ECKENÜBERDECKUNG

Instanz:	*Ein Graph (G, R) und $k \in \mathbb{N}$*
Frage:	*Gibt es eine Menge von Ecken $G' \subseteq G$ mit $card(G') \leq k$, so daß jede Kante einen Endpunkt in G' hat?*

3SAT

Instanz:	*Eine konjunktive Normalform F mit höchstens 3 Literalen pro Klausel*
Frage:	*Ist F erfüllbar?*

QUADRATISCHE DIOPHANTISCHE GLEICHUNGEN

Instanz:	$a, b, c \in \mathbb{N}$
Frage:	*Gibt es ein $x, y \in \mathbb{N}$ mit $a^2x + by = c$*

HAMILTONSCHE KREISE

Instanz:	*Ein Graph (G, R) mit $card(G) = n$*
Frage:	*Gibt es ein Aufzählung $k_1, \ldots k_m$ aller Ecken, sodaß alle Paare $\{k_i, k_{i+1}\}$ und $\{k_1, k_m\}$ durch eine Kante verbunden sind?*

Theorem 28 (Satz von Cook)
Das Erfüllbarkeitsproblem für aussagenlogische Formeln, SAT, ist NP-vollständig.

Beweis: Es ist einfach zu sehen, daß SAT $\in$ NP gilt.

Es bleibt für jedes $L \in NP$ noch $L \leq_p SAT$ zu zeigen. Sei dazu $T = (\Sigma, \Gamma, Q, q_0, \delta, F)$ eine nichtdeterministische Turingmaschine und $p(x)$ ein Polynom, so daß für jedes $w \in \Sigma^*$ genau dann eine akzeptierende Berechnung der Länge $\leq p(|w|)$ für w existiert, wenn $w \in L$ gilt.

Wir konstruieren zu jedem w eine AL-Formel $G(T,w)$ in konjunktiver Normalform, so daß

$$w \in L \Leftrightarrow G(w) \in SAT$$

Außerdem wird $G(T,w)$ in polynomieller Zeit aus w konstruierbar sein.

Zu Vereinfachung nehmen wir an, daß die Menge F der Endzustände von T aus genau zwei Zuständen besteht $F = \{q^+, q^-\}$, wovon q^+ der akzeptierende und q^- der nicht-akzeptierender Endzustand ist.

Kodierungsidee Wir zählen des Bandalphabet und die Zustände von T auf $\Gamma = \{\gamma_0 \dots \gamma_g\}$, $Q = \{q_0 \dots q_h\}$. Ausgehend von einem Eingabewort w mit $|w| = n$ und $m = p(n)$ werde nach t Schritten die folgende Konfiguration erreicht.

$-m$					0			k			m
a_{-m}	$\cdot$	$\cdot$	$\cdot$	$\cdot$	a_0	$\cdot$	$\cdot$	a_k	$\cdot$	$\cdot$	a_m
								q_r			

Zur Beschreibung dieser Konfiguration durch eine aussagenlogische Formel führen wir aussagenlogische Variable ein. Mit einer AL-Variablen können wir gerade einmal ein Bit Information kodieren. So soll z.B. die Variable $P_{i,t}$ genau dann wahr sein (den Wahrheitswert 1 haben) wenn zum Zeitpunkt t das Feld mit der Nummer i fokussiert wird. Ebenso soll $Q_{i,t}$ wahr sein, wenn sich die Turingmaschine zum Zeitpunkt t im Zustand q_i befindet. Als drittes soll $Z_{j,p,t}$ wahr sein, wenn zum Zeitpunkt t auf dem Feld mit der Nummer p das Symbol γ_j steht. Die obige Konfiguration zum Zeitpunkt t führt also zu der folgenden Belegung der aussagenlogischen Variablen:

$$\begin{array}{lcllcll}
P_{k,t} & = & 1, & P_{i,t} & = & 0 & \text{für } -m \leq i \leq m,\ i \neq k \\
Q_{r,t} & = & 1, & Q_{i,t} & = & 0 & \text{für } 0 \leq i \leq h,\ i \neq r \\
Z_{j,p,t} & = & 1 & \multicolumn{4}{l}{\text{für } a_p = \gamma_j} \\
Z_{j,p,t} & = & 0 & \multicolumn{4}{l}{\text{für } 0 \leq j \leq g,\ -m \leq p \leq m,\ a_p \neq \gamma_j}
\end{array}$$

$P_{i,t}$,$Q_{i,t}$,$Z_{j,p,t}$ Aussagenvariable. Anzahl der AL-Variablen $= O(p^2(n))$.

Allgemeine Einschränkungen an die AL-Variablen Will man jetzt umgekehrt ausgehend von einer Belegung der AL-Variablen eine Konfiguration rekonstruieren, so merkt man, daß das nicht ohne weitere möglich ist. So kann es zum Beispiel nur ein einziges Feld geben, das zu einem festen Zeitpunkt fokussiert wird und es muß auch mindestens eins geben. Ähnliche Einschränkungen gelten für den Zustand und die Bandinschrift. Zusammengefaßt ergeben sich die folgenden Einschränkungen:

$$
\begin{array}{lcl}
\psi_{P,t} & = & \bigvee_{-m\leq i\leq m} P_{i,t} \wedge \bigwedge_{-m\leq i<j\leq m} \neg(P_{i,t} \wedge P_{j,t}) \\
\psi_P & = & \bigwedge_{0\leq t\leq m} \psi_{P,t} \\
\psi_{Q,t} & = & \bigvee_{0\leq i\leq h} Q_{i,t} \wedge \bigwedge_{0\leq i<j\leq h} \neg(Q_{i,t} \wedge Q_{j,t}) \\
\psi_Q & = & \bigwedge_{0\leq t\leq m} \psi_{Q,t} \\
\psi_{Z,p,t} & = & \bigvee_{0\leq i\leq g} Z_{i,p,t} \wedge \bigwedge_{0\leq i<j\leq g} \neg(Z_{i,p,t} \wedge Z_{j,p,t}) \\
\psi_{Z,t} & = & \bigwedge_{-m\leq p\leq m} \psi_{Z,p,t} \\
\psi_Z & = & \bigwedge_{0\leq t\leq m} \psi_{Z,t} \\
\psi & = & \psi_P \wedge \psi_Q \wedge \psi_Z
\end{array}
$$

Ist jetzt eine Wahrheitsbelegung der aussagenlogischen Variablen gegeben, die ψ wahr macht, dann kann man daraus eindeutig eine Konfiguration von T rekonstruieren. Als nächstes überlegen wir uns welche Einschränkungen zwischen den Wahrheitswerten zum Zeitpunkt t und zum Zeitpunkt $t+1$ gelten müssen.

Sei dazu $E: \quad \delta(q_i, \gamma_j) = (q_{i'}, \gamma_{j'}, R)$ eine Zeile der Übergangstabelle der Turingmaschine T.

$$
\begin{array}{lcl}
\chi_{E,p} & = & P_{p,t} \wedge Q_{i,t} \wedge Z_{j,p,t} \quad \to \quad P_{p+1,t+1} \wedge Q_{i',t+1} \wedge Z_{j',p,t+1} \wedge \\
 & & \qquad\qquad\qquad\qquad \bigwedge_{p_1\neq p} \bigwedge_{j_1\neq j} (Z_{j_1,p_1,t} \leftrightarrow Z_{j_1,p_1,t+1} \\
\chi_E & = & \bigwedge_{-m\leq p\leq m} \chi_{E,p} \\
\chi_T & = & \bigwedge\{\chi_E \mid \text{ E Zeile der Übergangstabelle von } T\} \\
\chi_w & = & \text{Kodierung der Anfangskonfiguration} \\
 & & \text{(nicht ausgeführt)}
\end{array}
$$

$$
G(T, w) = \psi \wedge \chi_T \wedge \chi_w \wedge Q_{h,m}
$$

wobei $q_h = q^+$ der akzeptierende Finalzustand von T ist.

Erforderliche technische Voraussetzung: Für δ gilt: $\delta(q^z, \gamma) = (q^z, \gamma, U)$ für die beiden Finalzustände q^z von T und alle $\gamma \in \Gamma$.

∎

2.4.1 Ein PUZZLE-Problem

Betrachten Sie folgendes Problem PUZZLE:

Instanz: Gegeben ein Rahmen und eine endliche Menge von Puzzleteilen

Frage: Kann der Rahmen durch die Puzzleteile passend ausgefüllt werden?

Das Ziel bei diesem Puzzlespiel ist, einen vorgegebenen Rahmen mit einer begrenzten Anzahl quadratischer Puzzleteile vollständig auszulegen. Ein Puzzleteil hat vier Ränder, die jeweils eine Beschriftung tragen. Wie bei einem echten Puzzlespiel können zwei Puzzleteile nur dann nebeneinander gelegt werden, wenn die sich berührenden Ränder gleich beschriftet sind. Der Unterschied dieses Puzzlespiels zu einem echten Puzzle ist nun der, daß es hier eine geringe Anzahl von Puzzleteiltypen gibt. Von jedem Typ können jedoch beliebig viele Exemplare eingesetzt werden. Mit den Exemplaren dieser Typen soll ein gegebener Rahmen ausgefüllt werden. Dieser Rahmen ist ebenfalls mit Beschriftungen versehen, zu denen die angelegten Puzzleteile passen müssen. Die Frage ist nun, ob der Puzzlerahmen mit Teilen der vorgegebenen Typen $T_1, \dots, T_k$ passend ausgefüllt werden kann. Als PUZZLE-Problem bezeichnet man also das Problem, für einen gegebenen Rahmen und gegebene Puzzleteiltypen zu entscheiden, ob der Rahmen ausgefüllt werden kann.

Hier sehen Sie zwei Puzzlespiele mit 3x3-Spielfeldern. Bei dem oberen Puzzlespiel (Abb. 2.4) kann der Rahmen mit den vorgegebenen Puzzleteiltypen komplett und passend ausgelegt werden, bei dem unteren Spiel (Abb.2.5) ist das unmöglich. Das Teil (2,5,2,5) muß in die rechte obere Ecke gesetzt werden. Dann muß auch oben in die Mitte und oben links dieses Teil gesetzt werden. Das paßt aber nicht mit der 4 am linken Rand zusammen.

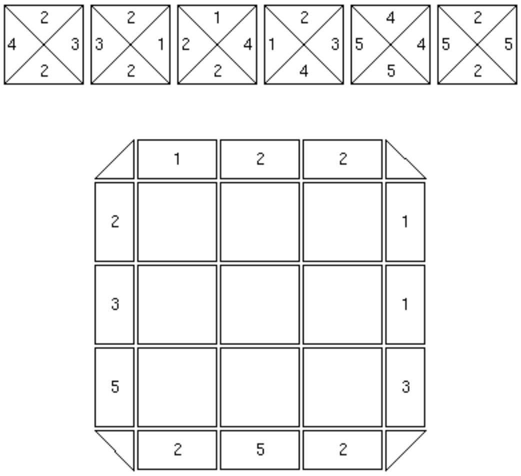


Abbildung 2.4: Ein komplettierbares Puzzlespiel

Ein Puzzleteiltyp kann durch Anklicken in der Leiste über dem Rahmen ausgewählt werden. Ein Exemplar dieses Typs wird durch Anklicken des Zielfeldes innerhalb des Rahmens eingesetzt. Durch Anklicken mit der rechten Maustaste kann ein schon gesetztes Teil im Rahmen wieder gelöscht werden. Passen die Ränder zweier benachbarter Puzzleteile oder der Rand eines Puzzleteils und ein angrenzendes Rahmenstück nicht zusammen, werden diese Ränder bzw. Rahmenstücke schwarz markiert. Ein Rahmen gilt als

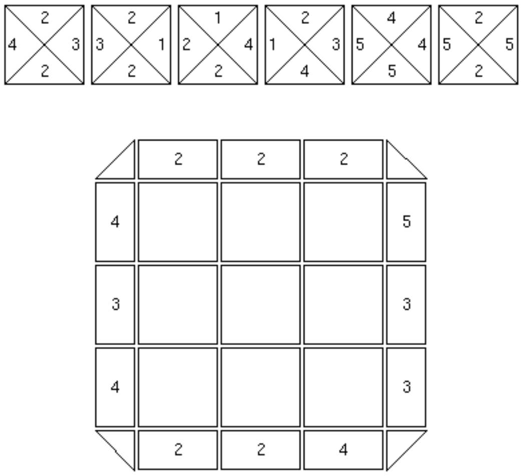


Abbildung 2.5: Ein nicht komplettierbares Puzzlespiel

komplett ausgelegt, wenn auf jedem Feld ein Puzzleteil gesetzt ist und kein Puzzleteilrand oder Rahmenstück markiert ist.

Formale Beschreibung des PUZZLE-Problems

Zur formalen Definition einer Instanz des PUZZLE-Problems müssen der Rahmen und die k vorgegebenen Puzzleteiltypen in geeigneter Weise codiert werden. Die Beschriftungen der Puzzleteile und des Rahmens seien durch Symbole aus dem Alphabet Σ gegeben. Ein Typ T wird durch 4 Symbolen aus Σ beschrieben. Der Rahmen R der Größe $t \times t$ wird durch $4t$ Symbole aus Σ festgelegt. Das PUZZLE-Problem definiert man so:

Instanz: $m, k, t \in \mathbf{N}$, $m, k, t \geq 1$, $R \in \Sigma^{4t}$, $T_1, \ldots, T_k \in \Sigma^4$

Frage: Kann der Rahmen R durch Puzzleteile der Typen $T_1, \ldots, T_k$ passend ausgefüllt werden?

Die zu diesem Problem gehörende Sprache wird folgendermaßen definiert:

$PUZZLE := \{(k, t, R, T_1, \ldots, T_k) \mid k, t \geq 1,\ R \in \Sigma^{4t},\ T_1, \ldots, T_k \in \Sigma^4$, und der Rahmen kann durch Puzzleteile der Typen $T_1, \ldots, T_k$ passend ausgefüllt werden$\}$.

Zur Beschreibung des ersten, komplettierbaren Puzzlespiels im vorigen Beispiel würde man also folgende Werte angeben:

$$\Sigma = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$k = 6$$

$$t = 3$$

$$R = (1, 2, 2, 1, 1, 3, 2, 5, 2, 5, 3, 2)$$

(Die Rahmenbeschriftung wird ab der linken oberen Ecke im Uhrzeigersinn angegeben.)

$$T_1 = (2, 3, 2, 4)$$

$$T_2 = (2, 1, 2, 3)$$

$$T_3 = (1, 4, 2, 2)$$

$$T_4 = (2, 3, 4, 1)$$

$$T_5 = (4, 4, 5, 5)$$

$$T_6 = (2, 5, 2, 5)$$

(Die Randbeschriftungen der Puzzleteile werden in der Reihenfolge oben, rechts, unten, links angegeben.)

PUZZLE ist NP-vollständig (Beweisidee)

Es folgt ein elementarer Beweis für die NP-Vollständigkeit von PUZZLE, d.h. der Beweis wird geführt ohne ein schon bekanntes NP-vollständiges Problem vorauszusetzen, das dann auf das PUZZLE-Problem reduziert wird.

PUZZLE liegt in NP Ein Problem liegt in NP, falls für eine erratene Lösung mit polynomiellen Zeitaufwand überprüft werden kann, ob sie tatsächlich eine Lösung des Problems darstellt.

Beim PUZZLE-Problem bedeutet dies, daß für eine gegebene Belegung eines $t \times t$- Rahmens mit Puzzleteilen getestet werden muß, ob alle benachbarten Ränder zusammenpassen. Dies ist mit polynomiellen Zeitaufwand möglich, da zu diesem Zweck insgesamt $2(t+1)t$ Vergleiche zweier Symbole aus Σ durchgeführt werden müssen.

PUZZLE ist NP-hart Zu zeigen ist, daß für jedes beliebige Problem $A \in NP$ gilt: $A \leq_p PUZZLE$, d.h. es muß eine polynomielle Reduktion von A auf das PUZZLE-Problem gefunden werden. Mit anderen Worten: Zu einer beliebigen Sprache $A \in NP$ muß eine, in polynomieller Zeit berechenbare, Funktion $f : \Gamma^* \to \Gamma^*$ konstruiert werden, so daß für $\omega \in \Gamma^*$ gilt:

$$\omega \in A \Leftrightarrow f(\omega) \in PUZZLE.$$

Sei also eine beliebige Sprache $A \in NP$ gegeben. $A \in NP$ bedeutet, daß es eine nichtdeterministische Turingmaschine M mit einseitig unendlichem Band und polynomiell beschränkter Zeitkomplexität gibt, die A akzeptiert. Die Zeitkomplexität sei beschränkt durch das Polynom $p : \mathbf{N} \to \mathbf{N}$.

Aus der Beschreibung von M und zu einem gegebenen Eingabewort $\omega \in \Gamma^*$ muß jetzt eine Instanz $f(\omega)$ des PUZZLE-Problems, d.h. ein Puzzlespiel konstruiert werden, das genau dann komplett ausgelegt werden kann, wenn M das Eingabewort ω akzeptiert. Dieses Puzzlespiel besteht aus einem Rahmen $R_{M,\omega}$ und einer Anzahl Puzzleteiltypen $T_1^M, \ldots, T_k^M$, also $f(\omega) = (k,\, p(n),\, R_{M,\omega},\, T_1^M, \ldots, T_k^M)$ mit $n = |\omega|$ (Länge des Eingabewortes).

Der quadratische Rahmen $R_{M,\omega}$ hat die Höhe und Breite p(n). In der oberen Zeile des Rahmens steht das Eingabewort ω, der Rest ist mit Leer-Zeichen aufgefüllt. Dies ist der Bandinhalt bei Start der Turingmaschine M. Da der Kopf der Turingmaschine zu Beginn der Berechnung auf dem ersten Zeichen $\omega[0]$ des Eingabewortes steht, ist in der oberen Zeile des Rahmens an dieser Stelle $q_0, \omega[0]$ eingetragen, wobei q_0 der Anfangszustand der Turingmaschine ist.

In der unteren Zeile des Rahmens ist der Bandinhalt nach p(n) Berechnungsschritten von M eingetragen. In dieser Zeit kann der Kopf auch nur höchstens p(n) Schritte nach rechts gemacht haben.

Falls M nach p(n) Schritten sich in einem Haltezustand q_f befindet, wird in der unteren Zeile des Rahmens das Feld, auf dem der Kopf steht mit q_f, x beschriftet, wobei x das Bandsymbol in diesem Feld ist. War die Turingmaschine schon vorher in einem Haltezustand q_f (das ist ein Zustand, für den beim aktuellen Bandinhalt kein Übergang in der Übergangsfunktion definiert ist) stellt man sich vor, daß die Berechnung mit Übergängen der Form $q_f, x \to q_f, x, U$, bei denen sich am Zustand und am Bandinhalt nichts ändert, bis zum p(n)-ten Schritt fortgesetzt wird.

Falls M nach p(n) Schritten noch nicht in einen Haltezustand gelangt ist, wird dieses Feld mit x beschriftet, der aktuelle Zustand wird dort also nicht vermerkt.

Der linke und rechte Teil des Rahmens ist mit einem speziellen Zeichen (#) beschriftet. Dieses Zeichen darf nicht im Bandalphabet der Turingmaschine vorkommen. Die formale Beschreibung des Rahmens ist also:

$R_{M,\omega} :=$

	$q_0, \omega[0]$	$\omega[1]$	$\omega[2]$	$\ldots$	$\omega[n-1]$	B	$\ldots$	B	
#									#
#									#
#									#
#									#
	$q_f, *$	$*$	$\ldots$	$*$	$*$	B	$\ldots$	B	

falls die TM nach p(n) Schritten in einen Haltezustand q_f gelangt ist, bzw.

$R_{M,\omega} :=$

	$q_0, \omega[0]$	$\omega[1]$	$\omega[2]$	$\ldots$	$\omega[n-1]$	B	$\ldots$	B	
#									#
#									#
#									#
#									#
	$*$	$*$	$\ldots$	$*$	$*$	B	$\ldots$	B	

falls die TM nach p(n) Schritten noch nicht in einen Haltezustand gelangt ist („$*$“ steht für ein beliebiges Bandsymbol).

Ein Beispiel für einen Rahmen sehen Sie in Abb. 2.6. Dieser Rahmen gehört zu einer Turingmaschine mit Startzustand q_1 und Eingabewort „01“. Der Zustand q_2 ist ein Haltezustand, das Eingabewort wurde also akzeptiert.

Jetzt soll dieser Rahmen Zeile für Zeile mit Puzzleteilen gefüllt werden. Jede Zeile von Puzzleteilen entspricht dabei einem Schritt des Berechnungsvorgangs der Turingmaschine M (siehe Abb. 2.7).

Je ein Puzzleteil trägt dabei am oberen Rand das Bandsymbol eines Feldes vor dem Übergang und am unteren Rand dessen Bandsymbol nach dem Übergang. Das ist bei dem gezeigten Rahmen (Abb. 2.6) für die erste und die letzte Zeile von Puzzleteilen angedeutet. Befindet sich vor oder nach dem Übergang der Kopf der Turingmaschine in dem Feld, so wird das durch eine Beschriftung der Art q, x angezeigt, wobei q der aktuelle Zustand der TM und x das Bandsymbol in dem Feld ist.

Die genaue Ausführung dieses Beweises überlassen wir als Übungsaufgabe.

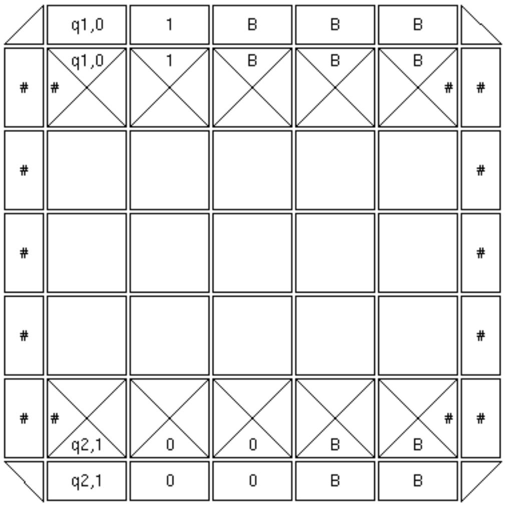


Abbildung 2.6: Beispiel eines Rahmens zu einer Turingmaschine

q1,0	1	B	B	B
1	q1,1	B	B	B
1	0	q1,B	B	B
1	0	q2,0	B	B
1	q2,0	0	B	B
q2,1	0	0	B	B

Abbildung 2.7: Beispiel eines Berechnungsvorgangs einer Turingmaschine

2.5 Andere Komplexitätsmaße

1. Algebraische Komplexitätstheorie
2. Durchschnittliche Komplexität (average case complexity)
3. Kolmogorov Komplexität

2.5.1 Algebraische Komplexitätstheorie

Bei der naiven Berechnung des Funktionswertes eines Polynoms

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots a_1 x + a_0$$

an der Stelle x_0 braucht man mindestens $2n-1$ Multiplikationen. Nach dem HORNER Schema kommt man mit n Multiplikationen aus:

$$f(x_0) = (\dots((a_n x + a_{n-1})x + a_{n-2})x + \dots a_1)x + a_0$$

$$\begin{array}{lcl} y_n & = & a_n x_0 + a_{n-1} \\ y_{n-1} & = & y_n x_0 + a_{n-2} \\ \vdots & = & \vdots \\ y_1 & = & y_2 x_0 + a_0 \end{array}$$

V. Ya. Pan veröffentlichte 1966 in den *Russian Math. Surveys, Vol. 21(1), pp. 105 -136* das folgende Resultat:

Theorem 29
Die Berechnung des Funktionswertes eines Polynoms

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots a_1 x + a_0$$

an der Stelle x_0 benötigt mindestens n Multiplikationen oder Divisionen. Das HORNER Schema ist also in dieser Hinsicht optimal.

Sei ω das Infimum aller reellen Zahlen τ, so daß die Multiplikation zweier $n \times n$-Matrizen mit $\mathcal{O}(n^\tau)$ arithmetischen Operationen ausgeführt werden kann. Nach dem gegenwärtigen Stand der Forschung ist

$$\omega < 2.50$$

(Coppersmith-Winograd, 1981)

2.5.2 Kolmogorov Komplexität

Sei $T_1, T_2, \ldots, T_n, \ldots$ eine Aufzählung aller Turingmaschinen über einem festen Alphabet Σ und $w \in \Sigma^*$. Die **Kolmogorov Komplexität** (Beschreibungskomplexität) $K(w)$ von w ist definiert:

$$K(w) = min\{\text{ Größe von } T_j \mid T_j(\varepsilon) = w\}$$

Anwendungsgebiete:

- Untere Komplexitätsschranken
- Zufällige 0-1-Folgen.

2.6 Lehrbücher

1. [37]
   Enthält ein sehr empfehlenswertes Kapitel über NP-Vollständigkeit mit vielen Beispielen für polynomielle Reduktion.

2. [40]
   Verständliche Einführung. Hervorzuheben ist, daß im Unterschied zu dem meisten anderen Lehrbüchern und Monographien zur Komplexitätstheorie nicht SAT als Ausgangsbeispiel für ein NP-vollständiges Problem benutzt wird, sondern ein Puzzle-Problem, das unmittelbarer verständlich ist. Betont formale Darstellungsweise.

3. [23]
   Umfangreiches Lehrbuch. Verständlich aber knapp geschrieben. Umfangreiche Übungsaufgaben mit Angabe des Schwierigkeitsgrades.

4. [10]
   Behandelt in kurzen thematisch abgerundeten, anschaulich aufbereiteten Kapiteln fast alle Themen, die wir hier behandeln. Wendet sich allerdings mehr an Schüler der 13. Klasse als an Universitätsstudenten.

5. [41]
   Kein Lehrbuch, aber ein interessanter neuer Ansatz. Es wird versucht die wesentlichen Ideen und Motivationen zu vermitteln, ohne dabei allzu sehr in technische Details einzusteigen. Gedacht als Ergänzung zu Lehrbuch und Vorlesung. Enthält keine Aufgaben, aber Testfragen mit Antworten.

6. [13]
   Kein Lehrbuch, aber eine umfangreiche Sammlung NP-vollständiger Probleme mit einer leicht verständlichen Einführung.

7. [31]
   Klar geschriebene, aber anspruchsvolle Monographie. Als weiterführende Literatur empfohlen.

8. [4]
   Weiterführende Lektüre.

2.7 Übungsaufgaben

Übungsaufgabe 2.7.1

Betrachten Sie folgendes Entscheidungsproblem c-Hamiltonkreis:

Definition 31 (c-Hamiltonkreis)
INSTANZ: Gegeben ein ungerichteter Graph $G = (V, E)$ *mit Knotenmenge* V*, Kantenmenge* E *und eine Konstante* $c \in \mathbb{N}$.
FRAGE: Gibt es einen Teilgraph $G' = (V', E')$ *von* G *mit* $|V'| = c$*, so daß* G' *einen Hamiltonkreis enthält?*

Zeigen Sie, daß c-Hamiltonkreis NP-vollständig ist.

Übungsaufgabe 2.7.2

Definition 32 (Handlungsreisender)
INSTANZ: Geben n *Orte und die Kosten* $c_{ij} \geq 0$*, die eine Reise von Ort* i *zum Ort* j *verursachen.*
FRAGE: Gibt es eine Rundreise durch alle Orte, so daß jeder Ort genau einmal besucht wird und die Gesamtkosten ein vorgegebenes $K \geq 0$ *nicht übersteigen?*

Zeigen Sie, daß das Problem des Handlungsreisenden NP-vollständig ist.

Übungsaufgabe 2.7.3

Definition 33 (Flaschenhals Handlungsreisender (FHR))
INSTANZ: Geben eine Menge $S = \{c_1, \ldots, c_n\}$ *von Städten, Entfernungen* $d(c_i, c_j) \geq 0$ *und ein* $K > 0$.
FRAGE: Gibt es eine Rundtour durch alle Städte, dessen längste Kante kleiner als K *ist, d.h. existiert eine Permutation* π *der Zahlen* $1, \ldots, n$ *mit*

$$d(\pi(n), \pi(1)) < K \text{ und } d(\pi(i), \pi(i+1)) < K \text{ für } i = 1, \ldots, n-1 \ ?$$

Zeigen Sie, daß das Problem des Flaschenhals-Handlungsreisenden NP-vollständig ist.

Übungsaufgabe 2.7.4

Definition 34 (Vereinfachung von 3SAT)
INSTANZ: Gegeben sei eine aussagenlogische Formel F *in konjunktiver Normalform, so daß jede Variable höchstens drei mal und jedes Literal höchstens zwei mal in* F *vorkommt; und jede Klausel höchstens drei Literale enthält.*
FRAGE: Ist F *erfüllbar?*

Zeigen Sie, daß diese Vereinfachung von 3SAT NP-vollständig ist.

Übungsaufgabe 2.7.5

Definition 35 (MONOTONE 3SAT)
INSTANZ: Eine konjunktive Normalform F *mit höchstens 3 Literalen pro Klausel und jede Klausel enthält entweder nur positive oder nur negative Literale.*
FRAGE: Ist F *erfüllbar?*

Zeigen Sie, daß MONOTONE 3SAT NP-vollständig ist.

Übungsaufgabe 2.7.6

Betrachten Sie folgendes Entscheidungsproblem:

Definition 36 (D-SAT)
INSTANZ: Gegeben eine aussagenlogische Formel F *in disjunktiver Normalform, d.h.* F *hat die Form* $(x_{1,1} \wedge \ldots \wedge x_{n,1}) \vee \ldots \vee (x_{1,k} \wedge \ldots \wedge x_{n,k})$ *mit aussagenlogischen Li teralen* $x_{i,j}$.
FRAGE: Ist F *erfüllbar?*

(a) Zeigen Sie, daß folgende Instanz von D-SAT erfüllbar ist:

$$F_1 = (p \wedge q \wedge \neg q) \vee (p \wedge \neg q) \vee (\neg p \wedge \neg q)$$

(b) Zeigen Sie, daß folgende Instanz von D-SAT unerfüllbar ist:

$$F_2 = (p \wedge q \wedge \neg q) \vee (p \wedge \neg q \wedge \neg p) \vee (q \wedge \neg p \wedge \neg q)$$

(c) Zeigen Sie, daß D-SAT in NP liegt.

(d) Betrachten Sie folgende Transformation f von 3SAT nach D-SAT ($x_{i,j}$ sind aussagenlogische Literale):

Zu einer Instanz $F = (x_{1,1} \vee x_{2,1} \vee x_{3,1}) \wedge \ldots \wedge (x_{3,k} \vee x_{2,k} \vee x_{3,k})$ von 3SAT sei $N_j := \{x_{1,j}, x_{2,j}, x_{3,j}\}$ und

$$f(F) := \bigvee_{(u_1,\ldots,u_k) \in N_1 \times \ldots \times N_k} (u_1 \wedge \ldots \wedge u_k)$$

$f(F)$ entsteht also durch „ausmultiplizieren“.

Geben Sie $f((a \vee b \vee c) \wedge (u \vee v \vee w))$ an.

(e) Ist es möglich, mit f zu zeigen, daß D-SAT NP-hart ist? Begründen Sie Ihre Antwort.

Übungsaufgabe 2.7.7

Betrachten Sie folgendes Optimierungsproblem:

Definition 37 (BIN-PACKING)
INSTANZ: $U = \{u_1, \ldots, u_n\}$ *sei eine Menge von Gegenständen mit Größe* $g(u_i) \in [0,1] \subset \mathbb{Q}, i = 1, \ldots, n$.
GESUCHT: Eine Aufteilung von U *in disjunkte Teilmengen* $U_1, \ldots, U_k$ *mit minimalen* k*, so daß die Summe der Größe aller Gegenstände in* U_j *kleiner gleich Eins ist, d.h.* $\sum_{u_i \in U_j} g(u_i) \leq 1$.

(a) I sei Instanz von BIN-PACKING und OPT(I) bezeichne die Anzahl k der Behälter einer optimale Lösung von I. Geben sie eine genaue Abschätzung von OPT(I) in Abhängigkeit von der Größe aller Gegenstände von I an.

(b) Formulieren Sie den folgenden Algorithmus zur Approximation von $\mathrm{OPT}(I)$ in Pseudo-Code und schätzen Sie dessen Zeitaufwand ab:

Wir gehen von einer unendlichen Sequenz $B_1, B_2, \ldots$ von leeren Behältern aus. Der Algorithmus füllt diese Behälter mit den Gegenständen in aufsteigender Reihenfolge der Indizes der Behälter. Dabei wird folgende Regel beachtet: Ein Gegenstand u_i wird immer in den ersten Behälter gelegt, in dem er zuerst paßt (first-fit), d.h. dessen Index am kleinsten ist.

(c) $\mathrm{FF}(I)$ sei die Anzahl Behälter, die obiger Algorithmus für eine Instanz I benötigt. Geben Sie eine möglichst gute Abschätzung von $\mathrm{FF}(I)$ in Abhängigkeit der Größe aller Gegenstände aus I an.

(d) Zeigen Sie, daß der Algorithmus ein Ergebnis liefert, welches höchstens doppelt so schlecht ist, wie das Optimum, d.h. $\mathrm{FF}(I) < 2 \cdot \mathrm{OPT}(I)$.

Übungsaufgabe 2.7.8

Betrachten Sie noch einmal das PUZZLE-Problem und das zugehörige Beweisfragment.

Ihre Aufgabe ist es, den Beweis zu vervollständigen. Dazu müssen Sie die richtige Beschriftung der Puzzleteile finden, so daß der Rahmen genau dann komplett passend ausgelegt werden kann, wenn die Turingmaschine das Eingabewort akzeptiert.

Dazu folgende Hinweise: Da die Puzzleteile den Berechnungsvorgang der Turingmaschine darstellen sollen, besteht ein enger Zusammenhang zwischen den Randbeschriftungen und den Zeilen der Übergangsfunktion.

Man kann die beschriebenen Puzzleteile in 6 Kategorien einteilen, je nachdem, welche Aktion des Kopfes durch das Puzzleteil beschrieben werden soll. Welche Kategorie für ein Feld verwendet werden muß, hängt also von der Position des Kopfes in Bezug auf das Feld ab. Die Kategorien können durch exemplarische Puzzleteile dargestellt werden. Die möglichen Randbeschriftungen dieser exemplarischen Puzzleteile sind:

a	ein Symbol
q	ein Zustand
q,a	ein Zustand und ein Symbol
b	ein neues Symbol
p	ein neuer Zustand
p,b	ein neuer Zustand und ein neues Symbol
#	das Hash-Zeichen

Man stellt sich vor, daß für a und b,c ein beliebiges Symbol und für p und q jeweils ein beliebiger Zustand aus der Übergangsfunktion der Turingmaschine eingesetzt wird. So erhält man eine (unter Umständen große) Anzahl von konkreten Puzzleteiltypen. Die Kategorien werden folgendermaßen beschrieben:

1. Für ein beliebiges Bandsymbol a: Weder vor noch nach dem Schritt der TM befindet sich der Kopf im Feld. Die Beschriftung des Feldes ändert sich also nicht.

2. Zu einer Transition $q, a \rightarrow p, b, R$: Der Kopf befindet sich im Feld. Es wird ein neues Bandsymbol geschrieben und der Kopf bewegt sich mit neuem Zustand in das rechte Nachbarfeld.

3. Zu einer Transition $q, a \rightarrow p, b, R$: Der Kopf kommt von links ins Feld. Die Beschriftung des Feldes ändert sich nicht.

4. Zu einer Transition $q, a \rightarrow p, b, L$: Der Kopf befindet sich im Feld. Es wird ein neues Bandsymbol geschrieben und der Kopf bewegt sich mit neuem Zustand in das linke Nachbarfeld.

5. Zu einer Transition $q, a \rightarrow p, b, L$: Der Kopf kommt von rechts ins Feld. Die Beschriftung des Feldes ändert sich nicht.

6. Zu einer Transition $q, a \rightarrow p, b, U$: Der Kopf befindet sich im Feld. Es wird ein neues Bandsymbol geschrieben und der Zustand ändert sich. Der Kopf bleibt im Feld.

Puzzle-Applet

Die Aufgabe Das Puzzle-Applet gibt Ihnen die Möglichkeit, durch Probieren die richtige Umsetzung der Transitionen in die 6 Kategorien von Puzzleteilen zu finden. Die erzeugten Puzzleteile können dann in den Rahmen eingesetzt werden.

Sie gehen also in 3 Schritten vor:

1. Eingabe einer Turingmaschine
   Es muß eine TM eingegeben werden, samt Übergangsfunktion, Startzustand, Haltezustand und der Beschriftung des Bandes bei Start und Halt der Turingmaschine. Sie müssen also schon einmal die Berechnung der TM von Hand durchgeführt haben, um Haltezustand, Position des Kopfes und Bandbeschriftung bei Halt korrekt angeben zu können. Es wird jedoch eine einfache TM als Beispiel vorgegeben.

2. Festlegen der Transformationsvorschrift
   Die Transformation der Übergänge in Puzzleteile muß festgelegt werden. Es müssen also die 6 Kategorien von Teilen angegeben werden, indem die Randbeschriftungen der exemplarischen Puzzleteile festgelegt werden. Bei allen Kategorien sind die Ränder, an denen Hash-Zeichen (#) stehen, schon vorgegeben. Zusätzlich ist die Beschreibung der 2. Kategorie komplett vorgegeben.

3. Auslegen des Rahmens
   Aus der Überführungsfunktion der eingegebenen TM werden durch die Transformationsvorschrift Puzzleteiltypen erzeugt. Exemplare dieser Typen können jetzt in den Rahmen eingesetzt werden.

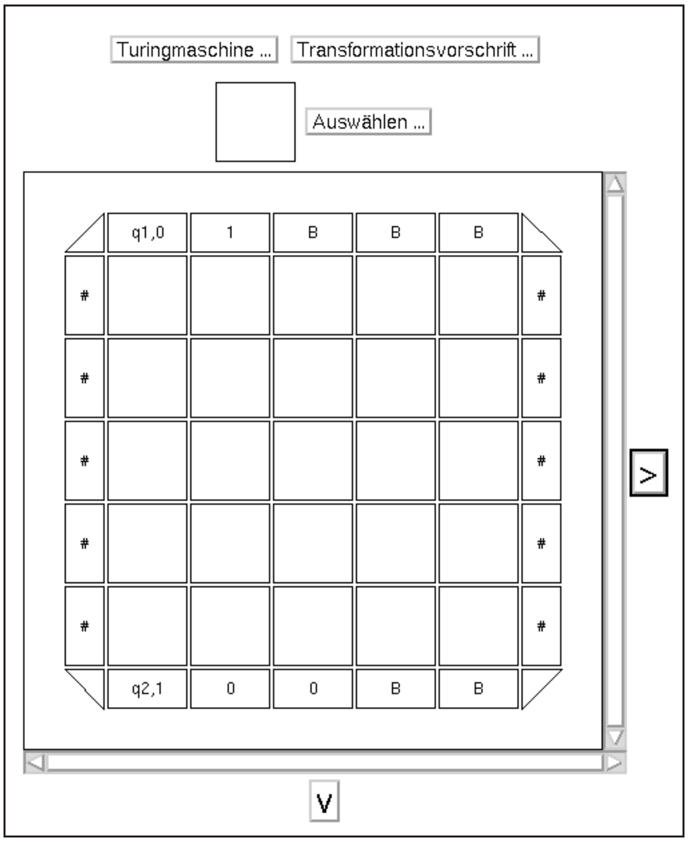


Abbildung 2.8: Das Puzzle-Applet.

Bedienungsanleitung Das Puzzle-Applet umfaßt vier Bereiche, die in verschiedenen Fenstern dargestellt werden. Das sind:

- das Puzzlespielfeld (der Rahmen mit den gelegten Puzzleteilen),
- das Eingabefenster für die Turingmaschine,
- die Transformationsvorschrift (die Beschreibung der 6 Kategorien)
- die Menge aller Puzzleteile, die in das Spielfeld eingesetzt werden können (Teilauswahlfenster).

Zu Beginn ist nur das leere Puzzlespielfeld zu sehen. Über dem Spielfeld gibt es zwei Buttons mit den Beschriftungen „Turingmaschine...“ und „Transformationsvorschrift...“. Durch Anklicken des Buttons „Turingmaschine...“ wird das Fenster zur Eingabe der Turingmaschine geöffnet. Bei Anklicken des Buttons „Transformationsvorschrift...“ erscheint das Fenster zum Festlegen der Transformationsvorschrift.

Direkt über dem Spielfeld wird der momentan zum Einsetzen ausgewählte Puzzleteiltyp angezeigt. Mit dem Button „ändern...“ wird das Teilauswahlfenster geöffnet und es kann eine neuer Puzzleteiltyp ausgewählt werden.

Jedes der drei zusätzlich zu dem Spielfeld angezeigten Fenster kann vorübergehend wieder geschlossen werden, ohne daß sein aktueller Inhalt verloren geht.

Eingabe einer Beispiel-Turingmaschine Die Übergangsfunktion wird zeilenweise eingegeben. Eine Transition besteht aus 5 Strings, die durch Kommata getrennt sind:

$$Zustand,\ Symbol,\ neuer\ Zustand,\ neues\ Symbol,\ Richtung$$

Bei Richtung steht ein „L“ für eine Kopfbewegung nach links, ein „R“ für eine Kopfbewegung nach rechts. Falls der Kopf sich nicht bewegt, wird für die Richtung ein „U“ eingetragen. Die anderen vier Strings können durch den Benutzer beliebig festgelegt werden. Als Blanksymbol (d.h. an dieser Position hat das Band noch keine Beschriftung) wird ein „B“ verwendet.

Die Bandbeschriftungen bei Start und Halt werden als Folge von Zeichenketten eingegeben, die durch Leerzeichen getrennt sind:

$$0\ 0\ 1\ 0\ 1\ \ldots$$

Als Start- und Haltezustand werden ebenfalls Strings eingegeben. Für die Kopfpositionen bei Start und Halt werden Zahlen eingegeben. Die Kopfposition 0 liegt bei dem ersten Symbol in der Bandbeschriftung.

Durch Anklicken des „Übernehmen"-Buttons wird das Teilauswahlfenster aktualisiert und die Beschriftung des Rahmens geändert. Lagen zu diesem Zeitpunkt schon Puzzleteile im Spielfeld, werden alle diese Puzzleteile gelöscht.

Durch einen Button „Zurücksetzen" können die vorgenommenen Änderungen rückgängig gemacht werden und auf den Stand beim letzten Übernehmen zurückgesetzt werden.

Bei Betätigung des „Löschen"-Buttons wird die im Eingabefenster sichtbare TM gelöscht.

Mit den „Laden"- und „Speichern"-Buttons können Dialogfenster zum Laden und Speichern einer TM geöffnet werden.

Festlegen der Transformationsvorschrift Jede der 6 Kategorien wird durch ein exemplarisches Puzzleteil dargestellt. Die Ränder eines exemplarischen Teils können eine der Beschriftungen aus Abb. 2.9 tragen:

a	ein Symbol
q	ein Zustand
q,a	ein Zustand und ein Symbol
b	ein neues Symbol
p	ein neuer Zustand
p,b	ein neuer Zustand und ein neues Symbol
#	das Hash-Zeichen
	keine Beschriftung

Abbildung 2.9: Mögliche Beschriftungen der Ränder der exemplarischen Teile

Neben jedem exemplarischen Puzzleteil wird ein „Ändern"-Button angezeigt, mit dem man ein Dialogfenster zum Ändern der Randbeschriftungen öffnet. Die dort gemachten Änderungen können mit „OK" bestätigt werden oder mit „Abbrechen" verworfen werden.

Sind bei dem exemplarischen Teil einer Kategorie alle Randbeschriftungen komplett ausgewählt, wird die entsprechende Zeile im Teilauswahlfenster aktualisiert. Außerdem werden die schon in das Spielfeld gelegten Teile dieser Kategorie angepaßt.

Mit den „Laden"- und „Speichern"-Buttons können Dialogfenster zum Laden und Speichern einer Transformationsvorschrift geöffnet werden.

Auswählen von Puzzleteilen („Teilauswahlfenster") Im Teilauswahlfenster werden für jede der 6 Kategorien die konkreten Puzzleteiltypen in einer Zeile dargestellt. Diese Typen wurden mit Hilfe der Transformationsvorschrift aus den Zeilen der Übergangsfunktion erzeugt. Ein Puzzleteiltyp kann durch Anklicken mit der Maus ausgewählt werden. Dieser Teiletyp wird durch inverse Darstellung hervorgehoben und gleichzeitig über dem Spielfeld angezeigt.

Das Puzzlespielfeld Mit einem Mausklick in ein Feld innerhalb des Rahmens wird dort ein Exemplar des über dem Spielfeld angezeigten Puzzleteiltyps eingesetzt. War in einem angeklickten Feld vorher schon ein anderes Puzzleteil gesetzt, so wird dieses überschrieben.

Paßt ein eingefügtes Puzzleteil nicht mit den benachbarten Puzzleteilen oder dem benachbarten Rahmen zusammen, werden auf beiden Seiten die betroffenen Ränder schwarz markiert.

Durch einen Mausklick mit der rechten Maustaste auf ein gelegtes Puzzleteil kann dieses wieder gelöscht werden.

Da das Spielfeld nicht komplett auf dem Bildschirm angezeigt werden kann, ist immer nur ein Ausschnitt des Spielfeldes sichtbar. Mit einer horizontalen und einer vertikalen Scrollbar kann der sichtbare Ausschnitt über dem Puzzlespielfeld verschoben werden.

Mit den Erweiterungs-Buttons $\boxed{>}$ und $\boxed{\vee}$ kann das Spielfeld vergrößert werden. Es wird eine neue Zeile bzw. Spalte werden am unteren bzw. rechten Rand des Spielfeldes eingefügt.

Wenn man die Rahmenbeschriftung anklickt und in der zugehörigen Zeile bzw. Spalte noch kein Puzzleteil gesetzt war, wird diese Zeile bzw. Spalte gelöscht. Spalten können nur dann gelöscht werden, wenn ihr oberes und unteres Rahmenteil mit „B“ beschriftet ist.

Übungsaufgabe 2.7.9

Welche der folgenden Aussagen sind richtig und welche sind falsch? Bitte kreuzen Sie die richtigen Kästchen an!

Jedes NP-vollstndige Problem ist NP-hart	◯ Ja	◯ Nein
3SAT ist polynomiell entscheidbar, falls alle Klauseln hchstens 2 Literale enthalten	◯ Ja	◯ Nein

2.8 Lösungen zu den Aufgaben

Lösung zu Aufgabe 2.7.1

(a) c-Hamiltonkreis liegt in NP: Rate nichtdeterministisch einen Teilgraph $G' = (V', E')$ von G mit c Knoten, das heißt, wähle nichtdeterministisch Teilmengen $V' \subseteq V$ und $E' \subseteq E$ mit $|V'| = c$. Dies läßt sich in linearer Zeit (also polynomiell) berechnen. Löse dann das Hamiltonkreisproblem in G'. Dieses liegt in NP, also ist c-Hamiltonkreis auch in NP.

(b) c-Hamiltonkreis ist NP-hart: Wir reduzieren das Problem HK Hamiltonscher Kreise auf das Problem c-Hamiltonkreis, indem eine Instanz von HK, also ein ungerichteter Graph $G = (V, E)$ wie folgt in eine Instanz von c-Hamiltonkreis, (einen Graph $G' = (V', E')$ und Konstante c) transformiert wird: $G' = G$ und $c = |V|$. Die Transformation ist in linearer Zeit, also polynomiell, berechenbar. Die Transformation ist lösungserhaltend: Ein Hamiltonkreis in G besteht nach Definition von HK aus $c = |V|$ Knoten und daher ist dieser Kreis eine Lösung von G' und umgekehrt.

Lösung zu Aufgabe 2.7.2

(a) Um von einem Problem zu zeigen, daß es in NP liegt, reicht es eine potentielle Lösung zu raten, um danach mit polynomiellem Zeitaufwand nachzuweisen, daß es wirklich eine Lösung des Problems darstellt.

Dies ist in diesem Fall recht einfach. Wir bezeichnen die Orte mit Zahlen von 1 bis n. Rate (nichtdeterministisch) eine Folge $i_1, \ldots, i_n$ aller Orte ($i_1, \ldots, i_n \in \{1, \ldots, n\}$ paarweise verschieden) und vergleiche die Gesamtkosten $c_{i_n i_1} + \Sigma_{i=1}^{n-1}$ mit K. Da es sich hier nur um Vergleichs- und einfache arithmetische Operationen handelt, ist der Zeitaufwand polynomiell.

(b) Das Problem HR des Handlungsreisenden ist dem Problem HK Hamiltonscher Kreise recht ähnlich. Wir zeigen deswegen HK $\leq_p$ HR.

Das Problem des Handlungsreisenden läßt sich wie folgt umformulieren: Gegeben ein (ungerichteter) Graph (G, R) mit $card(G) = n$, eine Kostenfunktion $c : R \to \mathbb{N}$ und eine Zahl $K \geq 0$. Gibt es eine Pfad

$k_1, k_2, \ldots, k_n$ durch G, so daß jeder Knoten $k_i, i = 1, \ldots, n$ genau einmal durchlaufen wird und $c(k_n, k_1) + \Sigma_{i=1}^{n-1} c(k_i, k_{i+1}) \leq K$ gilt.

f transformiere eine Instanz von Hamiltonkreis, das heißt ein ungerichteter Graph (G, R) und ein $n \in \mathbb{N}$ wie folgt in eine Instanz des Handlungsreisenden:

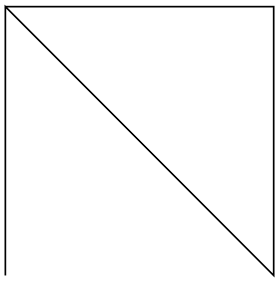
(i) Die Knoten des Graphs bleiben unverändert.

(ii) Zu jeder existierenden Kante (k_i, k_j) in (G, R) wird die Kante übernommen und $c(k_i, k_j) = 0$ gesetzt.

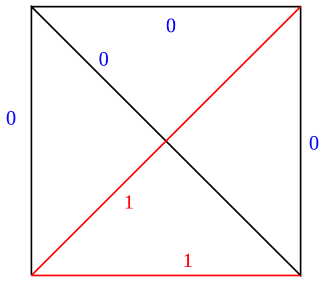
(iii) Zu jeder nicht existierenen Kante (k_i, k_j) in (G, R) wird die Kante (k_i, k_j) übernommen und $c(k_i, k_j) = 1$ gesetzt.

(iv) K sei 0.

Abbildung 2.10 (a) zeigt einen einfachen Graphen mit Hamiltonkreis und die



(a) Instanz von Hamiltonkreis



(b) Transformierte Instanz des Handlungsreisenden

Abbildung 2.10: Beispiel einer Transformation einer Instanz von Hamiltonkreis in eine Instanz des Handlungsreisenden: Die grauen Kanten sind neu hinzugekommen, die schwarzen sind geblieben. Jede zuvor existierenden Kante wurde mit den Kosten 0, und jede neue Kante mit den Kosten 1 versehen.

f ist mit quadratischen Aufwand berechenbar, da allenfalls n^2 neue Kanten hinzukommen.

Hat eine Instanz (G, R) von Hamiltonkreis eine Lösung, so besitzt das transformierte Problem einen Rundtour mit Gesamtkosten 0. Hat das transformierte Problem eine Lösung, dann gibt es eine Rundtour mit

Gesamtkosten 0. Jede benutzte Kante hat also Kosten 0. Da dies genau die existierenden Kanten des ursprünglichen Problems sind, besitzt dieses einen Hamiltonkreis.

Lösung zu Aufgabe 2.7.3

(a) (i) Für eine geratene Rundtour läßt sich in linearer Zeit überprüfen, ob sie eine Lösung des Flaschenhals-Handlungsreisenden ist. Also kann eine nichtdeterministische Turingmaschine das Problem in polynomieller Zeit lösen, d.h. der Flaschenhals-Handlungsreisende liegt in NP.

(ii) Wir Transformiere eine Instanz w von des Problems HK Hamiltonscher Kreise wie folgt in eine Instanz des Problem FHR des Flaschenhals-Handlungsreisenden:

Die Menge der Städte ist die Menge der Knoten von w. Für jede Kante von einer Stadt zu einer anderen setzte die Entfernung zu 1, für nicht existente Kanten wähle Entfernung 0. Wähle irgendein $K > 1$.

Dann hat eine Instanz HK genau dann eine Lösung, wenn die entsprechende transformierte Instanz von FHR eine Lösung besitzt.

(iii) Man kann zur Reduktion auch das Problem HR des Handlungsreisenden benutzen.

Die Transformation eines Problems von HR in eines von FHR ist recht einfach: Städte und Entfernungen bleiben wie sie sind und man wählt zusätzlich K um Eins größer als die weiteste Entfernung zwischen zwei Städten des Problems aus HR.

Lösung zu Aufgabe 2.7.4

(a) Das Problem ist in NP, da genau wie bei 3SAT eine Belegung von F nichtdeterministisch geraten und anschließend mit polynomiellen Aufwand auf Erfüllbarkeit getestet werden kann.

(b) Da das gegebene Problem eine Vereinfachung von 3SAT ist, liegt es nahe dieses für die Reduktion zu benutzen.

Sei F eine Formel in konjunktiver Normalform, die höchstens 3 Literale pro Klausel enthält. Aus F konstruieren wir eine erfüllbarkeitsäquivalente Formel F', so daß jede Variable höchstens drei mal und jedes Literal höchstens zwei mal in F' vorkommt:

Falls die Variable x k-mal in F vorkommt, so ersetze die einzelnen Vorkommen von x durch neue Variablen $x_1, \ldots, x_k$. In der so bereinigten Formel kommt dann keine Variable mehrfach vor. Um sicherzustellen, daß alle x_i den gleichen Wahrheitswert wie x besitzen, fügen wir zusätzlich $(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge \ldots \wedge (\neg x_n \vee x_1)$ zur Formel hinzu. Die resultierende Formel F' ist in konjunktiver Normalform und (i) jede Variable kommt höchstens drei mal vor, (ii) jedes Literal kommt höchstens zweimal vor und (iii) jede Klausel enthält nicht mehr als drei Literale.

Die Länge der Formel F' ist linear (in Abhängigkeit von der Länge von F). Diese Transformtion f kann offenbar in polynomieller Zeit berechnet werden und es gilt: F ist erfüllbar gdw. $F' = f(F)$ ist erfüllbar.

Lösung zu Aufgabe 2.7.5

Wir müssen zwei Dinge zeigen: (a) MONOTONE 3SAT liegt in NP und (b) jedes NP-vollständige Problem läßt sich auf MONOTON 3SAT reduzieren.

(a) Jeder Algorithmus der SAT löst kann auch für MONOTONE 3SAT verwendet werden, da offenbar MONOTONE SAT$\subset$ 3SAT gilt. Weil SAT in NP liegt, ist auch MONOTONE 3SAT in NP.

