Prof. Dr. Hans-Jörg Kreowski, Dr. Sabine Kuske
Studiengang Informatik
Universität Bremen
Linzer Str. 9a, OAS 3001
Tel.: 2956, 2335, 3697 (Sekr.), Fax: 4322
E-Mail: {kreo,kuske}@informatik.uni-bremen.de
www.informatik.uni-bremen.de/theorie

Januar 2011

Theoretische Informatik 1 (Teil 2)

Wintersemester 2010/2011

Vorbemerkung

Im Zuge der aktuellen Veränderungen im Bachelor-Studium der Informatik ist auch das Curriculum für Theoretische Informatik 1 und 2 umgestellt worden. Theoretische Informatik 1 ist nun ganz auf Automatentheorie und die Theorie der formalen Sprachen konzentriert, während in Theoretische Informatik 2 Berechenbarkeit und Komplexität behandelt werden.

Dieser neue Zuschnitt wird ab diesem Wintersemester realisiert. Deshalb werden die Kapitel 18 bis 20 aus dem Skript Theoretische Informatik 1 nicht bearbeitet, sondern stattdessen das folgende Material, das bisher zu Theoretische Informatik 2 gehörte.

Inhaltsverzeichnis

1 Formale Sprachen

Die Theorie formaler Sprachen stellt eines der ältesten und am weitesten entwickelten Gebiete der Theoretischen Informatik dar. Die Bedeutung dieser Theorie rührt daher, dass ihre Konzepte und Methoden die Grundlage für die Definition der Syntax von Programmiersprachen und für den Bau ihrer Compiler bilden, wobei insbesondere die Syntaxanalyse unterstützt wird (vgl. Kap. 8 aus dem ersten Teil des Skripts). Ein zentraler Aspekt der Syntaxanalyse ist die Lösung des sogenannten Wortproblems.

1.1 Wortproblem

Die Syntaxanalyse ist eine zentrale Aufgabe bei der Übersetzung von Programmen einer Programmiersprache. Den Kern bildet dabei die Lösung des Wortproblems für die Menge aller syntaktisch korrekten Programme, das darin besteht, algorithmisch festzustellen, ob ein beliebiges eingegebenes Wort ein Programm ist oder nicht. Aber das Wortproblem ist auch für andere Sprachen signifikant.

Betrachtet man als *formale Sprache* eine beliebige Menge L von Wörtern aus A^* für ein Alphabet A, so definiert L ein *Wortproblem*:

Gibt es einen Algorithmus, der für jedes $x \in A^*$ bestimmt, ob $x \in L$ oder $x \notin L$ gilt?

Als Beispiel betrachte die Menge aller Palindrome aus A^*. Das Wortproblem ist in diesem Fall gerade die Frage, ob ein gegebenes Wort ein Palindrom ist oder nicht. Die Lösung kann in diesem Fall durch einen Kellerautomaten konstruiert werden.

1.1.1 Wortproblem selten lösbar

Wenn das unterliegende Alphabet A nicht leer ist, gibt es unendlich viele Wörter, so dass nach einer bekannten Überlegung aus der Mathematik die Menge aller Teilmengen von A^* überabzählbar unendlich ist. Da es aber nur abzählbar viele Algorithmen gibt, müssen die Wortprobleme der meisten Sprachen unlösbar sein. Aber obwohl die Lösbarkeit des Wortproblems grundsätzlich in den seltensten Fällen gegeben ist, sind doch die meisten Sprachen, denen man üblicherweise begegnet, ziemlich gutartig. So sind die Wortprobleme aller regulären und kontextfreien Sprachen bekanntlich durch endliche bzw. Kellerautomaten lösbar. Es ist sogar relativ schwierig, Sprachen zu konstruieren, deren Wortproblem nicht lösbar ist (siehe zwei derartige Beispiele in den Abschnitten 9.1 und 9.2 in [HMU07, HMU02]). Dem Wortproblem wird dennoch in den nächsten Abschnitten viel Aufmerksamkeit gewidmet, weil man nicht nur an Lösbarkeit interessiert ist, sondern auch an praktisch benutzbaren Lösungen. So muss ein Algorithmus, der in einen Compiler eingebaut werden soll, insbesondere auch schnell sein.

2 Chomsky-Grammatiken

Lässt man bei den kontextfreien Regeln, wie sie meist bei der syntaktischen Beschreibung von Programmiersprachen verwendet werden, auf der linken Seite auch beliebige Zeichenketten zu, erhält man Produktionen (Regeln) von Chomsky-Grammatiken (nach dem amerikanischen Sprachwissenschaftler Noam Chomsky, geb. 1928, einem Pionier der Theorie der formalen Sprachen).

2.1 Grammatik allgemein

Eine Chomsky-Grammatik besteht aus endlich vielen Produktionen der Form $u ::= v$ für $u, v \in A^*$, wobei A ein Alphabet ist, aus einem Startsymbol $S \in A$ und einem terminalen Alphabet $T \subseteq A$. Meist wird noch verlangt, dass $S \in A \setminus T$ ist und in den linken Seiten von Produktionen Zeichen vorkommen, die nicht terminal sind. Die Differenzmenge $A \setminus T$ wird nichtterminales Alphabet genannt und mit N bezeichnet. Es ist manchmal auch bequemer, mit einem Startwort statt mit einem Startsymbol zu beginnen.

1. Für ein gegebenes Alphabet A ist eine *Produktion* (*Regel*) ein Paar $p = (u, v) \in A^* \times A^*$, das meist als $u ::= v$ geschrieben wird. Die Zeichenkette u wird *linke Seite*, v *rechte Seite* von p genannt.
   Zur Abkürzung können mehrere Produktionen $u ::= v_1, \ldots, u ::= v_k$ $(k \geq 2)$ mit derselben linken Seite zu $u ::= v_1 \mid \cdots \mid v_k$ zusammengefasst werden.
2. Eine *Chomsky-Grammatik* ist ein System $G = (N, T, P, S)$, wobei N eine Menge *nichtterminaler Zeichen*, T eine Menge *terminaler Zeichen*, P eine endliche Menge von Produktionen und $S \in N$ ein *Startsymbol* ist.
   Soweit nichts anderes gesagt wird, nimmt man an, dass kein nichtterminales Zeichen gleichzeitig terminal ist, d.h. $N \cap T = \emptyset$, dass alle in Produktionen vorkommenden Zeichen terminal oder nichtterminal sind, d.h. $p \in (N \cup T)^* \times (N \cup T)^*$ für alle $p \in P$, und dass in jeder linken Seite mindestens ein nichtterminales Zeichen vorkommt, d.h. $u \in (N \cup T)^* N (N \cup T)^*$ (bzw. $u \notin T^*$) für jede Produktion $u ::= v \in P$.

