Prof. Dr. Hans-Jörg Kreowski
Studiengang Informatik
MZH 3260
Tel.: 2956, 3697 (Sekr.), Fax: 4322
email: kreo@informatik.uni-bremen.de

Wintersemester 1996/97

Theoretische Informatik 2

Inhaltsverzeichnis

1 Syntax von Programmiersprachen **3**

2 Chomsky-Grammatiken **5**

3 Beispiele **6**

4 Immerhin aufzählbar **7**
4.1 Aufzählbarkeit erzeugter Sprachen 8
4.2 Die halbe Miete 9
4.3 Erzeugbarkeit aufzählbarer Sprachen 9
4.4 Unlösbarkeit des Wortproblems 9
4.5 Ausschöpfende Suche in die Breite mit roher Gewalt 9

5 Lösbarkeit des Wortproblems für monotone Grammatiken **10**
5.1 Monotone Grammatiken 10
5.2 Lösung des Wortproblems für monotone Grammatiken 11
5.3 So ein Aufwand 12

6 Endliche Automaten **12**
6.1 Definition endlicher Automaten 12
6.2 Der Potenzautomat 13
6.3 Pumping-Lemma für erkannte Sprachen 14

7 Rechtslineare Grammatiken und endliche Automaten **14**
7.1 Übersetzung endlicher Automaten in Chomsky-Grammatiken 15
7.2 Rechtlineare Grammatiken 16
7.3 Übersetzung rechtslinearer Grammatiken ohne Abweichung in endliche Automaten 16
7.4 Verkleinern der Abweichung 17
7.5 Übersetzung rechtslinearer Grammatiken in endliche Automaten 18

8 Reguläre Sprachen **18**
8.1 Reguläre Operationen 18
8.2 Endliche Automaten erkennen reguläre Sprachen 19

9 Kellerautomaten **20**
9.1 Konzept des Kellerautomaten 20
9.2 Deterministische Kellerautomaten 21
9.3 Beispiel: Reguläre Ausdrücke 22

10 Von kontextfreien Grammatiken zu Kellerautomaten **23**
10.1 Der Übersetzer 23

11 Linksableitungen **24**
11.1 Linksvertauschung 24
11.2 Linksableitung 25
11.3 Transformation von Ableitungen in Linksableitungen 25

12 Kontextfreiheitslemma **26**

13 Das Cocke-Kasami-Younger-Verfahren **26**

14 Ableitungsbäume **28**
14.1 Konstruktion 28
14.2 Beziehung zwischen Baumhöhe und Resultatslänge 29
14.3 Ein langer Weg von der Wurzel zu einem Blatt 30
14.4 Anhängen und Abhängen von Ableitungsbäumen 31

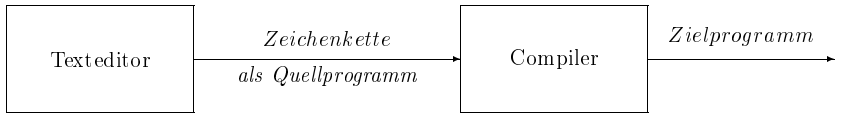
15 Pumping-Lemma für kontextfreie Sprachen **31**

16 Ein unentscheidbares Problem für kontextfreie Grammatiken **33**

17 Literatur **35**

1 Syntax von Programmiersprachen

Wer ein Programm in einer Programmiersprache X schreiben möchte, wählt sich häufig einen Texteditor Y aus und erstellt mit dessen Hilfe eine bestimmte Zeichenkette, die dann als Eingabe für den Compiler von X dient.