(b) Wir reduzieren 3SAT auf MONOTONE 3SAT. Wir transformieren dazu Instanzen S aus 3SAT (d.h. Klauselmengen) in Instanzen $\phi(S)$ (Klauselmengen) von MONOTONE 3SAT. Die Transformation ϕ sei durch folgende Vorschrift festgelegt:

   Für positive und negative Klauseln $C \in S$ sei $\phi(\{C\}) = \{C\}$. $C \in S$ sei nicht positiv oder negativ, O.E. nehmen wir an, daß $C = x \vee y \vee \neg z$ gilt (andere Fälle analog). Dann setzen wir $\phi(\{x \vee y \vee \neg z\}) = \{x \vee y \vee u, \neg u \vee \neg z\}$ für eine neu eingeführte aussagenlogische Variable u. ϕ ist wohldefiniert und bildet offenbar Instanzen von 3SAT in Instanzen von MONOTONE 3SAT ab. Die Abbildung kann in polynomieller Zeit berechnet werden, da sie jeder Klausel höchstens zwei Klauseln zuordnet.

Bleibt noch zu zeigen, daß S erfüllbar ist genau dann, wenn $\phi(S)$ erfüllbar: I sei eine Interpretation von S, dann erfüllt I offenbar alle von ϕ unverändert gelassenen Klauseln. Sei also $C \in S$ von der Form $\{x \vee y \vee \neg z\}$. I ist eine partielle Interpretation von $x \vee y \vee u, \neg u \vee \neg z$ und kann zu einer Interpretation $I' = I \cup \{u\}$, falls $z \in I$ gilt, oder $I' = I \cup \{\neg u\}$, falls $\neg z \in I$ gilt, erweitert werden.[1]

Umgekehrt ist offenbar jede Interpretation von $\phi(S)$ eine Interpretation von S.

Lösung zu Aufgabe 2.7.6

(a) $p = 1$ und $q = 0$ ist ein Modell von F_1

(b) Da jedes Konjunkt von F_1 ein komplementäres Paar enthält, ist F_2 nicht erfüllbar.

(c) Jede Instanz von D-SAT ist auch eine Instanz von SAT, d.h. es gilt D-SAT $\subset$ SAT. Da SAT $\in$ NP, folgt erst recht D-SAT $\in$ NP.

Es gilt sogar D-SAT $\in$ P, denn eine Formel in disjunktiver Normalform ist genau dann erfüllbar, wenn kein Konjunkt ein komplementäres Paar enthält. Letzteres ist mit quadratischen Zeitaufwand nachprüfbar.

(d) $$\begin{array}{l}(a \wedge u) \vee (a \wedge v) \vee (a \wedge w) \vee \\ (b \wedge u) \vee (b \wedge v) \vee (b \wedge w) \vee \\ (c \wedge u) \vee (c \wedge v) \vee (c \wedge w)\end{array}$$

(e) Nein, da die Berechnung von f exponentiellen Zeitaufwand verursacht. Falls es uns gelänge eine polynomielle Reduktion zu finden, dann gilt $P = NP$, da D-SAT $\in$ P liegt.

Lösung zu Aufgabe 2.7.7

Dieses Problem und die Lösung dazu werden ausgiebig in [13, Abschnitt 6.1, Seiten 123ff] diskutiert.

[1] Wir identifizieren Interpretationen mit den Literalen, die von ihr wahr gemacht werden, d.h. $L \in I$ gdw. $I(L) = true$.

(a) Das Minimum muß mindestens so groß wie die Summe der Größe aller Gegenstände sein, d.h.

$$\mathrm{OPT}(I) \geq \lceil \sum_{i=1}^{n} g(u_i) \rceil \ .$$

(b) Eingabe: Menge U von Gegenständen
Ausgabe: k

```
k := 1;
B[k] := 0;
foreach u ∈ U do
    fit := false;
    i := 1;
    repeat
        if B[i]+g(u) ≤ 1 then
            B[i] := B[i]+g(u);
            fit := true;
        else
            i := i+1;
        end
        if i > k then
            k := k+1;
            B[k] := g(u);
            fit := true;
        end
    until fit;
end
```

Im schlimmsten Fall ($g(u) = 1$ für alle $u \in U$) besitzt der Algorithmus den Zeitaufwand $O(n^2)$. Im günstigsten Fall ($\sum_{u \in U} g(u) \leq 1$) verursacht er linearen Zeitaufwand.

(c) Der Algorithmus liefert eine Lösung bei der höchstens ein Behälter weniger als zur Hälfte voll ist. Andernfalls müßte der letzte Gegenstand im zweiten halbleeren Behälter schon vorher in den ersten halbleeren Behälter eingefügt worden sein. Es gilt also

$$FF(I) < \lceil 2 \cdot \sum_{i=1}^{n} g(u_i) \rceil \ .$$

(d) Mit obigen Abschätzungen bekommt man direkt $\mathrm{FF}(I) < 2 \cdot \mathrm{OPT}(I)$.

Lösung zu Aufgabe 2.7.9

Welche der folgenden Aussagen sind richtig und welche sind falsch? Bitte kreuzen Sie die richtigen Kästchen an!

Jedes NP-vollstndige Problem ist NP-hart	√ Ja	◯ Nein
3SAT ist polynomiell entscheidbar, falls alle Klauseln hchstens 2 Literale enthalten	√ Ja	◯ Nein

% This file is in -*- LaTeX -*- mode

Kapitel 3

Reguläre Sprachen

3.1 Die Chomsky Hierarchie

Eine wichtige Aufgabe in vielen Teilgebieten der Informatik ist die saubere und verständliche Festlegung einer Menge von Wörtern, zum Beispiel der Wörter, oder vielleicht sollte man in diesem Zusammenhang besser von Zeichenfolgen sprechen, welche die erlaubten Programme einer Programmiersprache ausmachen. Auf einer etwas einfacheren Ebene innerhalb dieser sehr komplexen Gesamtaufgabe ist z.B. die Menge der erlaubten arithmetischen Ausdrücke festzulegen. Neben dem ersten Aspekt einer Beschreibung oder Erzeugung einer Menge von erlaubten Wörtern tritt der zweite Aspekt, von einem vorgegebenem Wort nachzuprüfen, ob es zur definierten Sprache gehört, Wir nennen ist beiden Aspekte der formalen Sprachtheorie den *generativen* bzw. den *analytischen* Aspekt. Als eine erste Möglichkeit Sprachen zu definieren betrachten wir, angelehnt an die Mittel, die zur Präzisierung natürlicher Sprachen benutzt werden, Grammatiken.

Definition 38 (Grammatik)
Eine Grammatik G *ist gegeben durch die Angabe der folgenden Bestimmungsstücke*

1. *einem Alphabet* N *für* Nichtterminalzeichen
2. *einem Alphabet* T *für* Terminalzeichen
3. *einem* Startsymbol $S \in N$

4. eine Menge von Produktionsregeln *der Form*

$$l \to r$$

mit $l, r \in (N \cup T)^*$, wobei l mindestens ein Symbol aus N enthält.

Um zu erklären wie eine Grammatik eine Menge von Wörtern erzeugt, brauchen wir ein bißchen Notation:

Definition 39 (Ableitbarkeit)
Es sei eine Grammatik $G = (N, T, S, \mathcal{P})$ vorgelegt. Wir führen die folgende Notation ein:

$u \Rightarrow w$	*es gibt eine Produktion $l \to r$ in $\mathcal{P}$, so daß l ein Teilwort von u ist, etwa $u = u_1 l u_2$ und $w = u_1 r u_2$.*
$\Rightarrow^+$	*transitiver Abschluß von $\Rightarrow$*
$\Rightarrow^*$	*transitiver, reflexiver Abschluß von $\Rightarrow$*

Falls erforderlich schreiben wir $u \Rightarrow_G w$ anstelle von $u \Rightarrow w$

Definition 40 (Sprachschatz)
Die von einer Grammatik G erzeugte Sprache $L(G)$ ist jetzt definiert durch

$$L(G) = \{u \in T^* \mid S \Rightarrow^* u\}$$

$L(G)$ wird auch der Sprachschatz *von G genannt.*

Beispiel 24
Die folgende Grammatik G_{NL} kann benutzt werden um einige Sätze in deutscher Sprache zu generieren. Hier sind zunächst die Produktionsregeln:

$Satz \to SPO.$	$No \to bauer \mid königin \mid klee$
$S \to NaNo$	$P \to lobt \mid kennt$
$O \to DaNo$	$Na \to der \mid die \mid das$
	$Da \to den \mid die \mid das$

Das Startzeichen von G_{NL} ist $Satz$.

Die Nichtterminalzeichen sind

$$N = \{Satz, S, P, O, Na, Da, No\}$$

Die Terminalzeichen sind

$$T = \{., bauer, königin, klee, lobt, kennt, der, die, das, den\}$$

Zu dem Sprachschatz $L(G_{NL})$ gehören zum Beispiel: „der bauer lobt die königin ." und „die königin lobt den klee .". Wir überzeugen uns, daß der erste Satz zum Sprachschatz von G_{NL} gehört, indem wir eine Kette von $\Rightarrow$-Relationen ausgehen von Startsymbol verfolgen:

Satz
SPO.
NaNoPO
der NoPO.
der bauer PO.
der bauer lobt O.
der bauer lobt DaNo.
der bauer lobt die No.
der bauer lobt die Königin .

Das Beispiel erklärt einleuchtend den Unterschied zwischen Terminal- und Nichtterminalzeichen. Letztere sind hier grammatische Kategorien

Zeichen	*Bedeutung*
Satz	*Satz*
S	*Subjekt*
P	*Prädikat*
O	*Objekt*
No	*Nomen*
Na	*Artikel im Nominativ*
Da	*Artikel im Dativ*

Die Nichtterminalzeichen für G_{NL} sind die Wörter, die normalerweise in einem Lexikon stehen würden. Wir haben die Produktionsregeln schon so angeordnet, daß auf der linken Seite die Regeln stehen, die man in einer Grammatik der deutschen Sprache finden würde. In der rechten Spalte stehen die Regeln, die Einträgen in einem Lexikon entsprechen.

Eine zweite Beobachtung kann man an diesem einfachen Beispiel machen. Die Zeichen in den Symbolmengen N und T müssen nicht aus einem einzelnen Buchstaben bestehen. Die genaue Angabe von N und T dient gerade dazu festzulegen, was man für eine bestimmte Grammatik als **Grundsymbole** *(engl. token) ansehen möchte. Das Wort Satz ist* **ein** *Symbol in N, das nicht weiter in Einzelteile zerlegt wird. Wir werden in den weiteren Beispielgrammatiken der Einfachheit halber als Zeichen in N und T meist einzelne Buchstaben wählen.*

Die Grammatik G_{NL} hat nur exemplarischen Wert, als Grammatik für die deutsche Sprache ist sie unbrauchbar. Sie bietet z.B. keine Möglichkeit morphologische Information zu berücksichtigen, etwa die Information, daß lobt von dem Infinitiv loben kommt. Auch können keine Attribute von Nomen, wie z.B. das grammatikalische Geschlecht, berücksichtigt werden. Das führt dazu, daß auch der falsche Satz „der bauer lobt den königin ." in G_{NL} abgeleitet werden kann.

Beispiel 25
Die Grammatik G_5 sei gegeben durch:

- *Startzeichen S,*
- *Terminalzeichen a, b,*
- *Nichtterminalzeichen S*
- *und die Produktionen*

$$\begin{array}{ll} S \to \varepsilon & \qquad S \to aSb \\ S \to SS & \qquad S \to bSa \end{array}$$

Der Sprachschatz der Grammatik G_5 kann nicht so einfach aus den Grammatikregeln abgelesen werden. Wir behaupten:

Für $w \in \{a,b\}^*$ und $c \in \{a,b\}$ sei $N_c(w)$ die Anzahl der Vorkommen von c in w. Es gilt

$$L(G_5) = \{w \in \{a,b\}^* \mid N_a(w) = N_b(w)\}$$

Erklärung *Wir überlegen uns zunächst, daß jedes in G_5 ableitbare Wort w die Eigenschaft*

$$N_a(w) = N_b(w)$$

hat. Das geschieht am bequemsten durch Induktion über die Länge k einer Ableitung von w. Ist $k = 1$, dann kommt nur $w = \varepsilon$ in Frage und wegen $N_a(w) = N_b(w) = 0$ ist die Behauptung in diesem Fall wahr. Für $k > 1$ machen wir eine Fallunterscheidung nach der ersten Produktionsregel in der Ableitung von w:

$$\begin{array}{lll} S \Rightarrow aSb & \text{und} & w = aw_1b \\ S \Rightarrow bSa & \text{und} & w = bw_1a \\ S \Rightarrow SS & \text{und} & w = w_1w_2 \end{array}$$

Da die Ableitungen für w_1 und w_2 um mindestens um einen Schritt kürzer sind als die für w folgt aus der Induktionsvoraussetzung, daß $N_a(w_i) = N_b(w_i)$ für $i \in \{1,2\}$ gilt. Daraus folgt aber in allen drei Fällen unmittelbar auch $N_a(w) = N_b(w)$.

Gelte jetzt umgekehrt für ein Wort w die Gleichung $N_a(w) = N_b(w)$. Wir wollen $S \Rightarrow^* w$ zeigen. Wieder empfiehlt sich eine Induktion, diesmal über die Länge m von w. Für $m = 0$ muß w das leere Wort sein und das ist ableitbar. Ist $m > 0$ dann unterscheiden wir vier Fälle

$w = aw_1b$
$w = bw_1a$
$w = aw_1a$
$w = bw_1b$

In den ersten beiden Fällen gilt offensichtlich auch $N_a(w_1) = N_b(w_1)$. Nach Induktionsvoraussetzung folgt also $S \Rightarrow^* w_1$, woraus durch Vorschalten der Regel $S \rightarrow aSb$ oder $S \rightarrow bSa$ eine Ableitung $S \Rightarrow^* w$ erhalten werden kann. Betrachten wir jetzt den dritten Fall. Sei $w = c_1 \ldots c_m$. Mit w_i bezeichnen wir das Anfangsstück $c_1 \ldots c_i$ von w für $1 \leq i \leq m$. Nach der Fallannahme gilt $1 = N_a(w_1) > N_b(w_1) = 0$. Da sich beim Übergang von i nach $i+1$ genau eine der Zahlen $N_a(w_i)$ oder $N_a(w_i)$ um eins ändert und $N_a(w_m) = N_b(w_m)$ vorausgesetzt wurde , muß es ein kleinstes j mit $1 < j \leq m$ geben mit $N_a(w_j) = N_b(w_j)$. Dieses j muß aber kleiner als $m-1$ sein, denn aus $N_a(w_{m-1}) > N_b(w_{m-1})$ würde wegen der Fallannahme der Widerspruch $N_a(w_m) > N_b(w_m)$ folgen. Somit läßt sich w schreiben als

$$w = w_j u_j$$

mit $u_j = c_{j+1} \ldots c_m$ mit $N_a(w_j) = N_b(w_j)$ und $N_a(u_j) = N_b(u_j)$. Da beide Wörter kürzer sind als w gilt nach Induktionsvoraussetzung $S \Rightarrow^* w_j$ und $S \Rightarrow^* u_j$. Durch Vorschalten von $S \rightarrow SS$ wird eine Ableitung von w erhalten. Der vierte Fall wird analog behandelt.

Der in Definition 38 eingeführte Begriff einer Produktionsregel ist noch sehr allgemein. In vielen Fällen wird die volle Stärke des allgemeinen Begriffs nicht gebraucht. Die wichtigste Klassifizierung von Grammatiken ist die *Chomsky Hierarchie*, benannt nach Naom Chomsky. Chomsky hat in entscheidendem Maße die moderne Linguistik geprägt. In den letzten Jahren hat er sich mehr sozialwissenschaftlichen Themen und politischem Engangement gewidmet.

Die Klassifikation von Grammatiken geschieht durch eine Klassifikation von Produktionsregeln in verschiedene Typen. Dazu benötigen wir die folgende Terminologie.

Definition 41 (Typen von Produktionsregeln)
N Nichtterminalzeichen, T Terminalzeichen,
$V = N \cup T$

Produktion	*Name*	*Bedingung*
$l \to r$	*allgemein*	$l, r \in V^*$, $l \notin T^*$
$l \to r$	*beschränkt (od. erweiternd)*	*wie allgemein und* $\|l\| \leq \|r\|$
$uAv \to urv$	*kontextsensitiv*	$u, v, r \in V^*$ $A \in N$, $r \neq \varepsilon$
$A \to r$	*kontextfrei*	$A \in N$, $r \in V^*$
$A \to Bx$	*linkslinear*	$A, B \in N$, $x \in T^*$
$A \to xB$	*rechtslinear*	
$l \to \varepsilon$	*ε-Produktion*	$l \in V^*$
$A \to x$	*terminierend*	$x \in T^*$

Definition 42 (Die Chomsky Hierarchie)

Chomsky Typ	*Bezeichnung*	*Eigenschaft aller Produktionen*
0	*allgemein*	*beliebig*
1	*kontextsensitiv*	*kontextsensitiv*
2	*kontextfrei*	*kontextfrei*
3	*regulär*	*rechtslinear oder terminierend*

Für die Informatik sind im wesentlichen nur die unteren beiden Stufen der Chomsky Hierarchie wichtig.

Lemma 30 *Es lassen sich die folgenden Inklusionen beweisen:*

$$\textit{Typ } 0 \supset \textit{Typ } 1 \supset \textit{Typ } 2 \supset \textit{Typ } 3$$

Beweis: Die schwachen Inklusionsbeziehungen Typ 0 $\supseteq$ Typ 1 $\supseteq$ Typ 2 $\supseteq$ Typ 3 sind bis auf den Fall Typ 1 $\supseteq$ Typ 2 offensichtlich. Das Problem in dem restlichen Fall ist ein (wichtiges) Detail: In Typ 1 Grammatiken sind ε-Produktionen nicht erlaubt, wohl aber in Typ 2 Grammatiken. Theorem 58 zeigt, daß man in in Typ 1 Grammatiken auf ε-Produktionen verzichten kann.

Die strikten Inklusionsbeziehungen sind schwieriger zu beweisen. Eine Sprache, die kontextfrei aber nicht regulär ist, wird in Beispiel 36 vorgestellt, eine kontextsensitive, nicht kontextfreie Sprache in Beispiel 50. Die Existenz einer Sprache, die nicht durch eine kontextsensitiv Grammatik produziert werden kann, folgert man am besten aus Entscheidbarkeitseigenschaften von Grammatiken, siehe Abschnitt 4.5.

■

3.2 Endliche Automaten

Endliche Automaten spielen in vielen Teilen der Informatik eine große Rolle, wo es um die Beschreibung von diskreten Zustandsänderungen infolge externer oder internen Ereignisse geht. Unzählige Varianten des Begriffs eines Automaten haben in der Informatik nützliche Anwendungen gefunden. Einige davon werden wir im Verlauf dieses Kurses noch näher kennenlernen, auf weitere Varianten und Modifikationen werden wir wenigstens hinweisen.

Wir motivieren die Grundbegriffe eines endlichen Automaten anhand eines allgemeinverständlichen Beispiels.

Die folgende Denksportaufgabe ist weithin bekannt. In der Ausgangssituation befinden sich ein Mensch, ein Wolf, eine Ziege und ein Salatkopf auf einer Seite eines Flusses. Es steht ein kleines Ruderboot zur Verfügung um die ganze Menagerie auf die andere Flußseite zu bringen. Bei jeder Fahrt muß der Mensch zum rudern mit dabeisein und außerdem kann höchstens ein weiteres Tier oder der Salatkopf mittransportiert werden. Weiterhin muß als Randbedingung eingehalten werden, daß niemals der Wolf und Ziege oder die Ziege und der Salatkopf allein auf einem Ufer sind. Gibt es eine Möglichkeit der Flußüberquerung unter diesen Randbedingungen? und welche?

Wir notieren die verschiedenen Stationen, oder Zwischenzustände, auf dem Weg zu einer Lösung als untereinanderstehende Folgen der Buchstaben M,W,Z,S. Wobei die Buchstaben in naheliegender Weise die beteiligten Akteure bezeichnen und die beiden Zeilen die beiden Flußufer. So steht

$\begin{matrix} MWZS \\ \emptyset \end{matrix}$ für die Anfangssituation und $\begin{matrix} \emptyset \\ MWZS \end{matrix}$ für die angestrebte Zielsituation. Durch eine Flußüberquerung ändert sich der Zustand. Wir notieren die möglichen Änderungen in einem Graphen, in dem die Zustände die Ecken und die Zustandsübergänge die Kanten bilden, siehe Abb. 3.1. Eine Flußüberquerung, die der Mensch allein unternimmt ist dabei mit M gekennzeichnet. Setzt er zusammen mit dem Wolf oder der Ziege oder dem Salat über schreiben wir an die entsprechende Kante W,Z oder S. Die Richtung der Überfahrt muß nicht explizit angegeben werden, sie ergibt sich aus dem Zusammenhang. Die Überfahrt muß auf dem Ufer starten, auf dem sich der Mensch gerade befindet. Die insgesamt sechs verbotenen Zustände sind garnicht erst eingezeichnet worden. Damit entfallen auch alle Kanten, die zu verbotenen Zuständen führen würden. Aus der Abbildung läßt sich nun leicht ein Lösungsweg ablesen, z.B. $Z - M - S - Z - W - M - Z$.

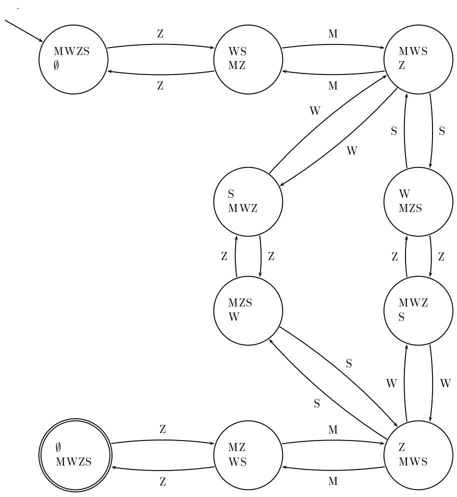


Abbildung 3.1: Zustandsübergangsgraph zur Flußüberquerung

3.2.1 Deterministische endliche Automaten

Nach dieser Vorbereitung kommen wir nun zur formalen Definition.

Definition 43
Ein endlicher Automat $\mathcal{A}$ ist bestimmt durch

- *eine endlichen Menge Q von* Zuständen*,*
- *ein* Startzustand $q_0 \in Q$*,*
- *einem endlichen* Alphabet Σ*,*
- *einer Menge $F \subseteq Q$ von* Endzuständen*,*
- *eine* Übergangsfunktion $\delta : Q \times \Sigma \to Q$

Die Einsatzmöglichkeiten für endliche Automaten sind, wie das einführende Beispiel schon nahelegt, sehr vielfältig. Wir konzentrieren uns zunächst auf die Möglichkeit mit endlichen Automaten eine Menge von Wörtern festzulegen, um nicht korrekte Wörter von korrekt gebildeten zu unterscheiden. Dazu liest man, im Anfangszustand beginnend, nacheinander die Buchstaben eines Wortes w ein Dabei folgt ma den Kanten, die mit dem gelesenen Buchstaben markiert sind. Wird nach Einlesen des letzten Buchstabens ein Zustand in F erreicht gilt das Wort als akzeptiert in allen anderen Fällen als zurückgewiesen.

Der Wert der Übergangsfunktion $\delta(q, a)$ gibt den Zustand an, der ausgehend von Zustand q beim Einlesen des Buchstabens a erreicht wird. Liest man nacheinander die Buchstaben eines Wortes w ein, so läßt sich durch wiederholte Anwendung von δ der erreichte Zustand ermitteln. Da diese Funktion häufig gebraucht wird vereinbaren wir eine feste Bezeichnung dafür. Für Wörter w mit nur einem Buchstaben stimmt diese Funktion mit δ überein. Sie ist also eine Fortsetzung von δ. Wir wollen diese Fortsetzung ebenfalls mit δ bezeichnen.

Definition 44
Die Funktion $\delta : Q \times \Sigma^* \to Q$ ist für $a \in \Sigma$, $w \in \Sigma^*$ definiert durch

$$\begin{aligned} \delta(q, a) &= \delta(q, a) \\ \delta(q, wa) &= \delta(\delta(q, w), a) \\ \delta(q, \varepsilon) &= q \end{aligned}$$

Definition 45
Es sei ein endlicher Automat $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$ gegeben. Die von $\mathcal{A}$ akzeptierte Sprache $L(\mathcal{A})$ ist definiert durch

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

Beispiel 26
Der in Abb. 3.2 gezeigte endliche Automat $\mathcal{A}_{Scons}$ akzeptiert genau die in Pascal erlaubten Konstanten für Zeichenketten (string constants). Sie müssen mit einem Hochkomma ' beginnen und enden, dürfen beliebige Buchstaben enthalten, was in $\mathcal{A}_{Scons}$ durch den einen Buchstaben c abstrahiert wird. Wenn innerhalb der Zeichenkette ein Hochkomma vorkommen soll, so muß es dupliziert werden.

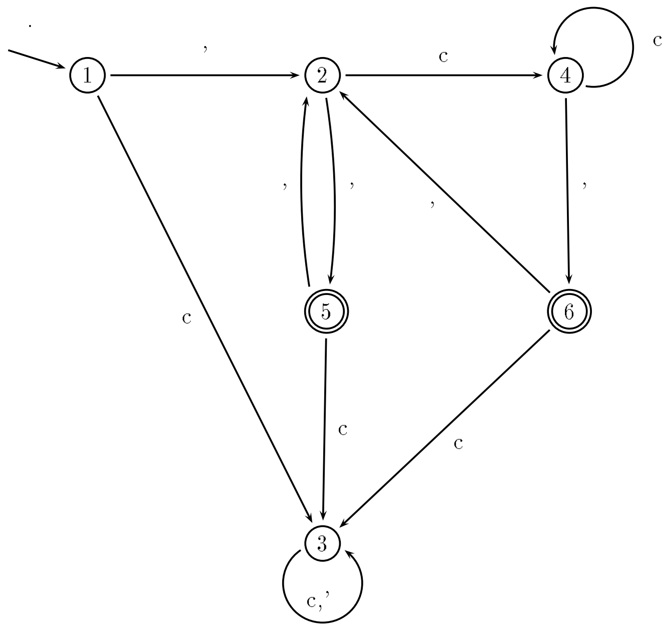


Abbildung 3.2: Automat $\mathcal{A}_{Pcons}$ für Pascal Zeichenketten

Definition 46
Gelegentlich wird die Übergangsfunktion δ eines endlichen Automaten $\mathcal{A}$ nicht an allen Stellen von $Q \times \Sigma$ definiert. Soll dieser Umstand besonders hervorgehoben werden, so redet man von einem partiellen *endlichen Automaten. Umgekehrt nennt man einen Automaten, für den $\delta : Q \times \Sigma \to Q$ eine totale Funktion ist einen* vollständigen *endlichen Automaten. Die Definition von $L(\mathcal{A})$ bleibt zwar wörtliche dieselbe, muß aber mit größerer Sorgfalt gelesen werden. Die Forderung $\delta(q_0, w) \in F$ beinhaltet, daß für jedes Teilwort w' von w der Funktionswert $\delta(q_0, w')$ auch definiert ist. Der endliche Automat aus Abbildung 3.3 akzeptiert das Wort bac nicht, weil $\delta(q_0, bac) = \delta(q_1, c)$ nicht definiert ist. Zwar ist der letzte definierte Zustand beim Einlesen des Wortes bac, nämlich q_1, ein Endzustand. Aber das ist nicht ausschlaggebend. Das*

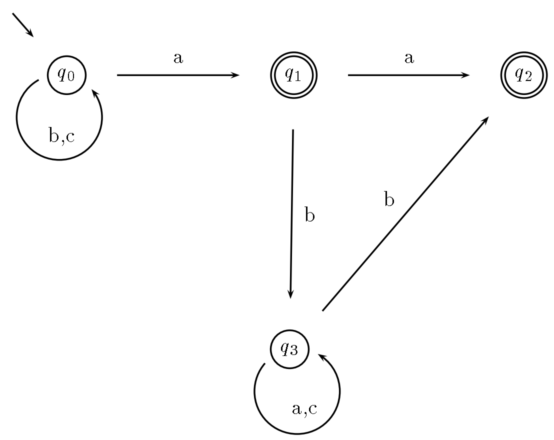


Abbildung 3.3: Partieller Akzeptor

ganze Wort muß erst eingelesen werden, der dann erreicht Zustand entscheidet über Akzeption oder Zurückweisung. In partiellen endlichen Automaten gibt es also zwei Möglichkeiten ein Wort zurückzuweisen: nach Einlesen des gesamten Wortes wird ein nicht-Finalzustand erreicht oder während des Einlesen des Wortes wird eine Stelle erreicht, an der δ nicht definiert ist.

Lemma 31
Zu jedem partiellen endlichen Automaten $\mathcal{A}$ gibt es einen vollständigen endlichen Automaten $\mathcal{A}_f$ mit

$$L(\mathcal{A}) = L(\mathcal{A}_f)$$

Beweis: Sei $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$ der gegebene Automat. Der gesuchte Automat $\mathcal{A}_f$ besitzt dasselbe Alphabet Σ wie $\mathcal{A}$ und auch denselben Anfangszustand und dieselben Endzustände. Die Menge der Zustände Q_f von $\mathcal{A}_f$ entsteht aus Q durch Hinzunahme eines neuen Zustands q_f, d.h. $Q_f = Q \cup \{q_f\}$. Für jeden Zustand q wird nun für alle Buchstaben $x \in \Sigma$ für die von q aus keine Übergang definiert ist, ein Übergang von q nach q_f hinzugefügt. Insbesondere gilt das für q_f selbst. Man nennt q_f häufig den *Fehlerzustand* (error state) von A_f.

∎

Der vervollständigte Automat zu dem partiellen Automaten aus Abbildung 3.3 ist in Abbildung 3.4 zu sehen.

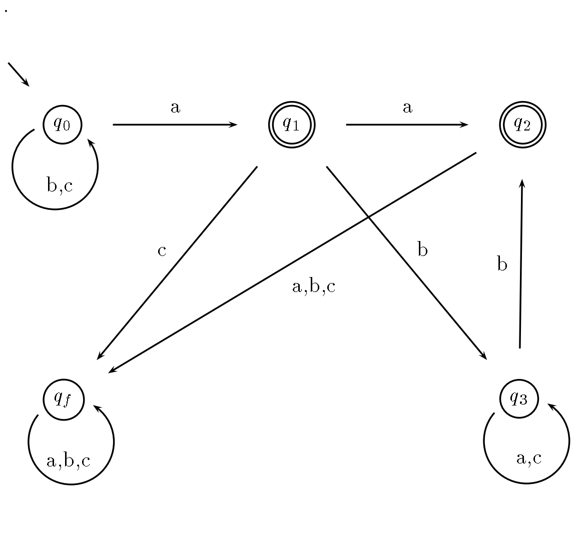


Abbildung 3.4: Vervollständigung des partiellen Akzeptors aus Abb. 3.3

3.2.2 Nichtdeterministische endliche Automaten

Für die bisher betrachteten endlichen Automaten ist in jedem Zustand für jeden Buchstaben des Eingabealphabets höchstens ein Übergang möglich. Für manche diskreten Zustandsübergangssysteme ist das eine zu einschneidende Forderung. Deshalb führen wir jetzt nichtdeterministische endliche Automaten ein.

Wer zum ersten Mal mit dem Begriff eines nichtdeterministischen Automaten konfrontiert wird, hat in der Regel Schwierigkeiten dessen Nützlichkeit einzusehen. Deswegen soll hier eine kurze Erklärung eingeschoben werden.

Wir werden uns im folgenden davon überzeugen, daß jede Sprache, die durch eine reguläre Grammatik, etwa einer rechtslinearen Grammatik, erzeugt werden kann, auch durch einen geeignet gewählten endlichen Automaten berechnet werden kann.

Das Erzeugung eines Wortes durch Grammatikregeln ist ein inhärent nichtdeterministischer Prozeß. Selbst in der einfachen Grammatik aus Beispiel 24 treten an mehreren Stellen Alternativen für die Anwendung von Produktionsregeln auf, z.B. $P \to lobt \mid kennt$.

Soll die oben aufgestellte Behauptung richtig sein, muß auch in dem Konzept eines endlichen Automaten ein Nichtdeterminismus möglich sein.

Die wichtigste Änderungen in der Definition nichtdeterministischer endlicher

Automaten gegenüber deterministischen, ist die Tatsache, daß die Übergangsfunktion δ nicht einen einzelnen Zustand als Funktionswert besitzt, sondern eine Menge möglicher Folgezustände.

Definition 47 (Nichtdeterministische endl. Automaten)
Ein nichtdeterministischer endlicher Automat $\mathcal{A}$ ist bestimmt durch

- *eine endlichen Menge Q von* Zuständen,
- *ein* Startzustand $q_0 \in Q$,
- *einem endlichen* Alphabet Σ,
- *einer Menge $F \subseteq Q$ von* Endzuständen,
- *eine* Übergangsfunktion $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$

Die Übergangsfunktion δ wird fortgesetzt zu einer Funktion $\delta : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$ durch

$$\begin{array}{ll} \delta(q,a) & = \delta(q,a) \\ \delta(q,wa) & = \bigcup\{\delta(r,a) \mid r \in \delta(q,w)\} \\ \delta(q,\varepsilon) & = \{q\} \end{array}$$

$a \in \Sigma, w \in \Sigma^*$

Es sei ein nichtdeterministischer endlicher Automat $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$ gegeben. Wir müssen festlegen, was die von einem nichtdeterministischen Automaten akzeptierte Sprache sein soll. Zu einem Wort $w \in \Sigma^*$ ist jetzt $\delta(q_0, w)$ die Menge aller von q_0 aus beim Einlesen von w erreichbaren Wörter. Für die Akzeption von w reicht die schwache Forderung aus, daß wenigstens einer der möglichen entlang w erreichbaren Zustände ein Finalzustand ist.

Definition 48
Die von einen nichtdeterministischen endlichen Automaten $\mathcal{A}$ akzeptierte Sprache $L(\mathcal{A})$ ist definiert durch

$$L(\mathcal{A}) = \{w \in \Sigma^* \mid \delta(q_0, w) \cap F \neq \emptyset\}$$

Theorem 32

1. *Zu jeder rechtslinearen Grammatik G gibt es einen endlichen Automaten A, so daß die von G generierte und die von A akzeptierte Sprache zusammenfallen:*
$$L(G) = L(A)$$
2. *Zu jedem endlichen Automaten A gibt es eine rechtslineare Grammatik G mit*
$$L(G) = L(A)$$

Beweis, Teil 1 Zu einer rechtslinearen Grammatik G mit N als Menge der Nichtterminalzeichen, T Menge der Terminalzeichen und Menge $\mathcal{P}$ von Produktionen bilden wir den endlichen Automaten A mit

	Automat	Grammatik
Σ	Alphabet	$= T$
Q	Zustandsmenge	$= N \cup \{q_f\}$
q_0	Startzustand	$= S$ (Startsymbol)
F	Endzustände	$= \{q_f\}$

dabei ist q_f ein Symbol, das nicht in N vorkommt. Die Zustandsübergangsfunktion δ für A berechnet sich wie folgt aus den Produktionsregeln von G: Sind $A \to aB_1, \ldots A \to aB_n$ alle Produktionen in $\mathcal{P}$ mit A auf der linken Seite und a als Anfang der rechten Seite, dann setzt man

$$\delta(A, a) = \{B_1, \ldots, B_n\}.$$

Für jede terminale Produktion $A \to a$ in $\mathcal{P}$ setzt man

$$\delta(A, a) = \{q_f\}.$$

Jetzt muß man sich noch davon überzeugen, daß $L(G) = L(A)$ gilt. Dazu zeigt man durch Induktion über k die Äquivalenz

$$S \Rightarrow^* a_1 \ldots a_k B \text{ genau dann, wenn } B \in \delta(S, a_1 \ldots a_k).$$

Beweis, Teil 2 Das können wir nach dem vorangegangenen Teil schnell erledigen. Die Terminalzeichen der gesuchten Grammatik ist das Alphabet des gegebenen Automaten. Die Nichtterminalzeichen entsprechen den Zuständen des Automaten, inklusive Endzustände, das Startsymbol ist der Anfangszustand. Für jedes $B \in \delta(A, a)$ enthält G die Produktion $A \to aB$. Gilt $B \in \delta(A, a)$ und ist B ein Endzustand, dann nehmen wir zusätzlich noch die Produktion $A \to a$ mit hinzu.

■

Theorem 33
Zu jedem nichtdeterministischen endlichen Automaten $\mathcal{A}_{nd}$ gibt es einen deterministischen endlichen Automaten $\mathcal{A}_d$ mit

$$L(\mathcal{A}_{nd}) = L(\mathcal{A}_d)$$

Beweis: Theorem 34 behauptet eine etwas stärkere Aussage als die des vorliegenden Satzes. Wer an einem direkten Beweis der schwächeren Aussage interessiert ist kann ihn in [14] nachschlagen.

Als nächste Variante endlicher Automaten kommen wir nun zu den Automaten mit ε-Übergängen oder, wie man auch häufig dazu sagt, spontanen Übergängen. Bisher war ein Übergang von einem Zustand aus immer durch das Einlesen eines Buchstabens des Eingabealphabets ausgelöst worden. Das muß nicht immer der Fall sein, der Übergang könnte auch ohne Einleseaktion,eben spontan, erfolgen.

Definition 49 (Nichtdet. endl. Automaten mit ε-Übergängen)
Ein nichtdeterministischer endlicher Automat $\mathcal{A}$ mit ε-Übergängen ist bestimmt durch

- *eine endlichen Menge Q von* Zuständen,
- *ein* Startzustand $q_0 \in Q$,
- *einem endlichen* Alphabet Σ,
- *einer Menge $F \subseteq Q$ von* Endzuständen,
- *eine* Übergangsfunktion $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$

Die einzige Änderung gegenüber Definition 47 liegt in in dem zustäzlichen Symbol ε im Argumentbereich von δ.Die nach Einlesen eines Eingabewortes erreichbaren Zustände sind jetzt nicht mehr so einfach zu beschreiben, da an jeder Stelle eine beliebige Anzahl spontaner Übergänge erfolgen kann. Wir führen daher den folgenden Hilfsbegriff ein.

Der ε-Abschluß einer Teilmenge $P \subseteq Q$ ist die kleinste Obermenge $Cl_\varepsilon(P)$ von P, die abgeschlossen ist unter ε-Übergängen, d.h.

$$\begin{array}{ll} Cl^0_\varepsilon(P) & = P \\ Cl^{n+1}_\varepsilon(P) & = \{q \in Q \mid \exists p \in Cl^n_\varepsilon(P)(q \in \delta(p, \varepsilon))\ \} \\ Cl_\varepsilon(P) & = \bigcup_{n \geq 0} Cl^n_\varepsilon(P) \end{array}$$

Ist $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$ ein nichtdeterministischer endlicher Automat mit ε-Übergängen, so besteht $L(\mathcal{A})$ aus allen Wörtern $w = x_1 \ldots x_n \in \Sigma^*$ für die eine Folge von Zuständen $q_0, q_0', q_1, q_1', \ldots q_n, q_n'$ existiert, so daß

- $q_{i+1} \in \delta(q_i', x_{i+1})$ für alle $0 \leq i < n$ und
- q_i' ist für alle i von q_i aus durch endliche viele ε-Übergänge erreichbar
- $q_n' \in F$

Formale Definition:

Definition 50
Für $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$ wird die Funktion $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ fortgesetzt zu einer Funktion $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$ durch

$$\begin{array}{lcl} \hat{\delta}(q, \varepsilon) & = & \{q\} \\ \hat{\delta}(q, wa) & = & Cl_\varepsilon(\bigcup\{\delta(p, a) \mid p \in \hat{\delta}(q, w)\} \end{array}$$

Die von $\mathcal{A}$ akzeptierte Sprache $L(\mathcal{A})$ kann jetzt definiert werden als

$$L(\mathcal{A}) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

Eine Menge L von Wörtern, die sich in der Form $L = L(\mathcal{A})$ für einen endlichen Automaten $\mathcal{A}$ darstellen läßt heißt eine reguläre Sprache.

Theorem 34
Zu jedem nichtdeterministischen endlichen Automaten mit ε-Übergängen gibt es einen äquivalenten deterministischen endlichen Automaten ohne ε-Übergänge.

Ein Beweis Dieser Beweis hat Modellcharakter. Der Übergang von einem nichtdeterministischen Verfahren zu einem deterministischen kommt häufiger in der Informatik vor und die Idee dazu ist immer dieselbe. Anstatt die von einem Zustand q_1 aus durch unterschiedliche nichtdeterministische Auswahl erreichbaren Zustände hintereinander zu betrachten, betrachten wir auf einmal die Menge dieser Zustände. Indem wir Mengen von Zuständen des nichtdeterministischen Automaten zu Zuständen des deterministischen Automaten machen verschwindet der Nichtdeterminismus. Wir zahlen dafür aber den Preis in der mengentheoretischen Hierarchie eine Stufe höher zu steigen.

Wir führen zunächst einen rein theoretischen Existenzbeweis. Gegeben sei der nichtdeterministische endliche Automat $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$ mit spontanen

Übergängen. Mit $\mathcal{B} = (\mathcal{P}(Q), \Sigma, \{q_0\}, F_d, \Delta)$ bezeichnen wir den gesuchten äquivalenten deterministische Automat. $\mathcal{B}$ besitzt natürliche dasselbe Alphabet wie $\mathcal{A}$, die Menge der Zustände von $\mathcal{B}$ ist die Menge aller Teilmengen von Q, der Anfangszustand ist die Einermenge $\{q_0\}$, Endzustände sind die Mengen, die einen Endzustand von $\mathcal{A}$ enthalten, d.h.

$$F_d = \{X \in \mathcal{P}(Q) \mid F \cap X \neq \emptyset\}$$

Es bleibt als wichtigster Teil der Definition die Transitionsfunktion Δ für jede Teilmenge $X \subseteq Q$ und jedes Symbol $x \in \Sigma$ zu erklären:

$$\Delta(X, x) = Cl_\varepsilon(\bigcup_{q \in X} \delta(q, x))$$

Ist $\mathcal{A}$ äquivalent zu $\mathcal{B}$? Dazu müssen wir zeigen, daß beide Automaten dieselben Wörter aus Σ^* akzeptieren. Es gilt $w \in L(\mathcal{A})$ genau dann, wenn $\hat{\delta}(q_0, w) \cap F \neq \emptyset$. Andererseits gilt $w \in L(\mathcal{B})$ genau dann, wenn $\Delta(\{q_0\}, w) \in F_d$, d.h. wenn $\Delta(\{q_0\}, w) \cap F \neq \emptyset$. Es genügt also für alle $w \in \Sigma^*$

$$\hat{\delta}(q_0, w) = \Delta(\{q_0\}, w)$$

zu zeigen. Das läßt sich durch Induktion über die Länge von w zeigen. Für $w \in \Sigma$ gilt

$$\begin{array}{rcl} \Delta(\{q_0\}, w) & = & Cl_\varepsilon(\delta(q_0, w)) \\ & = & \hat{\delta}(q_0, w) \end{array}$$

Der Induktionsschritt für $w \in \Sigma^*$ und $a \in \Sigma$ wird wie folgt bewiesen:

$$\begin{array}{rcll} \Delta(\{q_0\}, wa) & = & \Delta(\Delta(\{q_0\}, w), a) & \text{Def. von } \Delta \text{ auf } wa \in \Sigma^* \\ & = & Cl_\varepsilon(\bigcup_{q \in \Delta(q_0, w)} \delta(q, a)) & \text{Def. } \Delta \text{ für } a \in \Sigma \\ & = & Cl_\varepsilon(\bigcup_{q \in \hat{\delta}(q_0, w)} \delta(q, a)) & \text{Induktionsvoraussetzung} \\ & = & \hat{\delta}(q_0, wa) & \text{Def. von } \hat{\delta} \end{array}$$

Ein konstruktiver Beweis Der Nachteil der Konstruktion des äquivalenten deterministischen endlichen Automaten $\mathcal{B}$ ist die Tatsache, daß er immer exponentiell viele Zustände im Vergleich zu $\mathcal{A}$ enthält. Zustände von $\mathcal{B}$ sind aber nur dann notwendig, denn sie vom Anfangszustand $\{q_9\}$ erreichbar sind. Das folgende konstruktives Verfahren berücksichtigt nur solche Zustände. Ausgehend von einem endlichen Automaten $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$ legen wir eine Tabelle an mit einer Spalte für Teilmengen von Zuständen von Q und einer zusätzlichen Spalte für jeden Buchstaben $x \in \Sigma$. Siehe Beispiel 27. In der ersten Spalte der ersten Zeile steht der ε-Abschluß das Anfangszustands, in diesem Beispiel ist das $\{q_0\}$. Der Eintrag in die mit $x \in \Sigma$ markierten Spalten wird für jede Zeile wie folgt bestimmt. Ist Z die Menge von Zuständen, die in der ersten Spalte steht, dann bildet man zunächst die Menge Z_x aller Zustände, die von einem Zustand $q \in Z$ aus entlang eines mit x Übergangs erreicht werden können. In die entsprechende Tabellenposition wird jetzt der ε-Abschluß von Z_x eingetragen. Die beim Ausfüllen einer Zeile entstehenden Zustandsmengen werden in die erste Spalte eingetragen, sofern sie dort nicht schon vorgekommen sind. Das Verfahren endet, wenn alle Zeilen der Tabelle ausgefüllt sind und keine Teilmengen entstehen, die nicht schon in der ersten Spalte vorkommen. Da die Anzahl aller Teilmengen von Q endlich ist, ist die Terminierung dieses Verfahrens gesichert.

■

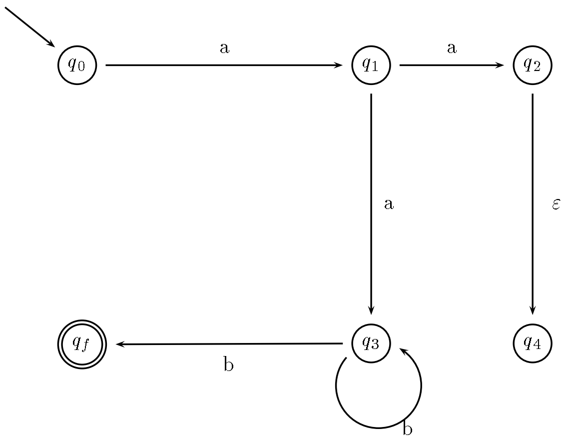


Abbildung 3.5: Elimination von Indeterminismus und ε-Übergängen

Beispiel 27

Gegeben sei der Automat aus Abb. 3.5.

Wir füllen der Reihe nach die Zeilen der folgenden Tabelle aus:

Zustandsmenge	a	b
$\{q_0\}$	$\{q_1\}$	$\emptyset$
$\{q_1\}$	$\{q_2, q_3, q_4\}$	$\emptyset$
$\{q_2, q_3, q_4\}$	$\emptyset$	$\{q_3, q_f\}$
$\{q_3, q_f\}$	$\emptyset$	$\{q_3, q_f\}$

Beispiel 28
Als ein zweites Beispiel zur Elimination von ε-Übergängen betrachten wir den endlichen Automaten aus Abb. 3.6.

neuer Zustand	*a*	*b*
$\{0,1\}$	$\{0,1,2\}$	$\{0,1\}$
$\{0,1,2\}$	$\{0,1,2,3\}$	$\{0,1,3\}$
$\{0,1,2,3\}$	$\{0,1,2,3,4\}$	$\{0,1,3,4\}$
$\{0,1,3\}$	$\{0,1,2,4\}$	$\{0,1,4\}$
$\{0,1,2,3,4\}$	$\{0,1,2,3,4\}$	$\{0,1,3,4\}$
$\{0,1,3,4\}$	$\{0,1,2,4\}$	$\{0,1,4\}$
$\{0,1,2,4\}$	$\{0,1,2,3\}$	$\{0,1,3\}$
$\{0,1,4\}$	$\{0,1,2\}$	$\{0,1\}$

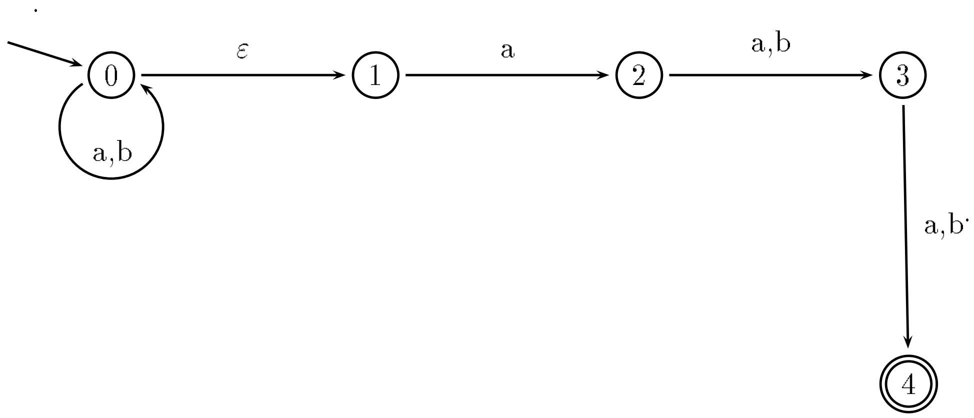


Abbildung 3.6: Elimination von ε-Übergängen

3.3 Reguläre Ausdrücke und endliche Automaten

Wir haben bisher zwei Möglichkeiten kennen gelernt reguläre Sprachen zu definieren, durch einen akzeptierenden endlichen Automaten oder durch eine rechts- oder links-lineare Grammatik. In diesem Abschnitt werden zwei weitere Möglichkeiten hinzukommen. Die erste Möglichkeit charakterisiert die Klasse aller regulären Sprachen durch eine induktive Definition; jede reguläre Sprache kann ausgehend von endlichen Wortmengen durch iterierte Anwendung geeigneter Abschlußoperationen erhalten werden. Die zweite Möglichkeit beschreibtjede reguläre Sprache explizit durch die Angabe eines regulären Ausdrucks.

Wir beginnen mit einer systematischen Übersicht über Operationen auf Sprachen, die auch in nachfolgenden Kapiteln eine Rolle spielen werden.

Definition 51 (Operationen auf Sprachen)
Für Sprachen $L, L_1, L_2 \subseteq \Sigma^*$ und Funktionen $g : \Sigma \to \mathcal{P}(\Gamma^*)$ definiert man die Operationen in Abbildung 3.7. Beachte: für jedes L gilt $\varepsilon \in L^*$.

Name	Symbol	Definition
Vereinigung	$L_1 \cup L_2$	$\{w \mid w \in L_1 \text{ oder } w \in L_2\}$
Verkettung	$L_1 L_2$	$\{w_1 w_2 \mid w_1 \in L_1 \text{ und } w_2 \in L_2\}$
Iteration	L^*	$\{w_1 \dots w_n \mid n \in \mathbb{N} \text{ und } w_i \in L \text{ für alle } i\}$
Spiegelung	L^r	$\{w^r \mid w \in L\}$ wobei: $(a_1 a_2 \dots a_{m-1} a_m)^r = a_m a_{m-1} \dots a_2 a_1$
Substitution	$g(L)$	$\bigcup_{w \in L} g(w)$ wobei: $g(a_1 \dots a_m) = \begin{cases} \varepsilon & m = 0 \\ g(a_1) \dots g(a_m) & m > 0 \end{cases}$

Abbildung 3.7: Operationen auf Sprachen

Beispiel 29 (Beispiel zur Substitution)

$$\begin{aligned}
\Sigma &= \{a, c\} \\
\Gamma &= \{0, 1\} \\
g(a) &= \{000, 111\} \\
g(c) &= \{101, 010\} \\
g(cac) &= g(c)g(a)g(c) \\
&= \{101000101, 101000010, 101111101, \\
&\quad 101111010, 010000101, 010000010, \\
&\quad 010111101, 010111010\}
\end{aligned}$$

Definition 52 (Reguläre Mengen)
Die Menge der regulären Teilmengen von Σ^* wird durch die folgende induktive Definition festgelegt:

1. *jede endliche Teilmenge von Σ^* ist eine reguläre Menge,*

2. *sind L_1, L_2 reguläre Mengen, dann auch $L_1 \cup L_2$,*

3. *sind L_1, L_2 reguläre Mengen, dann auch L_1L_2,*

4. *ist L eine reguläre Menge, dann auch L^*.*

Insbesondere ist $\emptyset$ eine reguläre Menge.

Definition 53 (Reguläre Ausdrücke)
Gegeben ist ein Alphabet Σ, $a \in \Sigma$. Die Menge der regulären Ausdrücke ist durch die folgende induktive Definition gegeben.

1. *$\emptyset$ ist ein regulärer Ausdruck.*

2. *a ist ein regulärer Ausdruck für jedes $a \in \Sigma$.*

3. *Sind R_1 und R_2 reguläre Ausdrücke, dann auch (R_1R_2),$(R_1 + R_2)$ und R_1^*.*

Ein regulärer Ausdruck R bezeichnet eine Menge $\mathcal{M}(R)$ von Wörtern. Der genaue Zusammenhang zwischen R und $\mathcal{M}(R)$ ist in der folgenden Tabelle zusammengefaßt.

regulärer Ausdruck R	*Interpretation* $\mathcal{M}(R)$
$\emptyset$	$\emptyset$
ε	$\{\varepsilon\}$
a	$\{a\}$
S^*	$\mathcal{M}(S)^*$
(ST)	$\mathcal{M}(S)\mathcal{M}(T)$
$(S+T)$	$\mathcal{M}(S) \cup \mathcal{M}(T)$

Beispiel 30

1. *$(a+b)^*aa(a+b)^*$ bezeichnet die Menge aller Wörter über den Alphabet $\Sigma = \{a, b\}$, die mindestens einmal zwei aufeinanderfolgende a's enthalten.*

2. *Für $R = '(c^{*\prime\prime}c^*)^{*\prime}$ ist $\mathcal{M}(R)$ genau die Menge Wörter, die der Automat $\mathcal{A}_{Scons}$ aus Abb. 3.2 akzeptiert.*

Eine Übersicht über die Äquivalenzbeweise zwischen den verschiedenen Charakterisierungen regulärer Sprachen ist in Abb. 3.8 zusammengestellt. Die Äquivalenz zwischen regulären Sprachen und den regulären Mengen aus Definition 52 ist hierin nicht aufgeführt, da sie nicht besonders tiefliegend ist.

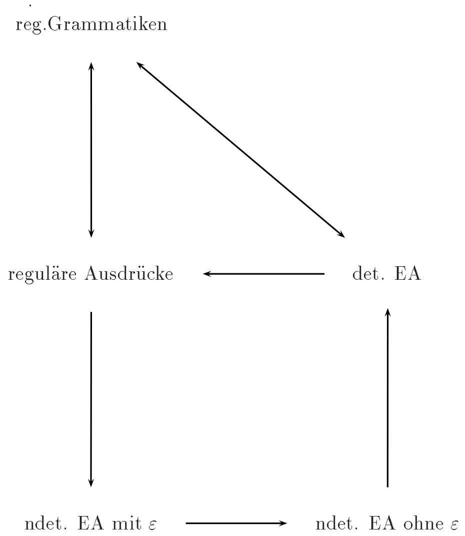


Abbildung 3.8: Übersicht über Äquivalenzbeweise

Lemma 35 *Die regulären Mengen sind genau die durch $\mathcal{M}(R)$ beschreibbaren Mengen für einen geeigneten regulären Ausdruck R.*

Beweis Ist R ein regulärer Ausdruck, dann liegt $\mathcal{M}(R)$ nach Definition 53 offensichtlich in der Klasse aller regulären Mengen. Da umgekehrt die Klasse aller $\mathcal{M}(R)$ unter den Abschlußoperationen für reguläre Mengen abgeschlossen ist, ist jedes $\mathcal{M}(R)$ eine reguläre Menge.