Produktionen werden auf Zeichenketten analog zum kontextfreien Fall angewendet. Man sucht in einer Zeichenkette eine Teilkette, die die linke Seite einer Produktion ist, und ersetzt sie durch die rechte Seite.

3. Seien $w, w', x, y, u, v \in A^*$. Dann wird w' aus w *direkt* durch Anwendung der Produktion $p = (u ::= v)$ *abgeleitet*, falls $w = xuy$ und $w' = xvy$. In diesem Falle wird $w \underset{p}{\longrightarrow} w'$ geschrieben.
   Die Anwendung einer Produktion wird *direkte Ableitung* genannt. Ist P eine Menge von Produktionen und $p \in P$, so kann man statt $w \underset{p}{\longrightarrow} w'$ auch $w \underset{P}{\longrightarrow} w'$ schreiben.
4. Die Iteration direkter Ableitungen ergibt das Konzept der *Ableitung*:

$$w_0 \underset{p_1}{\longrightarrow} w_1 \underset{p_2}{\longrightarrow} \cdots \underset{p_n}{\longrightarrow} w_n$$

für $w_0, \ldots, w_n \in A^*$ und Produktionen $p_1, \ldots, p_n$ ($n \geq 1$). Stammen alle angewendeten Produktionen aus P, so kann man die obige Ableitung auch schreiben als $w_0 \underset{P}{\longrightarrow} \cdots \underset{P}{\longrightarrow} w_n$ oder $w_0 \overset{n}{\underset{P}{\longrightarrow}} w_n$. Für manche Zwecke ist es sinnvoll, auch *Nullableitungen* zuzulassen: $w \overset{0}{\underset{P}{\longrightarrow}} w$ für alle $w \in A^*$. Statt $w \overset{n}{\underset{P}{\longrightarrow}} w'$ für $n \in \mathbb{N}$ darf auch $w \overset{*}{\underset{P}{\longrightarrow}} w'$ geschrieben werden. Außerdem kann man bei Ableitungen und direkten Ableitungen das Subskript P weglassen, wenn die Produktionsmenge aus dem Kontext klar ist.

Der Ableitungsprozess bildet die operationelle Semantik, die durch eine Produktionsmenge syntaktisch beschrieben ist. Betrachtet man diejenigen Zeichenketten, die aus dem Startsymbol einer Chomsky-Grammatik $G = (N, T, P, S)$ ableitbar sind und nur aus terminalen Zeichen bestehen, so erhält man auf der Basis des Ableitungsprozesses eine erzeugte Sprache.

5. Sei $G = (N, T, P, S)$ eine Chomsky-Grammatik. Dann enthält die von G *erzeugte Sprache* alle mit Produktionen in P aus dem Startsymbol S ableitbaren terminalen Zeichenketten:
$$L(G) = \{w \in T^* \mid S \overset{*}{\underset{P}{\longrightarrow}} w\}.$$

Auf diese Weise stellen Chomsky-Grammatiken ein syntaktisches Instrument dar, um formale Sprachen zu spezifizieren. Ausführliche Darstellungen von Chomsky-Grammatiken findet man in praktisch jedem Buch über formale Sprachen (siehe z.B. Hopcroft, Motwani, Ullman [HMU07] mit der deutschen Übersetzung [HMU02], Moll, Arbib und Kfoury [MAK88] und Salomaa [Sal73] mit der deutschen Übersetzung [Sal78]). Die Verbindung von formalen Sprachen und der syntaktischen Behandlung von Programmiersprachen wird umfassend in Aho und Ullman [ASU86] abgehandelt.

2.2 Beispiele

1. Mit der Produktion $S ::= aS$ lässt sich das Zeichen a hochzählen:
$$S \to aS \to a^2S \to \cdots \to a^nS.$$
Entsprechend kann man mit $S ::= a^kS$ ($k \in \mathbb{N}$) ein Vielfaches von k hochzählen. Terminieren lässt sich dieser Vorgang mit $S ::= \lambda$, so dass
$$L((\{S\}, \{a\}, \{S ::= a^kS \mid \lambda\}, S)) = \{a^{n \cdot k} \mid n \in \mathbb{N}\}.$$

2. Fast genauso einfach ist es, zwei Größen gleichzeitig hochzuzählen:
$$L((\{S\}, \{a, b\}, \{S ::= aSb \mid \lambda\}, S)) = \{a^nb^n \mid n \in \mathbb{N}\}.$$

3. Auch das getrennte Zählen zweier (bzw. mehrerer) Größen ist kein Problem. Sei $T_l = \{a_1, \ldots, a_l\}$ ($l \geq 1$), $N_l = \{S\} \cup \{A_1, \ldots, A_l\}$ und $P_l = \{S ::= A_1 \cdots A_l\} \cup \{A_i ::= a_iA_i \mid i = 1, \ldots, l\} \cup \{A_i ::= \lambda \mid i = 1, \ldots, l\}$.

Dann gilt:

$$L((N_l, T_l, P_l, S)) = \{a_1^{n_1} \cdots a_l^{n_l} \mid n_i \geq 0,\ i = 1, \ldots, l\}.$$

4. Etwas schwieriger wird es, die zahlenmäßige Ausgewogenheit zweier Größen zu garantieren, wenn die Reihenfolge nicht mehr wie in Punkt 2 fixiert ist.
   Die Grammatik $G_{equilibrium} = (\{A, B, S\}, \{a, b\}, P_{equilibrium}, S)$, wobei die Regelmenge die Regeln

   $$S ::= ASB,\ S ::= \lambda,\ AB ::= BA,\ A ::= a,\ B ::= b$$

   enthält, erzeugt als Sprache die Menge aller Wörter, in denen nur die Zeichen a und b vorkommen und das gleich oft. Ableitungen (d.h. wiederholte Regelanwendungen) sehen z.B. so aus:

   $$S \underset{(1)}{\longrightarrow} ASB \underset{(1)}{\longrightarrow} A^2SB^2 \underset{(2)}{\longrightarrow} A^2B^2 \underset{(3)}{\longrightarrow} ABAB \underset{(3)}{\longrightarrow} ABBA \underset{(4),(5)}{\longrightarrow} \cdots \underset{(4),(5)}{\longrightarrow} abba.$$

   Dabei verweisen die Nummern auf die Produktionen, die von links nach rechts gezählt sind.
5. Noch schwieriger wird es, wenn man drei Größen gleichzeitig nach Art des Punktes 2 kontrollieren möchte. Es ist zwar leicht zu sehen, dass mit den Produktionen der Form $ab^nc ::= a^2b^{n+1}c^2$ für $n \geq 1$ aus der Zeichenkette abc alle Zeichenketten der Form $a^nb^nc^n$ abgeleitet werden können; aber das sind unendlich viele Regeln. Um dasselbe mit endlich vielen Regeln zu erreichen, bedarf es der bisher kompliziertesten Grammatik im nächsten Punkt. Oder geht es einfacher?
6. Folgende Produktionen erlauben, die Sprache $\{a^nb^nc^n \mid n \geq 1\}$ zu erzeugen, wenn man mit S startet und $\{a, b, c\}$ als terminales Alphabet wählt:

(1)	$S ::= aABc$
(2)&(3)	$A ::= aABc \mid \lambda$
(4)	$cB ::= Bc$
(5)	$aB ::= ab$
(6)	$bB ::= bb$

   Mit (1) bis (3) erhält man $S \xrightarrow{*} a^n(Bc)^n$.