Man kann nicht erwarten, daß jeder Text, der mit dem Editor Y entstehen kann, bereits ein Programm in X ist. Denn mit einem Texteditor lassen sich in der Regel beliebige Zeichenketten aufbauen, während ein Programm eine bestimmte Form haben muß. Zeichenketten jedoch, die keine Programme sind, wird der Compiler als unübersetzbar zurückweisen. Woher weiß aber eine Programmiererin oder ein Programmierer, wie die Zeichenkette aussehen muß, um ein Programm zu sein? Wie findet der Compiler heraus, ob irgendeine Zeichenkette ein übersetzbares Programm darstellt? Die Form von Programmen einer Programmiersprache wird durch ihre Syntax festgelegt. Die Syntaxdefinition macht meist äußerst restriktive Vorschriften über die Anordnung und Plazierung von Zeichen, damit eine Zeichenkette ein syntaktisch richtiges Programm ist. Eine Person, die ein Programm mit Hilfe eines Texteditors erstellen will, sollte die Syntax recht gut kennen, weil andernfalls sicherlich häufig Syntaxfehler auftreten. Der Compiler dagegen übersetzt die eingegebene Zeichenkette in ein Zielprogamm, falls die Eingabe ein syntaktisch korrekt gebildetes Programm ist. Um das festzustellen, besitzen Compiler eine Syntaxanalyse-Komponente, die diese Aufgabe übernimmt.
Bei der Syntaxanalyse wird für eingegebene Zeichenketten untersucht, ob sie Programme sind oder nicht. Im positiven Fall wird außerdem der syntaktische Aufbau ermittelt, weil diese Information bei der weiteren Übersetzung maßgeblich genutzt wird. Im negativen Fall werden meist noch Hinweise auf Syntaxfehler gegeben. Die Trennung in "richtige" und "falsche" Zeichenketten ist in der Regel das entscheidende algorithmische Problem, weil bei der Lösung die zusätzlichen Informationen ohne allzu große Mühe nebenbei gewonnen werden können.

Die syntaktisch richtigen Programme einer Programmiersprache bilden als Menge von Zeichenketten eine "formale Sprache". Solche Zeichenkettenmengen sind Gegenstand der Untersuchung in der Theorie formaler Sprachen, die Konzepte für die syntaktische Definition formaler Sprachen bereitstellt und Methoden liefert, um die Eigenschaften formaler Sprachen analysieren zu können. Zu den wichtigsten Anwendungsfeldern der Theorie formaler Sprachen gehören die Syntaxdefinition von Programmiersprachen und die Syntaxanalyse. Die Schlüsselfrage der Syntaxanalyse, ob eine Zeichenkette ein Programm ist oder nicht, wird auch *Wortproblem* genannt. Betrachtet man die Menge aller richtigen Programme als formale Sprache, dann besteht das Problem darin, ob eine Zeichenkette in der Sprache liegt oder nicht. Da die Lösung des Wortproblems eine zentrale Rolle bei der Implementierung von Programmiersprachen spielt, aber keineswegs immer auf der Hand liegt, zieht sich die Behandlung des Wortproblems wie ein roter Faden durch die Theorie formaler Sprachen – und durch diese Lehrveranstaltung.

Aber erst einmal zurück zur Syntaxdefinition. Eine grundlegende Weise, formale Sprachen, einschließlich Programmiersprachen, zu spezifizieren, besteht darin, die übliche Art der Begriffsbildung (unter *dem und dem* versteht man *das und das*) in formalisierter From zu nutzen. Einem zu definierenden, also noch undefinierten, syntaktischen Konstrukt wird ein definierender Ausdruck zugeordnet. Beides zusammen bildet eine Syntaxregel, das (noch) Undefinierte wird linke, das Definierende rechte Regelseite genannt. Der definierende Ausdruck der rechten Seite ist eine Zeichenkette, die rekursiv auch wieder undefinierte Konstrukte enthalten darf. Ist das noch Undefinierte der linken Seite durch ein einziges Zeichen dargestellt, spricht man von einer kontextfreien Regel. Die Syntaxdefinition einer Programmiersprache besteht in einem ersten Anlauf meist in der Angabe von kontextfreien Regeln, die dann später um Syntaxteile ergänzt werden, die sich nicht durch kontextfreie Regeln ausdrücken lassen.