■

3.3.1 Von regulären Ausdrücken zu endlichen Automaten

In diesem Abschnitt werden wir uns von der Richtigkeit der folgenden Aussage überzeugen.

Theorem 36
Zu jedem regulären Ausdruck R gibt es einen endlichen Automaten $\mathcal{A}(R)$ mit

$$\mathcal{M}(R) = L(\mathcal{A}(R))$$

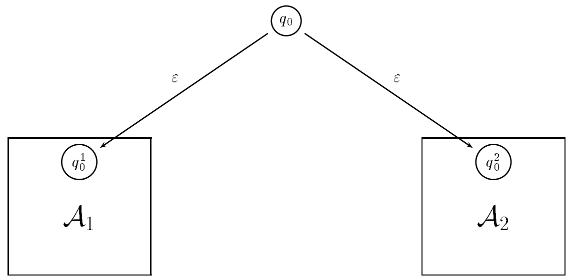
Zum Beweis von Satz 36 benötigen wir drei Konstruktionen die aus vorgegebenen endlichen Automaten neue Automaten kombinieren. Es folgen die präzisen Definitionen. Eine bildliche Darstellung ist in Abbildung 3.9 zu sehen.

Definition 54 (Kombinationen von Automaten)
Die Automaten $\mathcal{A}_i = (Q_i, \Sigma, q_0^i, F_i, \delta_i)$ für $i \in \{1,2\}$ mit $Q_1 \cap Q_2 = \emptyset$ und $\mathcal{B}$ seien gegeben.

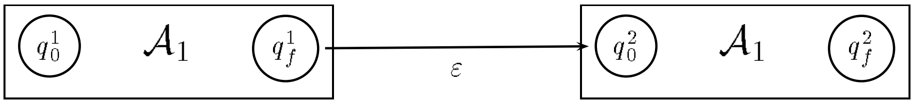
1. *Die* **disjunkte Summe** *der beiden Automaten $\mathcal{A} = \mathcal{A}_1 \uplus \mathcal{A}_2 = (Q, \Sigma, q_0, F, \delta)$ ist festgelegt durch:*
   - $Q = Q_1 \uplus Q_2 \uplus \{q_0\}$,
   - q_0 *ein neuer Zustand,*
   - $F = F_1 \uplus F_2$,
   - $\delta(q,a) = \begin{cases} \delta_1(q,a) & \textit{falls } q \in Q_1 \\ \delta_2(q,a) & \textit{falls } q \in Q_2 \\ \{q_0^1, q_0^2\} & \textit{falls } q = q_0 \textit{ und } a = \varepsilon \end{cases}$

2. *Die* **Verkettung** *$\mathcal{A} = \mathcal{A}_1\mathcal{A}_2 = (Q, \Sigma, q_0^1, F_2, \delta)$ der beiden Automaten $\mathcal{A}_1$ und $\mathcal{A}_2$ ist definiert durch:*
   - $Q = Q_1 \uplus Q_2$,
   - $\delta(q,a) = \begin{cases} \delta_1(q,a) & \textit{falls } q \in Q_1 \setminus F_1 \\ & \textit{od. } q \in Q_1 \ \&\ a \neq \varepsilon \\ \delta_1(q,\varepsilon) \cup \{q_0^2\} & \textit{falls } q \in F_1 \ \&\ a = \varepsilon \\ \delta_2(q,a) & \textit{falls } q \in Q_2 \end{cases}$

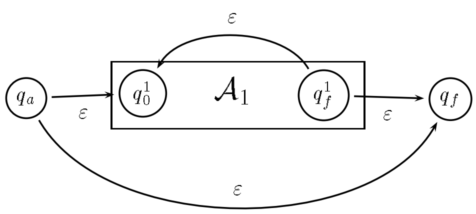
3. *Die Iteration $\mathcal{A} = \mathcal{B}^* = (Q \cup \{q_a, q_f\}, \Sigma, q_a, \{q_f\}, \delta)$ wird festgelegt durch:*
   - $\delta(q,a) = \begin{cases} \{q_0, q_f\} & \textit{falls } q = q_a, a = \varepsilon \\ \delta_B(q,\varepsilon) \cup \{q_0, q_f\} & \textit{falls } q \in F, a = \varepsilon \\ \delta_B(q,a) & \textit{sonst} \end{cases}$



(a) Summe



(b) Verkettung



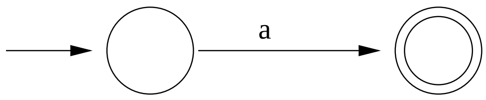
(c) Interation

Abbildung 3.9: Kombinationen von Automaten

Beweis von Theorem 36 Der endlichen Automaten $\mathcal{A}(R)$ wird durch Induktion über die Länge eines regulären Ausdrucks konstruiert. Entsprechend der Definition 53 sind dazu zwei Anfangsfälle und ein Induktionsschritt zu betrachten. Für die beiden Anfangsfälle kann man den zugehörigen Automaten direkt hinschreiben.

- $R = \emptyset$
  $\mathcal{A}(R)$ ist ein beliebiger Automat ohne Endzustände.

- $R = a$
  $\mathcal{A}(R)$:



Der Induktionsschritt gliedert sich wieder in drei Teile: Summenbildung, Verkettung und Iteration. Mit der Notation aus Definition 54 haben wir:

- $\mathcal{A}(R_1 + R_2) = \mathcal{A}(R_1) \uplus \mathcal{A}(R_1)$
- $\mathcal{A}(R_1 R_2) = \mathcal{A}(R_1)\ \mathcal{A}(R_1)$
- $\mathcal{A}(S^*) = \mathcal{A}(S)^*$

Damit ist die Erklärung der Konstruktion von $\mathcal{A}(R)$ abgeschlossen. Wir müssen uns nur noch überlegen, ob damit das gesteckte Ziel erreicht ist, d.h. wir müssen nachweisen, daß in allen drei Fällen $\mathcal{M}(R) = L(\mathcal{A}(R))$ gilt. Das geschieht natürlich wieder durch ein induktives Argument, wobei $\mathcal{M}(R_i) = L(\mathcal{A}(R_i))$ für $i = 1, 2$ und $\mathcal{M}(S) = L(\mathcal{A}(S))$ vorausgesetzt sind.

Es genügt demnach sich für beliebige Automaten $\mathcal{A}_i$ von der Richtigkeit der folgenden drei Behauptungen zu überzeugen:

1. $L(\mathcal{A}_1 \uplus \mathcal{A}_2) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$
2. $L(\mathcal{A}_1 \mathcal{A}_2) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$
3. $L(\mathcal{B}^*) = L(\mathcal{B})^*$

Die ersten beiden Teile sind einfach einzusehen.

Beweis von Teil (3) Die Inklusion $L(\mathcal{B}^*) \supseteq L(\mathcal{B})^*$ ist leicht einzusehen. Wir beschränken uns hier auf die kompliziertere Richtung. Sei $w = w_0 \dots w_m \in L(\mathcal{B}^*)$. Dann gibt es eine Folge von Zuständen

$$p_0 = q_a, p'_0, p_1, p'_1, \dots , p_m, p'_m = q_f,$$

so daß

- $p'_i \in Cl_\varepsilon(\{p_i\})$ und
- $p_{i+1} \in \delta_B(p'_i, w_i)$ gilt.

1.Fall: $p'_0 = q_f$

In diesem Fall finden nur spontane Übergänge statt, $w = \varepsilon$ und damit trivialerweise $w \in L(\mathcal{B})^*$.

2.Fall: sonst.

Da kein Übergang aus dem neuen Endzustand q_f herausführt, kommt q_f nur einmal vor. Außerdem kann q_f nur von einem Endzustand von $\mathcal{B}$ aus erreicht werden. Die einzige andere Möglichkeit führt zu Fall 1. An dieser Stelle ist es wichtig, daß einer neuer Anfangszustand hinzugenommen wurde.

Also gibt es ein $p^e_m \in F$ mit $p^e_m \in Cl_\varepsilon(\{p_m\})$. Falls alle ε-Übergänge von p_0 bis p^e_m in $\mathcal{B}$ möglich sind gilt $w \in L(\mathcal{B})$ und wir sind fertig. Anderenfalls listen wir alle Übergänge auf, die in $\mathcal{B}^*$ möglich sind, aber nicht in $\mathcal{B}$: $1 \leq i_1 < \dots i_r \leq m$ und $p^1_{i_j}, p^2_{i_j} \in Q$, so daß $p^1_{i_j} \in Cl^{\mathcal{B}}_\varepsilon(\{p_{i_j}\})$, $p'_{i_j} \in Cl^{\mathcal{B}}_\varepsilon(\{p^2_{i_j}\})$, aber der ε-Übergang von $p^1_{i_j}$ nach $p^2_{i_j}$ ist nur in $\mathcal{B}^*$ möglich. Nach Konstruktion von $\mathcal{B}^*$ muß für alle $1 \leq j \leq r$ gelten $p^1_{i_j} \in F$ und $p^2_{i_j} = q_0$. Setzt man $w_j = w_{i_{j-1}+1} \dots w_{i_j}$ für $1 < j \leq r$ und $w_1 = w_0 \dots w_{i_1}$, dann gilt $w = w_1 \dots w_r$ und $w_j \in L(\mathcal{B})$. Also $w \in L(\mathcal{B})^*$.

∎

Beispiel 31

In Abb. 3.10 sind die sukzessiven Stadien der Konstruktion eines endlichen Automaten zum regulären Ausdruck $01^* + 1$ gezeigt. $\mathcal{A}_1$ und $\mathcal{A}_2$ sind die Anfangsfälle mit $L(\mathcal{A}_1) = 0$ und $L(\mathcal{A}_2) = 1$. Der Automat $\mathcal{A}_3$ entsteht aus $\mathcal{A}_2$ durch die Iterationskonstruktion und es gilt $L(\mathcal{A}_3) = 1^*$. Bevor wir weitermachen eliminieren wir alle ε-Übergänge in $\mathcal{A}_3$ und vereinfachen das Ergebnis. Es entsteht der Automat $\mathcal{A}_4$, für den immer noch $L(\mathcal{A}_4) = 1^*$ gilt. $\mathcal{A}_5$ entsteht durch Hintereinanderschalten von $\mathcal{A}_1$ und $\mathcal{A}_4$ und $\mathcal{A}_6$ ist die disjunkte Summe von $\mathcal{A}_5$ und $\mathcal{A}_1$. Schließlich entsteht $\mathcal{A}_7$ aus $\mathcal{A}_6$ durch Elimination der ε-Übergänge und eine einfache Simplifikation.

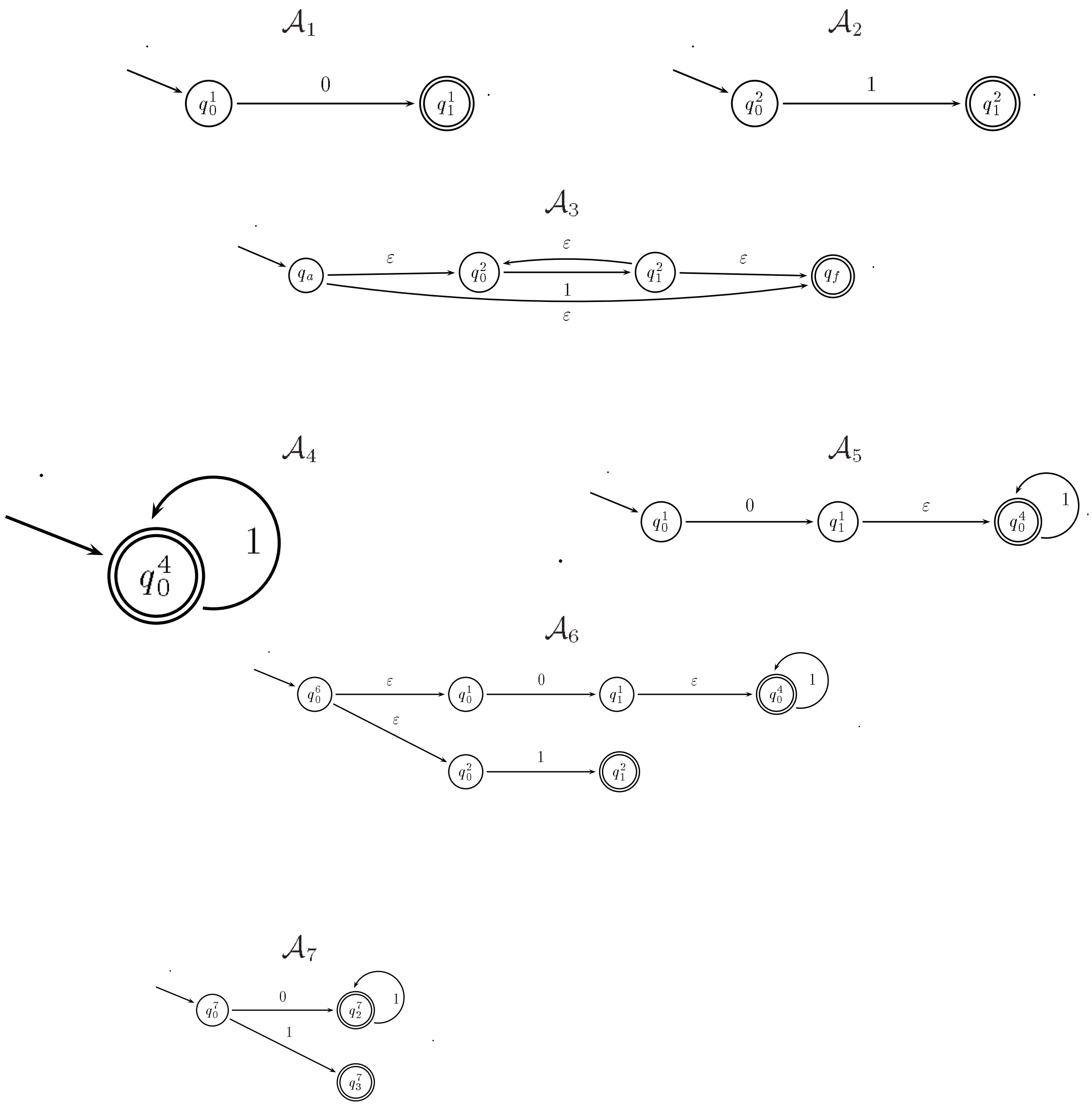


Abbildung 3.10: Konstruktion des endlichen Automaten zu $01^* + 1$

3.3.2 Von endlichen Automaten zu regulären Ausdrücken

In diesem Abschnitt geben wir einen konstruktiven Beweis von:

Theorem 37
Zu jedem endlichen Automaten $\mathcal{A}$ gibt es einen regulären Ausdruck R mit $\mathcal{M}(R) = L(\mathcal{A})$.

Beweis durch Angabe eines Konstruktionsverfahrens Gegeben sei der endliche Automat $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$ mit der Zustandsmenge $Q = \{q_0, q_1, \dots, q_n\}$. In einem ersten Schritt reduzieren wir das Problem. Dazu definieren wir

$$R_{i,j} = \{w \in \Sigma^* \mid \delta(q_i, w) = q_j\}$$

Offensichtlich gilt:

$$L(\mathcal{A}) = \bigcup_{j \in I_F} R_{0,j}$$

wobei $I_F = \{0 \leq j \leq n \mid q_j \in F\}$. Wenn wir reguläre Ausdrücke r_j für $R_{0,j}$ finden, dann ist

$$r_{j_1} + \ldots + r_{j_m}$$

für $I_F = \{j_1 \ldots j_m\}$ ein regulärer Ausdruck für $L(\mathcal{A})$.

Um R_{0,j_i} zu berechnen, brauchen wir als Zwischenresultate auch andere $R_{i,j}$. Wir geben daher ein Verfahren zur Berechnung von $R_{i,j}$ für beliebige i und j an. Dazu brauchen wir die Hilfsdefinition:

$$R^k_{i,j} = \{w \in \Sigma^* \mid \begin{array}{l} \delta(q_i, w) = q_j \text{ und für keine echtes An-} \\ \text{fangsstück } w' \text{ von } w \text{ gilt } \delta(q_i, w') = q_l \\ \text{für } l \geq k\} \end{array}$$

Es gilt $R^{n+1}_{i,j} = R_{i,j}$. Wir finden durch Induktion nach k reguläre Ausdrücke für $R^k_{i,j}$.

$$\begin{array}{lcl} R^0_{i,j} & = & \{s \in \Sigma \mid \delta(q_i, s) = q_j\} \text{ falls } i \neq j \\ R^0_{i,i} & = & \{s \in \Sigma \mid \delta(q_i, s) = q_i\} \cup \{\varepsilon\} \\ R^{k+1}_{i,j} & = & R^k_{i,j} \cup R^k_{i,k}(R^k_{k,k})^* R^k_{k,j} \end{array}$$

∎

Beispiel 32 *(siehe Abbildung 3.11)*

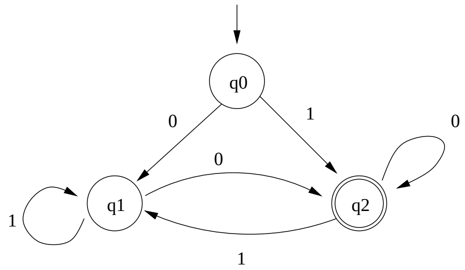


Abbildung 3.11: Beispielautomat

$$
\begin{array}{lcl}
L(\mathcal{A}) & = & R_{0,2} \\
 & = & R^3_{0,2} \\
 & = & R^2_{0,2} \cup R^2_{0,2}(R^2_{2,2})^* R^2_{2,2} \\
R^2_{0,2} & = & R^1_{0,2} \cup R^1_{0,1}(R^1_{1,1})^* R^1_{1,2} \\
R^2_{2,2} & = & R^1_{2,2} \cup R^1_{2,1}(R^1_{1,1})^* R^1_{1,2}
\end{array}
$$

$R^1_{0,2}$	$R^1_{0,1}$	$R^1_{1,1}$	$R^1_{1,2}$	$R^1_{2,2}$	$R^1_{2,1}$
1	*0*	*1*	*0*	*0*	*1*

$$
\begin{array}{lcl}
r^2_{0,2} & = & 1 + 01^*0 \\
r^2_{2,2} & = & 0 + 11^*0 \\
r & = & 1 + 01^*0 + \\
 & & (1 + 01^*0)(0 + 11^*0)^*(0 + 11^*0)
\end{array}
$$

3.3.3 Reguläre Ausdrücke im GNU Emacs Editor

In diesem Abschnitt wird ein Anwendungsbeispiel für reguläre Ausdrücke betrachtet. Dadurch soll deutlich werden, daß der Begriff der regulären Ausdrücke nicht nur ein theoretisches Konzept ist, sondern seinen festen Platz in der programmiersprachlichen Praxis hat.

Während einer Editiersitzung kann man das Kommando `C-M-s` eingeben, worauf in der Kommandozeile `Regexp I-search:` erscheint. Hinter dem Doppelpunkt kann man jetzt einen regulären Ausdruck t in der emacs-Syntax eingeben. Der Kursor springt dann unmittelbar auf die erste, von der aktuellen Kursorposition aus gesehen, Zeichenkette im Textpuffer die in $\mathcal{M}(t)$ liegt. Das Kommando `C-s` liefert das nächste Vorkommen einer Zeichenkette in $\mathcal{M}(t)$. Das Alphabet V_{emacs} besteht aus den 256 Zeichen des erweiterten ASCII Zeichensatzes. Auch Zeichen, die man nicht auf dem Bildschirm sieht, z.B. das ASCII Zeichen LF mit dem Oktalcode 012, können in einem regulären Ausdruck benutzt werden.

Ein erster Versuch

Aufgabe:	Wir suchen nach Zeichenketten, die mit einem x beginnen auf welches eine geraden Anzahl von y's folgt.
1. Versuch:	$x(yy)^*$ Die Klammern sind notwendig zur Unterscheidung von $(xyy)^*$ und $xy(y)^*$.
Problem:	Die Klammern (und) sind Zeichen in V_{emacs}. `Regexp I-search:` $x(yy)^*$ findet alle Ausdrücke der Form $x(yy)\ldots)$ mit einer beliebigen Anzahl schließender Klammern.

Sonderzeichen

Dieses Problem wird gelöst, wie an vielen anderen Stellen in der Informatik auch, in dem man V_{emacs} unterteilt in Sonderzeichen und normale Zeichen. An dieser Stelle geht die Einfachheit des theoretischen Konzepts verloren unter vielfachen Ausnahme- und Sonderregeln, Kompatibilitätsrücksichten und historischen Gewohnheiten. In der emacs Syntax regulärer Ausdrücke sind zunächst einmal die folgenden neun Zeichen Sonderzeichen:

`$`,`^`,`.`,`*`,`+`,`?`,`[`,`]`,`\`

Das Sonderzeichen \

Das Sonderzeichen `\` hat zwei Funktionen: Erstens dient es dazu die Sonderzeichen zu normalen Zeichen zu machen, d.h. für den reguläre Ausdruck `\$` gilt $\mathcal{M}(\texttt{\\\$}) = \{\texttt{\$}\}$ und konsequenterweise auch $\mathcal{M}(\texttt{\\\\}) = \{\texttt{\\}\}$. Zweitens macht `\` aus einigen normalen Zeichen Sonderzeichen. Hier ist eine Liste einiger auf diese Weise gebildeter Sonderzeichen:

`\|`, `\(`, `\)`, `'`, `` ` ``, `\b`, `\B`, `\<`, `\>`, `\w`, `\W`,`\scode`,`\Scode`
und `\z` für jede Ziffer `z`.

Semantik einiger Sonderzeichen

Für eine vollständige Beschreibung siehe:

Richard Stallman
Free Software Foundation
675 Mass Ave, Cambridge, MA 02139
GNU Emacs Manual, Sixth Edition, Emacs Version 18
1988, Abschnitt 13.5.

$$
\begin{array}{lcl}
\mathcal{M}(t_1\texttt{\textbackslash|}t_2) & = & \mathcal{M}(t_1) \cup \mathcal{M}(t_2) \\
\texttt{\textbackslash(}, \texttt{\textbackslash)} & & \text{Klammerung} \\
 & & \text{Beispiel } x\texttt{\textbackslash(}yy\texttt{\textbackslash)}* \\
* & & \text{wie in unserer Definition} \\
t_1 t_2 & & \text{wie in unserer Definition}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{M}(\texttt{.}) & = & \text{Menge aller Zeichen} \neq LF. \\
\mathcal{M}(t\texttt{+}) & = & \mathcal{M}(t(t)^*) \\
\mathcal{M}(t\texttt{?}) & = & \{\varepsilon\} \cup \mathcal{M}(t) \\
\mathcal{M}(\texttt{\^{}}t) & = & \{w \in \mathcal{M}(t) \mid w \text{ kommt am} \\
 & & \text{Anfang einer Zeile vor } \} \\
\mathcal{M}(t\$) & = & \{w \in \mathcal{M}(t) \mid w \text{ kommt am} \\
 & & \text{Ende einer Zeile vor } \} \\
\mathcal{M}(\texttt{[}a_1 \ldots a_k\texttt{]}) & = & \{a_1, \ldots, a_k\} \\
 & & \text{falls die } a_i \neq \text{Sonderzeichen} \\
\mathcal{M}(\texttt{[}a - z\texttt{]}) & = & \text{Menge aller Kleinbuchstaben} \\
\mathcal{M}(\texttt{[\^{}}a_1 \ldots a_k\texttt{]}) & = & \{z \in V_{emacs} \mid z \neq a_1 \text{ und} \\
 & & z \neq a_2 \ldots \text{ und } z \neq a_k\}
\end{array}
$$

. kann man auch in unserer Syntax hinschreiben als Summe mit 255 Summanden.

Eine weitere Sonderzeichenregelung

Innerhalb der `[...]` Paare gelten andere Sonderzeichen. Hier sind nämlich

`]`, `^`, `-`

die einzigen Sonderzeichen. Zur Aufhebung des Sonderzeichenstatus schreibt man `---` für `-`. `]` ist kein Sonderzeichen, wenn es unmittelbar nach `[` steht und `^` ist nur dann ein Sonderzeichen, wenn es unmittelbar nach `[` steht.

Reguläre Ausdrücke im Emacs Editor sind nur ein Beispiel. Sie treten in vielen anderen Programmiersprachen ebenfalls auf. Als ein weiteres Beispiel nennen wir die Skriptsprache Perl, siehe z.B. [34][Chapter 7].

3.3.4 Suche nach regulären Ausdrücken

Wir betrachten in diesem Abschnitt das folgende Problem. Gegeben sei eine reguläre Menge $M \subseteq \Sigma^*$ durch einen regulären Ausdruck R, i.e. $M = \mathcal{M}(R)$ und ein Texte T. Wie schreibt man ein Programm, das nacheinander alle Wörter $w \in M$ findet, die in T vorkommen. Fassen wir den regulären Ausdruck R als ein Muster auf, so können wir die Aufgabe umschreiben, als die Suche nach einem Teil von T, der zu dem Muster R paßt. Bevor wir uns dieser Aufgabe zuwenden betrachten wir das einfachere Problem, in dem der Text T zunächst noch nicht vorkommt. Wie schreibt man ein Programm, das als Eingabe ein Wort $w \in \Sigma^*$ erhält und die Ausgabe $Z = 1$ liefert, wenn $w \in M$ gilt und sonst $Z = 0$. Wie man zu R einen nichtdeterministischen endlichen Automaten $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$ konstruiert, der genau M akzeptiert haben wir in Theorem 36 gesehen. Wir kommt man nun von $\mathcal{A}$ zu einem Programm. Wir benutzen zur Beschreibung der Programme eine Pseudo-Programmiersprache, die sich eng an die *while*-Sprache aus dem ersten Kapitel anlehnt. In Abb. 3.12 ist das Hauptprogramm zu sehen, die dabei benötigten Makros sind in Abbildungen 3.13 zu sehen. Es handelt sich dabei um den Standardalgorithmus für diese Aufgabe, siehe z.B. [38]. Die Diskussion effizienterer Varianten des Standardalgorithmus wird in [7] geführt. Weiter Informationen zur Suche nach Wörtern einer regulären Menge in einem Text findet man in [8]. Um die Korrektheit des Algorithmus „RA-Test“ einzuse-

RA-Test(R,w)
Eingabe: regulärer Ausdruck R
Eingabewort w
Ausgabe: Z (Z = 1 für $w \in \mathcal{M}(R)$, sonst Z = 0)

Konstruiere NEA $\mathcal{A} = (Q, q_0, \Sigma, F, \delta)$ zu R;

W := w; C := *closure*($\{q_0\}, \delta$);
while W **loop**
A := **head** W; W := **tail** W;
B := *transition*(C,A,δ); C := *closure*(B,δ)
end;
if F $\cap$ C $\neq \emptyset$ **then** Z:=1 **else** Z:=0

Abbildung 3.12: Programm zur Akzeption einer regulären Menge

hen überlegt man sich, daß für ein Eingabewort $w = a_1 \dots a_n$ und für die Übergangsfunktion δ des Automaten $\mathcal{A}$ zu R und jedes i mit $1 \leq i \leq n$ gilt

$$C_i = \delta(q_0, a_1 \dots a_i)$$

wobei C_i der Wert der Programmvariablen C nach dem i-tem Schleifendurchlauf ist.

Jetzt kehren wir zurück zu der Problemstellung, die wir zu Beginn dieses Abschnitts eingeführt hatten. Durch einen simplen Trick können wir die Suche nach einem Muster R in einem Text $T \subseteq \Sigma^*$ zurückführen auf die soeben gelöste Aufgabe, indem wir anstelle von R den regulären Ausdruck $\Sigma^* R$ betrachten. Außerdem brauchen wir noch eine zweite kleine Änderung. Der

```
transitionen(X,A,δ)
Eingabe:     X ⊆ Q
             A ∈ Σ
             Übergangsfunktion δ
Ausgabe:     Z⊆ Q

   Z := ∅;
   while X loop
       P := head X;
       X := tail X;
       Z := append Z δ(P, A);
   end
```

```
closure(X,δ)
Eingabe:     X ⊆ Q
             Übergangsfunktion δ
Ausgabe:     Z⊆ Q

   Z := X;
   D := X;
   while D loop
       P := head D;
       D := tail D;
       for each R∈ δ(P, ε) loop
           if R∉ Z then
           Z := append Z {R};
           D := append D {R}
       end;
   end
```

Abbildung 3.13: Hilfprogramme für RA-Test

Algorithmus soll nicht erst den ganzen Text T einlesen, sondern sobald eine Anfangsstück T_0 von T zu dem Muster $\Sigma^* R$ paßt anhalten und die Länge Z von T_0 ausgeben. Das erreicht man offensichtlich, indem man den Test

$$\mathrm{F} \cap \mathrm{C} \neq \emptyset$$

nach jedem Schleifendurchlauf ausführt und im positiven Fall die Programmausführung beendet. Das Textstück T_0 ist von der Form uw mit $w \in \mathcal{M}(R)$ und u ein beliebiges Wort aus Σ^*. Somit beschreibt Z die Position eines Wortes in T vom Muster R. Genauer gibt Z die Position des letzten Buchstabens von w in T an. Der nichtdeterministische endliche Automat $\mathcal{A}_1$ zu $\Sigma^* R$ läßt sich leicht aus dem Automaten für R konstruieren. Es zeigt sich aber, daß es nicht nötig ist diese Konstruktion explizit durchzuführen. Der Suchalgorithmus in Abb. 3.14 arbeitet wie der Test-Algorithmus in Abb. 3.12 mit dem Automaten $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$ für R. Daß $\mathcal{A}_1$ gebraucht wird anstelle von $\mathcal{A}$ schlägt sich nur an einer einzigen Stelle nieder, indem die Zuweisung

B := *transition*(C,A,δ);

ersetzt wird durch

B := *transition*(C,A,δ) ∪ $\{q_0\}$;

```
RA-Suche(R,T)
Eingabe:    regulärer Ausdruck R
            Text T ⊆ Σ*
Ausgabe:    Z  (Position des letzten Buchstabens des gefundenen Wortes in T oder Z = 0)
   Konstruiere NEA 𝒜 = (Q, q0, Σ, F, δ) zu R;
   L := Länge von T; Z := 0;
   C := closure({q0}, δ);
   while T loop
       A := head T; T := tail T;
       B := transition(C,A,δ) ∪ {q0}; C := closure(B,δ);
       if F ∩ C ≠ ∅ then Z = L - Länge von T;
            STOP;
       endif
   end;
```

Abbildung 3.14: Suchalgorithmus nach dem Muster R im Text T

3.4 Abschlußeigenschaften reguläre Mengen

Reguläre Sprachen sind neben den kontextfreien Sprachen, die wir im nächsten Kapitel genauer vorstellen werden, die am häufigsten in der Informatik vorkommenden Sprachen. Neben allen anderen Vorzügen tragen dazu auch die Abgeschlossenheitseigenschaften der Klasse der regulären Sprachen bei.

Theorem 38
Sind $L_1, L_2 \subset \Sigma^*$ reguläre Sprachen (siehe Definition 49) und $g : \Sigma \to \mathcal{P}(\Gamma^*)$ eine Abbildung, so daß für alle $s \in \Sigma$ auch $g(s)$ regulär ist, dann sind auch die folgenden Sprachen regulär:

1. $L_1 \cup L_2$,
2. L_1^*,
3. $L_1 L_2$,
4. $\Sigma^* \setminus L_1$,
5. $L_1 \cap L_2$,
6. L_1^r,
7. $g(L_1)$

Beweise: Fall 1–3: Reguläre Mengen besitzten nach Definition diese drei Abschlusseigenschaften. Wegen der Äquivalenz zwischen regulären Mengen und regulären Sprachen (Sätze 35, 36 und 36) folgen die behaupteten Abgeschlossenheitseigenschaften.

Fall 4: Sei $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$ ein vollständiger deterministischer endlicher Automat mit $L(\mathcal{A}) = L_1$. Setzt man $\mathcal{A}^c = (Q, \Sigma, q_0, (Q \setminus F), \delta)$, dann gilt offensichtlich $L(\mathcal{A}^c) = \Sigma^* \setminus L_1$.

Fall 5: Bezeichnen wir mit L^c die Komplementärmenge $\Sigma^* \setminus L$ von L, dann gilt die mengentheoretische Gleichung $L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$. Aus der Abgeschlossenheit unter $\cup$ und Komplement folgt also unmittelbar die Abgeschlossen heit unter $\cap$.

Fall 6: $\mathcal{A}$ wie oben mit $L(\mathcal{A}) = L_1$. Setze $\mathcal{A}^r = (Q \cup \{q^0\}, \Sigma, q^0, \{q_0\}, \delta^r)$ mit

$$\begin{array}{l} \delta^r(q^0, \varepsilon) = F \\ \delta^r(q, a) = \{p\} \quad \Leftrightarrow \quad \delta(p, a) = q \end{array}$$

Fall 7: Einsetzen regulärer Ausdrücke.

3.4.1 Direktes Produkt zweier Automaten

Die Abgeschlossenheit unter Durchschnitten wurde in dem vorangegangenen Beweis mit minimalem Aufwand gezeigt; durch Rückführung auf die Abgeschlossenheit unter Vereinigung und Komplementbildung. Es gibt jedoch auch einen direkten Weg die Abgeschlossenheit regulärer Sprachen unter Durchschnitten direkt zu zeigen. Die dazu erforderliche Konstruktion des direkten Produktes endlicher Automaten gehört zum Standardrepertoire der Automatentheorie, auf die wir auch nicht verzichten wollen.

Definition 55 (Direktes Produkt von Automaten)
Seien $\mathcal{A}_i = (Q_i, \Sigma, q_0^i, \delta_i, F_i)$ für $i = 1, 2$ endliche Automaten. Das direkte Produkt

$$\mathcal{A}_1 \otimes \mathcal{A}_2 = (Q, \Sigma, q_0, \delta, F)$$

von $\mathcal{A}_1$ und $\mathcal{A}_2$ ist definiert durch:

$$\begin{array}{lcl} Q & = & Q_1 \otimes Q_2 \\ & = & \{(q_1, q_2) \mid q_i \in Q_i\} \\ q_0 & = & (q_0^1, q_0^2) \\ \delta((q_1, q_2), a) & = & (\delta_1(q_1, a), \delta_2(q_2, a)) \\ F & = & F_1 \otimes F_2 \\ & = & \{(q_1, q_2) \mid q_i \in F_i\} \end{array}$$

Theorem 39

$$L(\mathcal{A}_1 \otimes \mathcal{A}_2) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$$

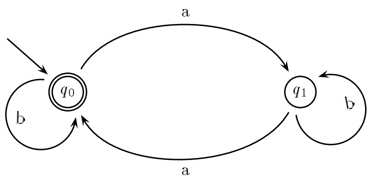
Beweis: Man zeigt leicht daß für jedes $w \in \Sigma^*$ gilt:

$$\delta(q_0, w) = (\delta_1(q_0^1, w), \delta_1(q_0^2, w))$$

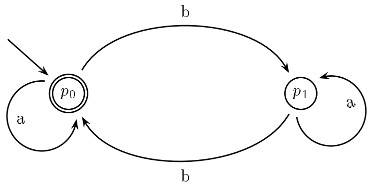
Somit

$$\begin{array}{rcl} w \in L(\mathcal{A}_1 \otimes \mathcal{A}_2) & \Leftrightarrow & \delta(q_0, w) \in F \\ & \Leftrightarrow & (\delta_1(q_0^1, w), \delta_1(q_0^2, w)) \in F \\ & \Leftrightarrow & \delta_1(q_0^1, w) \in F_1 \& \delta_1(q_0^2, w) \in F_2 \\ & \Leftrightarrow & w \in L(\mathcal{A}_1) \& w \in L(\mathcal{A}_2) \\ & \Leftrightarrow & w \in L(\mathcal{A}_1) \cap L(\mathcal{A}_2) \end{array}$$

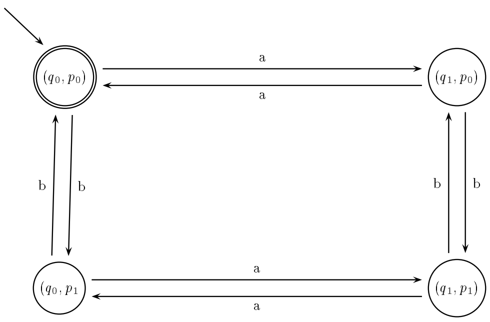
■



(a) $\mathcal{A}_{a-gerade}$



(b) $\mathcal{A}_{b-gerade}$

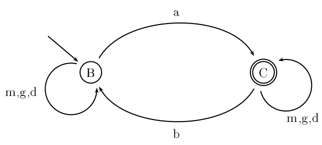


(c) Produktautomat $\mathcal{A}_{a-gerade} \otimes \mathcal{A}_{b-gerade}$

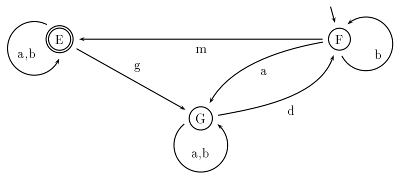
Abbildung 3.15: Automat für $\{w \in \{a, b\}^* \mid N_a(w)$ und $N_b(w)$ sind gerade $\}$

Beispiel 33
Die Abbildung 3.15 zeigt eine einfache aber typische Anwendung der Produktkonstruktion. Der Automat $\mathcal{A}_{a-gerade}$ akzeptiert genau die Wörter $w \in \{a, b\}^*$

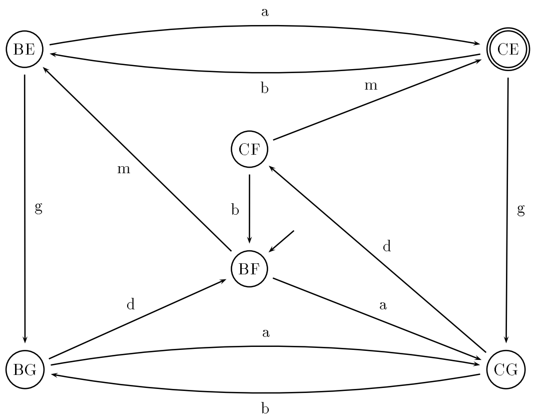
mit einer geraden Anzahl von a's, $\mathcal{A}_{b-gerade}$ akzeptiert alle $w \in \{a,b\}^*$ mit einer geraden Anzahl von b's und der Durchschnitt der beiden Sprachen, d.h. die Menge aller Wörter $w \in \{a,b\}^*$ mit einer geraden Anzahl von a's und mit einer geraden Anzahl von b's wird von den Produktautomaten $\mathcal{A}_{a-gerade} \otimes \mathcal{A}_{b-gerade}$ akzeptiert.



(a) Beispielautomat $\mathcal{A}_1$



(b) Beispielautomat $\mathcal{A}_2$



(c) Produktautomat aus $\mathcal{A}_1$ und $\mathcal{A}_2$

Abbildung 3.16: Beispiel eines Produktautomaten

Beispiel 34
In Abbildung 3.16 sind zwei Automaten $\mathcal{A}_1$ und $\mathcal{A}_2$ und ihr direktes Produkt $\mathcal{A}_1 \otimes \mathcal{A}_2$ zu sehen.

Das Beispiel ist eine Abwandlung eines Beispiels aus [19] zur konjunktiven Verknüpfung von "state charts".

3.5 Minimale endliche Automaten

Ziel dieses Abschnitts ist es einen Algorithmus vorzustellen, der zu jedem deterministischen endlichen Automaten einen äquivalenten Automaten mit minimaler Anzahl von Zuständen konstruiert. Der Nachweis, daß der Algorithmus tatsächlich den minimalen Automaten liefert, braucht als theoretisches Konzept den *Standardautomaten* zu einer vorgegeben Sprache.

Definition 56
Ein vollständiger deterministischer endlicher Automat A heißt minimal *für L, falls jeder andere Automat A_1 mit $L(A_1) = L(A)$ nicht weniger Zustände als A besitzt.*

Man kann diese Definition auch ohne Probleme für nicht vollständige Automaten benutzen. Da ein nicht vollständiger Automat durch die Hinzunahme eines *weiteren Zustandes zu einem vollständigen ergänzt werden kann, enthält ein minimaler nicht vollständiger Automat genau einen Zustand weniger als sein vollständiges Gegenstück.*

3.5.1 Standardautomaten

Gegeben sei eine reguläre Sprache L durch eine beliebige, z.B. natürlichsprachige Beschreibung. Um etwas Konkretes vor Augen zu haben kann man etwa an die Menge aller Pascal string Konstanten denken. Wir geben eine Methode an, wie man, zumindest im Prinzip, direkt aus L einen deterministischen endlichen Automaten $\mathcal{A}_L$ konstruieren kann, der L akzeptiert. Wir nennen $\mathcal{A}_L$ den *kanonischen Automaten* für L. Die Definition von $\mathcal{A}_L$ benötigt einige Vorbereitungen.

Definition 57 (Rechtskongruenz)
Eine Äquivalenzrelation R auf Σ^*, der Menge aller Wörter über einem Alphabet Σ, heißt eine Rechtskongruenz, *wenn für alle $w_1,w_2 \in \Sigma^*$ gilt*

$$\text{wenn } w_1Rw_2, \text{ dann auch } w_1uRw_2u \text{ für alle } u \in \Sigma^*$$

Für jedes Element $u \in \Sigma^*$ bezeichnet $[u]_R$ die Äquivalenzklasse von u:

$$[u]_R = \{w \mid R(u,w)\}$$

Die Anzahl der Äquivalenzklasse heißt der Index *von* R*,* $index(R)$.

Das wichtigste Beispiel einer Rechtskongruenz ist:

Definition 58 (Die Relation R_L)
Sei $L \subseteq \Sigma^*$ *eine Sprache.*

$$\begin{gathered} w_1 R_L w_2 \\ gdw \\ \textit{für alle } u \in \Sigma^* \textit{ gilt } w_1 u \in L \Leftrightarrow w_2 u \in L \end{gathered}$$

Zur Einübung in dem Umgang mit der Kongruenzrelation R_L und der Indexberechnung betrachten wir ein paar Beispiele:

Beispiel 35
Gegeben seinen die drei folgenden Sprachen:

$$\begin{array}{lcl} L_1 & = & \{w \in \{a,b\}^* \mid N_a(w) = 3\} \\ L_2 & = & \{w \in \{a,b\}^* \mid w \textit{ enthält nicht} \\ & & \textit{das Teilwort } ab\} \\ L_3 & = & \{w \in \{a,b\}^* \mid N_a(w) = N_b(w)\} \end{array}$$

Wir behaupten, daß sich die Indizes wie folgt berechnen:

$$\begin{array}{lcl} index(R_{L_1}) & = & 5 \\ index(R_{L_2}) & = & 3 \\ index(R_{L_3}) & = & \infty \end{array}$$

Die folgenden Hinweise sollten genügen um die Berechnungen nachvollziehen zu können.

1. $uR_{L_1}w \Leftrightarrow \begin{cases} \text{falls} & N_a(u) = N_a(w) \leq 3 \\ \text{oder} & N_a(u), N_a(w) > 3 \end{cases}$

2. Systematische Berechnung. Die Äquivalenzklassen sind $[\varepsilon]_{R_2}$, $[a]_{R_2}$, $[ab]_{R_2}$.

3. Die Wörter $\{a^n \mid n \geq 0\}$ sind paarweise nicht äquivalent bezüglich der Relation R_{L_3}. Für $n \neq m$ gilt $a^n b^n \in L_3$ aber $a^m b^n \notin L_3$

Definition 59 (Standardautomat)
Zu einer gegebenen Sprache L mit $index(R_L) < \infty$ konstruiert man den endlichen Automaten

$$\mathcal{A}_L = (Q, \Sigma, q_0, F, \delta)$$

wie folgt: $K_0, K_1, \ldots K_n$ sind alle Äquivalenzklassen von R_L, wobei $[\varepsilon]_{R_L} = K_0$ und $L = K_{j_1} \cup \ldots \cup K_{j_r}$.

$$\begin{array}{lcl@{\qquad}lcl} Q & = & \{K_i \mid 0 \leq i \leq n\} & \Sigma & = & \Sigma \\ q_0 & = & K_0 & \delta(K, a) & = & [wa]_{R_L} \text{ für } w \in K \\ F & = & \{K_{j_1}, \ldots, K_{j_r}\} & & & \end{array}$$

Lemma 40 (Myhill-Nerode)
Für jede Sprache L gilt:

1. *R_L ist eine Rechtskongruenz.*
2. *für jedes $u \in \Sigma^*$ gilt entweder $[u]_{R_L} \subseteq L$ oder $[u]_{R_L} \cap L = \emptyset$*
3. *$index(R_L) < \infty$ genau dann, wenn L regulär ist.*

Beweis:

1. Einfaches Nachrechnen.
2. Gelte $[u]_{R_L} \cap L \neq \emptyset$, etwa $w \in L$ und $w \in [u]_{R_L}$. Nach Definition von R_L gilt also für alle $v \in \Sigma^*$ die Äquivalenz $wv \in L \Leftrightarrow uv \in L$. Insbesondere folgt daraus für $v = \varepsilon$, daß $u \in L$ sein muß. Mit demselben Argument folgt für jedes weitere $w' \in [u]_{R_L}$ auch $w' \in L$.
3. Zum Beweis der Richtung von links nach rechts nehmen wir zunächst an, daß L regulär ist. Dann gibt es einen endlichen deterministischen Automaten $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$ mit $L(\mathcal{A}) = L$. Wir behaupten, daß aus $\delta(q_0, w_1) = \delta(q_0, w_2)$ folgt $w_1 R_L w_2$. Da $\mathcal{A}$ nur endlich viele Zustände hat, muß also auch der Index von R_L endlich sein. Zum Nachweis von $w_1 R_L w_2$ müssen wir für jedes $u \in \Sigma^*$ $w_1 u \in L \Leftrightarrow w_2 u \in L$ zeigen.

$$\begin{array}{lcl} w_1 u \in L & \Leftrightarrow & \delta(q_0, w_1 u) \in F \\ & \Leftrightarrow & \delta(\delta(q_0, w_1), u) \in F \\ & \Leftrightarrow & \delta(\delta(q_0, w_2), u) \in F \\ & \Leftrightarrow & \delta(q_0, w_2 u) \in F \\ & \Leftrightarrow & w_2 u \in L \end{array}$$

Nun zur umgekehrten Richtung. Hierbei wird $index(R_L) < \infty$ vorausgesetzt und damit ist der Standardautomat $\mathcal{A}_L$ nach Definition 59 ein endlicher Automat. Man zeigt durch Induktion über die Länge von w leicht für alle $w \in \Sigma^*$:

$$\delta(q_0, w) = [w]_{R_L}$$

und damit

$$\begin{array}{rcl} L(\mathcal{A}_L) & = & \{w \mid \delta(q_0, w) \in F\} \\ & = & K_{j_1} \cup \ldots \cup K_{j_r} \\ & = & L \end{array}$$

∎

Theorem 41 (Minimale Automaten) *Ist L eine reguläre Sprache, dann ist A_L ein minimaler endlicher Automat für L.*

Beweis: Sei $A_1 = (Q_1, \Sigma, q_0^1, F_1, \delta_1)$ ein beliebiger deterministischer endlicher Automat mit $L(A_1) = L$. Wir können o.B.d.A. annehmen, daß A_1 keine unerreichbaren Zustände besitzt. Für jedes $q \in Q_1$ wähle man ein $w \in \Sigma^*$ mit $\delta_1(q_0^1, w) = q$ und setze

$$\mu(q) = [w]_{R_L} \in Q_{\mathcal{A}_L}$$

Ist w' ein weiteres Wort mit $\delta_1(q_0^1, w') = q$, dann gilt

$$\begin{array}{llrcl} & \text{für alle } u \in \Sigma^* & \delta_1(q_0^1, wu) & = & \delta_1(q_0^1, w'u) \\ \Rightarrow & \text{für alle } u \in \Sigma^* & wu \in L & \leftrightarrow & w'u \in L \\ \Rightarrow & & & & wR_Lw' \\ \Rightarrow & & [w]_{R_L} & = & [w']_{R_L} \end{array}$$

Somit ist μ eine surjektive Abbildung von Q_1 auf die Menge aller Zustände von $\mathcal{A}_L$, d.h. A_1 hat nicht weniger Zustände als A_1.

∎

Der Beweis von Theorem 41 enthält noch eine weitere Information: Je zwei minimale vollständige endliche Automaten sind gleich. Wobei zwei Automaten gleich genannt werden, wenn sie im Sinner der folgenden Definition isomorph sind.

Definition 60 (Homomorphismen, Isomorphismen)
Seien $\mathcal{A}_1 = (Q_1, \Sigma, q_0^1, \delta_1, F_1)$, $\mathcal{A}_2 = (Q_2, \Sigma, q_0^2, \delta_2, F_2)$ zwei endliche deterministische Automaten über demselben Alphabet.

Eine Abbildung $\mu : Q_1 \to Q_2$ heißt eine Homomorphismus *von $\mathcal{A}_1$ nach $\mathcal{A}_2$ wenn gilt:*

1. $\mu(q_0^1) = q_0^2$

2. *für alle* $q \in Q_1$ *gilt* $q \in F_1 \Leftrightarrow \mu(q) \in F_2$

3. *für alle* $q \in Q_1$ *und alle* $a \in \Sigma$ *gilt* $\mu(\delta_1(q, a)) = \delta_2(\mu(q), a)$

Ist die Abbildung μ *surjektiv, so heißt* $\mathcal{A}_2$ *ein* homomorphes Bild *von* $\mathcal{A}_1$*.*

μ *heißt ein* Isomorphismus*, wenn* μ *ein Homomorphismus ist und zusätzlich bijektiv ist.*

Korollar 42 (Eindeutigkeit minimaler Automaten)
Je zwei vollständige minimale endliche Automaten für dieselbe reguläre Sprache L *sind isomorph.*

Beweis: Sei $\mathcal{A}_2$ ein beliebiger minimaler endlicher Automat mit $L(\mathcal{A}_2) = L$. Sei $\mu : \mathcal{A}_2 \to \mathcal{A}_L$ die im Beweis wie im Beweis von 41 definierte Abbildung.

1. μ ist ein Homomorphismus.

2. Da $\mathcal{A}_L$ und $\mathcal{A}_2$ beides minimale endliche Automaten sind, haben sie dieselbe endliche Anzahl von Zuständen. Dann ist die surjektive Abbildung μ aber schon bijektiv und damit ein Isomorphismus. Jeder minimale endliche Automat für L ist also isomorph zu $\mathcal{A}_L$.

■

3.5.2 Konstruktion minimaler Automaten

In diesem Abschnitt betrachten wir eine häufig auftretende Aufgabenstellung: ausgehend von einem gegebenen endlichen deterministischen Automaten $\mathcal{A}$ suchen wir den minimalen Automaten für $L(\mathcal{A})$. Rein theoretisch kann man aus $\mathcal{A}$ die Sprache $L = L(\mathcal{A})$ ablesen und nach Definition 59 den Standardautomaten zu L konstruieren. Das ist aber zu umständlich. Es gibt einen direkten Weg. Das Schlüsselkonzept zur Konstruktion sind Kongruenzrelationen auf Automaten.

Definition 61 (Kongruenzrelation auf Automaten)
Eine Kongruenzrelation $\approx$ *auf einem deterministischen endlichen Automaten* $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ *ist eine Äquivalenzrelation auf der Menge* Q *der Zustände des Automaten mit der zusätzlichen Eigenschaft:*

aus $q \approx p$ folgt für alle $a \in \Sigma$ auch $\delta(q,a) \approx \delta(p,a)$

Definition 62 (Reduktionskongruenz)
Das wichtigste Beispiel *einer Kongruenzrelation auf Automaten ist die Relation $\approx_{\mathcal{A}}$*

$$q \approx_{\mathcal{A}} p \quad \Leftrightarrow \quad \text{für alle } w \in \Sigma^* \text{ gilt } \delta(q,w) \in F \leftrightarrow \delta(p,w) \in F$$

Wir nennen $\approx_{\mathcal{A}}$ die Reduktionskongruenz *von $\mathcal{A}$.*

Definition 63 (Quotientenautomat)
Sei $\approx$ eine Kongruenzrelation auf dem deterministischen endlichen Automaten $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$, der Quotientenautomat $\mathcal{A}_\approx = (Q_\approx, \Sigma, q_0^\approx, \delta_\approx, F_\approx)$ ist wie folgt definiert:

$$\begin{array}{lcl} Q_\approx & = & \{[q]_\approx \mid q \in Q\} \\ \Sigma & = & \Sigma \\ q_0^\approx & = & [q_0]_\approx \\ \delta_\approx([q]_\approx, a) & = & [\delta(q,a)]_\approx \text{ für } a \in \Sigma \\ F_\approx & = & \{[q]_\approx \mid q \in F\} \end{array}$$

wobei $[q]_\approx = \{p \in Q \mid q \approx p\}$ die Kongruenzklasse von q ist.

Theorem 43
Sei $\mathcal{A}$ ein endlicher deterministischer Automat, $\approx$ eine Kongruenzrelation auf $\mathcal{A}$ und $\mathcal{A}_2 = \mathcal{A}_\approx$ der zugehörige Quotientenautomat. Dann gilt:

1. *$L(\mathcal{A}) = L(\mathcal{A}_2)$*
2. *$\mathcal{A}_2$ besitzt nie mehr Zustände als $\mathcal{A}$.*
3. *$\mathcal{A}_2 = \mathcal{A}$ gdw $\approx$ die Gleichheit ist.*

Beweis:

1. Aus der Definition des Quotientenautomaten folgt leicht, daß für alle $w \in \Sigma^*$ gilt $\delta_\approx([q_0]_\approx, w) = [\delta(q_0, w)]_\approx$. Somit gilt $w \in L(\mathcal{A})$ genau dann, wenn $\delta(q_0, w) \in F$. Was nachdem eben gesagten gleichbedeutend ist mit $\delta_\approx([q_0]_\approx, w) \in F_\approx$, d.h. $w \in L(\mathcal{A}_\approx)$
2. Offensichtlich, da jeder Zustand in $\mathcal{A}_2$ die Kongruenzklasse eines Zustands von $\mathcal{A}$ ist.

3. Die Gleichheit kann nur zutreffen, wenn jede Kongruenzklasse nur aus einem einzigen Element besteht.

■

Definition 64

1. *Ein Automat $\mathcal{A}$, dessen Reduktionsrelation $\approx_{\mathcal{A}}$ die Gleichheit ist, heißt* reduziert

2. *Ein Zustand $q \in Q$ eines endlicher deterministischer Automat $\mathcal{A}$ heißt* unerreichbar*, wenn es* kein *$w \in \Sigma^*$ gibt mit $\delta(q_0, w) = q$.*

Man sieht leicht, daß für jeden Automaten $\mathcal{A}$ der Quotientenautomat bezüglicher der Reduktionsrelation $\approx_{\mathcal{A}}$ eine reduzierter Automat ist.