   Mit (4) wird daraus: $a^nB^nc^n$.

   Mit (5) und (6) erhält man: $a^nb^nc^n$.

   Der Nachweis, dass man nichts anderes Terminales ableiten kann, erfordert diverse Fallunterscheidungen, die hier nicht im einzelnen durchgeführt werden sollen.

3 Immerhin aufzählbar

In diesem Kapitel wird der Zusammenhang zwischen Chomsky-Grammatiken und dem Konzept der Aufzählbarkeit beschrieben. Der Ableitungsprozess zusammen mit einem Terminalitätstest für Zeichenketten zählt die erzeugte Sprache einer Chomsky-Grammatik auf (3.1), deren Wortproblem sich damit auch als "halb" lösbar erweist (3.2). Es wird außerdem darauf hingewiesen, dass effektiv aufzählbare Mengen von Zeichenketten von Chomsky-Grammatiken erzeugt werden (3.3), allerdings ist das Wortproblem nicht immer "ganz" lösbar (3.4). In Abschnitt 3.5 schließlich wird auf Zusammenhänge zwischen dem hier angegebenen Aufzählungsverfahren und anderen bekannten Algorithmen hingewiesen.

3.1 Aufzählbarkeit erzeugter Sprachen

Für eine beliebige Chomsky-Grammatik $G = (N, T, P, S)$ bildet der Ableitungsmechanismus einen algorithmischen Prozess, den man mit beliebigen Zeichenketten beginnen und beliebig lange laufen lassen kann. Startet man zum Beispiel mit S und erlaubt bis zu k Ableitungsschritte, führt aber alle möglichen Alternativen bei Regelanwendungen aus, so erhält man die Menge $S(G)_k$ aller aus S in bis zu k Schritten ableitbaren Wörter; d.h.

$$S(G)_k = \{w \mid S \xrightarrow[P]{l} w,\ l \leq k\}.$$

Es gilt offenbar $S(G)_k \subseteq S(G)_m$ für $k \leq m$. Es gilt genauer: $S(G)_0 = \{S\}$ und $S(G)_{k+1} = S(G)_k \cup \{w \mid v \xrightarrow[P]{} w,\ v \in S(G)_k\}$. Denn nach Definition von Ableitungen kann man aus S in 0 Schritten nur S ableiten, und eine Ableitung mit höchstens $k+1$ Schritten hat sogar höchstens k Schritte oder setzt sich aus k Schritten gefolgt von einer weiteren direkten Ableitung zusammen.

Insbesondere erweisen sich die Mengen $S(G)_k$ für $k \in \mathbb{N}$ als endlich. Denn $S(G)_0$ ist einelementig, und wenn nach Induktionsvoraussetzung $S(G)_k$ endlich ist, so muss es auch $S(G)_{k+1}$ sein. Letzteres ergibt sich daraus, dass die endlich vielen Wörter in $S(G)_k$ nur endlich viele Teilwörter besitzen, also auch höchstens endlich viele linke Regelseiten, die durch höchstens endlich viele rechte Regelseiten ersetzt werden können, da die Regelmenge endlich ist.

Darüber hinaus ist $S(G)_{k+1}$ aus $S(G)_k$ effektiv, d.h. algorithmisch herstellbar, da Suchen und Ersetzen von Teilwörtern algorithmisch machbar sind.

Bezeichnet man die Menge aller aus S ableitbaren Zeichenketten mit $S(G)$, so gilt offenbar:

$$S(G) = \bigcup_{k \in \mathbb{N}} S(G)_k,$$

denn jedes ableitbare Wort ist in einer bestimmten Schrittzahl ableitbar. Die Elemente von $S(G)$ nennt man *Satzformen* von G. Besteht eine Satzform nur aus terminalen Zeichen, so gehört sie zur erzeugten Sprache, d.h.

$$L(G) = S(G) \cap T^*.$$

Bildet man also die Mengen $S(G)_k$ für wachsende k, beginnend mit $S(G)_0$, und filtert die terminalen Wörter heraus, so erhält man einen algorithmischen Vorgang, bei dem nach und nach jedes Wort aus $L(G)$ entsteht. Dieses Verfahren ist in Abbildung 1 dargestellt.

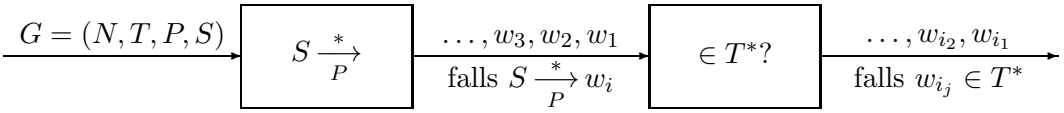


Abbildung 1: Aufzählung der von einer Grammatik erzeugten Sprache

Mit anderen Worten erweist sich die von der Chomsky-Grammatik G erzeugte Sprache als effektiv oder – wie man auch sagt – *rekursiv aufzählbar*.

3.2 Die halbe Miete

Will man von einer Zeichenkette wissen, ob sie zu einer erzeugten Sprache gehört oder nicht, so kann man den Aufzählungsprozess starten. Erscheint dabei irgendwann die fragliche Zeichenkette, liegt sie in der gegebenen Sprache. Der positive Fall des Wortproblems ist damit gelöst. Über den negativen Fall erfährt man auf diese Weise jedoch nichts, wenn der Aufzählungsprozess nicht anhält, was gerade bei unendlichen Sprachen passiert. Denn ist eine Zeichenkette zu einem bestimmten Zeitpunkt nicht aufgezählt, kann das noch später oder gar nicht geschehen. Das Wortproblem ist so nur "halb" gelöst, was man auch *semi-entscheidbar* nennt.

3.3 Erzeugbarkeit aufzählbarer Sprachen

Dass auch die Umkehrung gilt, dass also jede rekursiv aufzählbare – also durch einen Algorithmus herstellbare – Menge von Zeichenketten von einer Chomsky-Grammatik erzeugt werden kann, soll nicht bewiesen werden, da das zu viel Zeit und Mühe erforderte.

3.4 Unlösbarkeit des Wortproblems

In ihrer allgemeinen Form sind Chomsky-Grammatiken allerdings nur bedingt für die Definition der Syntax von Programmiersprachen geeignet, weil nicht zu jeder definierbaren Syntax auch eine Syntaxanalyse möglich ist. Auch das soll nicht formal bewiesen.