Der kontextfreie Anteil der Syntax von Programmiersprachen wird häufig in der sogenannten Backus-Naur-Form geschrieben, wobei die kontextfreien Regeln als linke Seiten nichtterminale Zeichen, die zu definierende syntaktische Konstrukte der Sprache benennen, und als rechte Seiten Zeichenketten aus terminalen und nichtterminalen Zeichen besitzen. Die rechten Seiten zur selben linken Seite werden als Alternativen nebeneinandergestellt, durch einen senkrechten Strich voneinander getrennt. Linke und rechte Seiten werden durch das Trennzeichen "::=" auseinandergehalten. Nichtterminale Zeichen sind in spitze Klammern eingeschlossen. Für Spezifikationen in CE-S sieht das im Ausschnitt so aus:

<*cesspec*>::=<*name*> opns: <*decllist*> eqns: <*celist*>
<*decllist*>::=<*decl*>|<*decl*>, <*decllist*>
<*celist*>::=<*ce*>|<*ce*>, <*celist*>
<*decl*>::=<*name*>:<*typelist*>→<*type*>
<*ce*>::=<*eq*> falls <*eqlist*> für <*varlist*>

Die Regeln können zum Aufbau syntaktisch korrekter Programme bzw. Programmstücke verwendet werden, indem mit dem gewünschten Konstrukt begonnen wird und dann nach und nach in den aktuellen Zeichenketten auftretende nichtterminale Zeichen durch zugehörige rechte Seiten ersetzt werden; z.B.:

<*cesspec*> ⟶ <*name*> opns: <*decllist*> eqns: <*celist*>
⟶ <*name*> opns: <*decl*> eqns: <*celist*>
⟶ <*name*> opns: <*decl*> eqns: <*ce*>, <*celist*>
⟶ <*name*> opns: <*decl*> eqns: <*ce*>, <*ce*>
⟶ <*name*> opns: <*name*>:<*typelist*>→<*type*> eqns: <*ce*>, <*ce*>
⋮
⟶ **sort**
opns: $sort\colon A^* \to A^*$
eqns: $sort(\lambda) = \lambda,$
$sort(xu) = insort(x, sort(u))$ für $x \in A, u \in A^*$

wobei allerdings die resultierende Spezifikation mit den obigen Regeln nicht erreichbar ist. Dazu müßte die Syntax von CE-S vervollständigt werden. Ziel solchen Ableitens ist eine Zeichenkette ohne nichtterminale Zeichen, die dann bezüglich der kontextfreien Syntax ein korrektes Konstrukt darstellt. Dieses Prinzip findet sich in syntaxgesteuerten Editoren wieder.

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). Eine Chomsky-Grammatik selbst besteht dann 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 $S \in A - T$ ist und in den linken Seiten von Produktionen Zeichen vorkommen, die nicht terminal sind. Die Differenzmenge $A - T$ 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 $u ::= v_1 | \cdots | v_k$ 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. $u \in (N \cup T)^* N (N \cup T)^*$ für jede Produktion $u ::= v \in P$. In einzelnen Fällen wird die Forderung eines Startssymbols durch ein Startwort $S \in (N \cup T)^*$ ersetzt. □

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 und Ullman [HU69] mit der deutschen Übersetzung [HU90], Moll, Arbib und Kfoury [MAK88] und Salomaa [Sal73]). Die Verbindung von formalen Sprachen und der syntaktischen Behandlung von Programmiersprachen wird umfassend in Aho und Ullman [AU72] abgehandelt.

3 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:

$$L((\{S\}, \{a, b\}, \{S ::= aSb \mid \lambda\}, S)) = \{a^n b^n \mid n \in \mathbb{N}\}.$$

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 $P_l = \{S ::= A_1 \cdots A_l\} \cup \{A_i ::= a_i A_i \mid i = 1, \ldots, l\} \cup \{A_i ::= \lambda \mid i = 1, \ldots, l\}$.
   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 $\{ab^nc ::= a^2b^{n+1}a^2\}$ 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:

$$\begin{array}{lllll} (1) & S & ::= & aXYZ \\ (2) & X & ::= & aXY \\ (3) & Y & ::= & BC \\ (4) & CB & ::= & BC \\ (5) & XB & ::= & bX \mid b \\ (6) & CZ & ::= & Zc \mid c \end{array}$$

Mit (1), (2) und (3) erhält man $S \overset{*}{\longrightarrow} a^nX(BC)^nZ$.