Lemma 44
Sei $\mathcal{A}$ ein endlicher deterministischer Automat $\mathcal{A}_0$ der Automat, der aus $\mathcal{A}$ durch Entfernen aller unerreichbaren Zustände (und aller Übergänge, die von solchen Zuständen ausgehen oder in sie einmünden), entsteht. Dann gilt:

$$L(\mathcal{A}) \quad = \quad L(\mathcal{A}_0)$$

Beweis: Offensichtlich.

■

Theorem 45
Ein vollständiger deterministischer endlicher Automat $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ ist minimal für die Sprache $L = L(\mathcal{A})$ genau dann, wenn $\mathcal{A}$ reduziert ist und keine unerreichbaren Zustände aufweist.

Beweis:

Einfache Richtung Wenn $\mathcal{A}$ minimal ist, dann muß $\mathcal{A}$ reduziert sein. Anderenfalls hätte der Quotientenautomat von $\mathcal{A}$ bezüglich der Reduktionskongruenz weniger Zustände als $\mathcal{A}$. Ein Widerspruch zur Minimalität. Außerdem kann $\mathcal{A}$ keine unerreichbaren Zustände besitzen, da man diese ohne die akzeptierte Sprache zu ändern weglassen kann. Was wieder der Minimalität widerspricht.

Schwierige Richtung Wir schreiben L für $L(\mathcal{A})$. Es sei

$$\mathcal{B} = \mathcal{A}_{R_L} = (Q_B, \Sigma, q_0^B, \delta_B, F_B)$$

der Standardautomat für L (siehe Definition 59).

Homomorphismus von $\mathcal{A}$ nach $\mathcal{B}$ Die Abbildung $\mu : Q \to Q_B$ werde für $q \in Q$ definiert, indem zuerst ein Wort $w \in \Sigma^*$ gewählt wird mit $\delta(q_0, w) = q$. Ein solches Wort existiert, da $\mathcal{A}$ keine unerreichbaren Zustände enthält. Wir setzen:

$$\mu(q) = [w]_{R_L}$$

Die Unabhängigkeit dieser Definition von der Wahl des Wortes w wird durch die folgende Überlegung gezeigt. Ist w' ein weiteres Wort mit $\delta(q_0, w') = q$, dann gilt

$$\begin{array}{llrcl}
 & \text{für alle } u \in \Sigma^* & \delta(q_0, wu) & = & \delta(_0, w'u) \\
\Rightarrow & \text{für alle } u \in \Sigma^* & wu \in L & \leftrightarrow & w'u \in L \\
\Rightarrow & & & & wR_Lw' \\
\Rightarrow & & [w]_{R_L} & = & [w']_{R_L}
\end{array}$$

Man zeigt leicht, daß μ ein Homomorphmus ist.

Surjektivität von μ Für jedes Element $[w]_{R_L} \in Q_B$ gilt $\mu(q) = [w]_{R_L}$ für $q = \delta(q_0, w)$.

μ ist ein Isomorphismus Nur noch die Injektivität ist zu zeigen.

$$\begin{array}{lrcll}
 & \mu(q_1) & = & \mu(q_2) & \text{mit } q_i = \delta(q_0, w_i) \\
\Rightarrow & [w_1]_{R_L} & = & [w_2]_{R_L} & \text{Def. von } \mu \\
\Rightarrow & w_1u \in L & \text{gdw} & w_2u \in L & \text{für alle } u \in \Sigma^* \\
\Rightarrow & \delta(q_0, w_1u) \in F & \text{gdw} & \delta(q_0, w_2u) \in F & \text{für alle } u \in \Sigma^* \\
\Rightarrow & \delta(\delta(q_0, w_1), u) \in F & \text{gdw} & \delta(\delta(q_0, w_2), u) \in F & \text{für alle } u \in \Sigma^* \\
\Rightarrow & \delta(q_1, u) \in F & \text{gdw} & \delta(q_2, u) \in F & \text{für alle } u \in \Sigma^* \\
\Rightarrow & q_1 & \approx_{\mathcal{A}} & q_2 & \\
\Rightarrow & q_1 & = & q_2 & \text{Reduzierheit von } \mathcal{A}
\end{array}$$

∎

Wir wollen an einem Beispiel skizzieren, wie man aus der Charakterisierung minimaler Automaten in Satz 45 einen Algorithmus gewinnen kann. Die Grobidee dabei ist: zu einem gegebenen Automaten $\mathcal{A}$ berechnet man den Quotientenautomaten $\mathcal{B}$ bzgl. der Reduktionskongruenz $\approx_A$. In einem zweiten Schritt entfernt man mögliche nicht erreichbare Zustände in $\mathcal{B}$.

Die Berechnung von $\approx_{\mathcal{A}}$ geschieht zweckmäßigerweise durch die Berechnung des Komplements der Relation, d.h. durch die Berechnung von Paaren (p, q) von Zuständen, so daß $p \not\approx_{\mathcal{A}} q$ gilt. Am Anfang weiß man, daß alle Paare (p, q) mit $p \in F$ und $q \notin F$ im Komplement von $\approx_{\mathcal{A}}$ liegen müssen. Im Iterationsschritt betrachten wir ein Paar (p_1, q_1) von dem wir $p_1 \not\approx_{\mathcal{A}} q_1$ schon wissen. Gibt es für das Paar (p, q) ein $a \in \Sigma$ mit $\delta(p, a) = p_1$ und $\delta(q, a) = q_1$, dann kann auch $p \approx_{\mathcal{A}} q$ nicht gelten. Dieses Verfahren wird iteriert bis keine weiteren Paar zum Komplement von $\approx_{\mathcal{A}}$ mehr hinzugenommen werden können.

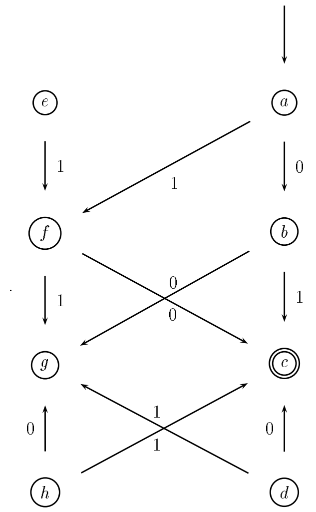


Abbildung 3.17: Beispielautomat

Wir führen dieses Verfahren für den Automaten aus Abbildung 3.17 vor.

Zunächst legen wir eine Tabelle aller Zustandspaare an.

b							
c							
d							
e							
f							
g							
h							
	a	b	c	d	e	f	g

Schritt für Schritt werden Zustandspaare mit einem Kreuz markiert, die nach $\approx_{\mathcal{A}}$ *nicht* äquivalent sind. Wir benutzen die Notation $\approx_i$ in der Bedeutung:

$p \approx_i q$ gilt genau dann, wenn in der i-ten Tabelle kein Kreuz an der Position (p, q) eingetragen ist.

$i = 0$ Für $i = 0$ sind das alle Paare (p, q), wobei p ein Endzustand und q keiner ist, oder umgekehrt.

	a	b	c	d	e	f	g
b							
c	X	X					
d			X				
e			X				
f			X				
g			X				
h			X				

$i = 1$ Es werden alle Felder in der Tabelle überprüft, die bisher nicht markiert sind. Dabei wird für alle $a \in \Sigma$ anhand der Tabelle für $i - 1$ geschaut, ob die erreichten Zustände $\delta(q, a)$ nach $\approx_{i-1} = \approx_0$ äquivalent sind. Ist dies für ein $a \in \Sigma$ nicht der Fall, sind die beiden Zustände nach $\approx_i$ nicht mehr äquivalent, daher wird an der entsprechenden Stelle in der Tabelle ein Kreuz gesetzt.

	a	b	c	d	e	f	g
b	X						
c	X	X					
d	X	X	X				
e		X	X	X			
f	X	X	X		X		
g		X	X	X		X	
h	X		X	X	X	X	X

$i = 2$ Das selbe wird für $i = 2$ auch gemacht. Damit erhält man folgende Tabelle:

	a	b	c	d	e	f	g
b	X						
c	X	X					
d	X	X	X				
e		X	X	X			
f	X	X	X		X		
g	X	X	X	X	X	X	
h	X		X	X	X	X	X

$i = 3, \ldots$ Für $i = 3, \ldots$ ändert sich nichts mehr an der Tabelle, also $\approx_2 = \approx_3 = \approx_{\mathcal{A}}$. Man erhält folgende kongruente Zustände:

$$e \approx_{\mathcal{A}} a, f \approx_{\mathcal{A}} d, h \approx_{\mathcal{A}} b$$

Es lassen sich jeweils die beiden äquivalenten Zustände zu einem neuen zusammenfassen. Damit erhält man folgenden reduzierten Automaten:

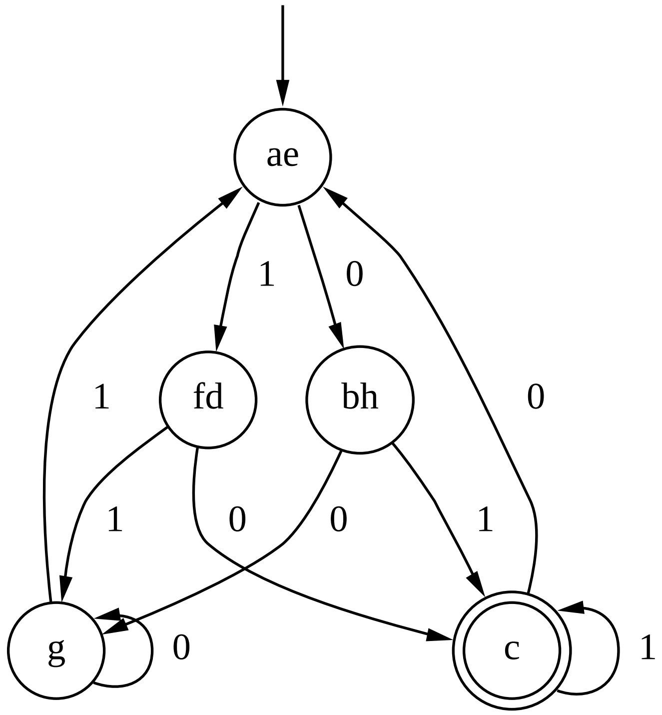


Abbildung 3.18: reduzierter Automat

Da der reduzierte Automat bereits keine unerreichbaren Zustände enthält, ist er auch der minimale Automat.

3.6 Grenzen der Ausdrucksstärke

Wir haben bisher eine Reihe von Beispielen für reguläre Sprachen gesehen, die aber aller recht einfach waren. Ist das ein Zufall oder eine Konsequenz didaktischer Überlegungen? In diesem Abschnitt geben wir einen ersten Hinweis auf die Grenzen der Ausdrucksstärke regulärer Sprachen. Die generelle Vorgehensweise ist einfach. Wir beweisen zunächst ein Lemma, das sogenannte *pumping lemma*, das aussagt, daß jede reguläre Sprache eine gewisse Periodizitätseigenschaft, die Aufblaseigenschaft (pumping property) hat. Treffen wir nun auf eine Sprache, die nicht die Aufblaseigenschaft besitzt, dann können wir daraus schließen, daß sie nicht regulär sein kann.

Die Schwierigkeit beim Verständnis und bei der Anwendung des *pumping lemmas* liegt darin, daß die Aufblaseigenschaft selbst kompliziert ist. Sie ist von der Form es gibt „ ... für alle ... es gibt ... für alle“(in Zeichen $\exists\forall\exists\forall$), weist also drei Quantorenwechsel auf, was dem menschlichen Verstand eine beträchtliche Konzentration abverlangt. Hier zunächst das Lemma. Wir benutzen dabei die Notation $|x|$ für die Länge des Wortes x. Andersgesagt ist $|x|$ die Anzahl der Buchstaben in x, wobei mehrfach vorkommende Buchstaben auch mehrfach gezählt werden.

Lemma 46 (Pumping Lemma)
Sei L eine reguläre Sprache. Dann gibt es ein $n \in \mathbb{N}$, so daß sich jedes Wort $x \in L$ mit $|x| \geq n$ zerlegen läßt in

$$x = uvw,$$

so daß die folgenden Eigenschaft erfüllt sind:

1. $|v| \geq 1$
2. $|uv| \leq n$
3. $uv^iw \in L$ *für alle* $i \in \mathbb{N}$.

Beweis: Sei $L \subseteq \Sigma^*$ eine reguläre Sprache. Es gibt also einen deterministischen endlichen Automaten $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$. Als n wählen wir $card(Q)+1$. Wir demonstrieren jetzt, daß für dieses n die im Lemma behauptete Zerlegungseigenschaft gilt. Dazu betrachten wir ein Wort $x \in L$ mit $|x| = r \geq n$. Nach Wahl von n heißt das $r > card(Q)$. Wir notieren alle Zustände, die der Automat $\mathcal{A}$ beim Einlesen von $x = x_1 \dots x_r$ durchläuft.

$$q^k = \delta(q_0, x_1 \dots x_k) \text{ für } 0 \leq k \leq r$$

Da r größer als die Anzahl aller möglichen Zustände ist muß ein Zustand zweimal vorkommen. Wir wählen das kleinste $k_1 < r$, so daß ein k_2 mit

$k_1 < k_2 \leq r$ existiert mit $q^{k_1} = q^{k_2}$. Die gesuchte Zelegung von x läßt sich nur angeben:

$$u = x_1 \ldots x_{k_1} \quad v = x_{k_1+1} \ldots x_{k_2} \quad w = x_{k_2+1} \ldots x_r$$

Da $k_1 < k_2$ gilt haben wir auf jeden Fall $|v| \geq 1$. Da k_1 und k_2 minimal gewählt waren gilt auch $|uv| \leq n$. Betrachten wir die Zustände, die der Automat $\mathcal{A}$ beim Einlesen eines Wortes uv^iw für ein $i \geq 0$ durchläuft, so ist nach dem vollständigen Einlesen jeder Kopie des Teilwortes v der Zustand q^{k_2} erreicht. Nach einlesen des gesamten Wortes wird derselbe Endzustand, q^r, erreicht, wie beim Einlesen von x. Man beachte, daß dieses Argument insbesondere für $i = 0$ funktioniert.

∎

Um die Anwendung des pumping Lemmas vorzubereiten formulieren wir seine Negation explizit:

Lemma 47
Gibt es für jedes $n \in \mathbb{N}$ eine Wort $x \in L$ mit $|x| \geq n$ und für jede Zerlegung

$$x = uvw,$$

mit $|v| \geq 1$ und $|uv| \leq n$, gibt es ein $i \in \mathbb{N}$ mit

$$uv^iw \notin L,$$

dann ist L nicht regulär.

Beweis: Negation von Lemma 46.

∎

Es folgen einige Anwendungsbeispiele.

Beispiel 36
Die Sprache $L_{mm} = \{0^m 1^m \mid m \geq 0\}$ ist nicht *regulär. Für jedes $n \in \mathbb{N}$ betrachten wir das Wort*

$$x = 0^{n+1}1^{n+1} \in L_{mm}$$

Ist eine Zerlegung $x = uvw$ gegeben mit $|v| \geq 1$ und $|uv| \leq n$, dann ist v von der Form $v = 0^i$ für $i > 0$. Somit liegt z.B. uv^2w nicht in L_{mm}.

Andererseits ist es einfach eine kontextfreie Grammatik für L_{mm} anzugeben. L_{mm} ist das Standardbeispiel für eine kontextfreie aber nicht-reguläre Sprache:

Beispiel 37
Die Sprache $L_{prim} = \{a^p \mid p \text{ ist eine Primzahl}\}$ *ist* nicht *regulär. Für jedes* $n \in \mathbb{N}$ *betrachten wir das Wort* $a^p \in L_{prim}$ *für die nächste Primzahl* $p \geq n+2$. *Sei eine beliebige Zerlegung* $a^p = uvw$ *gegeben mit* $|v| \geq 1$ *und* $|uv| \leq n$. *Also gilt* $|w| \geq 2$. *Wir müssen zeigen, daß für ein* $i \in \mathbb{N}$ uv^iw *nicht in* L_{prim} *liegt, d.h.* $|uw| + i \cdot |v|$ *keine Primzahl ist. Das ist aber für* $i = |uw|$ *der Fall*

$$|uw| + i \cdot |v| = |uw|\,(|v| + 1)$$

mit $|uw| \geq 2$ *und* $(|v| + 1) \geq 2$.

Beispiel 38 *Die Sprache* $L_{quad} = \{a^{n^2} \mid n \geq 0\}$ *ist* nicht *regulär. Für jedes* $n \in \mathbb{N}$ *betrachten wir das Wort* $a^{n^2} \in L_{quad}$ *und eine beliebige Zerlegung*

$$a^{n^2} = uvw$$

mit $|v| \geq 1$ *und* $|uv| \leq n$. *Wir behaupten, daß* uv^2w *nicht in* L_{quad} *liegen kann.*

$$\begin{array}{rcll} n^2 & < & |uv^2w| & \\ & = & |u| + 2|v| + |w| & \\ & = & n^2 + |v| & \\ & \leq & n^2 + n & \text{wegen } |v| \leq n \\ & < & (n+1)^2 & \end{array}$$

3.7 Endliche Automaten mit Ausgabe

In diesem Abschnitt führen wir die wichtige Klasse der endliche Automaten mit Ausgabe ein.

Es gibt zwei Typen von endlichen Automaten mit Ausgabe, die Mealy- und die Moore-Automaten.

Definition 65 (Mealy-Automaten)
Ein Mealy-Automat $A = (S, V_e, V_a, s_0, \delta, \lambda)$ *ist gegeben durch*

1. *eine endliche Menge* S *von Zuständen*
2. *einem Anfangszustand* s_0,
3. *einem Eingabealphabet* V_e,

4. *einer Übergangsfunktion* $\delta : S \times V_e \to S$

5. *einer Ausgabefunktion* $\lambda : S \times V_e \to V_a$

Definition 66 (Moore-Automaten)
Ein Moore-Automat $A = (S, V_e, V_a, s_0, \delta, \lambda)$ *ist wie ein Mealy-Automat mit dem einzigen Unterschied, daß die Ausgabefunktion nur vom Zustand abhängt, d.h.*

5. *einer Ausgabefunktion* $\lambda : S \to V_a$

Wie schon die Übergangsfunktion zu einer Funktion $\delta : S \times V_e^* \to S$ fortgesetzt werden konnte wird auch die Ausgabefunktion fortgesetzt zu einer Funktion $\lambda : S \times V_e^* \to V_a^*$.

Definition 67 (Erweiterte Ausgabefunktion)

$$\begin{array}{lcl} \lambda(s,\varepsilon) & = & \varepsilon \\ \lambda(s,wa) & = & \lambda(s,w)\lambda(\delta(s,w),a) \end{array}$$

Definition 68 (Übersetzungsfunktion)
Ist M *ein Mealy oder Moore Automat, so heißt die Funktion* $f_M : V_e^* \to V_a^*$, *definiert durch*

$$f_M(w) = \lambda(s_0, w)$$

die Übersetzungsfunktion *von* M. *Zwei Mealy oder Moore Automaten* M_1,M_2 *heißen* äquivalent, *falls* $f_{M_1} = f_{M_2}$.

In der Diagrammdarstellung eines Mealy-Automaten schreiben wir an eine Kante von Zustand q_i nach q_j nicht nur den Buchstaben a des Eingabealphabets, der als Vorbedingung für diese Transition vorliegen muß, sondern auch den Buchstaben A des Ausgabealphabets, der bei dieser Transition ausgegeben wird. Die beiden Buchstaben werden durch einen Schrägstrich voneinander getrennt, also in der a/A notiert.

Beispiel 39
In Abb. 3.19 ist ein Mealy-Automat dargestellt der Folgen von Zahlen einliestund Folgen von Buchstaben ausgibt. Für die Eingabe 11001 *ergibt sich die Ausgabe Hallo. Man beachte, daß keine eins-zu-eins Abbildung von Zahlen auf Buchstaben erfolgen kann. Je nachdem, in welchem internen Zustand eine* 1 *eingelesen wird, wird ein* H, *ein* o *oder ein* l *ausgegeben.*

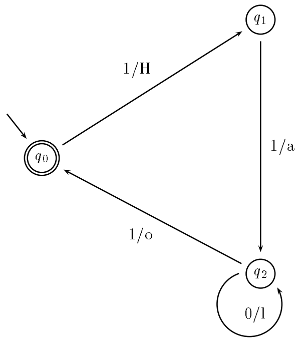


Abbildung 3.19: Beispiel eines Mealy-Automaten

3.8 Entscheidbare Probleme

In diesem Abschnitt stellen wir eine Liste entscheidbarer Probleme im Zusammenhang mit regulären Sprachen zusammen.

Im folgenden seien $L_1, L_2 \subseteq \Sigma^*$ reguläre Sprachen über einem festes Alphabet Σ und w ein Wort in Σ^*. Es spielt keine Rolle wie die Sprachen gegeben sind, als reguläre Ausdrücke, endliche Automaten oder lineare Grammatiken.

Theorem 48
Die folgenden Probleme sind entscheidbar:

1. $w \in L$,
2. $L = \emptyset$,
3. *L ist endlich,*
4. $L_1 = L_2$,
5. $L_1 \subseteq L_2$.

Beweise:

1. $w \in L$,
   Berechne in einem endlichen Automaten für L den Zustand $q = \delta(q_0, w)$ und stelle fest ob $q \in F$ gilt.

2. $L = \emptyset$,
   Anwendung des Algorithmus zur Bestimmung der Erreichbarkeit in Graphen: ist ein Endzustand von q_0 aus erreichbar?

3. L ist endlich,
   Sei n die Konstante aus dem pumping Lemma. L ist endlich genau dann, wenn jedes Wort in L eine Länge $< n$ hat.

4. $L_1 = L_2$,
   Berechne die minimalen endlichen Automaten $\mathcal{A}_1$, $\mathcal{A}_2$ für L_1,L_2. Wegen der Eindeutigkeit minimaler Automaten genügt es die Isomorphie von $\mathcal{A}_1$ mit $\mathcal{A}_2$ zu testen.

5. $L_1 \subseteq L_2$.
   $L_1 \subseteq L_2 \Leftrightarrow L_1 \cap L_2^c = \emptyset$

∎

3.9 Lehrbücher

1. An Introduction to Formal Language Theory [29]
   Bewährt gutes Lehrbuch des Autorentrios mit vielen Beispielen und Übungsaufgaben.

2. Kompendium Theoretische Informatik - eine Ideensammlung [41]
   Kein Lehrbuch, aber ein interessanter neuer Ansatz. Es wird versucht die wesentlichen Ideen und Motivationen zu vermitteln, ohne dabei allzu sehr in technische Details einzusteigen. Gedacht als Ergänzung zu Lehrbuch und Vorlesung. Enthält keine Aufgaben, aber Testfragen mit Antworten.

3. Introduction to Automata Theory [23]
   Umfangreiches Lehrbuch. Verständlich aber knapp geschrieben. Umfangreiche Übungsaufgaben mit Angabe des Schwierigkeitsgrades.

4. Formale Sprachen [33]
   Eine klassische Monographie zum Thema, allerdings ein bißchen in die Jahre gekommen und anspruchsvoll.

3.10 Übungsaufgaben

Übungsaufgabe 3.10.1

Zeigen Sie mit dem Pumpinglemma, daß die folgenden Sprachen nicht regulär sind:

(a) $L_1 = \{w \in \{0,1\}^* \mid w = x_1 \cdots x_n = w^r := x_n \cdots x_1, x_i \in \{0,1\}\}$

(b) $L_2 = \{0^k \mid \text{ ist keine Primzahl}\}$

(c) $L_3 = \{a^i b^j \mid 0 \leq i < j\}$

(d) $L_4 = \{w \in \{a,b\}^* \mid N_a(w) = N_b(w)\}$ wobei $N_x(w)$ die Anzahl Vorkommen von x in w ist.

Übungsaufgabe 3.10.2

Betrachten Sie die Sprache $L = \{1^i 0^j \mid i,j \geq 0\} \cup \{0^i 1^j 0^j \mid i,j \geq 0\}$.

(a) Zeigen Sie, daß L die Eigenschaften des Pumping-Lemmas besitzt.

(b) Zeigen Sie, daß L nicht regulär ist.

(c) Geben Sie eine kontextfreie Grammatik G mit $L(G) = L$ an.

Übungsaufgabe 3.10.3

Beweisen Sie die folgende Version des Pumping-Lemmas:

Lemma 49 *L sei eine reguläre Sprache. Dann gibt es ein $n \in \mathbb{N}$, so daß sich jedes Wort $x \in L$ mit $|x| \geq n$ in $x = uvw$ zerlegen läßt und die folgenden Eigenschaften erfüllt sind:*

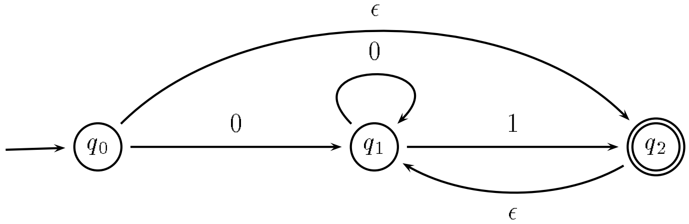
(a) $|v| \geq 1$,

(b) $|vw| \leq n$ *und*

(c) $uv^iw \in L$ *für alle* $i \in \mathbb{N}$.

Übungsaufgabe 3.10.4

Betrachten Sie folgenden endlichen Automaten $\mathcal{A}$:



(a) Geben sie die ϵ-Hüllen der Zustände q_0, q_1 und q_2 von $\mathcal{A}$ an.

(b) Geben Sie einen ϵ-freien endlichen Automaten $\mathcal{B}$ mit $L(\mathcal{B}) = L(\mathcal{A})$ an. Zeichnen Sie diesen Automaten.

Übungsaufgabe 3.10.5

(a) Geben Sie einen einen vollständigen deterministischen Automaten $\mathcal{C}$ an, der die Sprache

$$L := \{w \in \{0,1\}^* \mid N_0(w) \text{ ist gerade und } N_1(w) \text{ ist ungerade}\}$$

akzeptiert.

Dabei bezeichnet $N_x(w)$ die Anzahl Vorkommen von x in w.

(b) Erläutern Sie kurz wie Ihr Automat $\mathcal{C}$ geändert werden muß, damit er die Sprache

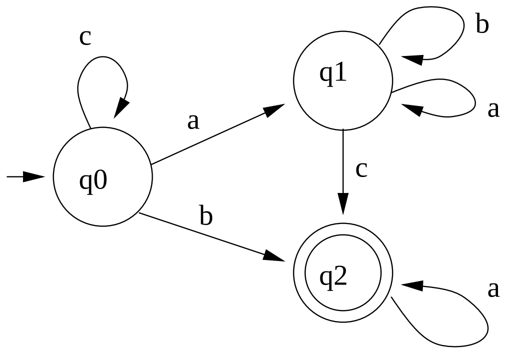
$$\{w \in \{0,1\}^* \mid N_0(w) \text{ ist gerade } \textit{oder } N_1(w) \text{ ist ungerade}\}$$

erkennt?

(c) Wie groß ist der Index von L?

Übungsaufgabe 3.10.6

Konstruieren Sie aus folgendem det. endl. Automaten schrittweise einen regulären Ausdruck:



Übungsaufgabe 3.10.7

Berechnen Sie zur Sprache

$$L = \{w \in \{0,1\}^* \mid 101 \text{ ist kein Teilwort von } w\}$$

den minimalen vollständigen deterministischen Automaten.

Übungsaufgabe 3.10.8

Gegeben Sie einen Mealy-Automaten an mit dem Eingabealphabet $\{a, b\}$ und dem Ausgabealphabet $\{a, b, c\}$, der alle Wörter mit höchstens zwei aufeinanderfolgenden a's akzeptiert. Für jedes akzeptierte Wort w gibt der Automat das Wort w' aus, das aus w entsteht, indem alle Vorkommen von aa durch ac ersetzt werden.

Übungsaufgabe 3.10.9

Betrachten Sie die reguläre Sprache

$$L = \{w \in \{0,1\}^* \mid w \text{ enthält } 00 \text{ als Teilwort}\} \ .$$

(a) Geben Sie einen zu L äquivalenten regulären Ausdruck an.

(b) Geben Sie alle Äquivalenzklassen von L bezüglich der Relation R_L an. Begründen Sie Ihre Antwort (z.B. durch Herleitung).

(c) Geben Sie einen vollständigen minimalen deterministischen endlichen Automaten an, der L akzeptiert.

(d) Wie groß ist der Index von L? Begründen Sie Ihre Antwort.

Übungsaufgabe 3.10.10

(a) Geben Sie einen deterministischen endlichen Automaten $\mathcal{A}$ an, der die durch den regulären Ausdruck $a(a+b)^n a$ definierte Sprache akzeptiert.

(b) Q sei die Menge der Zustände, $F \subset Q$ die Menge der Endzustände, $q_0 \in Q$ der Anfangszustand und δ die Zustandsübergangsfunktion des Automaten $\mathcal{A}$. Betrachten sie den nicht ϵ-freien endlichen Automaten $\mathcal{A}'$, der wie folgt aus $\mathcal{A}$ konstruiert wird: $Q' = Q \cup \{q_{-1}\}$, $F' = F$, q_{-1} ist statt q_0 Anfangszustand, $\delta' = \delta$ und zusätzlich gilt $\delta'(q_{-1}, a) = \delta'(q_{-1}, b) = q_{-1}$ und $\delta'(q_{-1}, \epsilon) = q_0$, d.h. $\mathcal{A}'$ akzeptiert die durch $(a+b)^* a(a+b)^n a$ definierte Sprache. Konstruieren Sie für $n = 2$ mit dem Teilmengenverfahren aus $\mathcal{A}'$ einen äquivalenten deterministischen Automaten.

(c) Wie groß (in Abhängigkeit von n) wird der durch das Teilmengenverfahren aus $\mathcal{A}'$ konstruierte Automat (ohne Begründung)?

Übungsaufgabe 3.10.11

Zeigen Sie, daß die regulären Sprachen von alle Grammatiken erzeugt werden, die nur linkslineare und terminierende Produktionen besitzen.

Übungsaufgabe 3.10.12

Betrachten Sie zu $k \in \mathbb{N}, k > 0$ die folgende reguläre Sprache:

$$L(k) = \{w \in \{0,1\}^* \mid N_1(w) \equiv 0 \bmod k\},$$

wobei $N_x(w)$ die Anzahl Vorkommen von x in w bezeichnet.

Geben Sie den Index von $L(k)$ an und Beweisen Sie Ihre Behauptung mit Hilfe der Kongurenzrelation R_L.

Übungsaufgabe 3.10.13

Welche der folgenden Aussagen sind richtig und welche sind falsch? Bitte kreuzen Sie die richtigen Kästchen an!

Jeder reduzierte Automat ist minimal	◯ Ja	◯ Nein
Jeder reduzierte Automat ist minimal	◯ Ja	◯ Nein

3.11 Lösungen zu den Aufgaben

Lösung zu Aufgabe 3.10.1

(a) Annahme L_1 ist regulär. Für jedes $n \in \mathbb{N}$ betrachte $x = 0^n 1 0^n \in L_1$. Für jede Zerlegung $x = uvw$ mit $|uv| \leq n$ und $|v| \geq 1$ gilt: $v = 0^i$ für $1 \leq i \leq n$. Nach dem Pumping-Lemma ist dann auch $uv^0w = uv = 0^{n-i}10^n \in L_1$ im Widerspruch zu $n - i < n$. Also ist L_1 nicht regulär.

(b) Annahme L_2 ist regulär, dann ist aufgrund der Abgeschlossenheit regulärer Sprachen auch das Komplement $(\{0\}^* - L_2) = \{0^k \mid k \text{ ist prim}\}$ von L_2 regulär. Von dieser Sprache wissen wir aber bereits, daß Sie nicht regulär ist. Also ist L_2 nicht regulär.

(c) Annahme L_3 ist regulär. Für jedes $n \in \mathbb{N}$ betrachte $x = a^n b^{n+1} \in L_3$. Für jede Zerlegung $x = uvw$ mit $|uv| \leq n$ und $|v| \geq 1$ gilt: $v = a^i$ für $1 \leq i \leq n$. w kann noch einige a's enthalten, d.h. $w = a^k b^{n+1}$. Nach dem Pumping-Lemma ist dann auch $uv^3w = a^{n-i-k}a^{3i}a^k b^{n+1} = a^{n+2i}b^{n+1} \in L_3$ im Widerspruch zu $n + 2i > n + 1$. Also ist L_3 nicht regulär.

(d) Für jedes $n \in \mathbb{N}$ sei $x = a^n b^n$. Für jede Zerlegung $x = uvw$ mit $|v| \geq 1$ und $|uv| \leq n$ gilt $|w| > n$, d.h. $v \neq \epsilon$ liegt vollständig in a^n. Dann gilt offenbar $N_a(uv^0w) < N_b(uv^0w)$ und $uv^0w \notin L_4$. Nach dem Pumpinglemma kann L_4 also nicht regulär sein.

Lösung zu Aufgabe 3.10.2

(a) Wir wählen $n = 1$ und führen eine Fallunterscheidung zwischen der Menge $L_1 = \{1^i 0^j \mid i, j \geq 0\}$ und $L_2 = \{0^i 1^j 0^j \mid i, j \geq 0\}$ durch. Insbesondere ist $x \neq \epsilon$. Im folgenden wird die Zerlegung immer so gewählt, daß $u = \epsilon$ und $|v| = 1$ ist. Offenbar gilt dann immer $|uv| \leq n = 1$ und $|v| \geq 1$.

<u>1.Fall:</u> $x = 1^i 0^j \in L_1$. Weiterhin sei

$i = 0$ In diesem Fall gilt $j > 0$. Zerlege $x = uvw$ so, daß $u = \epsilon, v = 0, w = 0^{j-1}$ ist. Dann gilt $uv^kw = 0^k 0^{j-1} \in L_1$ für alle $k \in \mathbb{N}$.

$i > 0$ Zerlege $x = uvw$ so, daß $u = \epsilon, v = 1, w = 1^{i-1}0^j$ ist . Dann gilt $uv^kw = 1^k 1^{i-1} 0^j \in L_1$ für jedes $k \in \mathbb{N}$.

2.Fall: $x = 0^i 1^j 0^j \in L_2$. Weiterhin sei

$i = 0$ In diesem Fall gilt $j > 0$. Zerlege $x = uvw$ so, daß $u = \epsilon, v = 1, w = 1^{j-1}0^j$ ist. Dann gilt $uv^kw = 1^k 1^{j-1} 0^j \in L_1$ für jedes $k \in \mathbb{N}$.

$i > 0$ Zerlege $x = uvw$ so, daß $u = \epsilon, v = 0, w = 0^{i-1}1^j 0^j$ ist. Dann gilt $uv^kw = 0^k 0^{i-1} 1^j 0^j \in L_2$ für alle $k \in \mathbb{N}$.

(b) Nach dem Satz von Myhill ist L regulär genau dann, wenn $index(L)$ endlich. Wir zeigen, daß der Index unendlich ist, d.h. es gibt unendlich viele verschiedene Äquivalenzklassen.

Für alle $i \neq j$ gilt: $[01^i]_\sim \neq [01^j]_\sim$, da $01^i\underline{0^i} \in L$ aber $01^j\underline{0^i} \notin L$ gilt. Es sind also alle $[01^i]_\sim$ paarweise verschieden.

(c) $G = (N, V, S, P)$ sei gegeben durch $N = \{0, 1\}, V = \{S, P_1, P_2, P_3\}$ und

$$\begin{array}{rcllll} P & = & \{ & S \to P_1, & S \to P_2, & \\ & & & P_1 \to 0P_1, & P_1 \to P_3, & \\ & & & P_3 \to 1P_30, & P_3 \to \epsilon, & \\ & & & P_2 \to 1P_2, & P_2 \to P_20, & P_2 \to \epsilon \quad \} \end{array}$$

Lösung zu Aufgabe 3.10.3

Wir benutzten für den Beweis das „normale“ Pumping-Lemma und die Abgeschlossenheit von reguläre Sprachen bzgl. Spiegelbild.

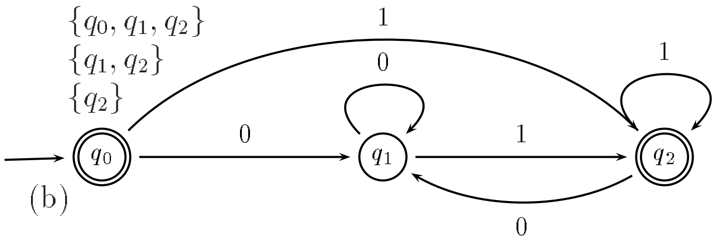
Es sei $y \in L$. Das Spiegelbild L^r von L ist regulär und erfüllt deswegen die Voraussetzungen für das (normale) Pumping-Lemma. D.h. es gibt $n \in \mathbb{N}$, so daß für $x = y^r \in L^r$ eine Zerlegung $x = uvw$ existiert mit $|uv| \leq n, |v| \geq 1$ und $|x| \geq n$ mit $\forall j \in \mathbb{N} : uv^jw \in L$

Dann gilt für jedes $y = (y^r)^r = x^r \in L$, daß eine Zerlegung $y = (uvw)^r = \underbrace{w^r}_{=u'}\underbrace{v^r}_{=v'}\underbrace{u^r}_{=w'}$ existiert mit $|y| = |x^r| \geq n, |v'| = |v^r| \geq 1$ und $|v'w'| = |u^rv^r| \leq n$ mit $\forall j \in \mathbb{N} : u'v'^jw' = w^r(v^r)^ju^r = w^r(v^j)^ru^r = (uv^jw)^r \in L$, da $uv^jw \in L^r$.

Lösung zu Aufgabe 3.10.4

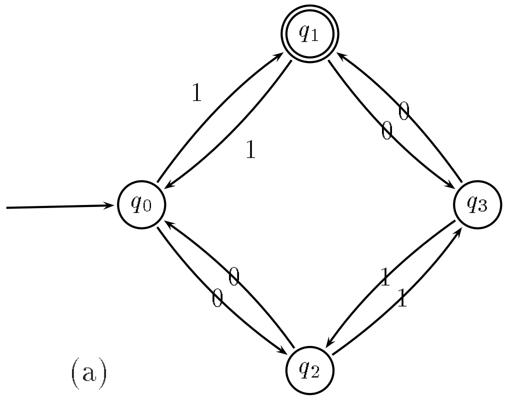
(a)

Zustand	Cl_ϵ
q_0	$\{q_0, q_1, q_2\}$
q_1	$\{q_1, q_2\}$
q_2	$\{q_2\}$



(b)

Lösung zu Aufgabe 3.10.5



(a)

(b) Die Zustände q_0 und q_3 werden zusätzlich Endzustände.

(c) Der Index ist 4 (der Automat, der L akzeptiert ist minimal).

Lösung zu Aufgabe 3.10.6

Wenn $\mathcal{A}$ den Automaten bezeichnet und $L(\mathcal{A})$ dessen akzeptierte Sprache ist, dann gilt:

$$
\begin{array}{lcllcl}
L(\mathcal{A}) & = & R_{0,2} & & = & \\
R^3_{0,2} & = & R^2_{0,2} \cup R^2_{0,2}(\underline{R^2_{2,2}})^* \underline{R^2_{2,2}} & & \overset{x \text{ s.u.}}{\sim} & x + xa^*a \\
R^2_{0,2} & = & R^1_{0,2} \cup R^1_{0,1}(R^1_{1,1})^* R^1_{1,2} & & & \\
 & \overset{\text{s.u.}}{\sim} & \overbrace{(b + cc^*b) + (a + cc^*a)(a+b)^*c}^{=:x} & & & \\
R^1_{0,2} & = & R^0_{0,2} \cup R^0_{0,0}(R^0_{0,0})^* R^0_{0,2} & & = & \{b\} \cup \{c\}\{c\}^*\{b\} \\
 & \sim & b + cc^*b & & & \\
R^1_{0,1} & = & R^0_{0,1} \cup R^0_{0,0}(R^0_{0,0})^* R^0_{0,1} & & = & \{a\} \cup \{c\}\{c\}^*\{a\} \\
 & \sim & a + cc^*a & & & \\
R^1_{1,1} & = & R^0_{1,1} \cup R^0_{1,0}(R^0_{0,0})^* R^0_{0,1} & & = & \{a,b\} \cup \emptyset \ldots \\
 & \sim & a + b & & & \\
R^1_{1,2} & = & R^0_{1,2} \cup R^0_{1,0}(R^0_{0,0})^* R^0_{0,2} & & = & \{c\} \cup \emptyset \ldots \\
 & \sim & c & & & \\
\underline{R^2_{2,2}} & = & R^1_{2,2} \cup R^1_{2,1}(R^1_{1,1})^* R^1_{1,2} & & = & \{a\} \cup \emptyset \ldots \\
 & \sim & a & & &
\end{array}
$$

Lösung zu Aufgabe 3.10.7

Es gibt hier zwei Möglichkeiten: Einmal kann man einen beliebigen deterministischen Automaten für L angeben und diesen Automaten minimieren, oder man zeigt, daß $index(L) = 4$ gilt und berechnet die Äquivalenzklassen von L. Anschließend konstruieren wir mit deren Hilfe den minimalen Automaten.

Der Beweis teilt sich in zwei Schritte auf. Wir bezeichnen die Äquivalenzklassen zu einem $x \in L$ mit $[x]_\sim$.

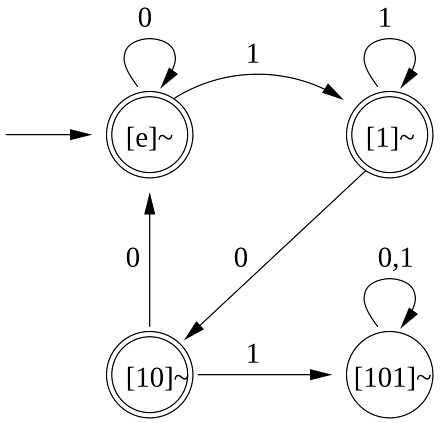
(a) Es gibt 4 *unterschiedliche* Äquivalenzklassen; deren Repräsentanten sind z.B. $\epsilon, 1, 10, 101$:

- $[\epsilon]_\sim \neq [1]_\sim$, da $\epsilon\underline{01} \in L$, aber $1\underline{01} \notin L$.
- $[1]_\sim \neq [10]_\sim$, da $1\underline{1} \in L$, aber $10\underline{1} \notin L$
- $[10]_\sim \neq [101]_\sim$, da $10\underline{\epsilon} \in L$, aber $101\underline{\epsilon} \notin L$
- $[\epsilon]_\sim \neq [10]_\sim$, da $\epsilon\underline{1} \in L$, aber $10\underline{1} \notin L$
- $[\epsilon]_\sim \neq [101]_\sim$, da $\epsilon\underline{\epsilon} \in L$, aber $101\underline{\epsilon} \notin L$
- $[1]_\sim \neq [101]_\sim$, da $1\underline{\epsilon} \in L$, aber $101\underline{\epsilon} \notin L$

(b) Es gibt nicht mehr als 4 unterschiedliche Äquivalenzklassen. Für ein $x \in (L - \{\epsilon, 1, 10, 101\})$ gibt es die folgende Fälle:

- x endet auf 101. Dann gilt $[x]_\sim = [101]_\sim$, da weder x noch 101 in L liegen.
- x endet auf 10. Dann gilt $[x]_\sim = [10]_\sim$, da für jedes $v \in L$ gilt: $10v \in L$ gdw. $x \in L$.
- x endet auf 0 aber nicht auf 10. Dann gilt $[x]_\sim = [0]_\sim$, da für jedes $v \in L$ gilt: $0v \in L$ gdw. $x \in L$.
- x endet auf 1 aber nicht auf 101. Dann gilt $[x]_\sim = [1]_\sim$, da für jedes $v \in L$ gilt: $1v \in L$ gdw. $x \in L$.

Der konstruierte Automat sieht wie folgt aus (Es ist $[0]_\sim = [\epsilon]_\sim$):



Lösung zu Aufgabe 3.10.9

(a) $(0+1)^*00(0+1)^*$

(b) Die Äquivalenzklasse zu einem Wort $w \in \{0,1\}^*$ seien mit $[w]$ bezeichnet. Für $u \in \{0,1\}^*$ gilt:

- $\epsilon u \in L \Leftrightarrow wu \in L$ gdw ($u \in L$ und) $w \notin L$ und w endet auf 1 (ansonsten gilt $\epsilon 0 \notin L$, aber $w0 \in L$). Also:

$$[\epsilon] = \{w \in \{0,1\}^* \mid w \text{ enthählt nicht } 00 \text{ und endet auf } 1\}$$

- $0u \in L \Leftrightarrow wu \in L$ gdw ($u \in L$ oder u beginnt mit 0 und) $w \notin L$ und w endet auf 0 (ansonsten gilt $00 \in L$, aber $w0 \notin L$). Also:

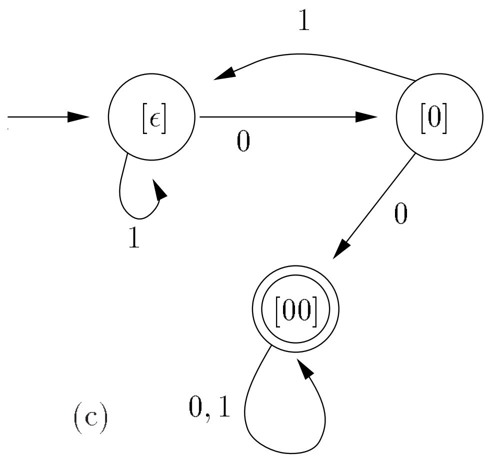
$$[0] = \{w \in \{0,1\}^* \mid w \text{ enthählt nicht } 00 \text{ und endet auf } 0\}$$

- $00u \in L \Leftrightarrow wu \in L$ gdw $w \in L$ (ansonsten gilt $00\epsilon \in L$, aber $w \notin L$). Also:

$$[0] = L$$

Dies sind alle Äquivalenzklassen, da $[\epsilon] \cup [0] \cup [00] = \{0,1\}^*$ gilt.

(c)



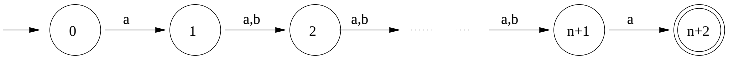
(d) Der Index ist 3. Es gibt zwei verschiedene Begründungen dafür:

  (i) Der Index einer Sprachen L ist definiert als Anzahl verschiedener Äquivalenzklassen bzgl R_L. Teil (b) ist der Index also 3.

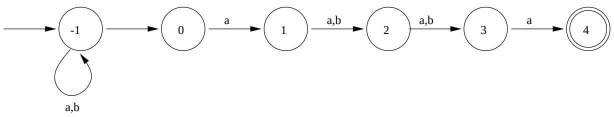
  (ii) Die Anzahl Zustände eines vollständigen minimalen det. endlichen. Automaten, der L akzeptiert ist gleich dem Index von L. Nach Teil (c) ist der Index also 3.

Lösung zu Aufgabe 3.10.10

(a) $\mathcal{A}$ sei durch folgendes Diagramm gegeben:



(b) Die Zustände werden durch die Zahlen $-1, 0, 1, 2, 3, 4$ gekennzeichnet. Der Automat $\mathcal{A}'$ sieht wie folgt aus (ϵ bei Übergang weggelassen):



Das Teilmengenkonstruktionsverfahren liefert das folgende Ergebnis (Mengenklammern sind weggelassen):

Zustandsmenge	a	b
$-1, 0$	$-1, 0, 1$	$-1, 0$
$-1, 0, 1$	$-1, 0, 1, 2$	$-1, 0, 2$
$-1, 0, 1, 2$	$-1, 0, 1, 2, 3$	$-1, 0, 2, 3$
$-1, 0, 2$	$-1, 0, 1, 3$	$-1, 0, 3$
$-1, 0, 1, 2, 3$	$-1, 0, 1, 2, 3, 4$	$-1, 0, 2, 3$
$-1, 0, 2, 3$	$-1, 0, 1, 3, 4$	$-1, 0, 3$
$-1, 0, 1, 3$	$-1, 0, 1, 2, 4$	$-1, 0, 2$
$-1, 0, 3$	$-1, 0, 1, 4$	$-1, 0$
$-1, 0, 1, 2, 3, 4$	$-1, 0, 1, 2, 3, 4$	$-1, 0, 2, 3$
$-1, 0, 1, 3, 4$	$-1, 0, 1, 2, 4$	$-1, 0, 2$
$-1, 0, 1, 2, 4$	$-1, 0, 1, 2, 3$	$-1, 0, 2, 3$
$-1, 0, 1, 4$	$-1, 0, 1, 2$	$-1, 0, 2$

Hier noch das Ergebnis für $n = 3$:

Zustandsmenge	a	b
$-1, 0$	$-1, 0, 1$	$-1, 0$
$-1, 0, 1$	$-1, 0, 1, 2$	$-1, 0, 2$
$-1, 0, 1, 2$	$-1, 0, 1, 2, 3$	$-1, 0, 2, 3$
$-1, 0, 2$	$-1, 0, 1, 3$	$-1, 0, 3$
$-1, 0, 1, 2, 3$	$-1, 0, 1, 2, 3, 4$	$-1, 0, 2, 3, 4$
$-1, 0, 2, 3$	$-1, 0, 1, 3, 4$	$-1, 0, 3, 4$
$-1, 0, 1, 3$	$-1, 0, 1, 2, 4$	$-1, 0, 2, 4$
$-1, 0, 3$	$-1, 0, 1, 4$	$-1, 0, 4$
$-1, 0, 1, 2, 3, 4$	$-1, 0, 1, 2, 3, 4, 5$	$-1, 0, 2, 3, 4$
$-1, 0, 2, 3, 4$	$-1, 0, 1, 3, 4, 5$	$-1, 0, 3, 4$
$-1, 0, 1, 3, 4$	$-1, 0, 1, 2, 4, 5$	$-1, 0, 2, 4$
$-1, 0, 3, 4$	$-1, 0, 1, 4, 5$	$-1, 0, 4$
$-1, 0, 1, 2, 4$	$-1, 0, 1, 2, 3, 5$	$-1, 0, 2, 3$
$-1, 0, 2, 4$	$-1, 0, 1, 3, 5$	$-1, 0, 3$
$-1, 0, 1, 4$	$-1, 0, 1, 2, 5$	$-1, 0, 2$
$-1, 0, 4$	$-1, 0, 1, 5$	$-1, 0$
$-1, 0, 1, 2, 3, 4, 5$	$-1, 0, 1, 2, 3, 4, 5$	$-1, 0, 2, 3, 4$
$-1, 0, 1, 3, 4, 5$	$-1, 0, 1, 2, 4, 5$	$-1, 0, 2, 4$
$-1, 0, 1, 2, 4, 5$	$-1, 0, 1, 2, 3, 5$	$-1, 0, 2, 3$
$-1, 0, 1, 4, 5$	$-1, 0, 1, 2, 5$	$-1, 0, 2$
$-1, 0, 1, 2, 3, 5$	$-1, 0, 1, 2, 3, 4$	$-1, 0, 3, 4$
$-1, 0, 1, 3, 5$	$-1, 0, 1, 2, 4$	$-1, 0, 2, 4$
$-1, 0, 1, 2, 5$	$-1, 0, 1, 2, 3$	$-1, 0, 2, 3$
$-1, 0, 1, 5$	$-1, 0, 1, 2$	$-1, 0, 2$

(c) Die Größe (Anzahl von Zuständen) von $\mathcal{A}'$ steigt exponentiell an.

Lösung zu Aufgabe 3.10.11

Bei der Klasse $\mathcal{L}_3$ von Sprachen, die von allen rechtslinearen Grammatiken erzeugt wird, handelt es sich nach Def. um die Sprachen vom Typ 3, d.h. um die regulären Sprachen.

Der folgende Beweis nutzt aus, daß $\mathcal{L}_3$ bezüglich Spiegelbild abgeschlossen ist. Dazu folgende Definition:

Das Spiegelbild G^r einer Grammatik $G = (N, T, S, P)$ ist gegeben durch $(N, T, S, P)^r = (N, T, S, P^r)$, mit:

$$
\begin{aligned}
P^r \quad := \quad & \{X \to Y w^r \mid X \to wY \in P; w \in N^*; X, Y \in N\} \cup \\
& \{X \to w^r \mid X \to w \in P; w \in N^*; X \in N\}
\end{aligned}
$$

Für links- und rechtslineare Grammatiken G gilt dann offenbar $L(G^r) = L(G)^r$.

Es sei $\mathcal{L}'_3 = \{L(G) \mid G \text{ ist linkslineare Grammatik}\}$. Zu zeigen sind beide Inklusionen von $\mathcal{L}_3 = \mathcal{L}'_3$.

- $\subseteq$ Da $\mathcal{L}_3$ abgeschlossen ist bezüglich Spiegelbild gibt es zu $L \in \mathcal{L}_3$ eine rechtslineare Grammatik G mit $L(G) = L^r$. Dann ist G^r linkslinear mit $L(G^r) = L(G)^r = (L^r)^r = L$.
- $\supseteq$ L sei aus $\mathcal{L}'_3$. Dann gibt es linkslineare Grammatik G mit $L(G) = L$. G^r ist rechtslineare Grammatik mit $L(G^r) = L(G)^r$. Wegen Abgeschlossenheit von $\mathcal{L}_3$ bezüglich Spiegelbild, gibt es eine rechtslineare Grammatik G' mit $L(G') = L(G)$.

Lösung zu Aufgabe 3.10.12

Der Index (Anzahl Äquivalenzklassen bzgl. R_L) ist k, denn es gibt folgende k verschiedenen Äquivalenzklassen (mit [] bezeichnet):

$$
\begin{aligned}
[\epsilon] \quad &= \quad \{w \in \{0,1\} \mid N_1(w) \equiv 0(mod\ \ k)\} \\
[1] \quad &= \quad \{w \in \{0,1\} \mid N_1(w) \equiv 1(mod\ \ k)\} \\
\vdots \quad &= \quad \vdots \\
[1^{k-1}] \quad &= \quad \{w \in \{0,1\} \mid N_1(w) \equiv k-1(mod\ \ k)\}
\end{aligned}
$$

Es handelt sich hier wirklich um Äquivalenzklassen, da für alle $w_1, w_2 \in [1^j], j = 0, \dots, k-1$ und $u \in \{0,1\}^*$ die Anzahl 1en modulo k in $w_1 u$ und $w_2 u$ gleich bleibt.

Die Äquivalenzklassen sind verschieden, da für $0 \leq i < j < k$ gilt: $1^i 1^{k-i} \in L$ aber $1^j 1^{k-i} \notin L$.

Es sind offenbar alle Äquivalenzklassen, da deren Vereinigung $\{0,1\}^*$ ergibt.

Lösung zu Aufgabe 3.10.13

Welche der folgenden Aussagen sind richtig und welche sind falsch? Bitte kreuzen Sie die richtigen Kästchen an!