3.5 Ausschöpfende Suche in die Breite mit roher Gewalt

Das der Aufzählung der erzeugten Sprache in Punkt 1 zugrundeliegende Verfahren, das in Abbildung 2 skizziert ist, lässt sich bei vielen Arten regelbasierter Systeme anwenden:

Man beginnt mit einem Anfangszustand oder einem Eingabezustand, wendet wiederholt auf alle entstehenden Zwischenzustände alle möglichen Regeln an, wie und wo immer es geht, und filtert dann nach einem bestimmten Kriterium Ausgabe- oder Endzustände aus den erreichten und produzierten Zwischenzuständen heraus.

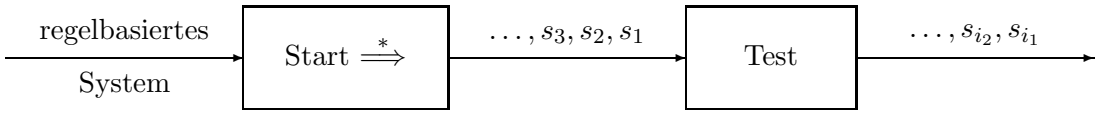


Abbildung 2: Allgemeine Funktionsweise regelbasierter Systeme

Im Zusammenhang mit regelbasierten Systemen der Künstlichen Intelligenz und mit Expertensystemen sowie in der Logistik wird das Verfahren oft *ausschöpfende Suche* genannt. In der Algorithmen- und Komplexitätstheorie ist das Verfahren ebenfalls bekannt und wird dort häufig als "rohe Gewalt" (brute force) eingesetzt, weil einfach alles durchprobiert wird. Zählt man alle Möglichkeiten nach der Systematik in Abschnitt 3.1 auf, d.h. nach wachsender Ableitungslänge bzw. wachsender Zahl von Regelanwendungen, so spricht man häufig auch von *Breitensuche* (*breadth first search*).

4 Lösbarkeit des Wortproblems für monotone Grammatiken

Wie im vorigen Kapitel erwähnt, kann die Lösbarkeit des Wortproblems nicht für alle Chomsky-Grammatiken garantiert werden, sondern höchstens für geeignete Spezialfälle. Außerdem hat sich gezeigt, dass die halbe Lösung, die durch den Aufzählungsprozess gegeben ist, deshalb nicht zu einer ganzen gemacht werden kann, weil man nicht weiß, wann der Prozess als erfolglos abgebrochen werden darf.

Die Situation ändert sich, wenn man monotone Grammatiken betrachtet, in deren Produktionen keine rechte Seite kürzer als die linke Seite ist. Dann können Wörter während des Ableitens auch nicht kürzer werden. Will man in einem solchen Fall von einer Zeichenkette wissen, ob sie zur erzeugten Sprache gehört oder nicht, kann man beim Aufzählungsprozess auf längere Zeichenketten verzichten. Die Zahl der Zeichenketten bis zu einer bestimmten Länge ist aber endlich, so dass jede Aufzählung solcher Zeichenketten nach endlich vielen Schritten abbrechen muss. Das ist der Schlüssel zur Lösung des Wortproblems für monotone Grammatiken. Allerdings ist der angegebene Algorithmus exponentiell, und ein polynomieller ist nicht bekannt, so dass auch monotone Grammatiken für praktische Anwendungen nicht geeignet sind, wenn die Lösbarkeit des Wortproblems wichtig ist.

4.1 Monotone Grammatiken

Eine Chomsky-Grammatik $G = (N, T, P, S)$ wird *monoton* genannt, wenn für jede Produktion $u ::= v \in P$ $|u| \leq |v|$ gilt.[1]

Für einen Ableitungsschritt $w = xuy \underset{u::=v}{\longrightarrow} xvy = w'$ gilt dann offenbar:

$$|w| = |x| + |u| + |y| \leq |x| + |v| + |y| = |w'|,$$

so dass durch einfache Induktion über die Länge von Ableitungen auch folgt:

$$|w| \leq |w'| \text{ für } w \xrightarrow[P]{*} w'.$$

Wenn man ein bestimmtes Wort der Länge n aus dem Startsymbol S ableiten will, treten zwischendurch also niemals längere Wörter auf. Wörter bis zur Länge n gibt es aber nur endlich viele, deren Zahl mit K_n bezeichnet wird. Denn es gilt für ein Alphabet A mit k Elementen:[2]

$$\#\{w \in A^* \mid |w| \leq n\} = 1 + k + k^2 + \ldots + k^n = \sum_{i=0}^{n} k^i = K_n < \infty.$$

[1] Für ein Wort v bezeichnet $|v|$ die Länge.

[2] Für eine endliche Menge X bezeichnet $\#X$ die Zahl der Elemente.

4.2 Lösung des Wortproblems für monotone Grammatiken

Theorem 1
Für jede monotone Grammatik $G = (N, T, P, S)$ ist das Wortproblem lösbar.

Beweis.
Sei $w_0 \in T^*$ mit $|w_0| = n$. Ohne Einschränkung kann man $n \geq 1$ annehmen, weil das leere Wort niemals von einer monotonen Grammatik erzeugt wird. Für jedes $k \geq 0$ sei S_k die Menge aller in höchstens k Schritten aus S ableitbaren Wörter, deren Länge n nicht überschreitet; d.h.

$$S_k = \{w \in (N \cup T)^* \mid S \xrightarrow[P]{j} w,\ j \leq k,\ |w| \leq n\}.$$

Offenbar gilt $S_0 = \{S\}$, und S_{k+1} lässt sich folgendermaßen aus S_k konstruieren:

$$S_{k+1} = S_k \cup \{w \in (N \cup T)^* \mid v \xrightarrow[P]{} w,\ v \in S_k,\ |w| \leq n\}. \qquad (*)$$

Damit ist nach Definition $S_k \subseteq S_{k+1}$, und wenn ein $m \in \mathbb{N}$ existiert mit $S_m = S_{m+1}$, so folgt

$$S_m = S_{m+1} = S_{m+2} = \ldots.$$

Da G nur endlich viele Produktionen hat, lässt sich leicht ein Algorithmus angeben, der ausgehend von $S_0 = \{S\}$ unter Verwendung von $(*)$ die Mengen S_k rekursiv konstruiert.

Falls es nun eine natürliche Zahl m gibt, so dass $S_m = S_{m+1}$ gilt, wäre unser Wortproblem entschieden, denn es gilt: $S \xrightarrow[P]{*} w_0$ gdw. $w_0 \in S_m$. Denn $S \xrightarrow[P]{*} w_0$ impliziert $w_0 \in S_k$, wobei k die Länge der Ableitungskette ist. Für $k \leq m$ gilt aber $S_k \subseteq S_m$ und für $k > m$ gilt $S_m = S_k$ nach Voraussetzung, also $w_0 \in S_m$. Die Umkehrung ist offenbar.