Jeder reduzierte Automat ist minimal	$\surd$ Ja	◯ Nein
Jeder reduzierte Automat ist minimal	◯ Ja	$\surd$ Nein

Kapitel 4

Kontextfreie Sprachen

Wir wenden uns in diesem Kapitel einem genaueren Studium kontextfreier Grammatiken zu, die in der Chomsky Klassifikation, Definition 42, als Grammatiken von Typ 2 geführt werden. Zum Einstieg betrachten wir noch einmal ein Beispiel einer kontextfreien Grammatik:

Beispiel 40 (Kontextfreie Grammatik)

$E \to E + E$	$E \to E * E$
$E \to (E)$	$E \to id$

In der Regeln benutzt man eine ökonomischere Notation, in der man Produktionen mit derselben rechten Seite zusammen faßt und die verschiedenen linken Seiten durch einen senkrechten Strich voneinander abtrennt.

$$E \to E + E \mid E * E \mid (E) \mid id$$

Ableitungen in einer Grammatik haben wir bisher in linearer Notation aufgeschrieben wie in dem folgenden Beispiel.

Beispiel 41 *Ableitung*

0	E
1	$E + E$
2	$(E) + E$
3	$(E) + E * E$
4	$(id) + E * E$
5	$(id) + E * id$
6	$(id) + id * id$

Herleitungen in einer Grammatik sind im allgemeinen hochgradig nicht deterministisch. Es lohnt sich die Stellen, an denen eine Auswahl möglich ist, etwas genauer anzuschauen. Da ist zunächst die Auswahl zu treffen, welches Vorkommen eines Nichtterminalzeichens in dem bisher letzten Wort einer Ableitung im nächsten Schritt durch eine Grammmatikregel ersetzt werden soll. In der obigen Ableitung wurde im Schritt 2 das erste Vorkommen von E durch Anwendung der Regel $E \rightarrow (E)$ ersetzt. Genausogut hätte man hier mit dem zweiten Vorkommen von E weiterarbeiten können. Der zweite Nichtdeterminismus liegt in der Auswahl einer Grammatikregel, die benutzt werden soll, um ein Vorkommen eines Nichtterminalzeichens zu ersetzen. In der obigen Ableitung hätte in jeder Zeile auch eine andere der vier Regel benutzt werden können. Es stellt sich heraus, daß die erste Quelle des Nichtdeterminismus harmlos ist. Man verbaut sich keine Ableitungsmöglichkeiten, wenn man z.B. immer das am weitesten links stehende Nichtterminalzeichen ersetzt. Die zweite Quelle des Nichtdeterminismus ist dagegen tiefliegend und wird uns im weiteren sehr intensiv beschäftigen.

Eine Variante zur linearen Notation einer Ableitung ist der Ableitungsbaum, auch Syntaxbaum genannt. An der Wurzel des Baumes steht das Startzeichen der Grammatik. Die Markierungen der Blattknoten ergeben von links nach rechts gelesen, das abgeleitete Wort. Wird im Ableitungsprozess ein Nichtterminalzeichen E ersetzt durch eine Worte $w = a_1 \dots a_n$, so wird der Syntaxbaum entsprechend erweitert, indem unterhalb von E n neue Knoten eingefügt werden, die nacheinander die Zeichen a_i als Markierung enthalten. Auf diese Weise kann man jeder Ableitung einen eindeutig bestimmten Syn-

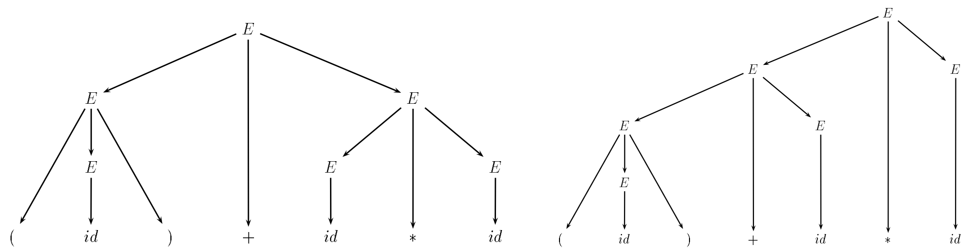


Abbildung 4.1: Zwei Ableitungsbäume für dasselbe Wort

taxbaum zuordnen. Eine Zuordnung in der umgekehrten Richtung ist auch möglich, doch haben wir verschiedene Möglichkeiten, die unabhängigen äste des Syntaxbaums in eine sequentielle Reihenfolge zu bringen. Trifft man hier eine Normierung, indem man z.B. immer das am weitesten links stehende Terminalzeichen ersetzt, so ergibt sich eine eindeutige Korrespondenz zwischen Ableitungsbäumen und Linksableitungen.

Beispiel 42 (Linksableitung)
E
$E + E$
$(E) + E$
$(id) + E$
$(id) + E * E$
$(id) + id * E$
$(id) + id * id$

Der Syntaxbaum ist durch ein abgeleitetes terminale Wort einer Grammatik nicht eindeutig bestimmt, wie Abbildung 4.1 für das Wort $(id) + id * id$ zeigt.

Definition 69
Eine Grammatik $\mathcal{G}$ heißt eindeutig*, wenn für jedes Wort $w \in L(\mathcal{G})$ genau ein Ableitungsbaum für w in $\mathcal{G}$ existiert.*

äquivalent ist die Forderung, daß für jedes Wort $w \in L(\mathcal{G})$ genau eine Linksableitung für w in $\mathcal{G}$ existiert.

4.1 Kellerautomaten

Bevor wir eine allgemeine Definition für Kellerautomaten geben, betrachten wir zwei Beispielklassen. In beiden Fällen handelt es sich um spezielle Kellerautomaten, die ausgehend von einer gegebenen kontextfreien Grammatik $\mathcal{G}$ konstruiert werden und zu einem vorgegeben Wort w feststellen, ob $w \in L(\mathcal{G})$ gilt oder nicht.

4.1.1 LL-Automaten

Die Namensgebung für LL-Kellerautomaten kommt daher, daß sie das Eingabewort von **l**inks nach rechts einlesen und im positiven Falle eine **L**inksableitung des Eingabewortes erzeugen. Die Konfigurationen eines LL-Kellerautomaten werde durch Zeichenfolgen der Form Wqw gegeben. Dabei ist W ein Wort über dem Alphabet $(N \cup T)$ und w ein terminales Wort. Dabei steht W für den Kellerinhalt - das am weitesten rechts stehende Zeichen ist das oberste Kellerelement - w steht für den noch nicht gelesenen Teil der Eingabe und q dient, zumindest für den Augenblick, als Trennungsmarkierung. Gestartet wird ein LL-Kellerautomat in der Konfiguration Sqw, wobei S das Startsymbol der Grammatik ist und w das zu testende Wort ist. Zustandsübergänge werden in der Form $Aqb \to uq$ notiert. Ein Übergang $Aqb \to uq$ ist anwendbar, wenn das oberste Kellerzeichen A und der nächste Buchstaben des Eingabewortes b ist. Die Anwendung der Regel ersetzt A durch das Wort u und streicht b aus dem Eingabewort. Die Angabe b kann auch fehlen. Dann hängt die Anwendung des Zustandsübergangs nur vom obersten Kellerzeichen ab. Das Eingabewort bleibt von der Zustandsänderung unberührt. Das Eingabewort wird akzeptiert, wenn aus der Startkonfiguration die Konfiguration q erreicht werden kann, in der das ganze Eingabewort eingelesen wurde und der Keller leer ist.

Wir betrachten zunächst ein Beispiel.

Beispiel 43 ($\mathcal{A}_1^{LL}$)
Gegeben sei die kontextfreie Grammatik G_1:

$E \to T$	$T \to F$	$F \to i$
$E \to E + T$	$T \to T * F$	$F \to (E)$

Die Zustandsübergangsregeln des LL-Automaten $\mathcal{K}_1^{LL}$ sind:

$Eq \to Tq$	$Tq \to Fq$	$Fq \to iq$
$Eq \to T + Eq$	$Tq \to F * Tq$	$Fq \to)E(q$
$iqi \to q$	$(q(\to q$	$)q) \to q$
$+q+ \to q$	$*q* \to q$	

Um die Arbeitsweise von $\mathcal{K}_1^{LL}$ zu demonstrieren, wollen wir zeigen, daß das Wort $i + i * i$ in $L(G_1)$ liegt.

Keller	*Eingabe*	*Regel*		*Grammatikableitung*
Eq	$i + i * i$			E
$T + Eq$	$i + i * i$	**produce**	$E \to E + T$	$E + T$
$T + Tq$	$i + i * i$	**produce**	$E \to T$	$T + T$
$T + Fq$	$i + i * i$	**produce**	$T \to F$	$F + T$
$T + iq$	$i + i * i$	**produce**	$F \to i$	$i + T$
$T + q$	$+i * i$	**compare**		
Tq	$i * i$	**compare**		
$F * Tq$	$i * i$	**produce**	$T \to T * F$	$i + T * F$
$F * Fq$	$i * i$	**produce**	$T \to F$	$i + F * F$
$F * iq$	$i * i$	**produce**	$F \to i$	$i + i * F$
$F * q$	$*i$	**compare**		
Fq	i	**compare**		
iq	i	**produce**	$F \to i$	$i + i * i$
q		**compare**		

Allgemein definieren wir:

Definition 70
Zur einer vorgegebenen kontextfreien Grammatik $G = (N, T, S, \mathcal{P})$ ist der LL-Automat $\mathcal{A}_G^{LL}$ bestimmt durch die Regeln $\mathcal{R} = \mathcal{R}_{produce} \cup \mathcal{R}_{compare}$

$\mathcal{R}_{compare} = \{aqa \to q \mid a \in T\}$
$\mathcal{R}_{produce} = \{Aq \to l^r q \mid A \to l \in \mathcal{P}\}$

Theorem 50
Sei $\mathcal{G}$ eine kontextfreie Grammatik, $\mathcal{A}_\mathcal{G}^{LL}$ der zugehörige LL-Automat. Dann gilt für jedes Wort $w \in \Sigma^*$:

$$\text{In } \mathcal{G}\text{: } S \Rightarrow^* w$$
$$gdw$$
$$\text{In } \mathcal{A}_\mathcal{G}^{LL}\text{: } Sqw \Rightarrow^* q$$

Beweis: Man zeigt genauer für jedes $u_1 \in T^*$,$A \in N$, $u_2 \in (T \cup N)^*$ und $w = u_1 w_1$, daß eine Linksableitung in $\mathcal{G}$

$$S \Rightarrow^* u_1 A u_2 \Rightarrow^* w$$

genau dann möglich ist, wenn eine Berechnungsfolge der Form

$$Sqw \Rightarrow_\ell^* u_2^r A q w_1$$

in $\mathcal{A}_\mathcal{G}^{LL}$ existiert.

Wir geben hier Hinweise wie die Implikation von oben nach unten bewiesen werden kann. Die Implikation in die umgekehrte Richtung ist dann einfach zu erschließen.

Induktionsschritt:

Wir betrachten die Ableitung in $\mathcal{G}$

$$S \Rightarrow_\ell^* u_1 A u_2 \Rightarrow u_1 v_1 B v_2 u_2 \Rightarrow_\ell^* w$$

mit $A \to v_1 B v_2 \in \mathcal{P}$, $w = u_1 w_1$ und $w_1 = v_1 w_2$. Nach Induktionsvoraussetzung gilt in $\mathcal{A}_\mathcal{G}^{LL}$ schon

$$Sqw \Rightarrow^* u_2^r A q w_1$$

Nach Konstruktion ist $Aq \to v_2^r B v_1^r q$ eine übergangsregel in $\mathcal{A}_\mathcal{G}^{LL}$. Also

$$Sqw \Rightarrow^* u_2^r A q w_1 \Rightarrow u_2^r v_2^r B v_1^r q w_1.$$

Wegen $w_1 = v_1 w_2$ kann durch *compare* Schritte

$$Sqw \Rightarrow^* u_2^r v_2^r B v_1^r q w_1 \Rightarrow^* u_2^r v_2^r B q w_2$$

erreicht werden.

■

Korollar 51
Es gibt eine eindeutige Korrespondenz zwischen Linksableitungen in einer Grammatik $\mathcal{G}$ und Berechnungsfolgen des LL-Automaten $\mathcal{A}_{\mathcal{G}}^{LL}$.

Insbesondere ist $\mathcal{G}$ genau dann eindeutig, wenn es für jedes Wort höchstens eine akzeptierende Berechnungsfolge von $\mathcal{A}_{\mathcal{G}}^{LL}$ gibt.

Beweis: Wir verzichten auf einen formalen Beweis. Die Beweisidee ist nach dem vorangegangenen Beweis und der Gegenüberstellung in Beispiel 43 ohnehin klar.

■

4.1.2 LR-Kellerautomaten

LR-Kellerautomaten werden so genannt, weil sie das Eingabewort von **l**inks nach rechts durchlesen und im positiven Fall eine **R**echtsableitung generieren. Das Format der Übergangsregeln ist dasgleiche wie bei den LL-Kellerautomaten und auch die allgemeine Arbeitsweise stimmt überein. Für LR-Automaten können allerdings in einem Schritt mehrere Kellersymbole ersetzt werden. Wir zeigen zunächst wieder ein Beispiel:

Beispiel 44 ($\mathcal{A}_1^{LR}$)
Wir gehen wieder von derselben Grammatik G_1 wie in Beispiel 43 aus:

$E \to T$	$T \to F$	$F \to i$
$E \to E + T$	$T \to T * F$	$F \to (E)$

Die Übergangsregeln des LR-Automat $\mathcal{A}_1^{LR}$ sind jetzt:

$Tq \to Eq$	$Fq \to Tq$	$iq \to Fq$
$E + Tq \to Eq$	$T * Fq \to Tq$	$(E)q \to Fq$
$qi \to iq$	$q(\to (q$	$q) \to)q$
$q+ \to +q$	$q* \to *q$	

Der Test für das Eingabewort $i + i * i$ sieht jetzt folgendermaßen aus:

Keller	*Eingabe*	*Regel*		*Grammatikableitung*
q	$i+i*i$			
iq	$+i*i$	**shift**		$i+i*i$
Fq	$+i*i$	**reduce**	$F \to i$	$F+i*i$
Tq	$+i*i$	**reduce**	$T \to F$	$T+i*i$
Eq	$+i*i$	**reduce**	$E \to T$	
$E+q$	$i*i$	**shift**		
$E+iq$	$*i$	**shift**		$E+i*i$
$E+Fq$	$*i$	**reduce**	$F \to i$	$E+F*i$
$E+Tq$	$*i$	**reduce**	$T \to F$	
$E+T*q$	i	**shift**		
$E+T*iq$		**shift**		$E+T*i$
$E+T*Fq$		**reduce**	$F \to i$	$E+T*F$
$E+Tq$		**reduce**	$T \to T*F$	$E+T$
Eq		**reduce**	$E \to E+T$	E

Allgemein definieren wir:

Definition 71
Für eine kontextfreie Grammatik $G = (N, T, S, \mathcal{P})$ ist der LR-Automat $\mathcal{A}_G$ bestimmt durch die Regeln $\mathcal{R} = \mathcal{R}_{reduce} \cup \mathcal{R}_{shift}$

$\mathcal{R}_{shift} = \{qa \to aq \mid a \in T\}$ $\mathcal{R}_{reduce} = \{lq \to Aq \mid A \to l \in \mathcal{P}\}$

Theorem 52
Sei $\mathcal{G}$ eine kontextfreie Grammatik mit Startsymbol S, $\mathcal{A}_{\mathcal{G}}^{LR}$ der zugehörige LR-Automat. Dann gilt für jedes Wort $w \in \Sigma^*$:

1. *In $\mathcal{G}$: $S \Rightarrow^* w$ gdw In $\mathcal{A}_{\mathcal{G}}^{LR}$: $qw \Rightarrow^* Sq$*

Genauer zeigt man

2. *$w_1Aqw_2 \Rightarrow^* Sq$ gdw $S \Rightarrow_r^* w_1Aw_2$ in $\mathcal{G}$*

Beweis: Offensichtlich genügt es die Behauptung 2 zu beweisen. Wir zeigen zunächst die Richtung von links nach rechts: daß aus einer akzeptierenden Berechnung $w_1Aqw_2 \Rightarrow^* Sq$ von $\mathcal{A}_{\mathcal{G}}^{LR}$ eine Rechtsableitung $S \Rightarrow_r^* w_1Aw_2$ in $\mathcal{G}$ konstruiert werden kann. Das geschieht durch Induktion über die Länge einer akzeptierenden Berechnung und das zwingt uns auch dazu Berechnungsfolgen zu betrachten die nicht in der Anfangskonfiguration von $\mathcal{A}_{\mathcal{G}}^{LR}$ starten.

Wir kümmern uns hier nur um den Induktionsschritt. Sei dazu eine Berechnung gegeben

$$w_3 q w_4 \Rightarrow^* w_1 u q w_2 \Rightarrow w_1 A q w_2 \Rightarrow^* S q$$

Dabei ist vorausgesetzt, daß

1. in dem Teilstück der Berechnungfolge von $w_3 q w_4$ nach $w_1 u q w_2$ nur shift-Regeln benutzt werden, also $w_3 w_4 = w_1 u w_2$ gilt,

2. die Produktion $A \rightarrow u$ in $\mathcal{G}$ vorkommt.

Nach Induktionsvoraussetzung gibt es eine Rechtsableitung

$$S \Rightarrow_r^* w_1 A w_2$$

Woraus sich mit der Regel $A \rightarrow u$ eine Ableitung

$$S \Rightarrow_r^* w_1 u w_2 = w_3 w_4$$

gewinnen läßt.

∎

Wie im Fall von LL-Automaten gilt auch hier die folgende konkretere Aussage:

Korollar 53
Es gibt eine eindeutige Korrespondenz zwischen Rechtsableitungen in einer Grammatik $\mathcal{G}$ und Berechnungsfolgen des LR-Automaten $\mathcal{A}_{\mathcal{G}}^{LR}$.

Insbesondere ist $\mathcal{G}$ genau dann eindeutig, wenn es für jedes Wort höchstens eine akzeptierende Berechnungsfolge von $\mathcal{A}_{\mathcal{G}}^{LR}$ gibt.

4.1.3 Allgemeine Kellerautomaten

Wir haben bisher zwei spezielle Typen von Kellerautomaten kennengelernt und kommen jetzt zur Einführung des allgemeinen Konzepts. Dieses Konzept ist allgemeiner als die Beispiele, indem noch endliche viele interne Zustände zugelassen werden. Auf der anderen Seite ist das allgemeine Konzept einschränkender als LR-Automaten, da bei jedem Zustandsübergang nur das oberste Kellerzeichen betrachtet werden darf.

Definition 72
Ein (nichtdeterministischer) Kellerautomat *wird bestimmt durch*

Q	*endliche Menge von Zuständen*
T	*Eingabealphabet*
K	*Kelleralphabet*
q_0	*Anfangszustand*
F	*Menge von Finalzuständen* $\subseteq Q$
s_0	*initiales Kellerzeichen*
δ	*Übergangsfunktion* $(K \times Q \times T \cup \{\varepsilon\}) \to \mathcal{P}(Q \times K^*)$

Die Arbeitsweise eines Kellerautomaten wird, wie wir es bei bisher allen Maschinenmodellen gehandhabt haben, durch die Abfolge von Konfigurationen beschrieben. Eine **Konfiguration** eines Kellerautomaten $\mathcal{K} = (Q, T, K, q_0, F, s_0, \delta)$ ist ein Tripel (y, q, x) mit

y	Kellerinhalt
q	aktueller Zustand
x	noch nicht gelesene Eingabe

Es gibt zwei Typen von Übergängen. Beim ersten wird ein Buchstabe des Eingabewortes eingelesen. Ausgehend von einer Konfiguration (y_0b, q_1, ax) und

$$(q_2, z) \in \delta(b, q_1, a)$$

mit $y_0, z \in K^*$, $b \in K$, $x \in T^*$, $a \in T$ und $q_i \in Q$ findet der Übergang

$$(y_0b, q_1, ax) \Rightarrow (y_0z, q_2, x)$$

statt.

Beim zweiten Übergangstypus wird nur der Keller verändert, das Eingabewort wird nicht berührt. Ausgehend von der Konfiguration (y_0b, q_1, ax) und

$$(q_2, z) \in \delta(b, q_1, \varepsilon)$$

findet der Übergang

$$(y_0b, q_1, ax) \Rightarrow (y_0z, q_2, ax)$$

statt.

$\Rightarrow^*$ steht für die transitive Hülle von $\Rightarrow$. Falls erforderlich schreiben wir $\Rightarrow^*_{\mathcal{K}}$

Definition 73
Sei $\mathcal{K} = (Q, T, K, q_0, F, s_0, \delta)$ *ein Kellerautomat.*

Es gibt zwei Möglichkeiten die von $\mathcal{K}$ *akzeptierte Sprache zu definieren.*

durch Finalzustand

$$L^F(\mathcal{K}) = \{x \in T^* \mid (s_0, q_0, x) \Rightarrow^*_{\mathcal{K}} (y, q_f, \varepsilon) \text{ } und \text{ } q_f \in F\}$$

durch leeren Keller

$$L^K(\mathcal{K}) = \{x \in T^* \mid (s_0, q_0, x) \Rightarrow^*_{\mathcal{K}} (s_0, q, \varepsilon)\}$$

Wir setzen voraus, daß kein Übergang das initiale Kellersymbol löscht oder zusätzliche Kopien von s_0 in den Keller schreibt. Man beachte, daß das Kellerende links steht!

Beispiel 45

$$\begin{array}{lcl}
Q & = & \{q_0, \ldots, q_7\} \\
F & = & \{q_4\} \\
T & = & \{a, b\} \\
K & = & \{s_0, A, B\} \\
\delta : & & \\
\delta(s_0, q_0, a) & = & \{(q_1, s_0A), (q_5, s_0)\} \\
\delta(A, q_1, a) & = & \{(q_1, AA)\} \\
\delta(A, q_1, b) & = & \{(q_2, \varepsilon)\} \\
\delta(A, q_2, b) & = & \{(q_2, \varepsilon)\} \\
\delta(s_0, q_2, a) & = & \{(q_3, s_0)\} \\
\delta(s_0, q_3, a) & = & \{(q_3, s_0)\} \\
\delta(s_0, q_3, \varepsilon) & = & \{(q_4, s_0)\} \\
\delta(s_0, q_5, a) & = & \{(q_5, s_0)\} \\
\delta(s_0, q_5, b) & = & \{(q_6, s_0B)\} \\
\delta(B, q_6, b) & = & \{(q_6, BB)\} \\
\delta(B, q_6, a) & = & \{(q_7, \varepsilon)\} \\
\delta(B, q_7, a) & = & \{(q_7, \varepsilon)\} \\
\delta(s_0, q_7, \varepsilon) & = & \{(q_4, s_0)\}
\end{array}$$

Aus: Duden Informatik, Dudenverlag, 1993.

Eine Ableitung:

Keller	Q	Eingabe	δ
s_0	q_0	a^2b^2a	
s_0A	q_1	ab^2a	$(q_1, s_0A) \in \delta(s_0, q_0, a)$
s_0AA	q_1	b^2a	$(q_1, AA) \in \delta(A, q_1, a)$
s_0A	q_2	ba	$(q_2, \varepsilon) \in \delta(A, q_1, b)$
s_0	q_2	a	$(q_2, \varepsilon) \in \delta(A, q_2, b)$
s_0	q_3	ε	$(q_3, s_0) \in \delta(s_0, q_2, a)$
s_0	q_4	ε	$(q_4, s_0) \in \delta(s_0, q_3, \varepsilon)$

Es gilt

$$L^F(\mathcal{M}) = \{a^n b^n a^m \mid n, m \geq 1\} \cup \{a^m b^n a^n \mid n, m \geq 1\}$$

Theorem 54

1. *Zu jedem Kellerautomaten $\mathcal{K}_1$ gibt es einen Kellerautomaten $\mathcal{K}_2$ mit*
$$L^F(\mathcal{K}_1) = L^K(\mathcal{K}_2)$$
2. *Zu jedem Kellerautomaten $\mathcal{K}_1$ gibt es einen Kellerautomaten $\mathcal{K}_2$ mit*
$$L^K(\mathcal{K}_1) = L^F(\mathcal{K}_2)$$

Beweis: siehe [29, Theorem 7] oder [23, Theorem 5.1 und 5.2, Seiten 114–115]

Theorem 55
Zu jeder kontextfreien Grammatik $\mathcal{G}$ gibt es einen Kellerautomaten $\mathcal{K}$ mit

$$L(\mathcal{G}) = L(\mathcal{K})$$

Beweis: Die Automaten $\mathcal{K}^{LL}$ und $\mathcal{K}^{LR}$ sind beides Beispiele für den gesuchten Automaten $\mathcal{K}$.

Theorem 56
Zu jedem Kellerautomaten

$$\mathcal{K}_1 = (Q_1, T, K_1, q_0^1, F_1, s_0^1, \delta_1)$$

gibt es einen Kellerautomaten

$$\mathcal{K}_2 = (Q_2, T, K_2, q_0^2, F_2, s_0^2, \delta_2)$$

mit

1. *Q_2 ist einelementig*
2. *$L^K(\mathcal{K}_1) = L^K(\mathcal{K}_2)$*

Beweis: z.B. [29, Theorem 5, Seite 61].

$Q_2 = \{q^2\}$
$F_2 =$ irrelevant
$K_2 = \{[pAq] \mid p, q \in Q_1, A \in K_1\} \cup \{s_0^2\}$
$s_0^2 =$ initiales Kellersymbol

Kellererhaltende Übergänge
$\{(q^2, [X_1 s_1 Y][X_2 s_2 X_1] \dots [q s_k X_k]) \mid X_i \in K\}$
$\subseteq \delta_2(\alpha, q^2, [pAY])$
für alle $Y \in K, \alpha \in T \cup \{\varepsilon\}$
falls $(q, s_1 \dots s_k) \in \delta_1(\alpha, p, A)$ und $k > 0$

Kellerverkürzende Übergänge
$(q^2, \varepsilon) \in \delta_2(\alpha, q^2, [pAq])$
für alle $Y \in K, \alpha \in T \cup \{\varepsilon\}$
falls $(q, \varepsilon) \in \delta_1(\alpha, p, A)$

Initialisierung
$\{(q^2, [q_0^1 s_0^1 X]) \mid X \in K\} \subseteq \delta_2(\varepsilon, q^2, s^2)$

Theorem 57
Zu jedem Kellerautomaten $\mathcal{K}$ gibt es eine kontextfreie Grammatik $\mathcal{G}$ mit

$$L^K(\mathcal{K}) = L(\mathcal{G})$$

Beweisskizze: $\mathcal{K} = (\{q\}, T, K, q, F, s_0, \delta)$ mit $T \cap K = \emptyset$

$\mathcal{G}$	$\mathcal{K}$
Terminalzeichen T	Eingabealphabet T
Nichtterminalzeichen N	Kelleralphabet K
Grammatikregel $B \to aW^r$ $B \to W^r$	Automatenübergang für $(q, W) \in \delta(a, q, B)$ für $(q, W) \in \delta(\varepsilon, q, B)$

$W \in N^*$, $a \in T$.

4.2 Normalformen

Definition 74

1. $A \to \varepsilon$
   ε-Produktionen

2. $A \to B$ *für* $A, B \in N$
   Kettenproduktionen

3. *Produktionen mit* nutzlosen *Nichtterminalzeichen.*

Sei $\mathcal{G} = (N, T, S, \mathcal{P})$ *eine kontextfreie Grammatik.*
Ein Nichtterminalzeichen $A \in N$ *heißt* nützlich, *wenn es eine Ableitung*

$$S \Rightarrow^* w_1 A w_2 \Rightarrow^* w$$

gibt mit $w_i \in (N \cup T)^*$ *und* $w \in T^*$.
Ein Nichtterminalzeichen, das nicht nützlich ist, nennen wir nutzlos.

Definition 75
Eine Grammatik $\mathcal{G} = (N, T, S, \mathcal{P})$ *heißt* ε*-frei, wenn*

- $\varepsilon \notin L(\mathcal{G})$ *und* $\mathcal{P}$ *enthält keine* ε*-Produktion oder*
- $\varepsilon \in L(\mathcal{G})$ *und die einzige* ε*-Produktion in* $\mathcal{P}$ *ist* $S \to \varepsilon$*. In diesem Fall wird zusätzlich verlangt, daß das Startzeichen in keiner rechten Seite einer Produktion auftritt.*

Theorem 58
Zu jeder Grammatik $\mathcal{G}$ *gibt es eine äquivalente* ε*-freie Grammatik* $\mathcal{G}_{ef}$.

Bevor wir mit dem Beweis des Theorems beginnen können brauchen wir einen Hilfbegriff:

Definition 76
Für eine Grammatik $\mathcal{G} = (N, T, S, \mathcal{P})$ *ist die Menge der nullproduzierenden Nichtterminalzeichen* $Null_{\mathcal{G}}$ *definiert durch:*

$$Null_{\mathcal{G}} = \{A \in N \mid A \Rightarrow^* \varepsilon\}$$

Die Menge der nullproduzierenden Nichtterminalzeichen kann durch die folgende iterative Prozedur berechnet werden.

$$\begin{array}{lcl} Null_{\mathcal{G}}^0 & = & \{A \in N \mid (A \to \varepsilon) \in \mathcal{P}\} \\ Null_{\mathcal{G}}^{n+1} & = & Null_{\mathcal{G}}^n \cup \{A \in N \mid \text{es gibt eine Regel } (A \to C_1 \ldots C_k) \in \mathcal{P} \\ & & \text{mit } C_i \in Null_{\mathcal{G}}^n \text{ für alle } 1 \leq i \leq k\} \end{array}$$

Wegen der Endlichkeit von N gibt es ein n mit

$$Null_{\mathcal{G}}^{n+1} = Null_{\mathcal{G}}^n$$

Man überzeugt sich leicht davon, daß für ein solches n gilt

$$Null_{\mathcal{G}} = Null_{\mathcal{G}}^n$$

Beweis von Theorem 58:

Gegeben sei die kontextfreie Grammatik $\mathcal{G} = (N, T, S, \mathcal{P})$. Die Grammatik $\mathcal{G}_{ef}$ entsteht aus $\mathcal{G}$, indem für jede Produktion $A \to X_1 \dots X_n$ in $\mathcal{P}$ mit $n > 0$ alle Produktionen der Form $A \to \alpha_1 \dots \alpha_n$ in $\mathcal{P}_{ef}$ aufgenommen werden mit

1. $\alpha_i = X_i$ falls $X_i \notin Null_{\mathcal{G}}$,
2. $\alpha_i \in \{X_i, \varepsilon\}$ falls $X_i \in Null_{\mathcal{G}}$,
3. nicht alle α_i sind ε. Ausnahme: $X = S$.

Beispiel 46
Für $A \to ABCD$ aus $\mathcal{P}$ und $B, D \in Null_{\mathcal{G}}$ und $A, C \notin Null_{\mathcal{G}}$ werden die Produktionen

$$A \to ABCD, A \to ABC, A \to ACD, A \to AC$$

in $\mathcal{P}_{ef}$ aufgenommen.

Falls in $\mathcal{G}$ eine Produktion

$$S \to N_1 \dots N_k$$

existiert, so daß $N_i \in Null(\mathcal{G})$ für alle i, dann kommt $S \to \varepsilon$ als Produktion in $\mathcal{G}_{ef}$ vor ohne möglicherweise die zusätzliche Beschränkung an das Vorkommen von S zu erfüllen. In diesem Fall fügt man ein neues Startzeichen S_1 und die beiden Produktionen $S_1 \to \varepsilon$,und $S_1 \to S$ hinzu und läßt die Produktion $S \to \varepsilon$ weg.

∎

Beispiel 47
Sei $\mathcal{G}_e$ gegeben durch

$$\begin{array}{lcll}
N & = & \{E, E_1, T, T_1, F\} & \\
T & = & \{(,), +, *, i\} & \\
E & = & \textit{Startzeichen} & \\
\mathcal{P}_e & = & \{E \to TE_1 & E_1 \to +TE_1 \mid \varepsilon \\
 & & T \to FT_1 & T_1 \to *FT_1 \mid \varepsilon \\
 & & F \to i \mid (E)\} & \\
Null^0_{\mathcal{G}} & = & \{E_1, T_1\} & \\
 & = & Null^1_{\mathcal{G}} &
\end{array}$$

Die ε-freie Grammatik $\mathcal{G}_{ef}$ zu $\mathcal{G}_e$ ist

$$\begin{array}{lcll}
\mathcal{P}_{ef} & = & \{E \to T \mid TE_1 & E_1 \to +T \mid +TE_1 \\
 & & T \to F \mid FT_1 & T_1 \to *F \mid *FT_1 \\
 & & F \to i \mid (E)\} &
\end{array}$$

Lemma 59
Ist $\mathcal{G}$ eine ε-freie kontextfreie Grammatik, dann gilt für je zwei Wörter u,w mit $u \Rightarrow^* w$

$$|w| \geq |u|$$

Beweis : Übungsaufgabe. Man achte vor allem darauf, daß die Aussage des Lemmas nur richtig ist wegen der zusätzlichen Anforderung an des Auftreten des Startzeichens im Fall $\varepsilon \in L(\mathcal{G})$.

∎

Lemma 60
Zu jeder Grammatik $\mathcal{G}$ gibt es eine äquivalente Grammatik $\mathcal{G}_{ok}$ ohne Kettenproduktionen.
War $\mathcal{G}$ ε-frei, dann ist es $\mathcal{G}_{ok}$ auch.

Beweis: Zusätzliche Übungsaufgabe.

Lemma 61
Zu jeder kontextfreien Grammatik

$$\mathcal{G}_1 = (N_1, T, S, \mathcal{P}_1)$$

gibt es eine äquivalente, kontextfreien Grammatik

$$\mathcal{G}_2 = (N_2, T, S, \mathcal{P}_2),$$

so daß für jedes Nichtterminalzeichen $A \in N_2$ ein terminales Wort $w \in T^*$ existiert mit

$$A \Rightarrow^*_{\mathcal{G}_2} w$$

$\mathcal{G}_2$ kann effektiv aus $\mathcal{G}_1$ konstruiert werden.

Beweisskizze: Man definiert induktiv:

$$\begin{array}{lcl} NEWV_0 & = & \{A \in N_1 \mid \text{es gibt } (A \to w) \in \mathcal{P}_1 \\ & & \text{mit } w \in T^*\} \\ NEWV_{n+1} & = & \{A \in N_1 \mid \text{es gibt } (A \to w) \in \mathcal{P}_1 \\ & & \text{mit } w \in (T \cup NEWV_n)^*\} \end{array}$$

Wegen der Endlichkeit von N_1 gibt es ein n mit $NEWV_n = NEWV_{n+1}$. Setze:

$$\begin{array}{lcl} N_2 & = & NEWV_n \\ \mathcal{P}_2 & = & \{(A \to r) \in \mathcal{P}_1 \mid A \in N_2 \text{ und } r \in (N_2 \cup T)^*\} \end{array}$$

Lemma 62
Zu jeder kontextfreien Grammatik

$$\mathcal{G}_1 = (N_1, T_1, S, \mathcal{P}_1)$$

gibt es eine äquivalente, kontextfreien Grammatik

$$\mathcal{G}_2 = (N_2, T_2, S, \mathcal{P}_2),$$

so daß für jedes $X \in N_2 \cup T_2$ *ein* $w_1, w_2 \in (T_2 \cup N_2)^*$ *existieren mit*

$$S \Rightarrow^*_{\mathcal{G}_2} w_1 X w_2$$

Beweisidee:

$$\begin{array}{lcl} N^0 & = & \{S\} \\ N^{k+1} & = & \{B \in N_1 \mid \text{es gibt } (A \to w) \in \mathcal{P}_1 \\ & & \text{mit } A \in N^k \text{ und } B \text{ kommt in } w \text{ vor} \\ T^0 & = & \emptyset \\ T^{k+1} & = & \{a \in T_1 \mid \text{es gibt } (A \to w) \in \mathcal{P}_1 \\ & & \text{mit } A \in N^k \text{ und } a \text{ kommt in } w \text{ vor } \} \end{array}$$

Wieder gibt es ein k mit $N^k = N^{k+1}$. Setze

$$\begin{array}{lcl} N_2 & = & N^k \\ T_2 & = & T^k \\ \mathcal{P}_2 & = & \{(A \to w) \in \mathcal{P}_1 \mid A \in N_2\} \end{array}$$

Theorem 63
Zu jeder Grammatik $\mathcal{G}$ *gibt es eine äquivalente Grammatik* $\mathcal{G}_{nf}$ *mit*

1. $\mathcal{G}_{nf}$ *ist* ε*-frei,*
2. $\mathcal{G}_{nf}$ *besitzt keine Kettenproduktionen,*
3. $\mathcal{G}_{nf}$ *besitzt keine nutzlosen Nichtterminalzeichen.*

Definition 77
Eine Grammatik $\mathcal{G} = (N, T, S, \mathcal{P})$ *heißt eine* Chomsky-Normalform*, wenn*

1. $\mathcal{G}$ ε*-frei ist,*
2. *jede Produktion in* $\mathcal{P}$ *ist von der Form*

$$A \to BC$$

für Nichtterminalzeichen $A, B, C \in N$ *oder*

$$A \to a$$

für $a \in T$.

3. Als einzige Ausnahme darf $S \to \varepsilon$ vorkommen, falls $\varepsilon \in L(\mathcal{G})$ gilt. Dann darf S allerdings nie auf einer rechten Seite vorkommen.

4.2.1 Transformation auf Chomsky-Normalform

Sei $\mathcal{G}_0$ eine ε-freie Grammatik ohne Kettenproduktionen. $\mathcal{G}_1$ entstehe aus $\mathcal{G}_0$, indem jede Produktion

$$A \to r$$

mit $r \in (N \cup T)^*$ wobei $r \notin T$ ist, ersetzt wird durch

$$\begin{array}{ll} A \to r' & \\ C_a \to a & \text{für jedes } a \in T, \\ & \text{das in } r \text{ vorkommt} \end{array}$$

wobei r' aus r entsteht, indem jedes Terminalzeichen a, das in r vorkommt durch ein neues Nichtterminalzeichen C_a ersetzt wird.

Beispiel 48 (Beispiel für Schritt 1)
$\mathcal{G}_0$ sei:

$$\begin{array}{lcl} S & \to & ABC \\ C & \to & BaB \mid c \\ B & \to & b \mid bb \\ A & \to & a \end{array}$$

Dann ist $\mathcal{G}_1$:

$$\begin{array}{lcl} S & \to & ABC \\ C & \to & BA_1B \mid c \\ A_1 & \to & a \\ B & \to & b \mid B_1B_1 \\ B_1 & \to & b \\ A & \to & a \end{array}$$

Ersetze alle Produktionen

$$A \to B_1 \dots B_m$$

mit $m \geq 3$ durch

$$\begin{array}{lcl} A & \to & B_1C_1 \\ C_1 & \to & B_2C_2 \\ \vdots & \vdots & \vdots \\ C_{m-3} & \to & C_{m-2}B_{m-2} \\ C_{m-2} & \to & B_{m-1}B_m \end{array}$$

für neue Nichtterminalzeichen $C_1, \ldots C_{m-2}$.

Beispiel 49 (Beispiel für Schritt 2)
Ausgehend von $\mathcal{G}_1$:

$$\begin{array}{lcl} S & \to & ABC \\ C & \to & BA_1B \mid c \\ A_1 & \to & a \\ B & \to & b \mid B_1B_1 \\ B_1 & \to & b \\ A & \to & a \end{array}$$

Ergibt sich $\mathcal{G}_{CH}$:

$$\begin{array}{lcl} S & \to & AD \\ D & \to & BC \\ C & \to & BE \mid c \\ E & \to & A_1B \\ A_1 & \to & a \\ B & \to & b \mid B_1B_1 \\ B_1 & \to & b \\ A & \to & a \end{array}$$

4.2.2 Greibach-Normalform

Definition 78
Eine Grammatik $\mathcal{G} = (N, T, S, \mathcal{P})$ heißt eine Greibach-Normalform, *wenn alle Produktionen in $\mathcal{P}$ von der Form*

$$A \to aW$$

sind, wobei $a \in T$ und $W \in N^*$.

Theorem 64
Zu der Grammatik $\mathcal{G}$ gibt es eine äquivalente Greibach-Normalform $\mathcal{G}_{grb}$.

4.3 Pumping Lemma

Theorem 65
Zu jeder kontextfreien Sprache L gibt eine Zahl $n \in \mathbb{N}$, so daß jedes Wort $z \in L$ mit $|z| > n$ zerlegt werden kann in

$$z = uvwxy$$

mit

1. $|vx| > 0$ *und*
2. $|vwx| \leq n$ *und*
3. $uv^iwx^iy \in L$ *für alle* $i \in \mathbb{N}$

Lemma 66
Sei $\mathcal{G} = (N, T, S, \mathcal{P})$ eine kontextfreie Grammatik und m die Länge der längsten rechten Seite einer Produktion in $\mathcal{P}$. Sei H ein Syntaxbaum für ein Wort $w \in L(\mathcal{G})$.

Falls jeder Pfad in H eine Länge $\leq j$ hat, dann gilt

$$|w| \leq m^j$$

Beispiel 50
Die Sprache

$$L_{nnn} = \{a^n b^n c^n \mid n \geq 0\}$$

ist nicht kontextfrei.

Sei $T = \{a_1, \ldots, a_k\}$ eine Menge von Terminalzeichen. Wir haben schon die Funktion

$$\#_{a_i}(w) = \text{Anzahl der } a_i \text{ in } w$$

von T^* nach $\mathbb{N}$ benutzt. Wir definieren die vektor-wertige Funktion $\psi : T^* \to \mathbb{N}^k$

$$\psi(w) = (\#_{a_1}(w), \ldots, \#_{a_k}(w))$$

Für $L \subseteq T^*$ ist $\psi(L)$ eine Menge von Vektoren:

$$\psi(L) = \{\psi(w) \mid w \in L\}$$

ψ heißt die **Parikh-Abbildung** zu T.

Beispiel 51

$$\begin{array}{lcl}
L_1 & = & \{a^m b^m \mid m \geq 0\} \\
L_2 & = & \{(ab)^m \mid m \geq 0\} \\
L_3 & = & \{a^i b^i c^j \mid i,j \geq 0\} \cup \{a^i b^j c^j \mid i,j \geq 0\} \\
 & = & \{a^i b^j c^k \mid i,j,k \geq 0 \textit{ und} \\
 & & (i=j) \textit{ oder } (j=k)\} \\
\psi(L_1) & = & \{(m,m) \mid m \geq 0\} \\
\psi(L_2) & = & \{(m,m) \mid m \geq 0\} \\
\psi(L_3) & = & \{(i,j,k) \mid i,j,k \geq 0 \textit{ und} \\
 & & (i=j) \textit{ oder } (j=k)\}
\end{array}$$

Eigenschaften von ψ:

$$\begin{array}{lcl}
\psi(\varepsilon) & = & (0,0,\ldots,0) \\
\psi(w_1 w_2) & = & \psi(w_1) + \psi(w_2)
\end{array}$$

Definition 79

1. *Eine Menge $M \subseteq \mathbb{N}^k$ von Vektoren heißt* **linear**, *wenn es Vektoren $v_0, v_1, \ldots, v_n \in \mathbb{N}^k$, so daß jedes $v \in M$ sich schreiben lässt als*

$$v = v_0 + \Sigma_{i=1}^n x_i v_i$$

   für geeignete Koeffizienten $x_i \in \mathbb{N}$.

2. *Eine Menge $M \subseteq \mathbb{N}^k$ heißt* **semilinear**, *wenn sie Vereinigung endlich vieler linearer Mengen ist.*

Beispiel 52

Die Menge

$$\psi(L_3) = \{(i,j,k) \mid i,j,k \geq 0 \textit{ und } (i=j) \textit{ oder } (j=k)\}$$

ist semilinear.

Setze:

$$\begin{array}{lclclclclcl}
v_0 & = & (0,0,0,) & v_1 & = & (1,1,0), & v_2 & = & (0,0,1) \\
v'_0 & = & (0,0,0), & v'_1 & = & (1,0,0), & v'_2 & = & (0,1,1)
\end{array}$$

Es gilt:

$$\begin{array}{lcl}
\psi(L_3) & = & \{v_0 + iv_1 + kv_2 \mid i,k \geq 0\} \cup \{v'_0 + iv'_1 + \\
 & & kv'_2 \mid i,k \geq 0\}
\end{array}$$

Theorem 67

Für jede kontextfreie Sprache L ist $\psi(L)$ semilinear.

4.4 Abschlußeigenschaften

4.4.1 Die Automatenhierarchie

Automatenklasse	Sprachklasse
deterministische endliche Automaten	reguläre Sprachen
nichtdeterministische endliche Automaten	reguläre Sprachen
nichtdeterministische Kellerautomaten	kontextfreie Sprachen
deterministische Kellerautomaten	deterministische kontextfreie Sprachen
linear beschränkte Automaten	kontextsensitive Sprachen
nichtdeterministische Turingmaschinen	rekursiv aufzählbare Sprachen
deterministische Turingmaschinen	rekursiv aufzählbare Sprachen

Definition 80
Linear beschränkte Automaten *sind wie Turingmaschinen, mit der Ausnahme, daß sie nur auf dem Abschnitt des Bandes arbeiten können, das zu Beginn von der Eingabe belegt ist.*
Das Eingabealphabet enthält dazu zwei Spezialsymbole E_l, E_r. Eine Eingabewort w wird als

$$E_l w E_r$$

auf das Band geschrieben. Der Schreib- und Lesekopf kann nicht über die Endmarkierungen hinaus bewegt werden, noch können E_l, E_r überschrieben werden.

4.4.2 Abschlußeigenschaften formaler Sprachen

Theorem 68

1. *Sei $\mathcal{G}$ eine beliebige Grammatik mit Terminalzeichenmenge T.*
   Dann ist $L(\mathcal{G})$ eine rekursiv aufzählbare Teilmenge von T^*.

2. *Ist $L \subseteq T^*$ eine beliebige rekursiv aufzählbare Menge, dann gibt es eine Grammatik $\mathcal{G}$ mit*
$$L(\mathcal{G}) = L.$$

Beweis: siehe [29, Theorem 5.3.4, Seite 94]

Sprachklasse Operation	reg	ktf	kts	ra	rek
Produkt	ja	ja	ja	ja	ja
*	ja	ja	ja	ja	ja
Vereinigung	ja	ja	ja	ja	ja
Durchschnitt	ja	nein	ja	ja	ja
Komplement	ja	nein	ja	nein	ja
Substitution	ja	ja			
Homomorphismus	ja	ja			

reg = regulär
ktf = kontextfrei
kts = kontextsensitive
ra = rekursiv aufzählbar
rek = rekursiv

Beweise:

Abgeschlossenheit unter $\cup$

Seien

$$\begin{aligned} \mathcal{G}_1 &= (N_1, T, S_1, \mathcal{P}_1) \\ \mathcal{G}_2 &= (N_2, T, S_2, \mathcal{P}_2) \end{aligned}$$

Grammatiken, wobei o.B.d.A. $N_1 \cap N_2 = \emptyset$. Setze

$$\begin{aligned} N_3 &= N_1 \cup N_2 \cup \{S\} \\ T &= \text{Terminalzeichen} \\ S &= \text{Startsymbol} \\ \mathcal{P}_3 &= \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{S \to S_1, S \to S_2\} \end{aligned}$$

Dann gilt für $\mathcal{G}_3 = (N_3, T, S, \mathcal{P}_3)$

$$L(\mathcal{G}_3) = L(\mathcal{G}_1) \cup L(\mathcal{G}_2)$$

Sind beide $\mathcal{G}_i$ kontextfrei, kontextsensitiv, dann ist auch $\mathcal{G}_3$ entsprechend kontextfrei, kontextsensitiv.

Sind beide $\mathcal{G}_i$ rechtslinear, dann ist $\mathcal{G}_3$ nicht rechtslinear. Die Regularität von $L(\mathcal{G}_1) \cup L(\mathcal{G}_2)$ war in diesem Fall schon früher gezeigt worden.

Abgeschlossenheit unter Verkettung für kontextfreie Sprachen

Seien $\mathcal{G}_1 = (N_1, T, S_1, \mathcal{P}_1)$, $\mathcal{G}_2 = (N_2, T, S_2, \mathcal{P}_2)$ kontextfreie Grammatiken, wobei o.B.d.A. $N_1 \cap N_2 = \emptyset$. Setze

$$\begin{array}{lcl} N_3 & = & N_1 \cup N_2 \cup \{S\} \\ T & = & \text{Terminalzeichen} \\ S & = & \text{Startsymbol} \\ \mathcal{P}_3 & = & \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{S \to S_1 S_2\} \cup \end{array}$$

Dann gilt für $\mathcal{G}_3 = (N_3, T, S, \mathcal{P}_3)$

$$L(\mathcal{G}_3) = L(\mathcal{G}_1)L(\mathcal{G}_2)$$

$\mathcal{G}_3$ ist ebenfalls kontextfrei.
Sind beide $\mathcal{G}_i$ rechtslinear, dann wurde die Regularität von $L(\mathcal{G}_1) \cup L(\mathcal{G}_2)$ schon früher gezeigt.

Was geht schief im kontextsensitiven Fall?

Seien

$$\begin{array}{lcl} \mathcal{G}_1 & = & (\{S_1\}, \{b\}, S_1, \{S_1 \to b\} \\ \mathcal{G}_2 & = & (\{S_2\}, \{b\}, S_2, \{bS_2 \to bb\} \end{array}$$

Es gilt $L(\mathcal{G}_1) = \{b\}$ und $L(\mathcal{G}_2) = \emptyset$. Für die verkettete Grammatik $\mathcal{G}_3$ mit:

$$\begin{array}{lcl} N_3 & = & \{S, S_1, S_2\} \\ T_3 & = & \{b\} \\ \mathcal{P}_3 & = & \{S \to S_1 S_2, S_1 \to b, bS_2 \to bb\} \end{array}$$

gilt aber

$$L(\mathcal{G}_3) = \{bb\},$$

wegen

$$S \to S_1 S_2 \to bS_2 \to bb \ .$$

Somit gilt

$$L(\mathcal{G}_3) \neq L(\mathcal{G}_1)L(\mathcal{G}_2) = \emptyset$$

Abgeschlossenheit unter Verkettung allgemeiner Fall

Seien $\mathcal{G}_1 = (N_1, T, S_1, \mathcal{P}_1)$, $\mathcal{G}_2 = (N_2, T, S_2, \mathcal{P}_2)$ Grammatiken, wobei o.B.d.A. $N_1 \cap N_2 = \emptyset$. Setze

$$\begin{array}{lcl}
X_t^i & = & \text{neue NT-Symbole für jedes } t \in T \\
X^1 & = & \{X_t^1 \mid t \in T\} \\
X^2 & = & \{X_t^2 \mid t \in T\} \\
N_3 & = & N_1 \cup N_2 \cup X^1 \cup X^2 \cup \{S\} \\
T & = & \text{Terminalzeichen} \\
S & = & \text{Startsymbol} \\
\mathcal{P}_3 & = & \mathcal{P}_1' \cup \mathcal{P}_2' \cup \{S \to S_1 S_2\} \cup \\
 & & \{X_t^i \to t \mid i \in \{1,2\}, t \in T\}
\end{array}$$

Dann gilt für $\mathcal{G}_3 = (N_3, T, S, \mathcal{P}_3)$

$$L(\mathcal{G}_3) = L(\mathcal{G}_1) L(\mathcal{G}_2)$$

$\mathcal{P}_i'$ entsteht aus $\mathcal{P}_i$, indem jedes $t \in T$ ersetzt wird durch X_t^i. Sind beide $\mathcal{G}_i$ kontextsensitiv, dann auch $\mathcal{G}_3$.

Abgeschlossenheit unter Iteration für kontextfreie Sprachen

Seien $\mathcal{G}_1 = (N_1, T, S_1, \mathcal{P}_1)$ eine kontextfreie Grammatik. Definiere $\mathcal{G}_2$ mit:

$$\begin{array}{lcl}
N_2 & = & \{S, s'\} \cup N_1 \\
T_2 & = & N_1 \\
\mathcal{P}_2 & = & \mathcal{P}_1 \cup \{S \to \varepsilon \mid S', \\
 & & S' \to S_1 \mid S_1 S'\}
\end{array}$$

Man zeigt leicht, daß

$$L(\mathcal{G}_2) = (L(\mathcal{G}_1))^*$$

gilt. Im kontextsensitiven Fall benutzt man dieselbe Idee, wie für die Verkettung.

Durchschnitte kontextfreier Sprachen Gegenbeispiel

Die beiden Sprachen

$$L_1 = \{a^j b^n c^n \mid j, n \geq 0\}$$

und

$$L_2 = \{a^n b^n c^j \mid j, n \geq 0\}$$

sind kontextfrei. Ihr Durchschnitt

$$L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$$

ist es nicht. Somit kann die Klasse der kontextfreien Sprachen auch nicht und Komplementbildung abgeschlossen sein.

Durchschnitte kontextsensitiver Sprachen

Die Frage, ob die Klasse der kontextsensitiven Sprachen unter Komplementen abgeschlossen ist wurde 1964 zum ersten Mal explizit gestellt wurde und erst 1988 unabhängig von

> Neil Immermann,
> Nondeterministic Space is Closed Under Complementation,
> SIAM J. Computing, 17, 935–938, 1988,

und

> R. Szelepcsényi,
> The Method of Forced Enumerations for Nondeterministic Automata,
> Acta Informatica, 26, 279–284, 1988.

gelöst wurde. Die Abgeschlossenheit unter Durchschnitten folgt dann wegen $A \cap B = c(c(A) \cup c(B))$.

4.5 Entscheidbarkeitseigenschaften

Sprachklasse Problem	reg	ktf	kts	ra	rek
$w \in L(\mathcal{G})$	e	e	e	u	e
$L(\mathcal{G}) = \emptyset$	e	e	u	u	u
$L(\mathcal{G})$ endlich	e	e	u	u	u
$L(\mathcal{G}_1) = L(\mathcal{G}_2)$	e	u	u	u	u
$L(\mathcal{G}_1) \subseteq L(\mathcal{G}_2)$	e	u	u	u	u

reg = regulär
ktf = kontextfrei
kts = kontextsensitive
ra = rekursiv aufzählbar
rek = rekursiv

e = entscheidbar
u = unentscheidbar

4.5.1 Das Problem $w \in L(\mathcal{G})$?

Ist $\mathcal{G}$ eine kontextsensitive Grammatik, so können wir für jede Produktion $r \to l$ von $\mathcal{G}$ annehmen, daß $|r| \leq |l|$ gilt. Zu einem gegebenen Wort $w \in T^*$ kann man also systematische alle Ableitungen von Wörtern der Länge $\leq |w|$ durchprobieren. Also ist $w \in L(\mathcal{G})$ entscheidbar für kontextsensitive Grammatiken und damit auch reguläre und kontextfreie. Für Typ-0-Grammatiken entspricht $w \in L(\mathcal{G})$ dem Elementproblem für rekursiv aufzählbare Mengen und ist damit unentscheidbar.

4.5.2 $L(\mathcal{G}) \neq \emptyset$?