Es bleibt also die Existenz eines m mit $S_m = S_{m+1}$ zu zeigen. Nun gilt aber für alle $k \geq 0$ und alle $w \in S_k$, dass $|w| \leq n$ und damit

$$\#S_k \leq \{w \in (N \cup T)^* \mid |w| \leq n\} = K_n < \infty.$$

Wegen $S_k \subseteq S_{k+1}$ muss also nach spätestens $m = K_n$ Schritten $S_{m+1} = S_m$ gelten. □

4.3 So ein Aufwand

Der Algorithmus, der das Wortproblem für monotone Grammatiken löst, sammelt – beginnend mit dem Startsymbol – alle Wörter bis zu einer vorgegebenen Länge auf, die sich mit wachsender Schrittzahl ableiten lassen, bis keine neuen Wörter mehr hinzukommen. Die resultierende Menge S_m kann im schlechtesten Fall K_n Elemente enthalten, also exponentiell viele, falls das Alphabet mindestens zwei Elemente enthält. Deshalb ist

der Algorithmus im schlechtesten Fall, der aber oft eintrifft, exponentiell und damit für praktische Zwecke unbrauchbar.

Es ist jedoch noch ungeklärt, ob es nicht eine polynomielle Lösung des Wortproblems monotoner Grammatiken gibt. Dies wird allerdings von Fachleuten für unwahrscheinlich gehalten.

Wenn man die jeweils nächste erfolgversprechende Regelanwendung richtig raten könnte, müsste man sich zwischendurch nur das bisher abgeleitete Wort merken, dessen Länge durch die Länge der Eingabe beschränkt ist. Probleme, die sich so lösen lassen, gehören zur Klasse NP-SPACE, wobei das "N" für *nichtdeterministisch* steht (und auf das Raten verweist) und "P-SPACE" auf den polynomiellen Platzbedarf verweist. Man weiß, dass die Klassen NP-SPACE und P-SPACE übereinstimmen, weil man den Nichtdeterminismus immer z.B. durch Backtracking beseitigen kann. Das Interessante an diesen Problemklassen ist, dass sie zu den größten bekannten gehören, die noch polynomiell lösbar sein könnten.

5 Das Cocke-Kasami-Younger-Verfahren

Das Verfahren von Cocke, Kasami und Younger ist eine Lösung des Wortproblems kontextfreier Grammatiken in Chomsky-Normalform. Es beruht auf dem Kontextfreiheitslemma und hat kubischen Aufwand.

Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik in *Chomsky-Normalform*,[3] d.h. für jede Produktion $A ::= r$ ist $r \in T$ oder $r \in N^2$. Will man wissen, ob ein Wort $w \in T^*$ aus einem nichtterminalen Zeichen $A \in N$ ableitbar ist, ob also $A \xrightarrow{*} w$ gilt, gibt es nur zwei relevante Fälle, in denen die Antwort positiv ausfällt:

1. $|w| = 1$ und $A ::= w \in P$ oder
2. $|w| > 1$ und es existieren $A ::= BC \in P$ sowie $B \xrightarrow{*} u$, $C \xrightarrow{*} v$ mit $w = uv$.

Wegen der Chomsky-Normalform leitet kein nichtterminales Zeichen das leere Wort ab. Außerdem hat die Ableitung $A \xrightarrow{*} w$ immer mindestens einen ersten Schritt $A \longrightarrow r$ für $A ::= r \in P$. Ist $r \in T$, muss die restliche Ableitung $r \xrightarrow{*} w$ die Länge 0 haben, d.h. $r = w$. Ist $r = BC$ für $B, C \in N$, dann induziert die restliche Ableitung $r = BC \xrightarrow{*} w$ nach dem Kontextfreiheitslemma zwei Ableitungen $B \xrightarrow{*} u$ und $C \xrightarrow{*} v$ mit $w = uv$. Außerdem hat damit w mindestens die Länge 2. Das ergibt insgesamt die beiden genannten Fälle, wenn man beachtet, dass die jeweiligen Rückrichtungen offensichtlich sind.

Um also für terminale Wörter der Länge 1 die Ableitbarkeit aus einem nichtterminalen Zeichen zu prüfen, muss man nur die terminierenden Regeln anschauen. Um sie für längere Wörter zu prüfen, muss man die nichtterminalen Regeln anschauen und das Wort in zwei Teile teilen. Das Anfangsstück muss aus dem ersten, das Endstück aus dem zweiten nichtterminalen Zeichen der rechten Regelseite ableitbar sein. Das ist dieselbe Frage, aber für kürzere Wörter, so dass diese Rekursion nach endlich vielen Schritten abbricht. Beachtet man noch, dass bei weiteren Zerlegungen der Wörter beliebige Teilwörter des ursprünglichen Wortes entstehen können, erhält man folgende Formulierung der obigen beiden Fälle, wobei als Gesamtwort $x_1 \cdots x_n$ (mit $x_l \in T$ für $l = 1, \ldots, n$) und als Teilwort $x_i \cdots x_{i+j-1}$ für $i = 1, \ldots, n$ und $j = 1, \ldots, n - i + 1$ betrachtet werden:

$A \xrightarrow{*} x_i \cdots x_{i+j-1}$ gdw.

- $j = 1$ und $A ::= x_i \in P$ oder
- $j > 1$, $A ::= BC \in P$ und es existiert k mit $1 \leq k < j$ derart, dass $B \xrightarrow{*} x_i \cdots x_{i+k-1}$ und $C \xrightarrow{*} x_{i+k} \cdots x_{i+j-1}$.

Das liefert ein Verfahren, um die nichtterminalen Zeichen zu bestimmen, aus denen sich Teilwörter von $x_1 \cdots x_n$ ableiten lassen.

[3] Zu jeder kontextfreien Grammatik, die nicht das leere Wort erzeugt, kann eine kontextfreie Grammatik in dieser Form konstruiert werden, die dieselbe Sprache erzeugt. Genaueres lässt sich z.B. in [HMU02, Kapitel 7] oder [EP00, Abschnitt 6.2 und 6.3] nachlesen.