Bei der Elimination nutzloser Nichtterminalzeichen war ein Verfahren zur Berechnung der Menge

$$NEWV = \{A \in N \mid \text{es gibt } w \in T^* \text{ mit } A \Rightarrow^* w\ \}$$

angegeben worden. Offensichtlich gilt:

$$L(\mathcal{G}) \neq \emptyset \Leftrightarrow S \in NEWV$$

Für jedes $n \in \mathbb{N}$ sei

$$N^n = \{A \in N \mid \exists w \in T^* \text{ mit } A \Rightarrow^* w \text{ und } |w| \geq n\}$$

$$\begin{array}{lcl} N_0^n & = & \{A \in N \mid \text{es gibt } A \to w \in \mathcal{P} \text{ mit } |w| \geq n\} \\ N_{j+1}^n & = & \{A \in N \mid \text{es gibt } A \to w \in \mathcal{P} \text{ mit Anzahl der Terminalzeichen in } w = n_0, \\ & & A_1, \ldots, A_k \text{ sind alle NT-Zeichen in } w, \\ & & \text{es gibt } n_1, \ldots n_k \leq n \text{ mit } n_0 + n_1 + \ldots + n_k = n \text{ und für alle } i \text{ gilt } A_i \in N_j^{n_i}\} \end{array}$$

Es gibt j mit $N_j^n = N_{j+1}^n = N^n$.

$$w \in L(\mathcal{G}) \text{ mit } |w| \geq n \Leftrightarrow S \in N^n$$

4.5.3 $L(\mathcal{G})$ endlich?

Sei n die Konstante aus dem pumping-Lemma für kontextfreie Sprachen. $L(\mathcal{G})$ ist genau dann unendlich, wenn es ein $w \in L(\mathcal{G})$ gibt mit $|w| \geq n$. Das kann in endlich vielen Schritten entschieden werden.

4.6 Unentscheidbare Probleme für Grammatiken

Zum Beweis der in der Übersichtstabelle enthaltenen Unentscheidbarkeitsresultate verweisen wir auf:

1. Hopcroft und Ullman, Introduction to Automata Theory, Languages and Computations, p. 201 ff
2. Salomaa, Formale Sprachen, Kapitel VIII.

Theorem 69 *Das Problem, von einer beliebigen kontextfreien Grammatik zu entscheiden, ob sie eindeutig ist, ist unentscheidbar.*

Beweis: Angenommen es gäbe einen Algorithmus $\mathcal{A}$, der für beliebige kontextfreie Grammatiken entscheidet, ob sie eindeutig sind oder nicht. Wir werden einen Algorithmus $\mathcal{B}$ angeben, der das Postsche Korrespondenzproblem löst. Das ist ein Widerspruch, da das PKP unentscheidbar ist. Zu einer Instanz des PKP

$$P = \{(x_i, y_i) \mid 1 \leq i \leq n\}$$

mit $x_i, y_i \in (\Sigma^* \setminus \{\varepsilon\})$ konstruieren wir die kontextfreie Grammatik G_P:

Terminalzeichen	$\Sigma \cup \{a_1, \ldots, a_n\}$
NTzeichen	$\{S, S_1, S_2\}$
Produktionen	$S ::= S_1 \mid S_2$
	$S_1 ::= x_1 S_1 a_1 \mid \ldots \mid x_n S_1 a_n$
	$S_1 ::= x_1 a_1 \mid \ldots \mid x_n a_n$
	$S_2 ::= y_1 S_2 a_1 \mid \ldots \mid y_n S_2 a_n$
	$S_2 ::= y_1 a_1 \mid \ldots \mid y_n a_n$

Man sieht leicht, daß gilt

$$\begin{array}{lcl} L(G_P) & = & \{x_{i_1} \ldots x_{i_k} a_{i_k} \ldots a_{i_1} \mid k \geq 1, 1 \leq i_j \leq n\} \\ & & \cup \\ & & \{y_{i_1} \ldots y_{i_k} a_{i_k} \ldots a_{i_1} \mid k \geq 1, 1 \leq i_j \leq n\} \end{array}$$

Die Grammatik G ist genau dann mehrdeutig, wenn es ein $k \geq 1$ und $i_j \in \{1, \ldots, n\}$ gibt mit

$$x_{i_1} \ldots x_{i_k} a_{i_k} \ldots a_{i_1} = y_{i_1} \ldots y_{i_k} a_{i_k} \ldots a_{i_1}$$

Diese Gleichung ist gleichbedeutend mit

$$x_{i_1} \ldots x_{i_k} = y_{i_1} \ldots y_{i_k}$$

Zusammenfassend gilt: G ist mehrdeutig gdw das Postsche System lösbar ist.

4.7 Lindenmayer Systeme

Literatur: [32].

Definition 81
Sei V ein Alphabet. Ein OL-System $L = (V, P, w)$ besteht aus

- *einer Menge P von Produktionsregeln der Form $a \to u$ mit $a \in V$ und $u \in V^*$.*
- *einem Anfangswort $w \in V^+$.*

Außerdem wird verlangt, daß für jedes $a \in V$ eine Produktion mit a als linker Seite existiert. Gegebenenfalls wird durch $a \to a$ ergänzt.

Ein OL-System, so daß für jedes $a \in V$ genau eine Produktion der Form $a \to u$ existiert heißt ein deterministisches kontextfreies Lindenmayer-System, DOL-System.

Direkte Ableitbarkeit $u \Rightarrow_L v$ in $L = (V, P, w)$:

$$\begin{array}{lcll} u & = & a_1 \dots a_k & a_i \in V \\ \multicolumn{4}{l}{\text{es gibt Produktionen aus } P} \\ a_i & \to & v_i & \text{für } 1 \leq i \leq k \\ v & = & v_1 \dots v_k & \end{array}$$

Ein $v \in V^*$ ist ableitbar in L,

$$w \Rightarrow_L^* v$$

wenn es $v_0 = w$, v_1, $\dots$, $v_n = v$ gibt mit

$$v_i \Rightarrow_L v_{i+1} \text{ für alle } 0 \leq i < n$$

$S(L) = \{u \in V^* \mid w \Rightarrow_L^* u\}$

Beispiel 53 (Ein DOL-System)
Produktionsregeln:

$$\begin{array}{lcl} b & \to & a \\ a & \to & ab \end{array}$$

Der Zustand des Graphiksystems ist gegeben durch die Koordinaten (x, y) und einen Winkel α, der die Richtung angibt.

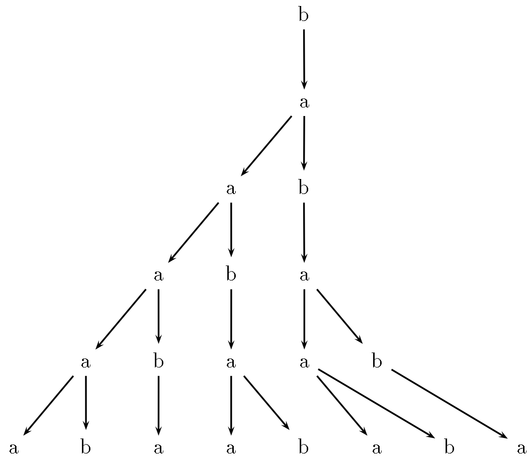


Abbildung 4.2: Ableitung zu Beispiel 53

F der Zustand (x_0, y_0, α) geht über in (x_1, y_1, α) mit $x_1 = x_0 + dcos(\alpha)$, $y_1 = y_0 + dsin(\alpha)$ und es wird eine Strecke von (x_0, y_0) nach (x_1, y_1) gezeichnet.

+ Übergang von (x, y, α) nach $(x, y, \alpha + \delta)$

- Übergang von (x, y, α) nach $(x, y, \alpha - \delta)$

f wie **F** jedoch ohne Zeichnen.

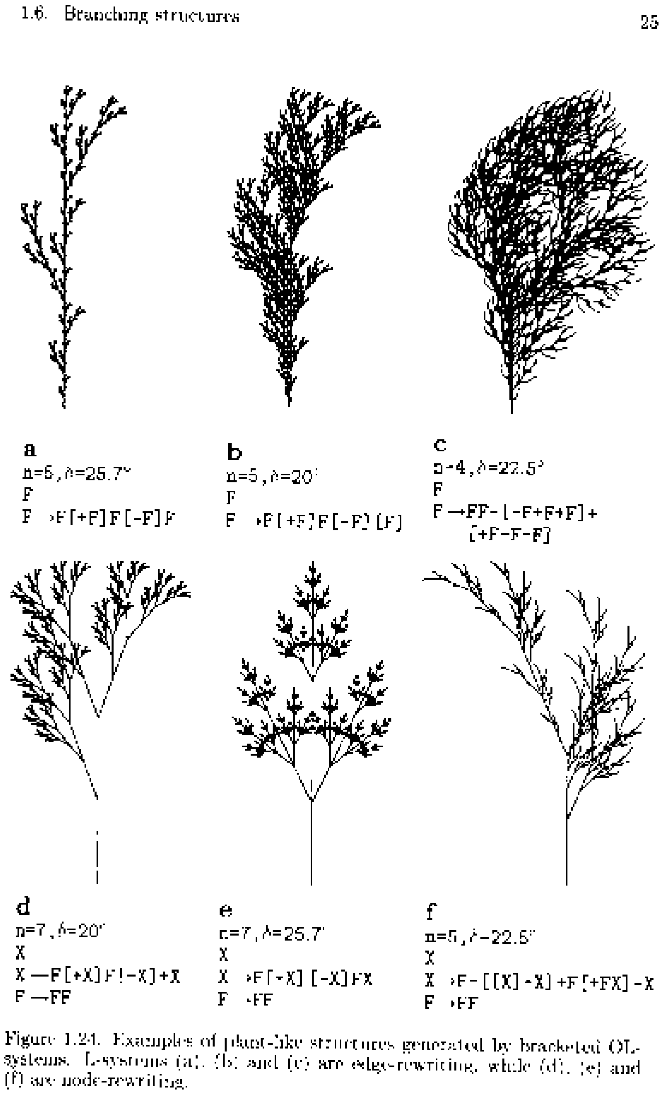
Beispiel 54 (Ein DOL-System für Schildkrötengraphik)
L_{Koch}

$$\begin{array}{lcl} w & = & F - F - F - F \\ P & = & F - F + F + FF - F - F + F \end{array}$$

Zu den Anweisungen an das Graphiksystem werden die folgenden beiden hinzugenommen:

[Speichere den aktuellen Zustand. Keine Zeichenaktion.

] Ersetze den aktuellen Zustand durch den zuletzt gespeicherten. Keine Zeichenaktion.


1.6. Branching structures 25

a
n=5, δ=25.7°
F
F → F[+F]F[-F]F

b
n=5, δ=20°
F
F → F[+F]F[-F][F]

c
n=4, δ=22.5°
F
F → FF-[-F+F+F]+[+F-F-F]

d
n=7, δ=20°
X
X → F[+X]F[-X]+X
F → FF

e
n=7, δ=25.7°
X
X → F[+X][-X]FX
F → FF

f
n=5, δ=22.5°
X
X → F-[[X]+X]+F[+FX]-X
F → FF

Figure 1.24. Examples of plant-like structures generated by bracketed OL-systems. L-systems (a), (b) and (c) are edge-rewriting, while (d), (e) and (f) are node-rewriting.

Abbildung 4.3: Durch Lindenmayer Systeme generierte Bilder

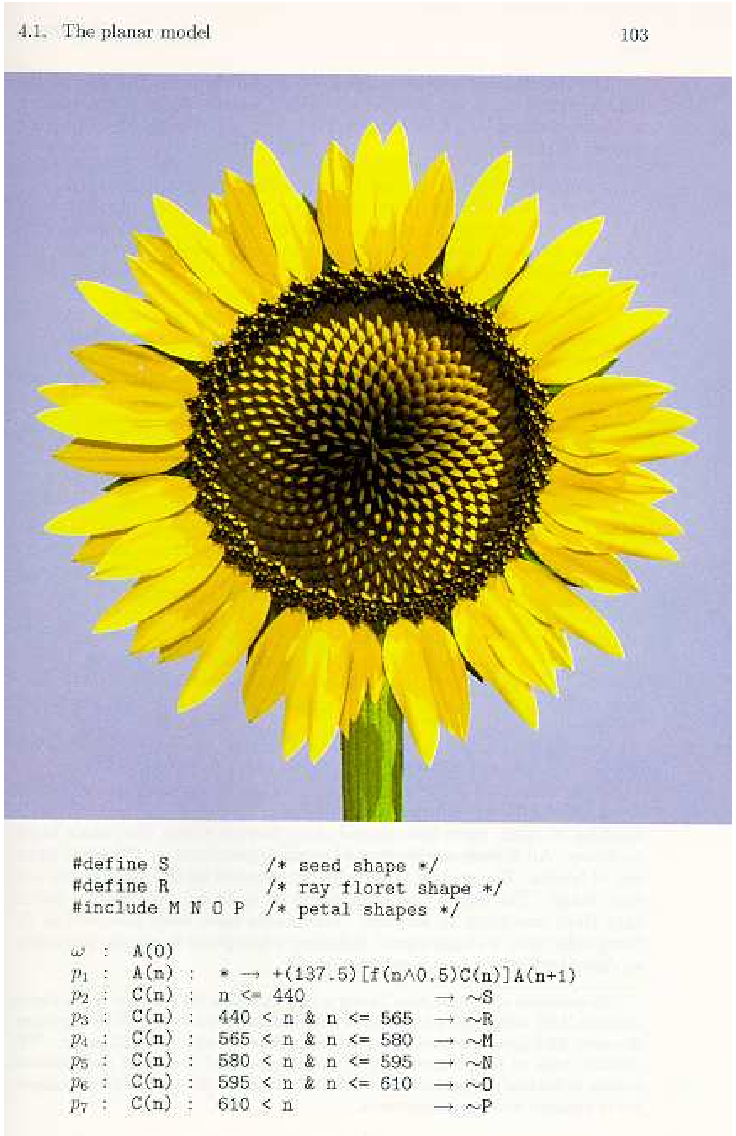
26 Chapter 1 Graphical modeling using L-systems

$n=7$, $\delta=22.5^\circ$

```
ω  :  A
p1 :  A → [&FL!A]/////'[&FL!A]///////'[&FL!A]
p2 :  F → S ///// F
p3 :  S → F L
p4 :  L → ['''∧∧{-f+f+f-|-f+f+f}]
```

Figure 1.25: A three-dimensional bush-like structure

Abbildung 4.4: Durch Lindenmayer Systeme generierte Bilder


4.1. The planar model 103

```
#define S            /* seed shape */
#define R            /* ray floret shape */
#include M N O P     /* petal shapes */

ω  :  A(0)
p1 :  A(n) :  * → +(137.5)[f(n^0.5)C(n)]A(n+1)
p2 :  C(n) :  n <= 440               → ~S
p3 :  C(n) :  440 < n & n <= 565     → ~R
p4 :  C(n) :  565 < n & n <= 580     → ~M
p5 :  C(n) :  580 < n & n <= 595     → ~N
p6 :  C(n) :  595 < n & n <= 610     → ~O
p7 :  C(n) :  610 < n                → ~P
```

Abbildung 4.5: Durch Lindenmayer Systeme generierte Bilder

4.8 Lehrbücher

- [35] und [36]
  Wohl die umfassendste Darstellung der Theorie der LL- und LR-Grammatiken in neuester Zeit. Anspruchsvolle Lektüre. Kann nur den theoretisch Interessierten als weiterführende Literatur empfohlen werden.
- [42]

4.9 Übungsaufgaben

Übungsaufgabe 4.9.1

Zeigen Sie, daß $L_2 = \{a^p \mid p \text{ Primzahl}\}$ nicht kontextfrei ist.

Übungsaufgabe 4.9.2

Definition 82 (lineare Grammatik, lineare Sprache)
Eine kontextfreie Grammatik $G = (N, T, S, P)$ heißt linear*, wenn alle Produktionen von der Form*

$$X \to aY, X \to Ya, X \to a, X \to \epsilon$$

mit $X, Y \in N, a \in T$ sind. Eine Sprache L heißt linear*, wenn es eine lineare Grammatik gibt, die L erzeugt.*

(a) Geben Sie eine lineare Grammatik für die Sprache $\{a^n b^n \mid n \in \mathbb{N}_0\}$ an.

(b) Zeigen Sie, daß jede reguläre Sprache linear ist.

(c) Beweisen Sie das folgende Pumping-Lemma für lineare Sprachen: Für jede lineare Sprache L gibt es eine Konstante $n \in \mathbb{N}$, so daß für alle $z \in L$ mit $|z| \geq n$ eine Zerlegung $z = uvwxy$, $|uvxy| \leq n, |vx| \geq 1$ existiert mit der Eigenschaft, daß für alle $i \in \mathbb{N}_0$ auch uv^iwx^iy zu L gehört.

(d) Weisen Sie mit diesem Pumping-Lemma nach, daß die Sprache $L = \{a^n b^n c^m d^m \mid n, m \in \mathbb{N}_0\}$ nicht linear ist.

(e) Zeigen Sie, daß $L = \{a^n b^n c^m d^m \mid n, m \in \mathbb{N}_0\}$ kontextfrei ist.

Übungsaufgabe 4.9.3

(a) Betrachten Sie Grammatiken folgenden Typs:

Definition 83 (Gleich-lineare Grammatiken und Sprachen)
Eine gleich-lineare *Grammatik $G = (N, T, S, P)$ ist eine kontextfreie Grammatik, bei der die Produktionen in P nur von folgender Form sind:*

$$X \to aYb, X \to a \textit{ oder } X \to \epsilon \textit{ mit } a, b \in T \textit{ und } X, Y \in N$$

Eine Sprache L heißt gleich-linear, *wenn es eine gleich-lineare Grammatik G mit $L = L(G)$ gibt.*

Geben Sie eine gleich-lineare Grammatik für die Sprache $\{a^n b^n \mid n \geq 0\}$ an.

(b) Betrachten Sie folgende Version des Pumpinglemmas für gleich-lineare Sprachen.

Theorem 70 (Pumpinglemma für gleich-lineare Sprachen)
Sei L eine gleich-lineare Sprache. Dann gibt es ein $n \in \mathbb{N}$, so daß für jedes $z \in L$ mit $|z| \geq 2n$ eine Zerlegung $z = uvwxy$ existiert mit

(i) $|v| = |x| > 0$,

(ii) $|uv| = |xy| \leq n$ *und*

(iii) für alle $i \in \mathbb{N}$ gilt $uv^i wx^i y \in L$.

Zeigen Sie mit dem Pumpinglemma für gleich-lineare Sprachen, daß

$$L = \{a^n b^n c^m \mid n, m \geq 0\}$$

nicht gleich-linear ist.

(c) Beweisen Sie das Pumpinglemma für gleich-lineare Sprachen.

(d) Geben Sie eine kontextfreie Grammatik G mit $L(G) = \{a^n b^n c^m \mid n, m \geq 0\}$ an.

(e) Ist der Index für eine gleich-lineare Sprache immer endlich?

Übungsaufgabe 4.9.4

Definition 84
Für Wörter $w \in \{0,1\}^*$ ist der Operator ^{I} wie folgt rekursiv definiert:

$$\begin{aligned}&\epsilon^I = \epsilon,\\ &(0w)^I = w^I 1, \quad (1w)^I = w^I 0\end{aligned}$$

Betrachten Sie die Sprache $L = \{w \in \{0,1\}^* \mid w = w^I\}$

(a) Geben Sie eine kontextfreie Grammatik G mit $L(G) = L$ an.

(b) Beweisen Sie, daß $L = L(G)$ gilt.

(c) Beweisen Sie, daß L nicht regulär ist.

Übungsaufgabe 4.9.5

Betrachten Sie folgende kontextfreie Grammatik $G = (\{S\}, \{a,b\}, S, P)$ mit

$$P = \{S \to aSbb \mid \epsilon\} \ .$$

Es gilt offenbar $L(G) = \{a^n b^{2n} \mid n \geq 0\}$.

(a) Zeigen Sie, daß G SLL(1) ist.

(b) Ist G eindeutig? Mit Begründung.

(c) Beweisen Sie mit vollständiger Induktion über die Länge von Ableitungen, daß $L(G) \subseteq L$ gilt.

(d) Zeigen Sie, daß L nicht regulär ist.

(e) Zeigen Sie, daß $LL := \{a^n b^{2n} c^k d^{2k} \mid n, k \geq 0\}$ kontextfrei ist.

(f) $L_1, L_2 \subseteq \Sigma^*$ seien kontextfreie Sprachen. Ist $L_1 - L_2$ immer kontextfrei? Begründen Sie Ihre Antwort.

Übungsaufgabe 4.9.6

Welche der folgenden Aussagen sind richtig und welche sind falsch? Bitte kreuzen Sie die richtigen Kästchen an!

Es gibt kontextfreie Sprachen mit endlichem Index ◯ Ja ◯ Nein

4.10 Lösungen zu den Aufgaben

Lösung zu Aufgabe 4.9.1

Annahme L_2 ist kontextfrei. Dann gibt es ein $n \in \mathbb{N}$ und eine Primzahl $p > n+2$ und eine Zerlegung $uvwxy$ von a^p mit $|vx| \geq 1, |vwx| \leq n$. Es gilt $i := |u| + |w| + |y| \geq 2$. Nach dem Pumping-Lemma müßte $|uv^iwx^iy| = i + i(|v| + |x|) = i(1 + |v| + |x|)$ eine Primzahl sein, was es aber nicht ist.

Also ist L_2 nicht kontextfrei.

Lösung zu Aufgabe 4.9.2

(a)
$$G = (\{X, Y\}, \{a, b\}, X, \{X \to \epsilon \mid aY, Y \to Xb \mid b\})$$

(b) Reguläre Sprachen werden von den Grammatiken (N, T, S, P) mit Produktionen von der Form

$$X \to wY, X \to w \text{ mit } X, Y \in N, w \in T^*$$

erzeugt.

Beh.: Jede Grammatik G obiger Form läßt sich in äquivalente lineare Grammatik G' mit Produktionen von der Form

$$X' \to aY', X' \to a, X' \to \epsilon$$

transformieren.

Ohne Einschränkung enthalte G keine Kettenproduktionen oder unnütze Nichtterminalzeichen

$p \in P, n \geq 2, a_i \in T, i = 1, \ldots, n$

- Betrachte Produktionen $p = X \to a_1 \ldots a_n$ und ersetze p durch

$$\begin{aligned} &X \to a_1X_1, \\ &X_1 \to a_2X_2 \\ &\vdots \\ &X_n \to a_n \end{aligned}$$

wobei X_i neue Nichtterminalzeichen sind.

Offenbar gilt für die daraus resultierende Grammatik G': $L(G') = L(G)$.

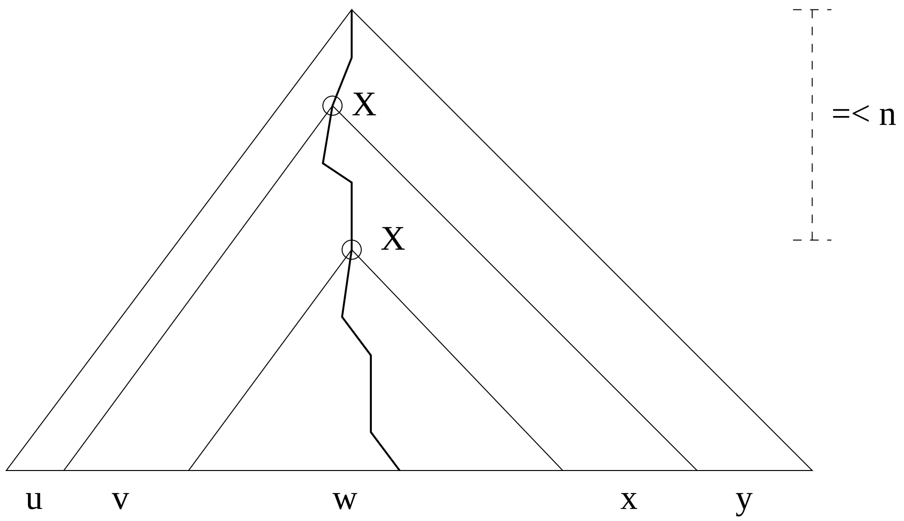
- Betrachte Produktionen $p = X \to a_1 \dots a_n Y$
  Ersetze p durch

$$
\begin{aligned}
&X \to a_1 X_1,\\
&X_1 \to a_2 X_2\\
&\vdots\\
&X_n \to a_n Y
\end{aligned}
$$

  wobei X_i neue Nichtterminalzeichen sind.

Die so konstruierte Grammatik ist offenbar linear.

(c) Wir benutzen zum Beweis die Beweisidee zum Pumpinglemmas für kontextfreie Sprachen. Wähle $n = |N|$ und betrachte einen Ableitungsbaum zu z. Dieser ist offenbar linear, das heißt, in jedem Knoten wird höchstens ein Terminalzeichen produziert. Es gibt also einen Pfad mit mehr als $|z|$ Nichtterminalzeichen. Sei X das Terminalzeichen, welches sich am weitesten oben zur Wurzel des Ableitungsbaumes befindet. Betrachte folgender Situation



Daraus ergibt sich: $|uvxy| \leq n$, $|vx| > 0$ und $uv^iwx^iy \in L$

(d) Annahme: L ist linear.

Betrachte zu jedem $n \in \mathbb{N}$ das Wort $z = a^n b^n c^n d^n \in L$ und eine beliebige Zerlegung $x = uvwxy$ mit $|vx| > 0$ und $|uvxy| \leq n$ gilt. Aus letzterem folgt: uv Teilwort von a^n und xy Teilwort von d^n Es gilt also $uv^0wx^0y = a^{n-|v|}b^n c^n d^{n-|x|}$. Aus $|vx| > 0$ folgt $|v| > 0$ oder $|x| > 0$.

Offenbar gilt dann $uv^0wx^0y = a^{n-|v|}b^nc^nd^{n-|x|} \notin L$, im Widerspruch zum Pumpinglemma für lineare Sprachen. Also ist L nicht linear

(e) Es gibt zwei Varianten, diese Aufgabe zu lösen

(i) Man gebe eine kontextfreie Grammatik an, die L erzeugt, z.B.: $G = (\{S, X\}, \{a, b, c, d\}, S, \{S \to XY, X \to aXb \mid \epsilon, Y \to cYd \mid \epsilon\}$ und zeige dann $L = L(G)$.

(ii) Mit Hilfe der Abschlußeigenschaften kontextfreier Sprachen: Zuerst zeigt man, daß $L' = \{a^nb^n \mid n \in \mathbb{N}\}$ kontextfrei ist: $G' = (\S, X\}, \{a, b\}, S, \{S \to aSb \mid \epsilon\})$ ist eine kontextfreie Grammatik mit $L(G') = L'$, da offenbar für jedes $n \in \mathbb{N}$ und jede Ableitung $S \to aSb \to^* a^nSb^n \to a^nb^n$ gilt.

Da kontextfreie Sprachen unter Homomorphismus abgeschlossen sind, ist auch $\{c^nd^n \mid n \in \mathbb{N}\}$ kontextfrei (wähle g mit $g(a) = \{c\}$ und $g(b) = \{d\}$).

L ist als Verkettung $\{a^nb^n \mid n \in \mathbb{N}\} \cdot \{c^nd^n \mid n \in \mathbb{N}\}$ zweier kontextfreien Sprachen selbst wieder kontextfrei.

Lösung zu Aufgabe 4.9.3

(a) $G = (\{S\}, \{a, b\}, S, P)$ mit $P = \{S \to aSb \mid \epsilon\}$ ist gleich-linear.

(b) Annahme L ist gleich-linear. Für jedes $n \in \mathbb{N}$ wähle $z = a^nb^nc^n$. Offenbar gilt $|z| \geq 2n$. Betrachte eine beliebige Zerlegung Zerlegung $z = uvwxy$ mit $|v| = |x| > 0$ und $|uv| = |xy| \leq n$. Aus $|uv| = |xy| \leq n$ (und Wahl von z) folgt, daß v vollständig in den a's und x vollständig in den $c's$ liegt. Es gilt $uv^0wx^0y - uwy = a^{n-|v|}b^nx^{n-|x|} \notin L$, da $|v| = |x| > 0$ gilt. Im Widerspruch zum Pumpinglemma. Also ist L nicht gleich-linear.

(c) Beweis: L sei eine gleich-lineare Sprache, die von der Grammatik $G = (N, T, S, P)$ erzeugt wird. Wir wählen $n = |N|$ (Anzahl der Nichtterminalzeichen von G). Zu einem Wort $z \in L$ mit $|z| \geq 2n$ betrachte man einen Ableitungsbaum. Dieser Ableitungsbaum besitzt einen Pfad von der Wurzel zu einem Blatt mit mindestens $n + 2$ Knoten, weil die Wurzel aus dem Startsymbol besteht, das Blatt aus einem Terminalzeichen oder ϵ und zur Ableitung des Wortes aus S mindestens n Produktionen der Form $X' \to aY'b$ benutzt werden müssen, um die $2n$

Terminalzeichen zu produzieren. Wir betrachten einen solchen Pfad. Er enthält mindestens $n+1$ innere Knoten (inklusive dem Wurzelknoten). Da $n+1 > |N|$ gilt, kommt ein Nichtterminalzeichen $X \in N$ mindestens zweimal auf dem Pfad vor. Wir betrachten das erste doppelte Vorkommen von X auf diesem Pfad von der Wurzel des Ableitungsbaums aus gesehen. Betrachte dazu Abbildung c.

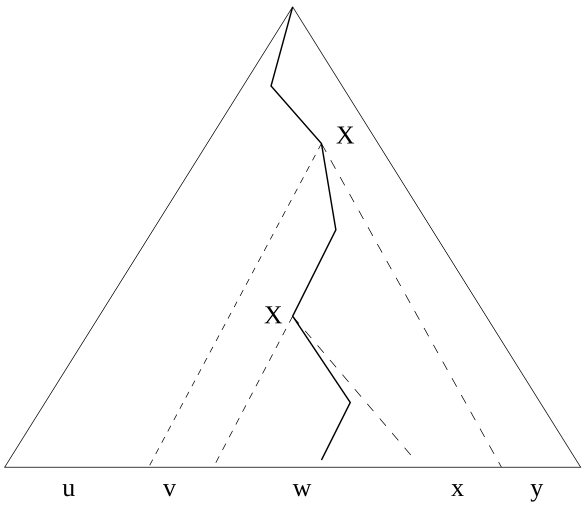


Abbildung 4.6: Syntaxbaum für ein abgeleitetes Wort

Es gilt $|uv| \leq n$ und $|xy| \leq n$, weil der Pfad zum zweiten Vorkommen von X (nach Wahl von X) höchstens n innere Knoten besitzt. Zu diesen inneren Knoten gehören Produktionen der Form $X' \to aY'b$, die links und rechts vom Pfad jeweils genau ein Terminalzeichen produzieren. uv besteht aus nicht mehr als n Terminalzeichen und aus anaologem Grund gilt $|xy| \leq n$. Es gilt $|uv| = |xy|$, weil jede Produktion zu einem inneren Knoten links und rechts vom Pfad die gleiche Anzahl Terminalzeichen produziert. Aus analogem Grund gilt $|v| = |x|$. Es gilt $|v| > 0$ und $|x| > 0$, weil beim ersten Vorkommen von X eine Produktion der Form $X \to aYb$ angewendet wurde, die jeweils links und rechts des Pfads ein Terminalzeichen produziert. Es gilt $uv^iwx^iy \in L$ für alle $i \in \mathbb{N}$, weil durch die i-fache Wiederholung des Teilstückes zwischen dem ersten und zweiten Vorkommen von X wieder ein korrekter Syntaxbaum entsteht.

(d) $G = (\{S, T\}, \{a, b, c\}, S, P)$ mit $P = \{S \to Sc|T, T \to aTb \mid \epsilon\}$

(e) Nein, da $\{a^n b^n \mid n \geq 0\}$ gleich-linear ist, aber nicht regulär und deswegen einen unendlichen Index besitzt.

Lösung zu Aufgabe 4.9.4

(a) $G = (\{S\}, \{0,1\}, S, P)$ mit

$$P = \{S \to 0S1 \mid 1S0 \mid \epsilon\}$$

(b) $L(G) \subseteq L$: vollständige Induktion über Länge der Wörter aus $L(G)$

Induktionsvoraussetzung:
$n \geq 0$ und $w \in L(G)$ mit $|w| = n$, dann gilt $w \in L$.

Induktionsanfang: $w \in L(G)$ mit
$|w| = 0$, d.h. $w = \epsilon \in L$
$|w| = 1$, d.h. $0, 1 \notin L$ und $0, 1 \notin L(G)$

Induktionsschluß von $n \geq 0$ auf $n+2$:

$w \in L(G)$ mit $|w| = n+2$

betrachte ersten Ableitungsschritt von $S \Rightarrow^* w$

Wegen $|w| \geq 2$ gilt $S \Rightarrow 0S1 \Rightarrow^* 0u1 = w$ ($S \Rightarrow 1S0$ analog)

$S \Rightarrow^* u \in L(G), |u| = n$

nach Induktionsvoraussetzung gilt $u \in L$, d.h. $u^I = u$

Insgesamt: $w^I = (0u1)^I = (u1)^I 1 = 0u^I 1 = 0u1 = w$. Also $w \in L$.

Zu zeigen ist noch: $(u1)^I = 0u^I$

Beh.: $(u1)^I = 0u^I$ ($(u1)^I = 0u^I$ analog)

Beweis: Induktion über Länge von $u1 \in \{0,1\}^*$

Induktionsbehauptung: Für alle $n \geq 1$ mit $|u1| = n, u \in \{0,1\}^*$ gilt $(u1)^I = 0u^I$

Induktionsanfang: $|u1| = 1$, d.h. $u = \epsilon$. $(u1)^I = (\epsilon 1)^I = (1\epsilon)^I = \epsilon^I 0 = \epsilon 0 = 0\epsilon = 0u$

Induktionsschluß: $|u1| = n+1, n \geq 1$

u nicht leer, etwa $u = 0u'$, dann gilt $|u'1| = n$ und nach Induktionsvoraussetzung $(u'1)^I = 0u'^I$

Also: $(u1)^I = (0u'1)^I = (u'1)^I 1 = 0u'^I 1 = 0(0u')^I = 0u^I$

$L \subseteq L(G)$: vollständige Induktion über Länge der Wörter aus L

Induktionsvoraussetzung:
$n \geq 0$ und $w \in L$ mit $|w| = n$, dann gilt $w \in L(G)$.

Induktionsanfang: $w \in L$ mit
$|w| = 0 : w = \epsilon \in L(G)$
$|w| = 1 : 0, 1 \notin L$ und $0, 1 \notin L(G)$

Induktionsschluß von $n \geq 0$ auf $n + 2$:

$w \in L$ mit $|w| = n + 2$

Aus $w = w^I$ folgt $w = 0u1$ mit $|u| = n$ ($w = 1u0$ analog, $0u0, 1u1$ nicht möglich)

Aus $w^I = 0u^I 1 = 0u1 = w$ folgt $u \in L$

nach Induktionsvoraussetzung gilt $u \in L(G)$, d.h. $S \Rightarrow^* u$

Insgesamt: $S \Rightarrow 0S1 \Rightarrow 0u1 = w \in L(G)$.

(c) Es gibt verschiedene Möglichkeiten:

1. Wende Pumpinglemma für reguläre Sprachen an:

Annahme L ist regulär

Zu jedem $n \in \mathbb{N}$ wähle $x = 0^n 1^n \in L$.

Betrachte Zerlegung $x = uvw$ mit (1) $|uv| \leq n$ und (2) $|v| \geq 1$

Aus (1) folgt, daß v vollständig in 0^n liegt,
etwa $v = 0^i$ mit $1 \leq i \leq n$ (2)

Dann gilt $uv^0w = 0^{n-i}1^n \notin L$ im Widerspruch zum Pumpinglemma
Also ist L nicht regulär

2. Man weißt nach, daß index(R_L) unendlich ist:

Beh. $[0^i] \neq [0^j]$ für $i \neq j$

Bew.: O.E. $i > j$

wähle $u = 1^i$ dann gilt $0^i u \in L$ und $0^j u \notin L$

Lösung zu Aufgabe 4.9.5

(a) $\text{First}_1(aBb\,\text{Follow}(S)) = \{a\}$ und $\text{First}_1(\epsilon\,\text{Follow}(S)) = \{\epsilon, b\}$. Beide Mengen sind disjunkt, also ist G SLL(1).

(b) Ja, da jede SLL(1)-Grammatik eindeutig ist.

(c) **Induktionsanfang:** $n = 1$: es gilt offenbar $S \Rightarrow^1 \epsilon$ und $\epsilon \in L$.

Induktionsvoraussetzung: für jedes $n \geq 1$ und jede Ableitungsfolge $S \Rightarrow^n w, w \in \{a,b\}^*$ gilt $w \in L$.

Induktionsschluß: von n auf $n+1$
$w \in L(G)$ mit $S \Rightarrow^{n+1} w$, dann gilt offenbar $S \Rightarrow aSbb \Rightarrow^n avbb = w$ und damit $S \Rightarrow^n v \in L(G)$. Nach IV gilt $v \in L$, etwa $v = a^i b^{2i}$. Dann gilt $w = a^{i+1} b^{2i+2} \in L$.

(d) Es gibt zwei verschiedene Lösungen

(i) Benutze Pumpinglemma für reguläre Sprachen. Annahme $L = L(G)$ ist regulär. Sei $n \in \mathbb{N}$. Für jede Zerlegung uvw von $x = a^n b^{2n}$ mit $|uv| \leq n$ und $|v| > 0$ liegt uv vollständig in a^n. Da v nicht leer ist gilt $v = a^k, k > 0$ und damit $uv^0w = a^{n-k}b^{2n} \notin L$ im Widerspruch zum Pumpinglemma für reguläre Sprachen. Also ist L nicht regulär.

(ii) Berechne Index von L. $[w]$ bezeichne die Äquivalenzklasse von $w \in \{a,b\}^*$ bezüglich R_L. Für $i < j$ gilt dann $a^i b^{2j} \notin L$, aber $a^j b^{2j} \in L$, d.h. $[a^i] \neq [a^j]$. Das heißt, es gibt unendliche viele Äquivalenzklassen, also ist L nicht regulär.

(e) Hier gibt's zwei Möglichkeiten:

(i) L ist kontextfrei: eine Richtung wurde oben bewiesen, die andere ($L(G) \supseteq L$) ist einfach: zu $a^n b^{2n} \in L$ gibt es die Ableitung $S \Rightarrow aSbb \Rightarrow^* a^n S b^{2n} \Rightarrow a^n b^{2n}$.

$\{c^k d^{2k} \mid k \geq 0\}$ ist kontextfrei, da es homomorphes Bild ($a \mapsto c, b \mapsto d$) von L ist.

Die kontextfreien Sprachen sind abgeschlossen bzgl Verkettung, deswegen ist auch $L \cdot \{c^k d^{2k} \mid k \geq 0\} = LL$ kontextfrei.

(ii) Angabe einer kontextfreien Grammatik G für LL und Argumentation für $LL = L(G)$.

(f) Nein,

(i) da für kontextfreies $L_1 = \Sigma^*$ dann folgt, daß kontextfreie Sprachen abgeschlossen sind unter Komplementbildung, was sie aber nicht sind (Vorlesung).

(ii) da sonst — mit Abschlußeigenschaft von $\cup$ — auch $(L_1 - L_2) \cup (L_2 - L_1) = L_1 \cap L_2$ kontextfrei ist.

Lösung zu Aufgabe 4.9.6

Welche der folgenden Aussagen sind richtig und welche sind falsch? Bitte kreuzen Sie die richtigen Kästchen an!

Es gibt kontextfreie Sprachen mit endlichem Index $\surd$ Ja $\bigcirc$ Nein

Kapitel 5

Syntaxanalyse

Die Aufgabe zu einer gegebenen Grammatik G mit terminalem Alphabet T und einem Wort $w \in T^*$ festzustellen, ob w zum Sprachschatz von G gehört und im positiven Fall auch einen Syntaxbaum für w zu konstruieren bezeichnet man als Syntaxanalyse. Sie ist der erste Schritt in der Übersetzung eines Quellprogramms in kompilierten Maschinencode. Auf dieser Stufe werden Syntaxfehler des Eingabeprogramms abgefangen. Der resultierende Syntaxbaum ist Ausgangspunkt für die weiteren Schritte der Kompilierung.

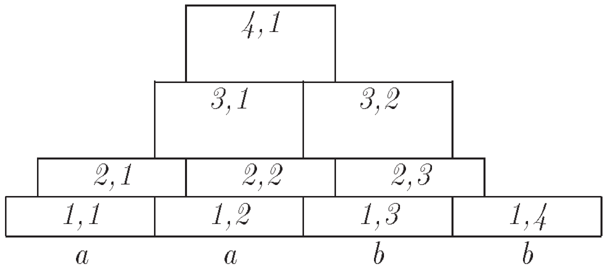
Aber auch in vielen anderen Situationen ist die Überprüfung einer festgelegten Syntax z.B. von Eingabedaten von Bedeutung.

5.1 Cocke-Kasami-Younger Algorithmus

Wir betrachten in diesem Abschnitt zunächst einen einfachen Algorithmus zur Entscheidung des Elementproblems für kontextfreie Grammatiken. Im Gegensatz zu den Überlegungen aus dem Abschnitt 4.5, die sich beschränkten auf die Erkundung der Grenze zwischen entscheidbaren und unentscheidbaren Eigenschaften, steigen wir hier ein die Entwicklung brauchbarer Algorithmen. Dabei werden die Algorithmen, die wir im folgenden erklären werden nicht nur die Antwort „ja“ oder „nein“ liefern, sondern im Falle einer positiven Antwort auch eine Folge von Grammatikproduktionen angeben, die das untersuchte Wort aus dem Startsymbol erzeugen. Deswegen nennt man solche Algorithmen auch **Zerteilungsalgorithmen** oder **Parsing-Algorithmen**.
Der Cocke-Kasami-Younger Algorithmus ist eher von didaktischem Interesse und setzt voraus, daß eine Grammatik in Chomsky-Normalform vorliegt. Charakteristisch für den Cocke-Kasami-Younger Algorithmus ist die dabei

benutzte pyramidenförmige Datenstruktur, wie sie in dem folgenden Beispiel für das Eingabewort *aabb* zu sehen ist. Es ist vor allem wichtig sich die Numerierung der Zellen einzuprägen.

Beispiel 55



Definition 85 (Der Cocke-Kasami-Younger Algorithmus)
Gegeben sind eine kontextfreie Grammatik $\mathcal{G} = (N, T, S, \mathcal{P})$ *in Chomsky-Normalform und ein Wort* $w = w_1 \dots w_n \in T^*$*. Wir beginnen mit der Erzeugung einer Pyramide, deren Basis aus n Zellen besteht, unter welche von links nach rechts die Buchstaben von w geschrieben werden. Im Laufe des Algorithmus werden die Zellen der Pyramide mit Mengen von Nichtterminalzeichen von* $\mathcal{G}$ *gefüllt und zwar nach der folgenden Vorschrift:*

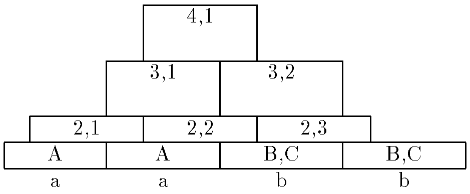
1. *Basis der Pyramide.*
   Für $1 \leq j \leq n$ *schreibe* $A \in N$ *in die Zelle* $(1, j)$ *genau dann, wenn es eine Produktion* $A \to w_j$ *in* $\mathcal{P}$ *gibt.*

2. *Weitere Reihen.*
   Seien alle Reihen $\leq i$ *schon eingetragen.*
   Für $1 \leq j \leq n - i$ *werden die Einträge in Zelle* $(i + 1, j)$ *wie folgt bestimmt:*
   Für $1 \leq m \leq i$ *betrachte alle Paare* X_1X_2 *mit*
   X_1 *aus Zelle* (m, j) *und*
   X_2 *aus Zelle* $(i - m + 1, j + m)$*.*
   Trage A in Zelle $(i + 1, j)$ *ein falls* $A \to X_1X_2$ *in* $\mathcal{P}$ *vorkommt.*

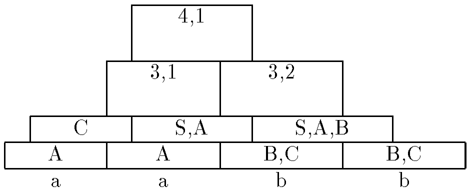
3. $w \in L(\mathcal{G})$ *genau dann, wenn S in Zelle* $(n, 1)$ *vorkommt.*

Beispiel 56
$\mathcal{G}$ *sei durch die folgenden Regeln gegeben:*

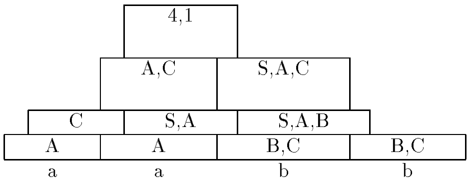
$$
\begin{aligned}
S &\to AB \mid BB \\
A &\to CC \mid AB \mid a \\
B &\to BB \mid CA \mid b \\
C &\to BA \mid AA \mid b
\end{aligned}
$$



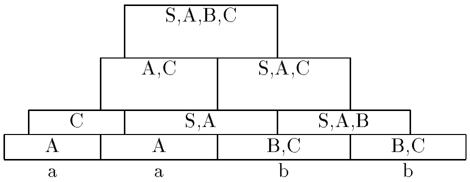
(a) Erste Reihe



(b) Zweite Reihe



(c) Dritte Reihe



(d) Schluß

Abbildung 5.1: Das sukzessive Ausfüllen der Pyramide

Abbildung 5.1 veranschaulicht wie der Cocke-Kasami-Younger Algorithmus für das Eingabewort aabb arbeitet. Teilabbildung (a) zeigt wie die erste Zeile der Pyramide ausgefüllt wird.

Das Ausfüllen der zweiten Ebene ist noch recht einfach. Um z.B. den Eintrag in die Zelle $(2,1)$ *zu ermitteln, muß man nach Produktionen suchen, die* AA *als rechte Seite haben. Da gibt es nur* $C \to AA$*. Entsprechend werden die restlichen Zellen der zweiten Reihe gefüllt.*

Um die Einträge in Zelle $(3,1)$ *zu finden, sind die Paare*

$$
\begin{array}{llll}
(1,1) & (2,2) & \textit{für } m=1 & AS, AA \\
(2,1) & (1,3) & \textit{für } m=2 & CB, CC
\end{array}
$$

zu betrachten. Das Ergebniss ist in der Teilabbildung (c) zu sehen.

Um die Einträge in Zelle $(4,1)$ *zu finden, sind die Paare*

$$
\begin{array}{llll}
(1,1) & (3,2) & \textit{für } m=1 & AS, AA, AC \\
(2,1) & (2,3) & \textit{für } m=2 & CS, CA, CB \\
(3,1) & (1,4) & \textit{für } m=3 & AB, AC, CB, CC
\end{array}
$$

zu betrachten. Das betrachtete Wort gehört also zum Sprachschatz der Grammatik $\mathcal{G}$. In Abbildung 5.2 ist ein Syntaxbaum für aabb abgebildet.

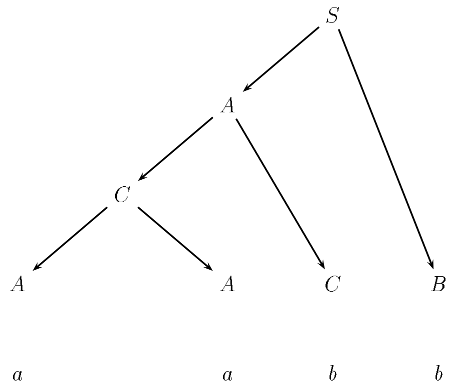


Abbildung 5.2: Ein Syntaxbaum für das Wort *aabb* in der Grammatik aus Beispiel 56

Theorem 71
Der CYK-Algorithmus ist korrekt.

Beweis: Gegeben sind eine kontextfreie Grammatik $\mathcal{G} = (N, T, S, \mathcal{P})$ in Chomsky-Normalform und ein Wort $w = w_1 \ldots w_n \in T^*$. Der Beweis geschieht durch Induktion nach den Stufen der Pyramide in derselben Reihenfolge, in welcher der Algorithmus vorgeht. Der wichtigste Schritt dabei ist die Wahl einer geeigneten Induktionsbehauptung. Hier ist sie:

A steht in Zelle (i, j) genau dann, wenn

$$A \Rightarrow^* w_j w_{j+1} \ldots w_{j+i-1} \text{ gilt.}$$

Wenn wir diese Aussage bewiesen haben, dann gilt insbesondere für die Spitze der Pyramide, $(n, 1)$:

A steht in Zelle $(n, 1)$ genau dann, wenn

$$A \Rightarrow^* w_1 w_2 \ldots w_n$$

Somit gilt $w \in L(\mathcal{G})$ gdw S in der Zelle $(n, 1)$ vorkommt.

Beginnen wir jetzt mit dem Induktionsbeweis. Für die Basis der Pyramide, d.h. für $i = 1$ reduziert sich die zu zeigende Behauptung auf:

A steht in Zelle $(1, j)$ genau dann, wenn

$$A \Rightarrow^* w_j \text{ gilt,}$$

was nach der Beschreibung des Algorithmus aber offensichtlich ist.

Zum Beweis des Induktionsschrittes betrachten wir eine Zelle $(i+1, j)$ und nehmen an, daß für alle Zellen (i', j') mit $i' \leq i$ die Behauptung schon bewiesen ist. Kommt A in der Zelle (i, j) vor so gibt es ein m , $1 \leq m \leq i$ und $X_1 \in (m, j)$, $X_2 \in (i - m + 1, j + m)$, so daß die Produktion $A \rightarrow X_1 X_2$ in $\mathcal{P}$ vorkommt. Nach Induktionsvoraussetzung gilt

$$X_1 \Rightarrow^* w_j w_{j+1} \dots w_{m+j-1}$$

und

$$X_2 \Rightarrow^* w_{m+j} w_{m+j+1} \dots w_{i+j}$$

Zusammengenommen ergibt sich

$$A \Rightarrow^* X_1 X_2 \Rightarrow^* w_j w_{j+1} \dots w_{i+j}$$

wie gewünscht.

Gelte jetzt umgekehrt $A \Rightarrow^* w_j w_{j+1} \dots w_{i+j}$. Da $\mathcal{G}$ eine Grammatik in Chomsky-Normalform ist gibt es eine Produktion $A \rightarrow X_1 X_2$ in $\mathcal{P}$, mit

$$A \Rightarrow^* X_1 X_2 \Rightarrow^* w_j w_{j+1} \dots w_{i+j}$$

Das von X_1 aus erzeugte Anfangsstück des Wortes $w_j w_{j+1} \dots w_{i+j}$ sei $w_j w_{j+1} \dots w_{j+m-1}$. Offensichtlich wird dann von X_2 aus der Rest des Wortes erzeugt, d.h. $w_{j+m} w_{j+m+1} \dots w_{i+j}$. Nach Induktionsvoraussetzung folgt also $X_1 \in (m, j)$ und $X_2 \in (i - m + 1, m + j)$.

∎

5.1.1 Komplexität des Cocke-Kasami-Younger Algorithmus

Wie groß ist der Aufwand des CYK-Algorithmus für ein Eingabewort der Länge n? Wir stellen zunächst fest, daß die Pyramide über diesem Wort aus $n + (n-1) + \ldots + 1 = \frac{n(n+1)}{2}$ Zellen besteht. Im schlimmsten Fall sind für das Ausfüllen einer Zelle $(n-1)$ viele Paare vorangegangener Zellen zu betrachten. Mit k wollen wir die Anzahl der Nichtterminalzeichen der zugrundeliegenden Grammatik bezeichnen. Fixiert man eine auszufüllende Zelle und ein

Paar vorangegangener Zellen, dann muß man im Extremfall k^2 viele Paare von Nichtterminalzeichen betrachten, die als rechte Seite einer Produktion vorkommen können. Da es höchstens k^3 viele Produktionen geben kann, dauert eine naive Suche nach einer bestimmten rechten Seite höchstens k^3 Schritte. Insgesamt sind also in der von uns fixierten Situation k^5 Rechenschritte durchzuführen. Insgesamt ist also n^3k^5 eine obere Schranke für die Anzahl der Gesamtkomplexität. Fast man die Grammatik als gegeben auf und nur das Eingabewort als variabel, so ergibt als Zeitkomplexität $O(n^3)$.

5.2 LL-Parser

Wir kommen jetzt zurück zu den in Definition 70 eingeführten LL-Kellerautomaten. Bisher wissen wir, daß ein Wort w von dem Automaten $\mathcal{A}_{\mathcal{G}}^{LL}$ genau dann akzeptiert wird, wenn $w \in L(\mathcal{G})$ gilt. Da der Automat $\mathcal{A}_{\mathcal{G}}^{LL}$ hochgradig nichtdeterministisch sein kann, ist es keine leichte Aufgabe festzustellen, ob eine akzeptierende Berechnungsfolge existiert. Ganz grob könnte ein nichtdeterministischer LL-Zerteilungsalgorithmus (LL-Parsing Algorithmus) etwa so aussehen:

Eingabe: G (Grammatik), $w \in T^*$
begin
    $K := S$; $W := w$;
    while $(K \neq \varepsilon \wedge W \neq \varepsilon)$ **do**
    begin
        if $(K = ut \wedge W = tw_1)$
        then $K := u$; $W := w_1$;
        if $(K = uA \wedge \mathcal{P}(A) \neq \emptyset)$
        then **wähle** $A \rightarrow v \in \mathcal{P}$;
            $K := uv^r$
        else $w \notin L(\mathcal{G})$.
    end
    if $K = \varepsilon \wedge W = \varepsilon$
    then $w \in L(\mathcal{G})$
    else $w \notin L(\mathcal{G})$.
end

dabei ist

$\mathcal{P}(A) = \{A \rightarrow v \mid \text{ in } \mathcal{P}\}$

Bevor wir uns überlegen wie dieser Algorithmus verbessert werden kann, führen wir noch einige Begriffe ein, die uns im folgenden von Nutzen sein werden.

Definition 86
$G = (N, T, S, \mathcal{P})$ eine kontextfreie Grammatik,
$w = a_1 \dots a_n \in (N \cup T)^*$, $L \subseteq T^*$ und $k > 0$ eine natürliche Zahl.

$$\begin{array}{lcl}
k : w & = & \begin{cases} w & \textit{falls } n \leq k \\ a_1 \dots a_k & \textit{falls } n > k \end{cases} \\
k : L & = & \{k : w \mid w \in L\} \\
L(\mathcal{G}) & = & \{\ v \in T^* \ \mid \ \ S \Rightarrow v\} \\
 & & \textit{von } \mathcal{G} \textit{ erzeugte Sprache,} \\
L_{\mathcal{G}}(w) & = & \{\ u \in T^* \ \mid \ \ w \Rightarrow u\} \\
\mathrm{First}_{G,k}(w) & = & k : L_{\mathcal{G}}(w) \\
\mathrm{Follow}_{G,k}(w) & = & \{k : y \mid S \Rightarrow_\ell^* uwy \textit{ für} \\
 & & y \in (N \cup T)^*, u \in T^*\} \\
\mathrm{Follow}_{G}(w) & = & \{y \mid S \Rightarrow_\ell^* uwy \textit{ für} \\
 & & y \in (N \cup T)^*, u \in T^*\}
\end{array}$$

Beispiel 57 (Beispielgrammatik $\mathcal{G}_3$)

$$\begin{array}{l}
S \to aB \mid bCc \mid c \\
B \to bB \mid aCC \\
C \to aSS \mid c
\end{array}$$

Wird im Laufe einer Berechnung des LL-Kellerautomaten $\mathcal{A}_3^{LL}$ zu $\mathcal{G}_3$ eine Konfiguration Bqb erreicht, so macht es nur Sinn einen „produce"-Schritt mit einer Grammatikregel $B \to u$ vorzunehmen, wenn das erste Symbol von u der Buchstabe b ist. Denn der erste Buchstabe von u wird zum obersten Kellerzeichen und nur b kann für einen nachfolgenden Vergleichsschritt benutzt werden. Ein Produktionsschritt mit der Regel $B \to aCC$ von $\mathcal{G}_3$ würde zu einem sofortigen Fehlschlag des Parsingversuchs führen. Wir können also zumindest in dem Beispiel $\mathcal{G}_3$ eine Verbesserung des LL-Zerteilungsalgorithmus erzielen wenn wir einen Buchstaben vorausschauen.

Wir verbessern den LL-Parsingalgorithmus für $\mathcal{G}_3$ aber beliebige Eingabewörter, indem wir die nichtdeterministische Auswahl ersetzen durch die folgende Vorschrift:

Falls $K = uA$ und $W = xw_1$ wähle $A \to v$ aus $\mathcal{P}$, so daß $v = xv_1$.

Das Ergebnis dieser Vorschrift ist in der folgenden LL-Parsertabelle zusammengefaßt:

Vorausschau $\rightarrow$ Keller $\downarrow$	a	b	c
S	$S \rightarrow aB$	$S \rightarrow bCc$	$S \rightarrow c$
B	$B \rightarrow aCC$	$B \rightarrow bB$	
C	$C \rightarrow aSS$		$C \rightarrow c$

Man bemerkt, daß für $\mathcal{G}_3$ ein deterministischer Algorithmus entsteht. Wir werden später sehen, daß das nicht immer so sein muß.

Betrachten wir ein weiteres nicht ganz so simples Beispiel:

Beispiel 58 (Beispielgrammatik $\mathcal{G}_4$)

$S \rightarrow aBBB \mid bC$
$B \rightarrow bB \mid aC$
$C \rightarrow SS \mid c$

Was soll man hier in der Konfiguration Cqa tun? Es gibt keine Produktion mit linker Seite C und a als erstem Buchstaben der rechten Seite. Eine Produktion mit $C \rightarrow SS$ könnte erfolgversprechend sein, wenn durch weitere Anwendungen aus S ein Wort mit dem ersten Buchstaben a hergeleitet werden kann. Mit anderen Worten, wenn $a \in \mathrm{First}_1(S)$ gilt. Man sieht sofort, daß das tatsächlich der Fall ist. Schon die erste Regel $S \rightarrow aBBB$ beweist das. Ebenso gilt $b \in \mathrm{First}_1(S)$.

Für $\mathcal{G}_4$ erhalten wir somit die folgende LL-Parsertabelle und wieder einen deterministischen LL-Zerteilungsalgorithmus.

Vorausschau $\rightarrow$ *Keller* $\downarrow$	a	b	c
S	$S \rightarrow aBBB$	$S \rightarrow bC$	
B	$B \rightarrow aC$	$B \rightarrow bB$	
C	$C \rightarrow SS$	$C \rightarrow SS$	$C \rightarrow c$

Nach den beiden Beispielen können wir eine allgemeine Konstruktionsvorschrift für Parsertabellen versuchen.

Definition 87 (LL-Parsertabelle,ε-freier Fall)
Für eine Vorausschau (engl. lookahead) der Länge $k = 1$ und eine ε-freie Grammatik $\mathcal{G}$ konstruiert man die LL-Zerteilertabelle nach der folgenden Vorschrift:

$A \rightarrow v$ wird in die Tabellenposition (A, a) eingetragen, falls $a \in \mathrm{First}_1(v)$ gilt.

Warum müssen wir in dieser Definition die ε-Freiheit von $\mathcal{G}$ voraussetzen? Betrachten wir ein weiteres Beispiel:

Beispiel 59 (Beispielgrammatik $\mathcal{G}_5$)

$$
\begin{array}{l}
S \to Cb \mid aB \\
B \to bB \mid S \\
C \to aC \mid \varepsilon
\end{array}
$$

Was soll man hier in die Tabellenposition (C, a) eintragen? Sicherlich $C \to aC$. Aber auch die Produktion $C \to \varepsilon$? Im Zweifelfall, wenn man nicht ausschließen kann, daß die Anwendung einer Produktion nicht doch zu einem erfolgreichen Ende führt, könnte man ja die umstrittene Regel mithinzunehmen und notfalls auch einen lokalen Nichtdeterminismus hinnehmen. Man kann aber ein bißchen mehr tun.

Betrachten wir eine Konfiguration

$$uCqaw_1,$$

in der man sich zwischen einer Anwendung der Produktion $C \to aC$ oder $C \to \varepsilon$ entscheiden muß. Führen wir einmal probehalber die letzte Produktion aus und erreichen damit die Konfiguration

$$uqaw_1.$$

Nach Theorem 50 ist das gleichbedeutend mit der Existenz einer Ableitung

$$S \Rightarrow_\ell^* u_1 C u^r \Rightarrow u_1 u^r$$

Wenn der erste Buchstabe von u^r ein Terminalzeichen ist, dann muß es der Buchstabe a sein, sonst scheitert der nächste Vergleichsschritt. Wenn der erste Buchstabe von u^r kein Terminalzeichen ist, dann muß es zumindest möglich sein in einigen weiteren Ableitungsschritten aus u^r eine Wort mit dem Anfangsbuchstaben a herzuleiten. Zusammenfassend kann man sagen: es ist nur dann sinnvoll C durch ε zu ersetzen, wenn es eine Rechtsableitung gibt, die zu einem Wort der Form $u_1 C a u_2$ führt. Man kann im Falle von $\mathcal{G}_5$ feststellen, daß überhaupt kein Wort ableitbar ist, in dem auf C der Buchstabe a folgt. Mit anderen Worten, es gilt $a \notin \mathrm{Follow}_1(C)$. Somit macht in der Konfiguration Cqa die Anwendung einer Produktionsregel mit $C \to \varepsilon$ keinen Sinn, sie führt früher oder später zu einem Fehlschlag. Hier ist die vollständige LL-Parsertabelle für $\mathcal{G}_5$:

Vorausschau $\rightarrow$ Keller $\downarrow$	a	b
S	$S \rightarrow aB$ $S \rightarrow Cb$	
B	$B \rightarrow S$	$B \rightarrow bB$
C	$C \rightarrow aC$	$C \rightarrow \varepsilon$

Definition 88 (LL-Parsertabelle, allgemeiner Fall)
Für $k > 0$ oder $\mathcal{G}$ nicht ε-frei:

> *$A \rightarrow v$ wird in die Tabellenposition $(A, a_1 \dots a_k)$ eingetragen, falls $a_1 \dots a_k \in \mathrm{First}_k(v\,\mathrm{Follow}(A))$ gilt.*

Hier ist noch ein weiteres Beispiel für eine LL-Parsertabelle mit lookahead 1.

Beispiel 60 (Beispielgrammatik $\mathcal{G}_6$)

$S \rightarrow aBB \mid bC$
$B \rightarrow CB \mid aC$
$C \rightarrow abS \mid c$

Die zugehörige LL-Parsertabelle sieht so aus:

Vorausschau $\rightarrow$ *Keller* $\downarrow$	a	b	c
S	$S \rightarrow aBB$	$S \rightarrow bC$	
B	$B \rightarrow aC$ $B \rightarrow CB$		$B \rightarrow CB$
C	$C \rightarrow abS$		$C \rightarrow c$

Man bemerkt, daß in diesem Beispiel, wie auch schon in Beispiel 59, ein Rest von Nichtdeterminismus erhalten bleibt. Man ist in erster Linie interessiert an Grammatiken. die zu einer eindeutigen Parsertabelle führen. Für eine Vorausschau von der Länge k nennt man solche Grammatiken SLL(k)-Grammatiken.

5.2.1 SLL(k)-Grammatiken

Die Bedingung, daß in jedem Feld der LL-Parsertabelle höchstens eine Produktion steht, läßt sich umsetzen in eine Bedingung über First und Follow Mengen.

Definition 89
Eine kontextfreie Grammatik $\mathcal{G} = (N, T, S, \mathcal{P})$ *heißt eine SLL(k)-Grammatik, falls für jedes Nichtterminalzeichen* A *und je zwei Produktionen* $A \to v_1$ *und* $A \to v_2$ *gilt*

$$\mathrm{First}_k(v_1\,\mathrm{Follow}(A)) \cap \mathrm{First}_k(v_2\,\mathrm{Follow}(A)) = \emptyset$$

Der besonders wichtige Spezialfall $k = 1$ *soll noch einmal besonders hervorgehoben werden:* $\mathcal{G} = (N, T, S, \mathcal{P})$ *heißt eine SLL(1)-Grammatik, falls für jedes Nichtterminalzeichen* A *und je zwei Produktionen* $A \to v_1$ *und* $A \to v_2$ *gilt*

$$\mathrm{First}_1(v_1\,\mathrm{Follow}(A)) \cap \mathrm{First}_1(v_2\,\mathrm{Follow}(A)) = \emptyset$$

Für den Spezialfall $k = 0$ *erhält man:* $\mathcal{G} = (N, T, S, \mathcal{P})$ *heißt eine SLL(0)-Grammatik, falls für jedes Nichtterminalzeichen* A *höchstens eine Produktion* $A \to v$ *in* $\mathcal{P}$ *existiert.*

Definition 90 (Alternative Definition von SLL(k)-Grammatiken)
Eine Grammatik $\mathcal{G} = (N, T, S, \mathcal{P})$ *ist eine SLL(k)-Grammatik, wenn für je zwei Produktionen in* $\mathcal{P}$

$$\begin{array}{l} A \to w_1 \\ A \to w_2 \end{array}$$

für welche zwei Ableitungen:

$$S \Rightarrow_\ell^* u_1 A v_1 \Rightarrow u_1 w_1 v_1 \Rightarrow_\ell^* u_1 \gamma_1$$

und

$$S \Rightarrow_\ell^* u_2 A v_2 \Rightarrow u_2 w_2 v_2 \Rightarrow_\ell^* u_2 \gamma_2$$

mit $\gamma_i \in T^*$ *und*

$$k : \gamma_1 = k : \gamma_2$$

existieren, gleich sein müssen, d.h. es muß

$$w_1 = w_2$$

gelten.

Der Wert des SLL(k)-Ansatzes hängt natürlich davon ab, daß man die zur Erstellung der Parsertabelle notwendigen First und Follow Mengen effizient berechnen kann.

Lemma 72
Gegeben sei eine ε-freie kontextfreie Grammatik $\mathcal{G} = (N, T, S, \mathcal{P})$. Die Mengen von Nichtterminalzeichen G^i werde wie folgt induktiv definiert:

$$\begin{array}{lcl} G^0(A) & = & \{A\} \\ G^{i+1}(A) & = & G^i(A) \cup \{X \in N \mid \textit{es gibt } Y \in G^i(A) \\ & & \textit{und es gibt } (Y \to Xw) \in \mathcal{P}\} \end{array}$$

Es gibt i mit $G^{i+1}(A) = G^i(A)$. Setze

$$G(A) = G^i(A)$$

für ein solches i. Dann gilt:

$$\begin{array}{l} \mathrm{First}_1(A) = \{a \in T \mid \textit{es gibt } (X \to aw) \in \mathcal{P} \\ \qquad\qquad\qquad\qquad \textit{mit } X \in G(A)\} \end{array}$$

Beispiel 61

$$\begin{array}{lcl} S & \to & X \\ X & \to & Yc \mid Yd \\ Y & \to & a \mid bY \\ \mathrm{Follow}(X) & = & \{\varepsilon\} \\ \mathrm{First}(Yc) & = & \{b^n ac \mid n \geq 0\} \\ \mathrm{First}(Yd) & = & \{b^n ad \mid n \geq 0\} \end{array}$$

Für die Beispielgrammatik gilt für alle k

$$\mathrm{First}_k(Yc\,\mathrm{Follow}(X)) \cap \mathrm{First}_k(Yd\,\mathrm{Follow}(X)) \neq \emptyset$$

$\mathcal{G}$ ist somit für **kein** *k eine SLL(k)-Grammatik.*

5.3 LR-Parser

Wir beginnen mit einem einführenden Beispiel, welches das Pferd vom Schwanz her aufzäumt. Dieses Beispiel zeigt das Endergebnis auf welches dieser Abschnitt hinsteuert. Bevor wir dazu Beispiel 62 betrachten, brauchen wir noch eine technische Definition. Bei der Beschreibung von LR-Kellerautomaten (ebenso bei LL-Kellerautomaten) war es wichtig festzustellen, wann das ganze Eingabewort eingelesen war. Bisher wurde nicht näher festgelegt, wie das geschehen könnte. Wir vereinbaren jetzt, daß jedes Eingabewort mit dem Sonderzeichen \$ enden soll, wobei natürlich implizit vorausgesetzt ist, daß \$ nicht in $N \cup T$ vorkommt. Es wird sich als hilfreich herausstellen, die folgende Definition zur Verfügung zu haben.

Definition 91 (Erweiterte Grammatik)
Zu einer kontextfreien Grammatik $G = (N, T, S, \mathcal{P})$ entsteht die erweiterte Grammatik

$$G' = (N \cup \{S'\}, T \cup \{\$\}, S', \mathcal{P} \cup \{S' \to S\$\})$$

durch Hinzunahme eines neuen Startsymbols und einer neuen Produktion. Manchmal nennt man G' genauer die \$ -erweiterte Grammatik (engl. \$-augmented grammar)

Offensichtlich gilt: $L(G') = L(G)\$$

Beispiel 62 (Die Grammatik G_{AEXP})

1	$S \to S + T$	4	$T \to F$
2	$S \to T$	5	$F \to (S)$
3	$T \to T \times F$	6	$F \to id$

Um festzustellen ob ein Wort w im Sprachschatz von G_{AEXP} liegt kann man die Parsertabelle aus Abbildung 5.3 benutzen. Das Berechnungsmodell ist

Zustand	Aktion						Sprung		
	id	+	×	(	)	\$	S	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Abbildung 5.3: Parsertabelle für G_{AEXP}

eine geringfügige Variante des LR-Kellerautomat aus Definition 71. Die Modifikation besteht darin, daß der neue Automatentypus, nennen wir ihn einen

LR-Parser, mehrere Zustände haben kann. Diese Zustände werden auch im Keller abgelegt und zwar immer abwechselnd ein Buchstabe des Kelleralphabets, ein Zustandssymbol, ein Buchstabe des Kelleralphabets usw. Das oberste Kellerzeichen wird immer ein Zustandssymbol sein, den wir den aktuellen Zustand nennen. Der Einfachheit halber benutzen wir Zahlen zur Bezeichnung von Zuständen. Der Anfangszustand wird durch die 0 dargestellt.

Eine LR-Parser startet in der Konfiguration mit leerem Keller und dem zu parsenden Wort w als Eingabe. Eine solche Konfiguration schreiben wir als $0 \mid w$. Die nicht-deterministische Auswahl der nächsten anzuwendenden Regel wird durch eine deterministische Anwendung der Parsertabelle ersetzt. In Abhängigkeit von dem ersten noch nicht gelesenen Buchstaben des Eingabewortes und dem aktuellen Zustand wird aus der Parsertabelle die nächste auszuführende Aktion ermittelt. Steht in dem entsprechenden Feld der LR-Parsertabelle kein Eintrag, so ist der Parsingversuch fehlgeschlagen und des Eingabewort gehört nicht zum Sprachschatz von G_{AEXP}. Steht dort acc so ist das Eingabewort akzeptiert. Ein Eintrag si signalisiert, daß eine shift Operation ausgeführt wird: das nächste Zeichen von w wird auf dem Keller abgelegt. Der neue aktuelle Zustand ist i. Die Zahl i wird jetzt das oberste Kellerzeichen.

Steht in dem Feld der LR-Parsertabelle der Eintrag rj, so ist eine reduce-Aktion mit der j-ten Regel, $A \to v$, der Grammatik G_{AEXP} auszuführen. Das geschieht in zwei Schritten. Der Keller wird in diesem Fall das Endstück der Form $s_0 b_1 s_1 \dots s_{k-1} b_k j$ aufweisen. Zunächst wird $b_1 s_1 \dots s_{k-1} b_k j$ durch die linke Seite der Produktion ersetzt, so daß jetzt der Keller auf $s_0 A$ endet. Der neue aktuelle Zustand wird jetzt wieder aus der Parsertabelle ermittelt unter dem Zustand s_0 und der Spalte für A.

Beispiel 63

Zeile	*Keller*	*Eingabe*			
1	0	$id \times id + id\$$	9	$0S1$	$+\ id\$$
2	$0id5$	$\times\ id + id\$$	10	$0S1{+}6$	$id\$$
3	$0F3$	$\times\ id + id\$$	11	$0S1{+}6id5$	$\$$
4	$0T2$	$\times\ id + id\$$	12	$0S1{+}6F3$	$\$$
5	$0T2{\times}\ 7$	$id + id\$$	13	$0S1{+}6T9$	$\$$
6	$0T2{\times}\ 7id5$	$+\ id\$$	14	$0S1$	$\$$
7	$0T2{\times}\ 7F10$	$+\ id\$$	15	$0S1\$acc$	ε
8	$0T2$	$+\ id\$$			

Folgt man konsequent den Regeln für LR-Kellerautomaten so kann man in der Konfiguration 15 noch eine Reduktion mit der Produktion $S' \to S\$$ vornehmen und erhält die **Endkonfiguration** $S' \mid \varepsilon$. Wir wollen im folgenden $S' \mid \varepsilon$ als Endkonfiguration für LR-Parser ansehen, auch wenn der letzte Reduktionsschritt nicht mehr in der Parsertabelle auftritt und der Keller nicht mehr das abwechselnde Muster von Buchstaben und Zuständen zeigt.

Wie konstruiert man zu einer gegebenen kontextfreien Grammatik eine LR-Parsertabelle? Gibt es zu jeder kontextfreien Grammatik eine solche Tabelle? Wie aufwendig ist die Konstruktion? Mit der Beantwortung dieser und ähnlicher Fragen werden wir uns für den Rest dieses Kapitels beschäftigen.

5.3.1 Lebensfähige Präfixe

Wir beginnen zunächst bescheiden, indem wir an einem Beispiel den Nichtdeterminismus beim LR-Parsing genauer zu studieren.

Beispiel 64 (Die Grammatik G_{ab})

$$\begin{array}{llll} 1 & S \to aA & 2 & S \to bB \\ 3 & A \to c & 4 & A \to dAd \\ 5 & B \to c & 6 & B \to dBd \end{array}$$

$$\mathcal{L}(G_{ab}) \;=\; \{ad^n cd^n \mid n \geq 0\} \cup \{bd^n cd^n \mid n \geq 0\}$$

Will man mit Hilfe des LR-Kellerautomaten zu G_{ab} feststellen, ob das Wort adcd zu $\mathcal{L}(G_{ab})$ gehört, so gibt es die beiden in Abbildung 5.4 aufgezeigten Möglichkeiten. Genau genommen gibt es im 4. Schritt noch eine weitere Wahlmöglichkeit; statt eines Ersetzungsschrittes hätte man auch eine shift-Aktion durchführen können. Allerdings sieht man dann schon im nächsten Schritt, daß man in eine Sackgasse gelaufen ist. Ist es auch möglich schon in Schritt 4 zu erkennen, daß die Rückwärtsanwendung der Grammatikproduktion $B \to c$ zu einem Fehlschlag führt? Die Antwort ist ja. Man braucht dazu allerdings eine aufwendigere Form der Vorausschau als bei LL-Parsern. Wenn die nach dem 4. Schritt in der rechten Ableitung erreichte Konfiguration $adBqd$ zu einem erfolgreichen Ende führen soll, so muß nach Theorem 52 gelten:

$$S \Rightarrow_r^* adBd \Rightarrow adcd$$

Das führt zu der folgenden Definition:

Schritt	Keller	Eingabe	Keller	Eingabe
1		adcd		adcd
2	a	dcd	a	dcd
3	ad	cd	ad	cd
4	adc	d	adc	d
5	adA	d	adB	d
6	adAd		adBd	
7	aA		aB	
8	S		Fehlschlag	

Abbildung 5.4: Zwei Parsingversuche für $ascd$ in G_{ab}

Definition 92 (Lebensfähige Präfixe)
Sei $G = (N, T, S, \mathcal{P})$ eine kontextfreie Grammatik.

1. $A \rightarrow u \cdot v$
   heißt eine punktierte Produktion für G, falls $A \rightarrow uv \in \mathcal{P}$.

2. *ein Wort $w \in (N \cup T)^*$ heißt ein lebensfähiger Präfix, wenn eine Ableitung*
   $$S \Rightarrow_r^* u_0Ay \Rightarrow_\ell^\pi u_0uvy$$
   existiert mit einer Produktion $\pi = A \rightarrow uv$ und $w = u_0u$

Man sieht leicht, daß die lebensfähigen Präfixe genau die Wörter sind, die im Keller einer erfolgreichen LR-Berechnung vorkommen können und zwar unabhängig vom Eingabewort. Die bisherigen Überlegungen machen keinen Gebrauch von dem nächsten einzulesenden Buchstaben. Um später auch diese Vorausschau zu ermöglichen führen wir hier schon die beiden folgenden Definitionen ein. Dabei ist k die Länge der Vorausschau. Letzten Endes werden wir nur an $k = 0$ und $k = 1$ interessiert sein. Diejenigen Definitionen und Behauptungen die ohne große Schwierigkeiten für beliebiges k gegeben werden können werden wir auch in dieser Allgemeinheit angeben.

Definition 93 (LR(k)-Elemente)
Sei $G = (N, T, S, \mathcal{P})$ eine kontextfreie Grammatik.

1. *Ein syntaktisches Objekt der Form $[A \rightarrow u \cdot v, y]$ heißt ein LR(k)-Element (engl. LR(k)-item), falls $A \rightarrow u \cdot v$ eine punktierte Produktion für G ist und $y \in T^*$ liegt mit $\mid y \mid \leq k$*
   Ein LR(0)-Element ist damit nichts anderes als eine punktierte Produktion.

2. In der folgenden Situation

$$S \Rightarrow_r^* u_0Ay \Rightarrow_r^\pi u_0uvy \text{ mit } u_0 \in V^*,\ y \in T^*,\ A \in N \text{ und } \pi = A \to uv$$

heißt $[A \to u \cdot v, z]$ *ein* **gültiges** *LR(k)-Element für* w*, falls* $z = k : y$.

Das Wort $w = u_0u$ *ist nach der vorangegangen Definition natürlich ein lebensfähiger Präfix von* G

Die Idee lebensfähige Präfixe zu betrachten, macht natürlich nur dann einen Sinn, wenn man sie einfach berechnen kann. Wir beginnen mit einem naiven Blick auf das Problem. Wir betrachten den vollständigen Ableitungsbaum für G_{ab}:

	S		
	aA	bB	
ac	$adAd$	$bdBd$	bc
$adcd$	ad^2Ad^2	bd^2Bd^2	$bdcd$
ad^2cd^2	ad^3Ad^3	bd^3Bd^3	bd^2cd^2
$\dots$	$\dots$	$\dots$	$\dots$

Hieraus läßt sich die Menge aller lebensfähigen Präfixe wie folgt ablesen

	S		
	aA	bB	
ac	$adAd$	$bdBd$	bc
$adcd$	ad^2Ad^2	bd^2Bd^2	$bdcd$
ad^2cd^2	ad^3Ad^3	bd^3Bd^3	bd^2cd^2
$\dots$	$\dots$	$\dots$	$\dots$

$$\begin{array}{lclc}
\{ad^n \mid n \geq 0\} & \cup & \{ad^nc \mid n \geq 0\} & \cup \\
\{ad^nA \mid n \geq 0\} & \cup & \{ad^nAd \mid n \geq 1\} & \cup \\
\{bd^n \mid n \geq 0\} & \cup & \{bd^nc \mid n \geq 0\} & \cup \\
\{ad^nB \mid n \geq 0\} & \cup & \{bd^nBd \mid n \geq 1\} & \cup \\
\{\varepsilon\} & & &
\end{array}$$

Bei dem rechten Parsingversuch in Abbildung 5.4 ist der Kellerinhal, $adBd$, in Zeile 5 kein lebensfähiger Präfix für G_{ab}. Das Scheitern des Parsingversuchs hätte schon hier entdeckt werden können. Für G'_{ab} anstelle von G_{ab} erhält man noch zusätzlich S und $S\$$ als lebensfähige Präfixe.

Berechnung lebensfähiger Präfixe

Definition 94 (Die nichtdet. LR(k)-Maschine)
Sei $G = (N, T, S, \mathcal{P})$ eine kontextfreie Grammatik, G' ihre \$ -Erweiterung, dann ist die LR(k)-Maschine

$$M_{G,k} = (Q, V, q_0, \delta, F)$$

ein nichtdeterministischer endlicher Automat mit spontanen Übergängen, wie folgt definiert:

$$\begin{array}{lcl}
Q & = & \textit{alle LR(k)-Elemente von } G', \\
V & = & N \cup T, \\
q_0 & = & [S' \to \cdot S\$, \varepsilon] \\
F & = & Q, \\
\ [A \to uX \cdot v, y] & \in & \delta([A \to u \cdot Xv, y], X) \\
 & & \textit{für } X \in V \\
\ [B \to \cdot v_1, z] & \in & \delta([A \to u \cdot Bv, y], \varepsilon) \\
 & & \textit{falls } B \in N \\
 & & \textit{und } z \in \mathrm{First}_k(vy) \\
 & & \textit{und } B \to v_1 \in \mathcal{P}
\end{array}$$

Die Tatsache, daß jeder Zustand von $M_{G,k}$ ein Finalzustand ist zeigt. daß es auf Finalzustände hier nicht ankommt. Die Auswahl der Wörter, die $M_{G,k}$ akzeptieren soll, wird durch die Partialität der Transitionsfunktion δ getroffen.

Lemma 73
Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik, G' ihre \$ -Erweiterung.

1. *Für $w \in V^*$ gilt*

   $w\$ \in L(M_{G,k})$ *gdw* w *ein lebensfähiger Präfix für* G *ist.*

2. *Für jedes $w \in V^*$ und jedes LR(k)-Element $[A \to u \cdot v, y]$ gilt*

   $[A \to u \cdot v, y] \in \delta(q_0, w)$ *gdw* $S \Rightarrow_r^* u_0 A z \Rightarrow_r u_0 u v z$

   wobei $w = u_0 u$, $A \to uv \in P$ und $y = k : z$

Beweis: Offensichtlich genügt es die Aussage (2) zu beweisen. Wir zeigen die beiden Richtungen getrennt.

Teil 1:

Implikation von links nach rechts. Induktion über die Definition von $\delta(q_0, w)$. Da in $M_{G,k}$ spontane Übergänge vorkommen können ist das nicht dasselbe wie eine Induktion über die Länge von w. Der Induktionsanfang, $w = \varepsilon$ ist offensichtlich. Wir unterscheiden, ob der letzte Übergang durch Einlesen eines Zeichens $X \in N \cup T \cup \{\$\}$ erfolgte oder spontan.

Einlesen von X

Wir betrachten also

$$[A \to uX \cdot v_0, y] \in \delta(q_0, wX)$$

mit

$$[A \to u \cdot Xv_0, y] \in \delta(q_0, w)$$

Nach Induktionsvoraussetzung gilt

$$S \Rightarrow_r^* u_0Az \Rightarrow_r u_0uXv_0z$$

mit

- $w = u_0u$, $A \to uXv_0 \in P$ und $y = k : z$

Diese Beziehungen bleiben ganz offensichtlich erhalten, wenn der Punkt um eine Stelle nach rechts verschoben wird.

spontane Übergänge

Hier gehen wir aus von einem Zustand

$$[A \to u \cdot Bv_0, y] \in \delta(q_0, w)$$

mit $B \in N$, für den die Induktionsvoraussetzung gilt, d.h.:

$$S \Rightarrow_r^* u_0Az \Rightarrow_r u_0uBv_0z$$

mit

- $w = u_0u$, $A \to uBv_0 \in P$ und $y = k : z$

Die Behauptung ist dann für einen Zustand

$$[B \to \cdot v_1, y_1]$$

mit $y_1 \in \text{First}_k(v_0 y)$ zu zeigen.

Nach Definition von First_k gibt es eine Ableitung in G der Form

$$v_0 y \Rightarrow_r^* y_2 \text{ mit } y_2 \in T^*, y_1 = k : y_2$$

Wegen $y = k : z$ gilt dann auch

$$v_0 z \Rightarrow_r^* z_1 \text{ mit } y_1 \in T^*, y_1 = k : z_1$$

Die nach Induktionsvoraussetzung existierende Ableitung $S \Rightarrow_r^* u_0 u B v_0 z$ läßt sich somit zu einer Ableitung $S \Rightarrow_r^* u_0 u B z_1$ fortsetzen. Zusammengefaßt erhalten wir:

$$S \Rightarrow_r^* u_0 u B z_1 \Rightarrow_r u_0 u v_1 z_1$$

mit

- $w = u_0 u$, $B \to v_1 \in P$, $y_1 = k : z_1$

2.Teil

Zum Beweis der umgekehrten Implikation gehen wir von einer Herleitung

$$S \Rightarrow_r^* u_0 A z \Rightarrow_r u_0 u v z$$

aus und wollen $[A \to u \cdot v, y] \in \delta(q_0, w)$ zeigen. Das geschieht durch Induktion über die Länge der Herleitung und über $|u|$. Der Induktionsanfang, $u_0 = z = \varepsilon, A = S$, ist wieder einfach. Auch der Fall, daß die Länge der Herleitung unverändert bleibt und nur das mit u bezeichnete Teilwort um einen Buchstaben länger wird, ist trivial.

Es bleibt also die Situation

$$S \Rightarrow_r^* u_0 A z \Rightarrow_r u_0 u B v_0 z \Rightarrow_r u_0 u v_1 v_0 z$$

zu betrachten mit $z, v_0 \in T^*$ und $B \to v_1 \in P$. Nach Induktionsvoraussetzung gilt

$$[A \to u \cdot B v_0, y] \in \delta(q_0, w)$$

für $w = u_0 u$ und $y = k : z$. Zu zeigen ist

$$[B \to \cdot v_1, y_1] \in \delta(q_0, w)$$

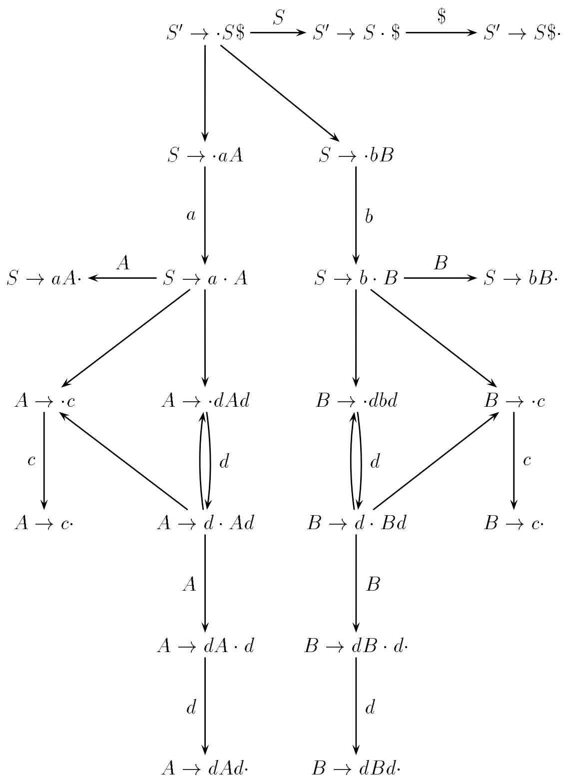


Abbildung 5.5: Der nichtdeterministische LR(0)-Automat zu G_{ab}

für $y_1 = k : v_0 z$. Es ist also zu zeigen, daß in $M_{G,k}$ ein spontaner Übergang von $[A \rightarrow u \cdot Bv_0, y]$ nach $[B \rightarrow \cdot v_1, y_1]$ möglich ist. Nach Definition ist dafür nur noch $y_1 \in \text{First}_k(v_0 y)$ zu zeigen. Im vorliegenden Fall sind v_0,y,z terminale Wörter, so daß $\text{First}_k(v_0 y) = \{k : v_0 y\}$ gilt und wir sind fertig.

∎

In Abbildung 5.5 ist $M_{G_{ab},0}$ zu sehen. Die Berechnungen eines nichtdeterministischen endlichen Automaten können immer noch recht aufwendig sein. Aber wir habe in Theorem 34 gesehen, wie man dem begegnen kann durch Transformation in einen deterministischen endlichen Automaten. Das wollen wir uns jetzt zunutze machen.

Definition 95 (Die deterministische LR(k)-Maschine)
Sei $G = (N, T, S, \mathcal{P})$ eine kontextfreie Grammatik. Dann ist die deterministische LR(k)-Maschine für G, bezeichnet mit

$$D_{G,k},$$

der zu $M_{G,k}$ äquivalente deterministische endliche Automat.

Die Konstruktion von $D_{G,k}$ aus $M_{G,k}$ erfolgt mit dem üblichen Verfahren (Myhill-Buechi-Verfahren, Teilmengenkonstruktion) Der deterministische LR(0)-Automat für die Beispielgrammatik G_{ab} ist in Abbildung 5.6, die dabei auftretenden Zustände, das müssen nach Konstruktion Teilmengen von Zuständen von $M_{G_{a,b},0}$ aus Abbildung 5.5 sein, sind dabei:

$$\begin{array}{lcl@{\qquad}lcl}
q_0 & = & \{S' \rightarrow \cdot S\$, S \rightarrow \cdot aA, S \rightarrow \cdot bB\} & q_8 & = & \{A \rightarrow c\,\cdot\} \\
q_1 & = & \{S \rightarrow a \cdot A, A \rightarrow \cdot c, A \rightarrow \cdot dAd\} & q_9 & = & \{S \rightarrow bB\,\cdot\} \\
q_2 & = & \{S \rightarrow b \cdot B, B \rightarrow \cdot c, B \rightarrow \cdot dBd\} & q_{10} & = & \{B \rightarrow c\,\cdot\} \\
q_3 & = & \{A \rightarrow d \cdot Ad, A \rightarrow \cdot c, A \rightarrow \cdot dAd\} & q_{11} & = & \{A \rightarrow dAd\,\cdot\} \\
q_4 & = & \{B \rightarrow d \cdot Bd, B \rightarrow \cdot c, B \rightarrow \cdot dBd\} & q_{11} & = & \{A \rightarrow dAd\,\cdot\} \\
q_5 & = & \{A \rightarrow dA \cdot d\} & q_{12} & = & \{B \rightarrow dBd\,\cdot\} \\
q_6 & = & \{B \rightarrow dB \cdot d\} & q_{13} & = & \{S' \rightarrow S \cdot \$\} \\
q_7 & = & \{S \rightarrow aA\,\cdot\} & q_{14} & = & \{S' \rightarrow S\$\,\cdot\}
\end{array}$$

Lemma 74
Für die deterministische LR(k)-Maschine $D_{G,k} = (Q, V, q_0, \Delta)$ zur ktf. Grammatik $G = (N, T, S, P)$ ist für alle $w \in V^*$, so daß $\Delta(q_0, w)$ definiert ist, $\Delta(q_0, w)$ die Menge aller gültigen LR(k)-Elemente für w.

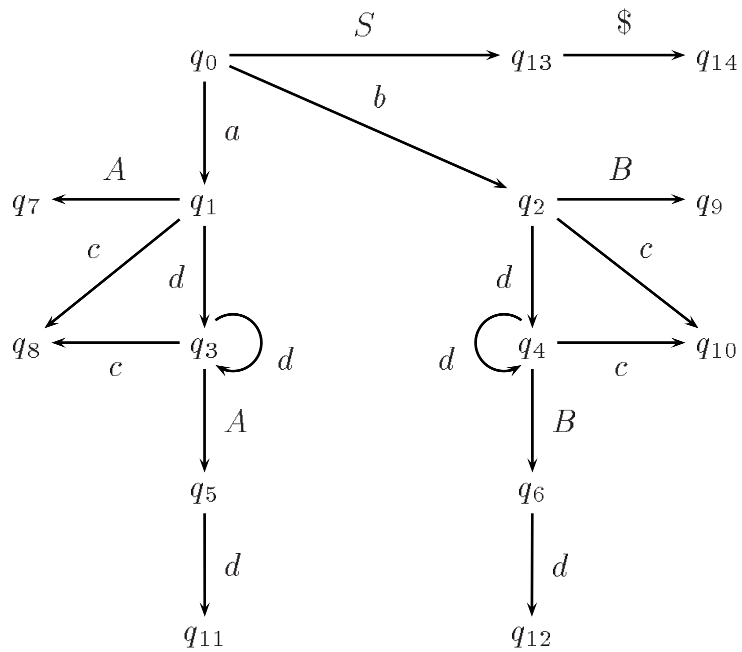


Abbildung 5.6: Der deterministische LR(0)-Automat zu G_{ab}

Beweis: Folgt aus

- der Definition gültiger LR(k)-Elemente
- Lemma 73
- dem Zusammenhang $\bar{\delta}(s_0, w) = \Delta(q_0, w)$ zwischen der nichtdeterministische LR(k)-Maschine, $M_{G,k} = (S, V, s_0, \delta)$ und der deterministische $D_{G,k} = (Q, V, q_0, \Delta, F)$ für G.

5.3.2 Der kanonische LR(k)-Parser

Wir sind jetzt in der Lage die eingangs gestellte Frage, wie man zu einer kontextfreien Grammatik eine Parsertabelle konstruiert, zu beantworten. Die folgende Definition geben wir nur für $k = 0$ und $k = 1$. Den allgemeinen, etwas komplizierteren Fall, kann man z.B. in [36, Abschnitt 6.3] nachlesen.

Definition 96 (Der kanonische LR(k)-Parser)
$G = (N, T, S, \mathcal{P})$ sei ktf Grammatik, $k \in \{0, 1\}$, $D_{G,k} = (Q, V, q_0, \Delta, F)$ die zugehörige deterministische LR(k)-Maschine.

Der kanonische LR(k)-Parser $K = K_{G,k}$ für G besitzt als Zustände die Zustände von $D_{G,k}$.

Sei $q \in Q$, $c \in T$ und $A \in N$. Die Parsertabelle für K wird durch die folgenden Regeln konstruiert:

- *falls $\Delta(q,c)$ definiert ist kommt* **shift**$\Delta(q,c)$ *im Feld (q,c) vor.*
- *falls $\Delta(q,A)$ definiert ist kommt $\Delta(q,A)$ im Feld (q,A) vor.*
- *falls ein Element $[A \to u\cdot, y]$ in q liegt kommt* **reduce** *$A \to u$ im Feld (q,y) vor.*
- *falls $[S' \to S \cdot \$, y]$ in q vorkommt steht acc im Feld $(q,\$)$.*

Probiert man diese Definition für die Beispielgrammatik G_{ab} aus Beispiel 64 anhand des deterministischen LR(0)-Automaten aus Abbildung 5.6 aus, so erhält man:

Beispiel 65 (Der kanonische LR(0)-Parser für G_{ab})

q	a	b	c	d	$\$$	A	B	S
0	*s1*	*s2*						*13*
1			*s8*	*s3*		*7*		
2			*s10*	*s4*			*9*	
3			*s8*	*s3*		*5*		
4			*s10*	*s4*			*6*	
5				*s11*				
6				*s12*				
7	*r1*	*r1*	*r1*	*r1*	*r1*			
8	*r3*	*r3*	*r3*	*r3*	*r3*			
9	*r2*	*r2*	*r2*	*r2*	*r2*			
10	*r5*	*r5*	*r5*	*r5*	*r5*			
11	*r4*	*r4*	*r4*	*r4*	*r4*			
12	*r6*	*r6*	*r6*	*r6*	*r6*			
13					*acc*			

Theorem 75
Sei $G = (N, T, S, \mathcal{P})$ eine kontextfreie Grammatik, $k \geq 0$, $K = (Z, \delta_1, \delta_2)$ der zugehörige LR(k)-Parser und $w \in T^*$, dann gilt

$$K \text{ akzeptiert ein Wort } w\$$$
$$gdw$$
$$\text{in } G' \text{ gilt } S' \Rightarrow_r^* w\$$$

Beweis:

1.Teil: Wir gehen davon aus, daß K die Eingabe w akzeptiert und wollen $S \Rightarrow_r^* w$ zeigen. Das ist einfach. Lassen wir in der erfolgreichen Berechnungsfolge von K alle Zustandssymbole im Keller weg, so erhalten wir eine erfolgreiche Berechnung des Automaten $\mathcal{A}_G^{LR}$. Nach Theorem 52 folgt die gewünscht Ableitbarkeit in G.

2.Teil: Vorausgesetzt ist jetzt $S' \Rightarrow_r^* w$. $D_{G,0} = (Q, V, q_0, \Delta)$ sei die deterministische LR(0)-Maschine zu G aus Definition 95. Wir brauchen für den vorliegenden Beweis eine Hilfskonstruktion. Zu jedem Wort $w = w_1 \dots w_n \in (N \cup T)^*$ sei $keller(w)$ von der Form $q_0 w_1 q_1 \dots w_n q_n$, wobei

- q_0 der Anfangszustand von $D_{G,0}$ ist und
- $q_{i+1} = \Delta(q_i, w_i)$ für $0 \leq i < n$.

Für das leere Wort setzt man sinnvollerweise $keller(\varepsilon) = q_0$. Die Einermenge $\{[S' \to S\$\,\cdot]\}$ ist nach Konstruktion ein Zustand in Q, den wir im folgenden mit q_f bezeichnen wollen.

Wir betrachten eine Ableitung

$$S' \Rightarrow_r^* w_1 A w_2 \$$$

in G', wobei A das am weitesten rechts stehende Nichtterminalzeichen ist, $w_1 = w_{1,1} \dots w_{1,n}$ und um der notationellen Einheitlichkeit willen $w_{1,n+1} = A$. Durch Induktion über die Länge dieser Ableitung beweisen wir jetzt:

1. in K gilt $keller(w_1 A) \mid w_2\$ \Rightarrow^* S' \mid \varepsilon$ und
2. für jedes $0 \leq j \leq n+1$ gibt es eine Produktion in q_j (zur Erinnerung: die Zustände von $D_{G,0}$ sind Mengen punktierter Produktionen von G') der Form $B \to v_{0,j} \cdot w_{1,j+1} v_{1,j}$.

Zum Induktionsbeginn betrachten wir die einschrittige Grammatikableitung $S' \Rightarrow S\$$. Die Behauptung reduziert sich in diesem Fall auf

$$keller(S) \mid \$ \Rightarrow S' \mid \varepsilon$$

Dabei ist $keller(S) = q_0 S q_1$. Nach Definition gilt $[S' \to \cdot S\$, \varepsilon] \in q_0$ und daher auch $[S' \to S \cdot \$] \in q_1 = \Delta(q_0, S)$. Die Parsertabelle für K erlaubt also an dieser Stelle das nächste Zeichen , also \$, auf den Keller zu legen. Das führt zu der Konfiguration $q_0 S q_1 \$ q_f \mid \varepsilon$, die wir vereinbarungsgemäß als $S' \mid \varepsilon$ schreiben.

Zum Beweis des Induktionsschrittes betrachten wir die Ableitung

$$S' \Rightarrow_r^* w_1 A w_2 \$ \Rightarrow w_1 u w_2 \$$$

Im letzten Ableitungsschritt wurde also die Grammatik Regel $A \to u$ angewendet. Für späteren Gebrauch schreiben wir schon einmal $u = u_1 \dots u_m$. Nach Induktionsvoraussetzung gilt

1. $keller(w_1 A) \mid w_2 \$ \Rightarrow^* S' \mid \varepsilon$

Aber was ist eigentlich die Induktionsbehauptung? Dazu müssen wir das am weitesten rechts stehende Nichtterminalzeichen, nennen wir es C, in $w_1 u w_2 \$$ lokalisieren. Wir führen hier den Fall weiter aus daß C das letzte Zeichen von w_1 ist. Die anderen Fälle, C liegt irgendwo in w_1, C liegt innerhalb u und es gibt überhaupt kein Nichtterminalzeichen in $w_1 u w_2 \$$, müssen natürlich auch nachgeprüft werden. Wer aber den hier vorgeführten Spezialfall verstanden hat, dem werden sie keine Schwierigkeiten bereiten. Auf den letzten Fall werden wir weiter unten noch einmal zurückkommen.

Wir wollen also zeigen, daß in K die Ableitung

$$keller(w_1) \mid u w_2 \$ \Rightarrow^* S' \mid \varepsilon$$

möglich ist.

Aus der Voraussetzung $S' \Rightarrow_r^* w_1 A w_2 \$ \Rightarrow w_1 u w_2 \$$ folgt mit Lemma 73, und Lemma 74 , daß $[A \to \cdot u, y]$ mit $y = \mathrm{First}_1(w_2)$ in $\Delta(q_0, w_1) = q_n$ liegt.

Nach Definition der Parsertabelle steht in dem Feld (q_n, u_1) der Eintrag sq^1 für $q^1 = \Delta(q_n, u_1)$. Damit ist in K der Übergang

$$keller(w_1) \mid u w_2 \$ \Rightarrow keller(w_1) u_1 q^1 \mid u_2 \dots u_m w_2 \$$$

möglich ist. Wiederholt man dieses Argument für $q^{i+1} = \Delta(q^i, u_{i+1})$ so erhält man nach m Schritten

$$keller(w_1) \mid u w_2 \$ \Rightarrow^* keller(w_1) u_1 q^1 \dots u^m q^m \mid w_2 \$$$

Es gilt $[A \to u \cdot, y] \in q^m$ und damit erlaubt die Parsertabelle von K in dieser Konfiguration einen Reduktionsschritt mit der Regel $A \to u$. Man beachte, daß auch die Vorausschau y mit dem ersten Buchstaben des noch einzulesenen Wortes w_2 übereinstimmt. Das führt zu der Ableitungskette

$$keller(w_1) \mid uw_2\$ \Rightarrow^* keller(w_1)Aq_{n+1} \mid w_2\$$$

und damit wie gewünscht zu

$$keller(w_1) \mid uw_2\$ \Rightarrow^* S' \mid \varepsilon$$

Schauen wir uns noch kurz den Fall an, daß in dem Wort $w_1uw_2\$$ kein Nichtterminalzeichen mehr vorkommt. In diesem Fall ist $keller(\varepsilon) \mid w_1uw_2\$ \Rightarrow^* S' \mid \varepsilon$ zu zeigen.

Aus $S' \Rightarrow_r^* w_1Aw_2\$ \Rightarrow w_1uw_2\$$ folgt, daß $w_{1,1} \dots w_{1,j+1}$ für alle $1 \leq j < n$ ein lebensfähiger Präfix ist. Nach den Lemmas 73,74 ist $\Delta(q_0, w_{1,1} \dots w_{1,j+1})$ definiert. Insbesondere ist der Übergang $\Delta(q_i, w_{1,j+1})$ definiert. Nach Definition der LR(k)-Maschinen muß also ein LR(k)-Element der Form $[B \to v_0 \cdot w_{1,j+1}v_1, y]$ für ein geeignetes y in q_j liegen. Für $j = 0$ kommt also ein Element $[B \to v_0 \cdot w_1v_1, y]$ in q_0 vor. Somit läßt die Parsertabelle von K den Übergang

$$q_0 \mid w_1uw_2\$ \Rightarrow q_0w_{1,1}q_1 \mid w_{1,2} \dots w_{1,n}uw_2\$$$

zu. Durch n-fache Wiederholung dieses Arguments kommt man zu:

$$q_0 \mid w_1uw_2\$ \Rightarrow q_0w_{1,1}q_1 \dots w_{1,n}q_n \mid uw_2\$$$

Von hier aus geht es weiter wie oben.

■

5.3.3 LR(k)-Grammatiken

Von besonderem Interesse sind solche Grammatiken, deren LR(k)-Parsertabelle eindeutige Anweisungen gibt. Die Entscheidung, ob ein vorgelegtes Wort zum Sprachschatz der Grammatik gehört, läßt sich dann in einem linearen Durchlauf durch das Wort treffen ohne verschiedene Alternativen durchzuprobieren.

Definition 97 (LR(k)-Grammatiken)
Eine Grammatik G heißt eine LR(k)-Grammatik, *wenn der kanonische LR(k)-Parser deterministisch ist, d.h. in jedem Feld der LR(k)-Parsertabelle für G steht höchstens ein Eintrag.*

Offensichtlich sind G_{AEXP} und G_{ab} Beispiele für LR(0)-Grammatiken.

Definition 98 (Konflikte)
Sei $D_{G,k}$ der deterministische LR(k)-Automat zur Grammatik G. Ein Zustand q von $D_{G,k}$ enthält einen reduce-reduce-Konflikt, *wenn es zwei verschiedene Produktionen $A \to v$, $B \to u$ und ein Wort $y \in k : T^*$ gibt, so daß*

$$[A \to v\cdot, y] \in q \text{ und } [B \to u\cdot, y] \in q$$

Ein Zustand q von $D_{G,k}$ enthält einen shift-reduce-Konflikt, *wenn es zwei Produktionen $A \to vcw$, $B \to u$ mit $c \in T$ und Wörter $y, z \in k : T^*$ mit $1 : y = c$ gibt, so daß*

$$[A \to v \cdot cw, z] \in q \text{ und } [B \to u\cdot, y] \in q$$

Lemma 76
Eine Grammatik G ist genau dann eine $LR(k)$-Grammatik, wenn ihr deterministischer LR(k)-Automat weder einen reduce-reduce-Konflikt noch einen shift-reduce-Konflikt enthält.

Beweis: Läßt sich mühelos durch Auflösen der Definitionen zeigen.

∎

Wir betrachten die folgende Grammatik als ein Beispiel einer LR(1)-Grammatik, die keine LR(0)-Grammatik ist.

Beispiel 66

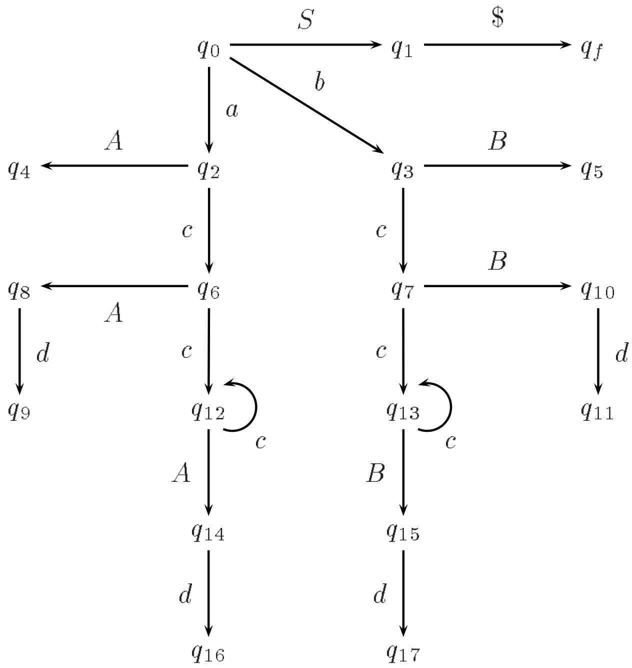
$$\begin{array}{llll} 1 & S \to aA & 2 & S \to bB \\ 3 & A \to \varepsilon & 4 & A \to cAd \\ 5 & B \to \varepsilon & 6 & B \to cBd \end{array}$$

$$\mathcal{L}(G_{ab\varepsilon}) = \{ac^n d^n \mid n \geq 0\} \cup \{bc^n d^n \mid n \geq 0\}$$

Die deterministische LR(1)-Maschine für $G_{ab\varepsilon}$ ist in Abbildung 5.7 zu sehen. Die daraus resultierende LR(1)-Parser-Tabelle für $G_{ab\varepsilon}$ in Abbildung 5.8.

Wir stellen zur Abrundung dieses Abschnitts ohne Beweis noch einige wichtige Sätze über LR(k)-Grammatiken zusammen.

Theorem 77
Ist G eine $LR(k)$-Grammatik, dann ist G auch eine $LR(k')$-Grammatik für jedes $k' \geq k$.



Die Zustände sind dabei gegeben durch

$$\begin{array}{lll}
q_0 = \{[S' \to \cdot S\$, \varepsilon], [S \to \cdot aA, \$], [S \to \cdot bB, \$]\} & & q_1 = \{[S' \to S \cdot \$, \varepsilon]\} \\
q_2 = \{[S \to a \cdot A, \$], [A \to \cdot cAd, \$], [A \to \cdot, \$]\} & & q_f = \{[S' \to S\$ \cdot, \varepsilon]\} \\
q_3 = \{[S \to b \cdot B, \$], [A \to \cdot cBd, \$], [A \to \cdot, \$]\} & & q_4 = \{[S \to aA \cdot, \$]\} \\
q_6 = \{[A \to \cdot cAd, d], [A \to c \cdot Ad, d], [A \to \cdot, d]\} & & q_5 = \{[S \to bB \cdot, \$]\} \\
q_7 = \{[B \to \cdot cBd, d], [B \to c \cdot Bd, d], [B \to \cdot, d]\} & & q_8 = \{[A \to cA \cdot d, \$]\} \\
q_9 = \{[A \to cAd \cdot, \$]\} & & q_{10} = \{[B \to cB \cdot d, \$]\} \\
q_{12} = \{[A \to c \cdot Ad, d], [A \to \cdot cAd, d], [A \to \cdot, d]\} & & q_{11} = \{[B \to cBd \cdot, \$]\} \\
q_{13} = \{[B \to c \cdot Bd, d], [B \to \cdot cBd, d], [B \to \cdot, d]\} & & q_{14} = \{[A \to cA \cdot d, d]\} \\
q_{16} = \{[A \to cAd \cdot, d]\} & & q_{15} = \{[B \to cB \cdot d, d]\} \\
q_{17} = \{[B \to cBd \cdot, d]\} & &
\end{array}$$

Abbildung 5.7: Die deterministische LR(1)-Maschine für $G_{ab\varepsilon}$

q	a	b	c	d	$	A	B	S
0	s2	s3						1
1					acc			
2			s6		r3	4		
3			s7		r5		5	
4					r1			
5					r2			
6			s12	r3		8		
7			s13	r5			10	
8				s9				
9					r4			
10				s11				
12			s12	r3		14		
13			s13	r5			15	
14				s16				
15				s17				
16				r4				
17				r6				

Abbildung 5.8: LR(1)-Parser-Tabelle für $G_{ab\varepsilon}$

Beweis : Siehe z.B. [36, Theorem 6.40, Seite 53].

Theorem 78
Für jedes $k \geq 1$ gibt es zu jeder LR(k)-Grammatik eine äquivalente LR(1)-Grammatik.

Beweis : Siehe z.B. [36].