Seien für $i = 1, \ldots, n$ und $j = 1, \ldots, n - i + 1$

$$CELL_{i,j} = \{A \in N \mid A \xrightarrow{*} x_i \cdots x_{i+j-1}\}.$$

Dann können diese "Zellen" nach der obigen Überlegung folgendermaßen berechnet werden:

- Für $i = 1, \ldots, n$: $CELL_{i,1} = \{A \in N \mid A ::= x_i \in P\}$;
- für $j = 2, \ldots, n$ und $i = 1, \ldots, n - j + 1$:

$$CELL_{i,j} = \bigcup_{k=1}^{j-1} \{A \in N \mid A ::= BC \in P,\ B \in CELL_{i,k},\ C \in CELL_{i+k,j-k}\}.$$

Damit ist auch das Wortproblem für G gelöst, denn es gilt:

$$x_1 \cdots x_n \in L(G) \text{ gdw. } S \xrightarrow{*} x_1 \cdots x_n \text{ gdw. } S \in CELL_{1,n}.$$

Da es für jedes $j = 1, \ldots, n$ jeweils $n - j + 1$ Zellen mit j als zweitem Index gibt, müssen insgesamt

$$\sum_{j=1}^{n} (n - j + 1) = \frac{n \cdot (n + 1)}{2} \leq n^2$$

Zellen berechnet werden. Für die Zellen $CELL_{i,1}$ geht das bei einer gegebenen Grammatik in konstanter Zeit, weil nur die gegebenen terminierenden Produktionen gefunden werden müssen. Um eine Zelle $CELL_{i,j}$ mit $j > 1$ zu bilden, muss man für $k = 1, \ldots, j - 1$ auf die Zellenpaare $CELL_{i,k}$ und $CELL_{i+k,j-k}$ zugreifen. Man kann annehmen, dass die bereits berechnet sind, weil ihre zweiten Indizes kleiner als das aktuelle j sind. Jede dieser Zellen enthält eine beschränkte Zahl von nichtterminalen Zeichen (höchstens $\#N$ viele). Aus den $j - 1$ Zellenpaaren sind die beschränkt vielen Elementpaare zu bilden (höchstens $\#N^2$ viele) und mit den rechten Seiten der Produktionen zu vergleichen. Auch das sind nur beschränkt viele Möglichkeiten. Für jede der höchstens quadratisch vielen Zellen muss man also höchstens linear viele Zellenpaare beschränkt lange bearbeiten, was insgesamt eine Zahl von Rechenschritten ergibt, die proportional zu n^3 ist, also kubisch.

6 Die Chomsky-Hierarchie

Wie in Abschnitt 3.4 erwähnt wurde, haben Chomsky-Grammatiken die ungünstige Eigenschaft, dass das Wortproblem für die erzeugten Sprachen im allgemeinen unentscheidbar ist. Grammatiken, die solche Sprachen erzeugen, sind z.B. für die Definition der Syntax von Programmiersprachen offenbar ungeeignet. Es stellt sich daher die Frage, ob man durch geeignete zusätzliche Bedingungen – insbesondere an die Form der Regeln – zu eingeschränkten Klassen von Chomsky-Grammatiken gelangen kann, die bessere Eigenschaften haben, aber trotzdem noch genügend Allgemeinheit besitzen. Dies führt zur sogenannten *Chomsky-Hierarchie*. Wie der Name andeutet, handelt es sich dabei um eine Hierarchie mehr oder weniger eingeschränkter Typen von Chomsky-Grammatiken.

Eine Chomsky-Grammatik $G = (N, T, P, S)$ ist

(1) *kontext-sensitiv*, falls alle Regeln in P die Form $u_1 A u_2 ::= u_1 v u_2$ haben, wobei $u_1, u_2, v \in (N \cup T)^*$, $v \neq \lambda$ und $A \in N$;
(2) *kontextfrei*, falls $u \in N$ für alle Regeln $u ::= v \in P$;
(3) *rechtslinear* (auch *regulär* genannt), falls für alle Regeln $u ::= v \in P$ gilt, dass $u \in N$ und entweder $v \in T^*$ oder $v = v'B$ mit $v' \in T^+$ und $B \in N$. (Hierbei bezeichnet T^+ die Menge $T^* \setminus \{\lambda\}$.)

Monotone und kontext-sensitive Grammatiken werden auch *Grammatiken vom Typ 1* genannt, kontextfreie werden als *Typ-2-* und rechtslineare als *Typ-3-Grammatiken* bezeichnet. Allgemeine Chomsky-Grammatiken werden Grammatiken vom *Typ 0* genannt. Eine Sprache L ist vom Typ i, wenn es eine Grammatik dieses Typs gibt, die L erzeugt.

Der folgende Satz rechtfertigt, warum diese Typeneinteilung nicht zwischen monotonen und kontext-sensitiven Grammatiken unterscheidet.

Theorem 2
Monotone und kontext-sensitive Grammatiken erzeugen dieselbe Klasse von Sprachen.

Aus der Definition von kontext-sensitiven Grammatiken folgt sofort, dass sie monoton sind. Umgekehrt lässt sich zeigen, dass zu jeder monotonen Grammatik eine kontext-sensitive konstruiert werden kann, die dieselbe Sprache erzeugt (in einem solchen Fall spricht man auch von einer *Normalform*-Grammatik). Wer genaueres wissen will, kann den Beweis z.B. in [EP00, Satz 8.1.1] nachlesen.

Die Berechtigung, von einer *Hierarchie* zu sprechen, liefert der nächste Satz.

Theorem 3
Sei $\mathcal{L}_i$ die Menge aller Sprachen des Typs i ($i \in \{0, \ldots, 3\}$). Dann gilt:

1. $\mathcal{L}_1 \subsetneqq \mathcal{L}_0$,
   d.h. Monotonie bzw. Kontext-Sensitivität ist eine echte Einschränkung.
2. $\{L \setminus \{\lambda\} \mid L \in \mathcal{L}_2\} \subsetneqq \mathcal{L}_1$,
   d.h. bis auf die bei monotonen Grammatiken offenbar nicht bestehende Möglichkeit, das leere Wort zu erzeugen, ist Kontextfreiheit eine echte Einschränkung gegenüber Kontext-Sensitivität.

3. $\mathcal{L}_3 \subsetneqq \mathcal{L}_2$,
   d.h. Rechtslinearität ist eine echte Einschränkung gegenüber Kontextfreiheit.

Aus der in Kapitel 4 behandelten Entscheidbarkeit des Wortproblems für Typ-1-Sprachen lässt sich die erste Aussage des obigen Satzes – $\mathcal{L}_1 \subsetneqq \mathcal{L}_0$ – als Folgerung ableiten. Zusammen mit der Unentscheidbarkeit des Wortproblems im allgemeinen Fall ergibt sich nämlich aus der Entscheidbarkeit für $\mathcal{L}_1$ die Ungleichheit $\mathcal{L}_1 \neq \mathcal{L}_0$ (und $\mathcal{L}_1 \subseteq \mathcal{L}_0$ gilt ohnehin per Definition).

Der Nachweis, dass die beiden anderen Inklusionen echt sind, wurde bereits im ersten Teil des Skripts mit Hilfe der Pumping-Lemmata gezeigt.