5.3.4 LALR-Parser

Motivation für LALR-Parser

Die Anzahl der Zustände eines LR(1)-Parsers nimmt noch sehr rasch mit der Größe der Grammatik zu, wobei man zusätzlich beobachtet, daß sich manche Zustände nur geringfügig voneinander unterscheiden. Diese Beobachtung führte zur Einführung der LALR-Parser, die heute in der Praxis hauptsächlich benutzt werden.

Definition 99 (Kern eines LR(k)-Elements)
Ist $[A \to u \cdot w, y]$ ein LR(k)-Element, so nennen wir die erste Komponente $A \to u \cdot w$, den Kern *(core) des LR(k)-Elements. Der Kern einer Menge Q von LR(k)-Elementen ist die Menge der Kerne ihrer Elemente, d.h. $Kern(Q) = \{Kern(q) \mid q \in Q\}$*

Lemma 79
Sei $D_{G,k} = (Q, V \cup \{\$\}, \Delta)$ der deterministische LR(k)-Automat für die kontextfreie Grammatik G, und $q_1, q_2 \in Q$ zwei Zustände. Dann gilt für alle $X \in V \cup \{\$\}$:

$$\begin{array}{ll} \textit{wenn} & kern(q_1) = kern(q_2) \\ \textit{dann} & kern(\Delta(q_1, X)) = kern(\Delta(q_2, X)) \end{array}$$

Beweis Ist $M_{G,k} = (S, V \cup \{\$\}, \delta)$ der nichtdeterministische LR(k)-Automat für G, dann genügt es wegen

$$\begin{array}{rcl} \Delta(q, X) & = & \bar{\delta}(q, X) \\ & = & \varepsilon\text{-}cl(\{s \in S \mid \exists s_1 \in \bar{\delta}(q, \varepsilon) \wedge s \in \delta(s_1, X)\}) \end{array}$$

1. für $s_1, s_2 \in S$ mit $kern(s_1) = kern(s_2)$ auch $kern(\delta(s_1, X)) = kern(\delta(s_2, X))$ und

2. für $Q_1, Q_2 \subseteq S$ mit $kern(Q_1) = kern(Q_2)$ auch $kern(\varepsilon\text{-}cl(Q_1)) = kern(\varepsilon\text{-}cl(Q_1))$

zu zeigen. Diese beiden Behauptungen lassen sich aber leicht aus den beteiligten Definitionen herleiten.

∎

Definition 100
Sei $D_{G,k}$ der deterministische LR(k)-Automat für die Grammatik G. Für $q \in Q$ setzen wir

$$\bar{q} = \bigcup \{q_1 \in Q \mid kern(q) = kern(q_1)\}$$

Dann konstruiert man den LA-Automaten für G, bezeichnet mit $LA_{G,k} = (\bar{Q}, V \cup \{\$\}, \bar{\Delta})$ wie folgt:

$$\begin{array}{lcl} \bar{Q} & = & \{\bar{q} \mid q \in Q\} \\ \bar{\Delta}(\bar{q}, X) & = & \overline{\Delta(q, X)} \end{array}$$

Damit diese Definition Sinn macht muß man wissen, daß aus $\bar{q_1} = \bar{q_2}$ auch $\overline{\Delta(q_1, X)} = \overline{\Delta(q_1, X)}$ gilt. Aber das folgt leicht aus dem vorangegangen Lemma.

Beispiel 67

$$\begin{array}{llll} 0 & S' \to S & & \\ 1 & S \to CC & & \\ 2 & C \to cC & 3 & C \to d \end{array}$$

Kanonische LR(1)-Parsertabelle für G_{CC}

q	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

LALR(1)-Parsertabelle für G_{CC}

q	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3			
5			r1		
89	r2	r2	r2		

5.3.5 Bison

Zu den in der Unix-Welt bekanntesten Compilergeneratoren, die aus kontextfreien Grammatiken LALR(1)-Parser erzeugen gehören YACC und Bison. Die Eingabedatei für das Bisonprogramm ist von der folgenden Form:

```
%{
C DECLARATIONS
%}

BISON DECLARATIONS

%%
GRAMMAR RULES
%%

ADDITIONAL C CODE
```

Eine solche Eingabedatei wird in Bison eine Grammatikdatei genannt. Dabei sind '`%%`', '`%{`' und '`%}`' Interpunktionszeichen, die verschiedenen Abschnitte voneinander trennen.

Bison Grammatikdatei für G_{CC}

```
%%
s:    c c
;
c:    'c' c
      | 'd'
;
%%
```

Nichtterminalzeichen werden mit kleinen Buchstaben geschrieben, Terminalzeichen in Anführungszeichen gesetzt. Anstelle des Pfeils in der Darstellung von Produktionsregeln steht hier nach dem Nichtterminalzeichen auf der linken Seite ein Doppelpunkt :. In diesem Beispiel gibt es keine Deklarationen und keine Aktionsteile.

Als Ausgabe produziert das Bison Programm C Kode für einen Parser, der uns hier nicht besonders interessiert. Läßt man Bison mit der *verbose* Option laufen, so erhält man zusätzlich einen Ausdruck der LALR(1)-Parsertabelle.

Beispiel 68 (Ausgabe für G_{CC})

```
rule 1    s -> c c
rule 2    c -> 'c' c
rule 3    c -> 'd'
Terminals, with rules where they appear
$ (-1) 'c' (99) 2 'd' (100) 3  error (256)
Nonterminals, with rules where they appear
s (5)    on left: 1
c (6)    on left: 2 3, on right: 1 2
state 0
    'c'         shift, and go to state 1
    'd'         shift, and go to state 2
    s           go to state 6
    c           go to state 3
state 1
    c  ->  'c' . c   (rule 2)
    'c'         shift, and go to state 1
    'd'         shift, and go to state 2
    c           go to state 4
state 2
    c  ->  'd' .   (rule 3)
    $default    reduce using rule 3 (c)
state 3
    s  ->  c . c   (rule 1)
    'c'         shift, and go to state 1
    'd'         shift, and go to state 2
    c           go to state 5
state 4
    c  ->  'c' c .   (rule 2)
    $default    reduce using rule 2 (c)
```

```
state 5
    s  ->  c c .   (rule 1)
    $default    reduce using rule 1 (s)
state 6         $       go to state 7
state 7         $       go to state 8
state 8         $default accept
```

Die folgende Tabelle gibt an, wie die Numerierung der Zustände im Bison Programm sich in die in der Parsertabelle für G_{CC} benutzten umrechnen lassen.

Bison-Ausgabe	Parser-Tabelle
0	0
1	36
2	47
3	2
4	89
5	5
6	1

Es sind jeweils die punktierten Produktionen, die den Kern eines LALR(1)-Zustandes bilden, angegeben. Der Abschluß unter spontanen übergängen wird allerdings weggelassen. In "state 1" wären z.B. noch die punktierten Produktionen `c -> . 'c' c` und `c -> .'d'` zu ergänzen. Die Vorausschaubuchstaben werden nur im Konfliktfall angezeigt. Dem Zustand 0 ist keine Menge von punktierten Produktionen zugeordnet. Wir hatten in der vorangegangen Darstellung durch Einführung eines neuen Startzeichens S' eine uniforme Beschreibung auch für den Zustand 0 benutzt. Es geht, wie man sieht, auch anders.

Beispiel 69 (G_{RPCALC})

0	$I \rightarrow \varepsilon$	1	$I \rightarrow IL$
2	$L \rightarrow N$	3	$L \rightarrow E\backslash$
4	$E \rightarrow num$	5	$E \rightarrow EE+$
6	$E \rightarrow EE-$	7	$E \rightarrow EE*$
8	$E \rightarrow EE/$	9	$E \rightarrow EE\hat{}$
10	$E \rightarrow E\ n$		

Bison-Grammatikdatei für G_{RPCALC}

```
%{
#define YYSTYPE double
#include <math.h>
%}
 %token NUM
 %%
input:  /* empty */
       | input line
;
 line:    '\n'
   | exp '\n' { printf ("\t%.10g\n", $1);}
;
 exp:     NUM          { $$ = $1;       }
       | exp exp '+'   { $$ = $1 + $2;  }
       | exp exp '-'   { $$ = $1 - $2;  }
       | exp exp '*'   { $$ = $1 * $2;  }
       | exp exp '/'   { $$ = $1 / $2;  }
     /* Exponentiation */
     | exp exp '^'   { $$ = pow ($1, $2);}
     /* Unary minus    */
       | exp 'n'       { $$ = -$1; }
%%
```

Bison-Echo für G_{RPCALC}

```
Grammar
rule 1    input ->              /* empty */
rule 2    input -> input line
rule 3    line -> '\n'
rule 4    line -> exp '\n'
rule 5    exp -> NUM
rule 6    exp -> exp exp '+'
rule 7    exp -> exp exp '-'
rule 8    exp -> exp exp '*'
rule 9    exp -> exp exp '/'
rule 10   exp -> exp exp '^'
rule 11   exp -> exp 'n'
Terminals, with rules where they appear
$ (-1)
'\n' (10) 3 4
'*' (42) 8
```

```
'+' (43) 6
'-' (45) 7
'/' (47) 9
'^' (94) 10
'n' (110) 11
error (256)
NUM (258) 5
Nonterminals, with rules where they appear
input (11)
    on left: 1 2, on right: 2
line (12)
    on left: 3 4, on right: 2
exp (13)
    on left: 5 6 7 8 9 10 11,
    on right: 4 6 7 8 9 10 11
```

5.4 Lehrbücher

- [29], [23]
  Diese bereits im Kapitel über reguläre Sprachen genannte Bücher decken auch den Bereich der kontextfreien Sprachen ab

- [2], [1]
  Beide Bücher legen nicht so großen Wert auf theoretische Strenge, bieten dafür anschauliche und motivierende Anwendungsbeispiele aus dem Compilerbau.

5.5 Übungsaufgaben

Übungsaufgabe 5.5.1

(a) Zeigen Sie: $G_1 = (\{S\}, \{a, b, c\}, S, P_1)$ mit

$$P_1 = \{S \to aSb \mid aSc \mid \epsilon\})$$

besitzt für kein $k \in \mathbb{N}$ die $SLL(k)$-Eigenschaft.

(b) Bestimmen Sie das kleinste $k \in \mathbb{N}$ für das die Grammatik $G_2 = (\{S, X, Y, Z\}, \{a, b\}, S, P_2)$ mit

$$P_2 = \{S \to aX \mid aY, X \to aY \mid aZ, Y \to bS \mid bZ, Z \to a\})$$

die $SLL(k)$-Eigenschaft besitzt.

Übungsaufgabe 5.5.2

Transformieren Sie $G = (\{S, X, Y, Z\}, \{a, b\}, S, \{S \to aX \mid aY, X \to aY \mid aZ, Y \to bS \mid bZ, Z \to a\})$ in Chomsky-Normalform.

Überprüfen Sie mit dem Cocke-Kasami-Younger Algorithmus, ob $aababa \in L(G)$ gilt.

Übungsaufgabe 5.5.3

Betrachten Sie die kontextfreie Grammatik $G = (\{S, A\}, \{i, +, *,), (\}, S, P)$ mit $P = \{S \to (A + A), S \to A * A, A \to S, A \to i\}$.

(a) Geben Sie den zu G gehörigen LL-Kellerautomat $\mathcal{A}_G^{LL}$ an.

(b) Geben Sie eine Konfigurationsfolge von $\mathcal{A}_G^{LL}$ an, so daß $(i+i)*i*(i+i) \in L(G)$ mit leerem Keller akzeptiert wird.

Übungsaufgabe 5.5.4

Geben Sie alle lebensfähigen Präfixe von

$$G = (S, \{a, b\}, S, \{S \to aSb, S \to bSa, S \to \epsilon\})$$

an.

Übungsaufgabe 5.5.5

Betrachten Sie folgende Grammatik $G = (\{S', S\}, \{a, \$\}, S', P)$ mit

$$P = \{S' \to S\$, S \to aSa \mid a\} \ .$$

(a) Geben Sie die nicht-deterministische LR(1)-Kellermaschine zu G an.

(b) Geben Sie die deterministische LR(1)-Kellermaschine zu G an.

(c) Ist G eine LR(1)-Grammatik?

(d) Geben sie den LR(1)-Parser zu G an.

Übungsaufgabe 5.5.6

Betrachten Sie die Grammatik $G = (\{a, b, c\}, \{S\}, S, \{S \to c, S \to aSb\})$.

(a) Bestimmen Sie die nichtdet. $LR(1)$-Maschine $M_{G,1}$ zu G.

(b) Bestimmen Sie die det. $LR(1)$-Maschine $D_{G,1}$ zu G.

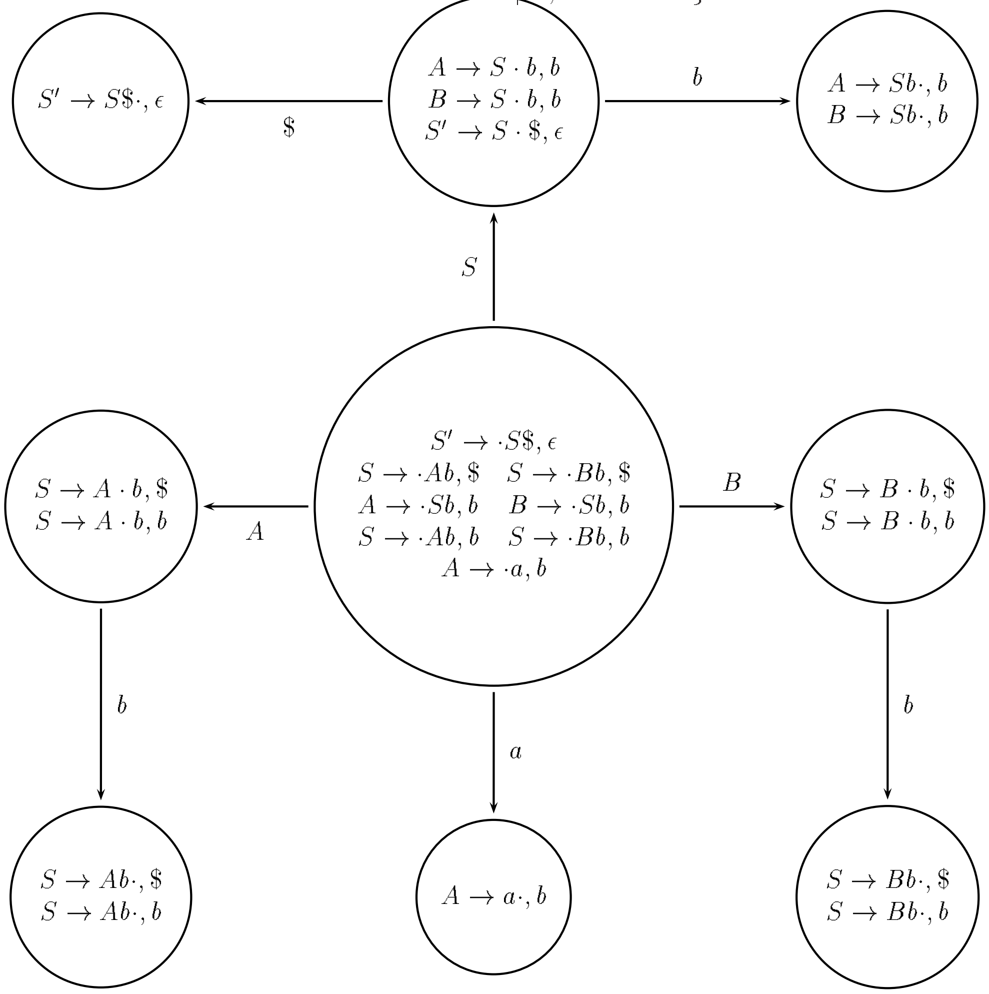
(c) Geben Sie den kanonischen $LR(1)$ Parser zu $D_{G,1}$ an

(d) Geben Sie eine Ableitung des Wortes $aacbb$ mit ihrem Parser an.

Übungsaufgabe 5.5.7

Die folgende Abbildung zeigt die deterministischen Kellermaschine zur Grammatik $(\{S', S, A, B\}, \{a, b\}, S', P)$ mit

$$P = \{ \quad S' \to S\$, \qquad S \to Ab \mid Bb, \\ A \to Sb \mid a, \quad B \to Sb\} \ .$$



Bestimmen Sie alle Konflikte, die sich für den zugehörigen LR-Parser ergeben.

Übungsaufgabe 5.5.8

Betrachten Sie die Grammatik $G = (\{a, b\}, \{S, A, B\}, S, \{S \rightarrow aAa, S \rightarrow aBa, A \rightarrow aSa, A \rightarrow a, B \rightarrow bBb, B \rightarrow a\})$.

Zeigen Sie, daß G nicht $LR(1)$ ist.

Übungsaufgabe 5.5.9

$G = (\{S', S, A, B\}, \{\$, a, b\}, S', P)$ sei die Grammatik mit den folgenden Produktionen:

$$
\begin{array}{llcll}
P = \{ & S' & \rightarrow & S\$, & (0) \\
 & S & \rightarrow & Aa, & (1) \\
 & S & \rightarrow & Bb, & (2) \\
 & A & \rightarrow & aBb, & (3) \\
 & A & \rightarrow & a, & (4) \\
 & B & \rightarrow & bAa, & (5) \\
 & B & \rightarrow & b & (6)\ \}
\end{array}
$$

Die Produktionen seien numeriert, wie durch die Zahlen in den Klammern angegeben.

(a) Bestimmen Sie die nicht-deterministische $LR(1)$-Maschine zu G.

(b) Bestimmen Sie die deterministische $LR(1)$-Maschine zu G.

(c) Bestimmen Sie die $LR(1)$-Parsertabelle zu G.

Übungsaufgabe 5.5.10

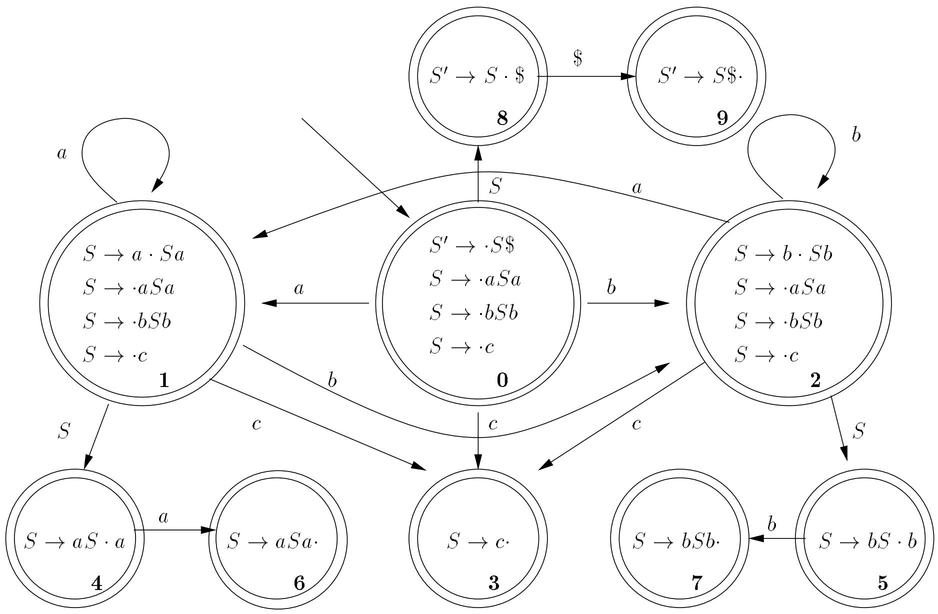
Betrachten Sie folgenden Grammatik $G = (\{S\}, \{a, b, c\}, S, P)$ mit

$$P = \{S \rightarrow aSa \mid bSb \mid c\}$$

(a) Geben Sie die Regeln des zu G gehörigen LR-Kellerautomaten $\mathcal{A}_G^{LR}$ an.

(b) Geben Sie eine vollständige Konfigurationsfolge von $\mathcal{A}_G^{LR}$ an, so daß $aca \in L(G)$ akzeptiert wird. Machen Sie in jedem Schritt deutlich, welche Regel Sie angewendet haben.

(c) Betrachten Sie die zu G zugehörige deterministische LR(0)-Kellermaschine:



Die fettgedruckte Zahl in einem Zustand ist der Name des entsprechenden Zustands.

Vervollständigen Sie bei der folgenden LR(0)-Parsertabelle zu G die Einträge in den Zeilen zu den Zuständen **2, 5** und **6** (Zeilen mit weißem Hintergrund). Die Produktionen sind dabei wie folgt numeriert:

Produktion	Nummer
$S \to aSa$	1
$S \to bSb$	2
$S \to c$	3

	a	b	c	$	S
0	s**1**	s**2**	s**3**		**8**
1	s**1**	s**2**	s**3**		**4**
2					
3	r3	r3	r3	r3	
4	s**6**				
5					
6					
7	r2	r2	r2	r2	
8				acc	

(d) Welche der Wörter aS, aca und $aaSa$ sind lebensfähige Präfixe von G und welche nicht? Begründen Sie jeweils Ihre Antwort.

5.6 Lösungen zu den Aufgaben

Lösung zu Aufgabe 5.5.1

(a) Wir betrachten $S \rightarrow aSc$ und $S \rightarrow aSc$. Es gilt:

$$
\begin{aligned}
\mathrm{First}_k(aSb\,\mathrm{Follow}_k(S)) &\supset \mathrm{First}_k(aSb)\\
&\supset \mathrm{First}_k(aaSbb)\\
&\vdots\\
&\supset \mathrm{First}_k(aa^nSb^nb)\\
&\supset k : \{a^n \mid n \leq k+1\}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{First}_k(aSc\,\mathrm{Follow}_k(S)) &\supset \mathrm{First}_k(aSc)\\
&\supset \mathrm{First}_k(aa^nSc^nc)\\
&\supset k : \{a^n \mid n \leq k+1\}
\end{aligned}
$$

Der Schnitt beider Mengen ist offenbar für kein k leer, also besitzt G_1 nicht $SLL(k)$-Eigenschaft.

(b) Im folgenden sei:

$$x\,\mathrm{First}_i(y) := \{xy' \mid y' \in \mathrm{First}_i(y)\}, x \in T^*, y \in (N \cup T)^*$$

(i) Betrachte $S \rightarrow aX$ und $S \rightarrow aY$:

$$
\begin{aligned}
\mathrm{First}(aX\ \mathrm{Follow}(S)) &= a\,\mathrm{First}(X)\\
&= \underline{aa}(\mathrm{First}(Y) \cup \mathrm{First}(Z))\\
\mathrm{First}(aY\ \mathrm{Follow}(S)) &= a\,\mathrm{First}(Y)\\
&= \underline{ab}(\mathrm{First}(S) \cup \{a\})
\end{aligned}
$$

$\mathrm{First}_2(aX\ \mathrm{Follow}(S)) \cap \mathrm{First}_2(aY\ \mathrm{Follow}(S)) = \emptyset$

(ii) Betrachte $X \rightarrow aY$ und $S \rightarrow aZ$:

$$
\begin{aligned}
\mathrm{First}(aY\ \mathrm{Follow}(X)) &= a\,\mathrm{First}(Y)\\
&= \underline{ab}(\mathrm{First}(S) \cup \mathrm{First}(Z))\\
\mathrm{First}(aZ\ \mathrm{Follow}(X)) &= a\,\mathrm{First}(Z)\\
&= \{\underline{aa}\}
\end{aligned}
$$

$\mathrm{First}_2(aY\ \mathrm{Follow}(X) \cap \mathrm{First}_2(aZ\ \mathrm{Follow}(X)) = \emptyset$

(iii) Betrachte $Y \rightarrow bS$ und $S \rightarrow bZ$:

$$
\begin{aligned}
\mathrm{First}(bS\ \mathrm{Follow}(Y)) &= b\,\mathrm{First}(S)\\
&= ba(\mathrm{First}(X) \cup \mathrm{First}(Y))\\
&= \underline{baa}(\mathrm{First}(Y) \cup \mathrm{First}(Z)) \cup\\
& \underline{bab}(\mathrm{First}(S) \cup \mathrm{First}(Z))\\
\mathrm{First}(bZ\ \mathrm{Follow}(Y)) &= b\,\mathrm{First}(Z)\\
&= \{\underline{ba}\}
\end{aligned}
$$

$$\mathrm{First}_3(bS\,\mathrm{Follow}(Y)) \cap \mathrm{First}_3(bZ\,\mathrm{Follow}(S)) = \emptyset$$

G_2 ist also $SLL(3)$ und G_2 ist nicht $SLL(2)$, da ba in dem Durchschnitt $\mathrm{First}_2(bS\,\mathrm{Follow}(Y)) \cap \mathrm{First}_2(bZ\,\mathrm{Follow}(S))$ liegt.

Lösung zu Aufgabe 5.5.2

(a) G ist ϵ-frei und besitzt keine Kettenproduktionen

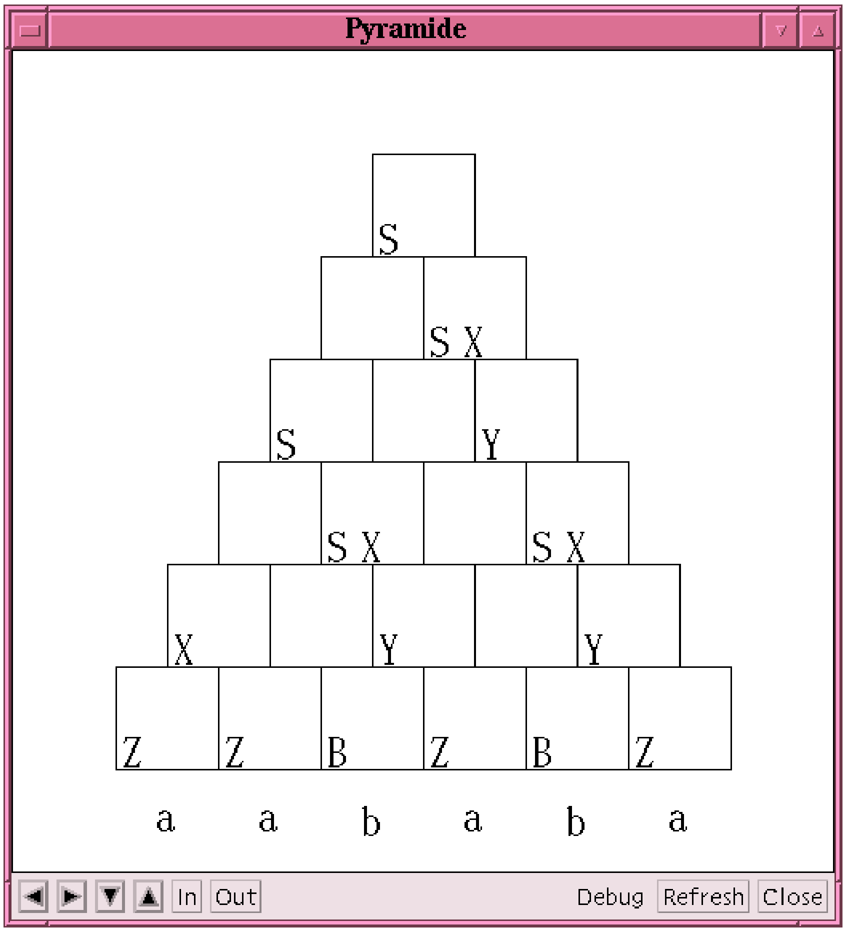
Ersetze a bzw. b durch A bzw. B und füge neue Produktionen $A \to a$ und $B \to b$ hinzu:

$$\begin{array}{ll} S & \to AX \mid AY \\ X & \to AY \mid AZ \\ Y & \to BS \mid BZ \\ Z & \to a \\ A & \to a \\ B & \to b \end{array}$$

Die resultierende Grammatik ist schon in CNF

Offenbar kann man a anstatt durch neues Nichtterminalzeichen A durch Z ersetzen und bekommt folgende äquivalente Chomsky-Normalform:

$$\begin{array}{ll} S & \to ZX \mid ZY \\ X & \to ZY \mid ZZ \\ Y & \to BS \mid BZ \\ Z & \to a \\ B & \to b \end{array}$$

(b) 

Startsymbol ist im obersten Kästchen, also gilt $aababa \in L(G)$.

Lösung zu Aufgabe 5.5.3

(a) Der LL-Kellerautomat besitzt die folgenden Regeln:

$$\begin{aligned}\mathcal{P}_{compare} &= \{iqi \to q, +q+ \to q, *q* \to q,)q) \to q, (q(\to q\} \\ \mathcal{P}_{produce} &= \{Sq \to)A + A(q, Sq \to A * Aq, Aq \to Sq, Aq \to iq\}\end{aligned}$$

(b)

Keller	Eingabe	Regel
Sq	(i+i)*i*(i+i)	**produce** $S \to A * A$
A*Aq	(i+i)*i*(i+i)	**produce** $A \to S$
A*Sq	(i+i)*i*(i+i)	**produce** $S \to (A + A)$
A*)A+A(q	(i+i)*i*(i+i)	**produce** $S \to (A + A)$
A*)A+Aq	i+i)*i*(i+i)	**compare**
A*)A+Aq	i+i)*i*(i+i)	**produce** $A \to i$
A*)A+iq	i+i)*i*(i+i)	**compare**
A*)A+q	+i)*i*(i+i)	**compare**
A*)Aq	i)*i*(i+i)	**produce** $A \to i$
A*)iq	i)*i*(i+i)	**compare**
A*)q	)*i*(i+i)	**compare**
A*q	*i*(i+i)	**compare**
Aq	i*(i+i)	**produce** $A \to S$
Sq	i*(i+i)	**produce** $S \to A * A$
A*Aq	i*(i+i)	**produce** $A \to i$
A*iq	i*(i+i)	**compare**
A*q	*(i+i)	**compare**
Aq	(i+i)	**produce** $A \to S$
Sq	(i+i)	**produce** $S \to (A + A)$
)A+A(q	(i+i)	**compare**
)A+Aq	i+i)	**produce** $A \to i$
)A+iq	i+i)	**compare**
)A+q	+i)	**compare**
)Aq	i)	**produce** $A \to i$
)iq	i)	**compare**
)q	)	**compare**
q		

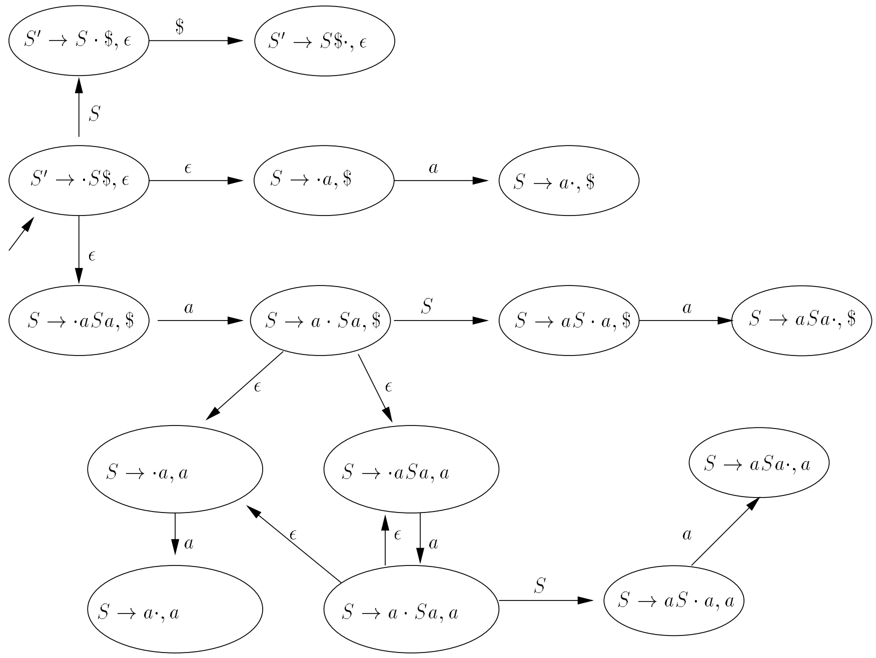
Die Ableitung ist weder deterministisch noch eindeutig: In der zweiten Zeile hätte man auch $S \to A * A$ produzieren und das Wort dann ableiten können.

Lösung zu Aufgabe 5.5.4

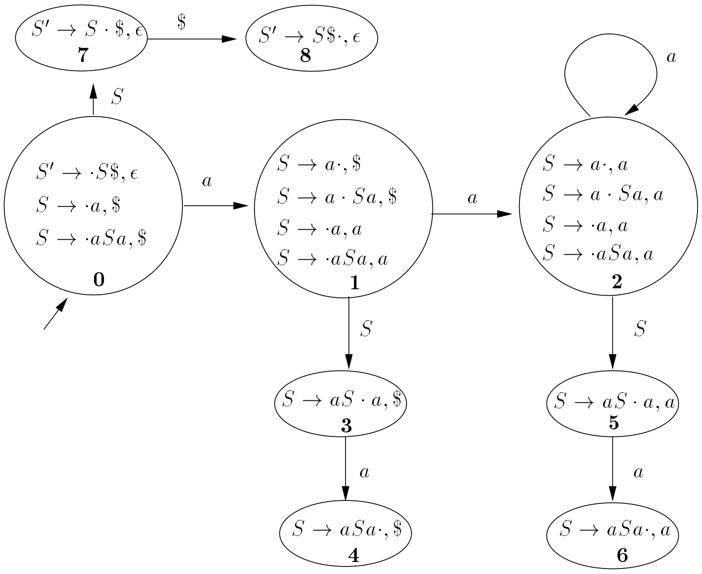
Es gilt $S \Rightarrow_r^* ySy^I \Rightarrow yaSb^Iy$, $S \Rightarrow_r^* ySy^I \Rightarrow yy^I$, und $S \Rightarrow_r^* ySy^I \Rightarrow ybSay^I, y \in \{a, b\}^*$. Also ist nach Definition $(a + b)^*(a + aS + aSb + b + bS + bSa + \epsilon)$ Menge aller lebensfähigen Präfixe.

Lösung zu Aufgabe 5.5.5

(a)



(b) Zahlen geben den Namen des jeweiligen Zustandes an. Alle Knoten sind Endzustände.



(c) Nein. Es gibt mehrere Begründungen:

(i) Knoten **2** enthält einen shift-reduce Konflikt.

(ii) G nicht $SLL(1)$ ist:
$First_1(aSaFollow(S)) = \{a\}$ und $First_1(Follow(S)) = \{a\}$,
aber jede LR(1) Grammatik ist eine SLL(1) Grammatik.

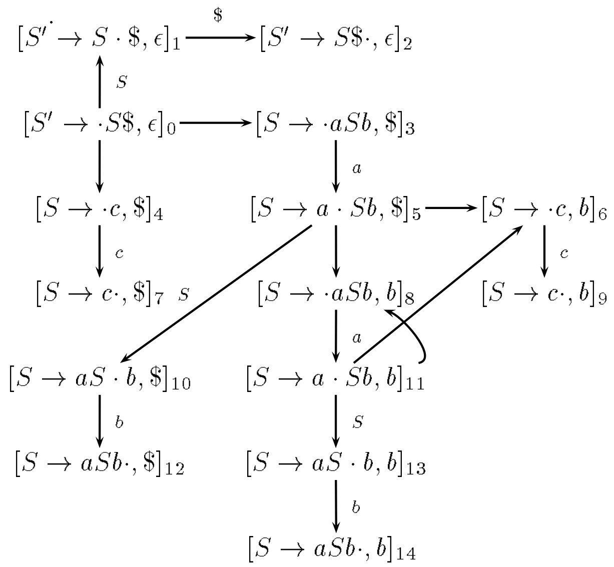
(d) Die Produktionen von G sind wie folgt numeriert:

Nummer	Produktion
1	$S \to aSa$
2	$S \to a$

	a	\$	S
0	s**1**		**7**
1	s**2**	r2	**3**
2	s**2**, r2		**5**
3	s**4**		
4		r1	
5	s**6**		
6	r1		
7		acc	
8			

Lösung zu Aufgabe 5.5.6

(a) Man betrachte die erweiterte Grammatik G' zu G. Die Zustände des Automaten $M_{G,1}$ sind numeriert. Der Anfangszustand hat Nummer 0. Die Transitionen ohne Kennzeichnung sind ϵ-Übergänge.



(b) Wird auf $M_{G,1}$ das Myhill-Buechi-Verfahren angewendet, so erhält man $D_{G,1}$. Dessen Zustände werden mit fett gedruckten Zahlen numeriert:

		a	b	c	\$	S
0	0,3,4	5,6,8	-	7	-	1
1	5,6,8	6,11,8	-	9	-	10
2	7	-	-	-	-	-
3	1	-	-	-	2	-
4	6,8,11	6,8,11	-	9	-	13
5	9	-	-	-	-	-
6	10	-	12	-	-	-
7	13	-	14	-	-	-
8	12	-	-	-	-	-
9	14	-	-	-	-	-

(c) Es folgt die zu $D_{G,1}$ gehörige LR(1)-Parsertabelle. $r1$ entspricht dabei einer **reduce**-Operation mit $S \to c$ und $r2$ einer mit $S \to aSb$.

	a	b	c	\$	S
0	s**1**		s**2**		**3**
1	s**4**		s**5**		**6**
2				r1	
3				acc	
4	s**4**		s**5**		**7**
5		r1			
6		s**8**			
7		s**9**			
8				r2	
9		r2			

(d)

Keller	Zeichenkette
0	aacbb\$
0a1	acbb\$
0a1a4	cbb\$
0a1a4c5	bb\$
0a1a4S7	bb\$
0a1a4S7b9	b\$
0a1S6	b\$
0a1S6b8	\$
0S3	\$

Lösung zu Aufgabe 5.5.7

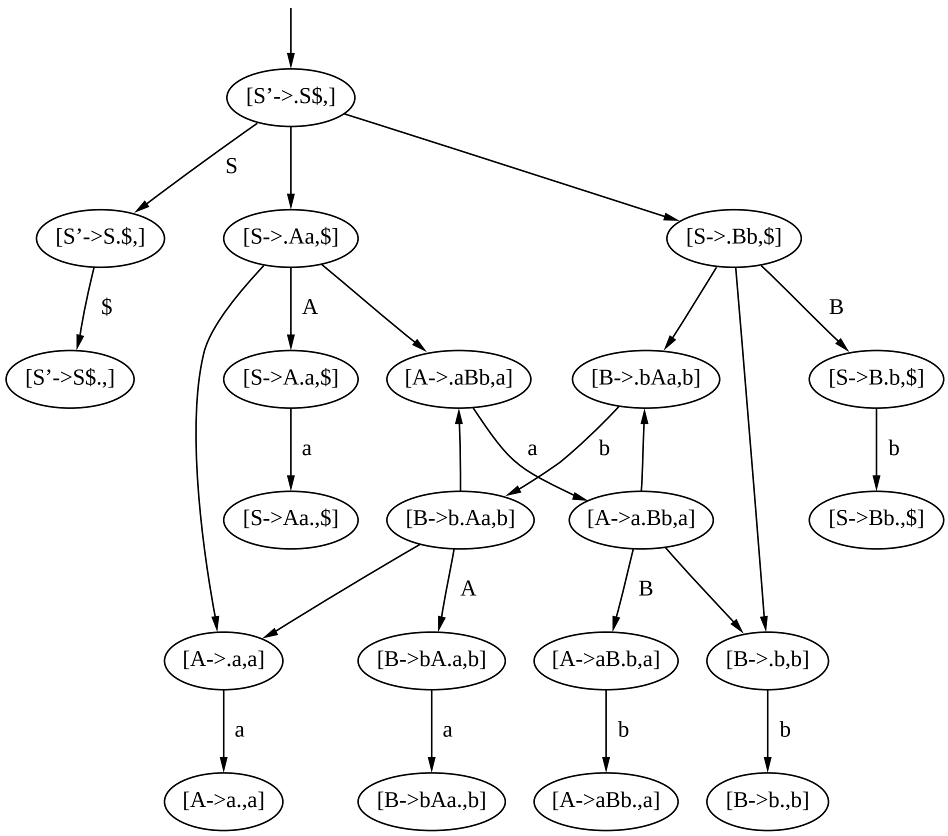
Der rechte obere Knoten enthält einen reduce-reduce Konflikt

Lösung zu Aufgabe 5.5.8

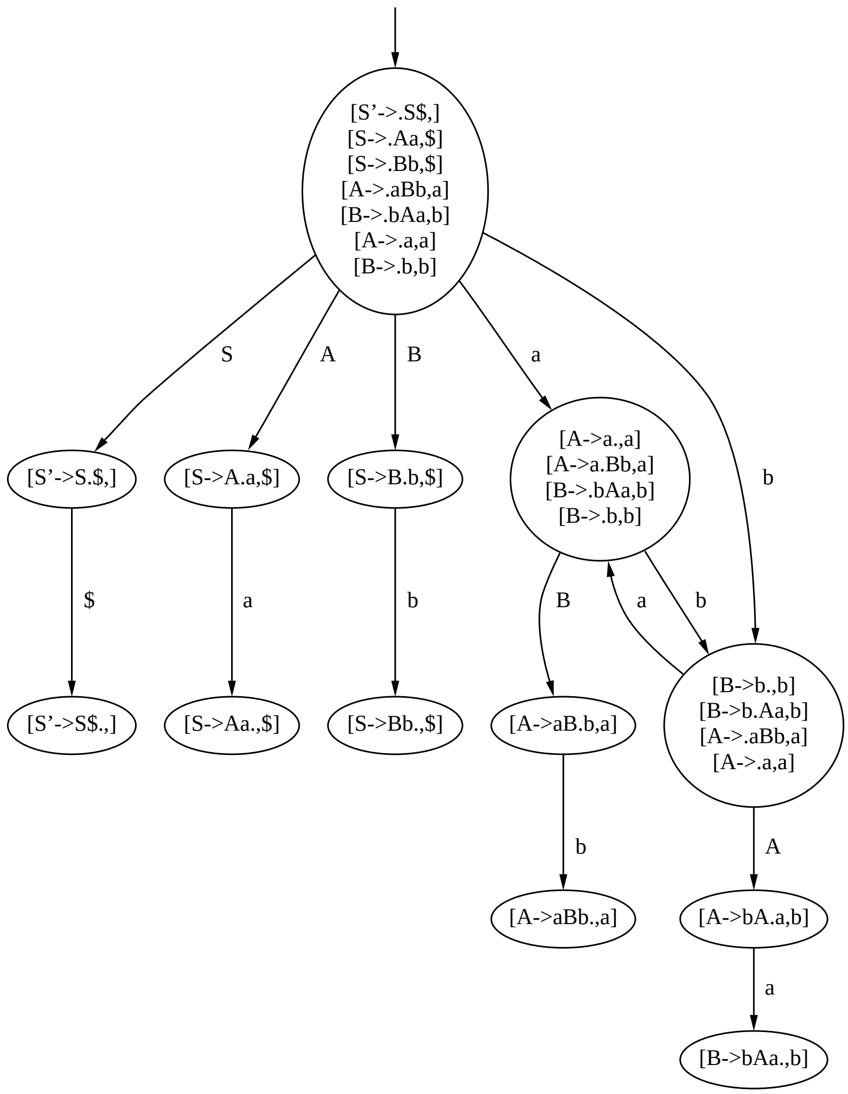
Betrachte folgenden Teil des nichtdet. $LR(1)$-Automaten $M_{G,1}$ (nicht gekennzeichnete Übergänge sind ϵ-Übergänge. Der Anfangszustand q_0 des zugehörigen det. $LR(1)$-Automaten $D_{G,1}$ umfaßt die Zustände mit Index $1, 2, 3$. δ sei Zustandsübergangsfunktion von $D_{G,1}$. Dann umfaßt $q_1 = \delta(q_0, a)$ mindestens die Zustände $4, 5, 6, 7$ und $q_2 = \delta(q_1, a)$ umfaßt mindestens 8 und 9. Letzteres bedeutet einen reduce-reduce-Konfilkt. G ist also nicht $LR(1)$.

$$
\begin{tikzcd}[column sep=small]
{[S^{'} \to \cdot S\$, \epsilon]_1} \arrow[d] \arrow[dr] & \\
{[S \to \cdot aAa, \$]_2} \arrow[d, "a"] & {[S \to \cdot aBa, \$]_3} \arrow[d, "a"] \\
{[S \to a \cdot Aa, \$]_4} \arrow[d] & {[S \to a \cdot Ba, \$]_5} \arrow[d] \\
{[A \to \cdot a, a]_6} \arrow[d, "a"] & {[B \to \cdot a, a]_7} \arrow[d, "a"] \\
{[A \to a \cdot, a]_8} & {[B \to a \cdot, a]_9}
\end{tikzcd}
$$

Lösung zu Aufgabe 5.5.9



(a)



[S'->.S$,]
[S->.Aa,$]
[S->.Bb,$]
[A->.aBb,a]
[B->.bAa,b]
[A->.a,a]
[B->.b,b]
S
A
B
a
b
[S'->S.$,]
[S->A.a,$]
[S->B.b,$]
[A->a.,a]
[A->a.Bb,a]
[B->.bAa,b]
[B->.b,b]
$
a
b
B
a
b
[S'->S$.,]
[S->Aa.,$]
[S->Bb.,$]
[A->aB.b,a]
[B->b.,b]
[B->b.Aa,b]
[A->.aBb,a]
[A->.a,a]
b
A
[A->aBb.,a]
[A->bA.a,b]
a
[B->bAa.,b]

(b)

(c)

q	$	a	b	A	B	S
0		s**1**	s**2**	**3**	**4**	**11**
1		r4	s**2**		**5**	
2		s**1**	r6	**6**		
3		s**7**				
4			s**8**			
5			s**9**			
6		s**10**				
7	r1					
8	r2					
9		r3				
10			r5			
11	s**12**					
12	acc					

Lösung zu Aufgabe 5.5.10

(a) Shift-Regeln sind

$$\{qa \to aq, qb \to bq, qc \to cq\}$$

Reduce-Regeln sind

$$\{Sq \to aSaq, Sq \to bSbq, Sq \to cq\}$$

(b)

Keller	Eingabe	Regel
q	aca	**shift**
aq	ca	**shift**
acq	a	**reduce** $S \to c$
aSq	a	**shift**
aSaq		**reduce** $S \to aSa$
Sq		

(c)

	a	b	c	$	S
0	s**1**	s**2**	s**3**		**8**
1	s**1**	s**2**	s**3**		**4**
2	s**1**	s**2**	s**3**		**5**
3	r3	r3	r3	r3	
4	s**6**				
5		s**7**			
6	r1	r1	r1	r1	
7	r2	r2	r2	r2	
8				acc	

(d) Der Automat in Teil (c) akzeptiert genau die lebensfähigen Präfixe von G. Er akzeptiert aS und $aaSa$, beide sind also lebensfähige Präfixe. Er akzeptiert aber nicht aca, dies ist also kein lebensfähiger Präfix

Literaturverzeichnis

[1] Aho, A. V., R. Sethi und J. D. Ullmann: *Compilers. Principles, Techniques, and Tools*. Addison-Weseley, 1988.

[2] Aho, A. V. und J. D. Ullmann: *Principles of Compiler Design*. Addison-Weseley, 1977.

[3] Alexandrov, P. S. (Hrsg.): *Die Hilbertschen Probleme*, Bd. 252 d. Reihe *Ostwalds Klassiker der exakten Wissenschaft*. Akademische Verlagsgesellschaft, 1971.

[4] Bovet, D. P. und P. Crescenzi: *Introduction to the Theory of Compexity*. Prentice Hall, 1993.

[5] Brady, A. H.: *The Busy Beaver Game and the Meaning of Life*. In: *The Universal Turing Machine. A Half-Century Survey*, S. 259–277. Kammerer & Unverzagt, 1988.

[6] Browder, F. E. (Hrsg.): *Mathematical developments arising from Hilbert problems*, Bd. 28 d. Reihe *Proceedings of the Symposia in Pure Mathematics*. American Math. Soc. Providence, RI, 1976.

[7] Chang, C.-H. und R. Paige: *From regular expressions to DFS's using NFA's*. Theoretical Computer Science, 178:1–36, 1997.

[8] Crochemore, M. und C. Hancart: *Automata for Matching Patterns*. In: *Handbook of Formal Languages*, Bd. 2. Linear Modeling: Background and Application, S. 399–462. Springer Verlag, 1997.

[9] Dewdney, A. K.: *Computer Recreations*. Scientific American, 252(4):12–16, april 1985.

[10] Dewdney, A. K.: *Der Turing Omnibus. Eine Reise durch die Informatik mit 66 Stationen*. Springer-Verlag, 1993.

[11] D.Jones, N.: *Computability and Complexity. From a programming perspective.* MIT Press, Cambridge, Mass., and London, 1997.

[12] Ebbinghaus, H.-D., J. Flum und W. Thomas: *Einführung in die Mathematische Logik. 3., vollständig überarbeitete und erweiterte Auflage.* BI Wissenschaftsverlag, 1992.

[13] Garey, M. R. und D. S. Johnson: *Computers and Intractability. A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., 1979.

[14] Goos, G.: *Vorlesung über Informatik. Band 1: Grundlagen und funktionales Programmieren.* Springer Lehrbuch. Springer-Verlag, 1995.

[15] Goos, G.: *Vorlesung über Informatik. Band 3: Berechenbarkeit, formale Sprachen, Spezifikationen.* Springer Lehrbuch. Springer-Verlag, 1997.

[16] Gurevich, Y.: *'Reconsidering Turing's Thesis: Toward More Realistic Semantics of Programs.* Techn. Ber. CRL-TR-36-84, University of Michigan, Computing Research Lab, 1984.

[17] Harel, D.: *First-Order Dynamic Logic*, Bd. 68 d. Reihe *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

[18] Harel, D.: *Dynamic Logic.* In: *Handbook of Philosophical Logic*, Bd. II Extensions of Classical Logic, S. 497 – 604. D.Reidel, 1984.

[19] Harel, D.: *Statecharts: A Visual Formalism for Complex Systems.* Science of Programming, 8(3):231–274, 1987.

[20] Herken, R. (Hrsg.): *The Universal Turing Machine. A Half-Century Survey.* Kammerer & Unverzagt, 1988.

[21] Hermes, H.: *Aufzählbarkeit Entscheidbarkeit Berechenbarkeit. Zweite Auflage.* Heidelberger Taschenbücher. Springer-Verlag, 1971.

[22] Hodges, A.: *ALAN TURING: The Enigma.* Simon and Schuster, New York, 1983.

[23] Hopcroft, J. E. und J. D. Ullmann: *Introduction to Automata Theory.* Addison-Weseley, 1979.

[24] Kfoury, A. J., R. N. Moll und M. A. Arbib: *A Programming Approach to Computability. Second printing.* Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[25] Marxen, H. und J. Buntrock: *Attacking the Busy Beaver 5*. Bulletin of the EATCS, 40:247–251, february 1990.

[26] Matijasevic, Y. V.: *Enumerable Sets are Diophantine*. Dokl. Akad. Nauk. SSSR (russisch). Englische Übersetzung in Soviet Math. Dokl.Vol.11, pp.354–357, 191:279–282, 1970.

[27] Matijasevic, Y. V. und J. Robinson: *Reduction of an Arbitrary Diophantine Equation to One in 13 Unknowns*. Acta Arithmetica, 27:521–553, 1975.

[28] Minsky, M. L.: *Computation: Finite and Infinite Machines*. Prentice-Hall International, 1972.

[29] Moll, R. N., M. A. Arbib und A. J. Kfoury: *An Introduction to Formal Language Theory*. Texts and Monographs in Computer Science. Springer-Verlag, 1988.

[30] Oberschelp, A.: *Rekursionstheorie*. B.I. Wissenschaftsverlag, 1993.

[31] Papadimitriou, C. H.: *Computational Complexity*. Addison-Wesley, 1994.

[32] Prusinkiewicz, P. und A. Lindenmayer: *The algorithmic beauty of plants*. Springer-Verlag, Berlin, 1990.

[33] Salomaa, A. K.: *Formale Sprachen*. Springer, 1978.

[34] Schwartz, R. L.: *Learning Perl*. UNIX Programming. O'Reilly & Associates, 1994.

[35] Sippu, S. und E. Soisalon-Soininen: *Parsing Theory. Vol. I: Languages and Parsing*. Nr. 15 in *EATCS Monographs on Theoretical Computer Science*. Springer, 1988.

[36] Sippu, S. und E. Soisalon-Soininen: *Parsing Theory. Vol. II: LR(k) and LL(k) Parsing*. Nr. 20 in *EATCS Monographs on Theoretical Computer Science*. Springer, 1988.

[37] Sperschneider, V. und B. Hammer: *Theoretische Informatik: eine problemorientierte Einführung*. Springer Verlag, 1996.

[38] Thompson, K.: *Regular expression search algorithm*. Communications of the ACM, 11:419–422, 1968.

[39] TURING, A. M.: *On computable numbers, with an application to the Entscheidungsproblem*. Proc. London Mathematical Society, 42:230–265, 1936.

[40] WAGNER, K. W.: *Einführung in die Theoretische Informatik. Grundlagen und Modelle*. Springer Lehrbuch. Springer-Verlag, 1994.

[41] WEGENER, I.: *Kompendium Theoretische Informatik - eine Ideensammlung*. Leitfäden der Informatik. B.G.Teubner Verlag, 1996.

[42] ZIMA, H.: *Compilerbau I, Analyse*, Bd. Band 36 d. Reihe *Reihe Informatik*. BI Wissenschaftsverlag, 1988.

Symbolverzeichnis

$L(G)$	Sprachschatz der Grammatik G	130
$L(T)$	von einer Turingmaschine T akzeptierte Sprache.	19
N	Menge der Nichtterminalzeichen einer Grammatik	130
N	konstante Funktion mit Funktionswert 0	31
S	Startsymbol einer Grammatik	130
T	Menge der Terminalzeichen einer Grammatik	130
U_i^n	Projektionsfunktion	31
$\blacktriangleright$	Anfangsmarkierung eines einseitig unendlichen Turingbandes	9
χ_S	Charakteristische Funktion für S	35
$\mathrm{Def}(g)$	Definitionsbereich einer Funktion g	7
μ	Operator für rekursive Funktionen	34
ρ	Interpretation von Programmen in der dynamischen Logik	30
succ	primitiv rekursive Nachfolgerfunktion	31
$f(d)\uparrow$	$f(d)$ ist undefiniert	34
$f(d)\downarrow$	$f(d)$ ist definiert	34
$\Rightarrow$	1-Schritt-Ableitbarkeit in Grammatiken	130
$\Rightarrow^*$	reflexiv-transitive Ableitbarkeit in Grammatiken	130
$\Rightarrow^+$	transitive Ableitbarkeit in Grammatiken	130
$l \to r$	Produktionsregel einer Grammatik	130
$\mathcal{A}_L$	Standardautomat zur Sprache L	168
B	Symbol für das Leerzeichen auf Turingmaschinen Bändern	5
$beaver$	die Biberfunktion	17
$\mathrm{Bild}(\phi)$	Bildbereich einer partiellen Funktion ϕ	49
$Cl_\varepsilon(P)$	ε-Hülle von P	143
$\mathrm{Def}(\phi)$	Definitionsbereich einer partiellen Funktion ϕ	49
δ	Übergangsfunktion einer Turingmaschine	5
δ	Übergangsfunktion eines endlichen Automaten	137
$\hat{\delta}$	fortgesetzte Übergangfunktion für EA mit spontanen Übergangen	144
$\uplus$	disjunkte Summe zweier Automaten	151

Index