7 Ein unentscheidbares Problem für kontextfreie Grammatiken

Was verrät die Syntax über die Semantik? Das ist eine Kernfrage der Informatik, weil die semantische Ebene das Gewünschte und Interessierende repräsentiert, während nur die syntaktischen Beschreibungen explizit verfügbar sind. Das Halteproblem für Programme (und Turingmaschinen) und das Wortproblem für Grammatiken sind typische Probleme dieser Art.

Dummerweise sind viele semantische Fragen an syntaktischen Gebilden unentscheidbar – selbst dann noch, wenn man die Syntax stark einschränkt. Ein Beispiel dieser Art wird in diesem Kapitel für kontextfreie Grammatiken vorgestellt. Während das Leerheitsproblem für kontextfreie Grammatiken ($L(G) = \emptyset\,?$) entscheidbar ist, erweist sich bereits die Frage nach der Leerheit des Durchschnitts zweier kontextfreier Sprachen ($L(G_1) \cap L(G_2) = \emptyset\,?$) als unentscheidbar.

Um das zu beweisen, werden Postsche Korrespondenzprobleme, deren Lösbarkeit bekanntlich unentscheidbar ist, auf das Durchschnittsleerheitsproblem reduziert. Solche Reduktionen sind typisch für den Nachweis von Unentscheidbarkeit.

Betrachte dazu ein Postsches Korrespondenzproblem $PCP = ((u_1, \ldots, u_n), (v_1, \ldots, v_n))$ über dem Alphabet T. PCP ist lösbar, wenn es eine nichtleere Indexfolge $i_1 \cdots i_k$ mit $u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$ gibt. Die Konkatenationen aus beiden Listen zu Indexfolgen lassen sich gleichzeitig kontextfrei erzeugen, wenn man die zweite Konkatenation transponiert. Die entsprechende Grammatik G_{PCP} hat folgende Produktionen:

$$\left.\begin{array}{l} S ::= u_i A\, \mathit{trans}(v_i) \\ A ::= u_i A\, \mathit{trans}(v_i) \mid \$ \end{array}\right\} \text{ für } i = 1, \ldots, n.$$

Offensichtlich lassen sich damit aus S die terminalen Wörter der Form

$$u_{i_1} \cdots u_{i_k} \$\, \mathit{trans}(v_{i_k}) \cdots \mathit{trans}(v_{i_1}) = u_{i_1} \cdots u_{i_k} \$\, \mathit{trans}(v_{i_1} \cdots v_{i_k})$$

ableiten. Mit anderen Worten ist PCP genau dann lösbar, wenn $L(G_{PCP})$ ein Wort der Form $w\$\, \mathit{trans}(w)$ enthält.

Solche Wörter lassen sich aber bekanntlich durch eine kontextfreie Grammatik G_{mirror} mit den Produktionen

$$S ::= \$ \mid xSx \text{ für } x \in T$$

erzeugen. Also ist PCP genau dann lösbar, wenn $L(G_{PCP}) \cap L(G_{mirror}) \neq \emptyset$.

Wäre nun das Durchschnittsleerheitsproblem für kontextfreie Grammatiken entscheidbar, gälte das insbesondere für die Grammatiken G_{PCP} und G_{mirror}, so dass sich die Lösbarkeit von Postschen Korrespondenzproblemen als entscheidbar erwiese. Der Widerspruch ist nur dadurch auflösbar, dass die Annahme falsch ist. Die Frage nach der Leerheit des Durchschnitts von Sprachen, die von kontextfreien Grammatiken erzeugt werden, muss also unentscheidbar sein.

Liefert eine Indexfolge eine Lösung für ein PCP, so bildet jede Wiederholung der Indexfolge ebenfalls eine Lösung. Ein PCP hat also unendlich viele Lösungen, wenn es überhaupt Lösungen hat. Mit der obigen Übersetzung ist PCP genau dann lösbar, wenn $L(G_{PCP}) \cap L(G_{mirror})$ unendlich ist. Die Frage nach der (Un-)Endlichkeit des Durchschnitts zweier kontextfrei erzeugter Sprachen erweist sich also ebenso wie die Frage nach der Leerheit des Durchschnitts als unentscheidbar.

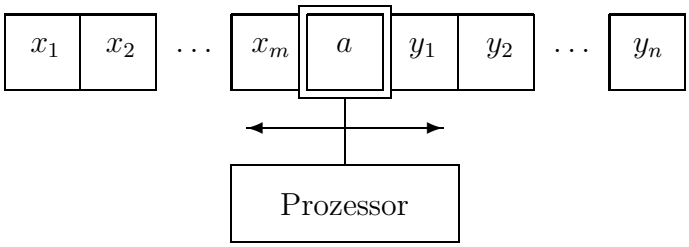
Das Wortproblem für den Durchschnitt ist übrigens entscheidbar, denn ein Wort liegt genau dann im Durchschnitt, wenn es in beiden Sprachen liegt. Das Wortproblem für die einzelnen kontextfreien Sprachen ist aber entscheidbar.

8 Turing-Maschinen

Das Konzept der Turing-Maschine wurde von Alan Turing in den 30er Jahren des letzten Jahrhunderts eingeführt und stellt damit eines der ältesten Berechenbarkeitsmodelle dar. Die Idee war, den mechanischen Anteil des Rechnens mit Bleistift und Radiergummi auf Papier formal zu fassen.

8.1 Begriff und Arbeitsweise

Eine Turing-Maschine *TM* funktioniert so: Sie befindet sich zu jeder Zeit in einem Zustand, der aus einer vorgegebenen endlichen Menge S von Zuständen stammt, mit denen sich also endlich viele verschiedene Fälle darstellen lassen. Sie hat ein Arbeitsband, das aus einer linearen Kette von Zellen besteht, die von links nach rechts geordnet sind. Jede Zelle ist mit einem Zeichen aus einem endlichen Alphabet A beschrieben, das wir Arbeitsalphabet nennen. Das Arbeitsalphabet enthält das Sonderzeichen $\Box$, das unbeschriebene Zellen repräsentiert. Ist somit eine Zelle mit $\Box$ beschrieben, wird sie – etwas widersprüchlich – als *unbeschrieben* bezeichnet. Zu jeder Zeit ist das Band endlich lang, kann aber unter bestimmten Umständen links und rechts durch unbeschriebene Zellen verlängert werden. Außerdem hat die Maschine einen Lese-Schreib-Kopf, der entweder am rechten Ende des Bandes oder auf einer der Zellen steht. Formal ist deshalb das Arbeitsband durch zwei Zeichenketten $u, v \in A^*$ beschrieben, wobei u den Bandinhalt links vom Kopf und v den Bandinhalt rechts vom Kopf – einschließlich der aktuellen Zelle, falls der Kopf nicht ganz rechts steht – darstellt. Abgesehen vom Arbeitsalphabet gibt es ein endliches Eingabealphabet I, welches von A umfasst wird und das Sonderzeichen $\Box$ nicht enthält. Zu Beginn befindet sich die Maschine in einem ausgezeichneten Anfangszustand, der Lese-Schreib-Kopf steht ganz links und alle Zellen sind mit Zeichen aus dem Eingabealphabet beschrieben (d.h. $u = \lambda$ und $v \in I^*$). Die Maschine kann Arbeitsschritte vollziehen, die von ihrem "Programm" – der Zustandsüberführungsrelation d – abhängen. Diese ordnet jedem Zustand und jedem gelesenen Zeichen aus A mögliche Folgezustände, mögliche zu schreibende Zeichen aus A und mögliche Bewegungen des Kopfes zu. Als Bewegungen stehen zur Verfügung "l" für eine Zelle nach links,"r" für eine Zelle nach rechts und "n" für Nicht-Bewegen. Ein konkreter Schritt ändert dann den aktuellen Zustand sowie den Inhalt der Zelle unter dem Lese-Schreib-Kopf und führt eine Bewegung aus – und das alles gemäß der Zustandsüberführungsrelation. Solche Arbeitsschritte können beliebig wiederholt werden. Die Schrittfolge endet notwendigerweise, wenn die Zustandsüberführungsrelation keinen Folgezustand mehr vorsieht. Das tritt insbesondere ein, wenn der aktuelle Zustand ein Endzustand ist. Alles was in diesem Fall unter dem Kopf und rechts davon steht, wird als Ergebnis der Berechnung angesehen, wenn da keine Zelle unbeschrieben ist.



Diese Konzeption kann folgendermaßen formalisiert werden:

1. Eine *Turing-Maschine* ist ein System $TM = (S, I, A, d, s_0, F)$, wobei S eine endliche Menge von *Zuständen*, I ein endliches *Eingabealphabet* mit $\square \notin I$, A ein endliches *Arbeitsalphabet* mit $I \subseteq A$ und $\square \in A$, $s_0 \in S$ ein *Anfangszustand*, $F \subseteq S$ eine Menge von *Endzuständen* und d eine *Zustandsüberführungsrelation* ist, die jedem Zustand $s \in S$ und jedem Zeichen $a \in A$ eine Teilmenge $d(s,a) \subseteq S \times A \times \{n, l, r\}$ zuordnet. Dabei sind n, l und r drei Steuerzeichen.
2. TM ist *deterministisch*, falls $d(s,a)$ für jedes $s \in S$ und $a \in A$ höchstens ein Element enthält.
3. Eine *Konfiguration* hat die Form usv mit $u, v \in A^*$ und $s \in S$.
4. Eine *Anfangskonfiguration* hat die Form $\lambda s_0 w$ mit $w \in I^*$.
5. Eine *Endkonfiguration* hat die Form $us'v\square z$ mit $u, z \in A^*, s' \in F$ und $v \in I^*$.
6. *Folgekonfigurationen* entstehen für alle $s, s' \in S$, $u, v \in A^*$ und $a, b, c \in A$ wie folgt:

(i)	$usav \vdash us'bv$,	falls $(s', b, n) \in d(s,a)$
(ii)	$usav \vdash ubs'v$,	falls $(s', b, r) \in d(s,a)$
(iii)	$ucsav \vdash us'cbv$	falls $(s', b, l) \in d(s,a)$
(iv)	$\lambda sav \vdash s'\square bv$	
(v)	$us\lambda \vdash us\square$	

   Beachte, dass für $v = \lambda$ im Fall (ii) der Lese-Schreib-Kopf in der Folgekonfiguration $ubs'v$ nicht über einer Zelle steht, sondern am rechten Ende des Bandes. Um in diesem Fall weitere Folgekonfigurationen zu bilden, muss man mit dem in Fall (v) beschriebenen Übergang ganz rechts eine neue unbeschriebene Zelle erschaffen, über der sich dann auch der Lese-Schreib-Kopf befindet.
7. Die Arbeitsweise der Turing-Maschine besteht in der beliebigen Iteration solcher Konfigurationsübergänge:

   $$u_1 s_1 v_1 \vdash u_2 s_2 v_2 \vdash \cdots \vdash u_k s_k v_k,$$

   wofür kurz auch $u_1 s_1 v_1 \stackrel{k-1}{\vdash} u_k s_k v_k$ geschrieben werden kann, wenn die Zwischenschritte nicht explizit gebraucht werden. Ist auch die Schrittzahl unwesentlich, kann $k-1$ durch $*$ ersetzt werden.
8. Eine (partielle) Funktion $f\colon I^* \to I^*$ wird von einer Turing-Maschine *TM berechnet*, falls für alle $v, w \in I^*$ gilt:

   $$f(w) = v \quad \text{gdw.} \quad \lambda s_0 w \stackrel{*}{\vdash} us'v\square z \text{ für geeignete } u, z \in A^*, s' \in F.$$

In diesem Fall wird f auch mit f_{TM} bezeichnet.

9. Turing-Maschinen können nicht nur Funktionen berechnen, sondern definieren auch Sprachen. Ein Wort w aus I^* gehört zu der Sprache einer Turing-Maschine TM, wenn sie bei Eingabe von w in einen Endzustand gelangen kann, d.h.

$$L(TM) = \{w \in I^* \mid \lambda s_0 w \vdash^* us'v, u, v \in A^*, s' \in F\}$$

8.2 Deterministische Turing-Maschinen

Während bei einer beliebigen Turing-Maschine eine Konfiguration mehrere Folgekonfigurationen besitzen kann, ist bei einer deterministischen Maschine immer höchstens ein Übergang möglich. Jede Anfangskonfiguration kann also in genau einer Weise durch Konfigurationsübergänge ausgerechnet werden, wobei die Übergänge entweder unendlich fortsetzbar sind und die Maschine nicht hält, oder aber die Folge in einer Konfiguration endet, zu der es keine Folgekonfiguration gibt. Ist diese eine Endkonfiguration, so ist die Berechnung erfolgreich. Sonst hält die Maschine ohne Ergebnis.

Ohne Beweis sei angemerkt, dass deterministische und nichtdeterministische Turing-Maschinen dieselbe Klasse von Funktionen berechnen.

Literatur

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles,Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[EP00] Katrin Erk and Lutz Priese. *Theoretische Informatik*. Springer, Berlin Heidelberg, 2000.

[HMU02] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Pearson Studium, 2002.

[HMU07] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation, third edition*. Pearson, 2007.

[MAK88] Robert N. Moll, Michael A. Arbib, and A.J. Kfoury. *An Introduction to Formal Language Theory*. Springer, New York, 1988.

[Sal73] Arto K. Salomaa. *Formal Languages*. Academic Press, New York, 1973.

[Sal78] Arto K. Salomaa. *Formale Sprachen*. Springer, Berlin, 1978.