Mit (4) wird daraus: $a^nXB^nC^nZ$.

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

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

4 Immerhin aufzählbar

In diesem Kapitel wird der Zusammenhang zwischen Chomsky-Grammatiken und dem Konzept der Aufzählbarkeit (vgl. ThI 1, 7.4) beschrieben. Der Ableitungsprozeß zusammen mit einem Terminalitätstest für Zeichenketten zählt die erzeugte Sprache einer Chomsky-Grammatik auf (4.1), deren Wortproblem sich damit auch als "halb" lösbar erweist (4.2). Es wird außerdem plausibel gemacht, daß effektiv aufzählbare Mengen von Zeichenketten von Chomsky-Grammatiken erzeugt werden (4.3), allerdings ist das Wortproblem nicht immer "ganz" lösbar (4.4). In Punkt 5 schließlich wird auf Zusammenhänge zwischen dem hier angegebenen Aufzählungsverfahren und anderen bekannten Algorithmen hingewiesen.

4.1 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.

$$S(G)_k = \{w \mid S \overset{l}{\underset{P}{\longrightarrow}} w, l \leq k\}.$$

Es gilt offenbar $S(G)_k \subseteq S(G)_m$ für $k \leq m$. Es gilt genauer: $S(G)_0 = \{S\}$ und $S(G)_{k+1} = S(G)_k \cup \{w \mid v \underset{P}{\longrightarrow} w, v \in S(G)_k\}$. 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.

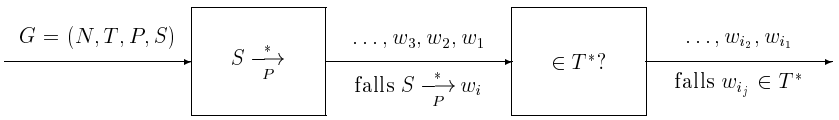
Bezeichnet man die Menge aller aus S ableitbaren Zeichenketten mit $S(G)$, so gilt offenbar:

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

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 jedesWort aus $L(G)$ entsteht.



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

4.2 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 später noch oder gar nicht geschehen. Das Wortproblem ist so nur "halb" gelöst, was man auch *semi-entscheidbar* nennt.

4.3 Erzeugbarkeit aufzählbarer Sprachen

Daß auch die Umkehrung gilt, daß also jede rekursiv aufzählbare Menge von Zeichenketten von einer Chomsky-Grammatik erzeugt werden kann, soll nicht bewiesen werden, da das zu viel Zeit und Mühe erforderte. Man kann sich aber die Richtigkeit dieser Behauptung mit den bisherigen Erkenntnissen der theoretischen Informatik recht gut plausibel machen.

Eine Menge von Zeichenketten ist rekursiv aufzählbar, falls es einen Algorithmus gibt, der die Elemente der Menge nach und nach "aufzählt". Nach der Churchschen These ist alles Algorithmische auch mit einer Turingmaschine machbar. Die Arbeitsweise einer Turingmaschine jedoch kann von einer Chomsky-Grammatik simuliert werden. Denn Konfigurationen sind bereits Zeichenketten, die den aktuellen Bandinhalt, den aktuellen Zustand und die Position des Lese/Schreibkopfes repräsentieren. Die Zustandsüberführung läßt sich auch durch Produktionen ausdrücken. Es ist dann nicht mehr allzu schwierig, die Grammatik erzeugen zu lassen, was die Turingmaschinen aufzählt. Details zu der Verbindung zwischen Turingmaschinen und Chomsky-Grammatiken finden sich in vielen Büchern über formale Sprachen.

4.4 Unlösbarkeit des Wortproblems

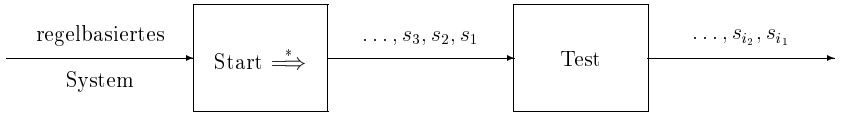
In ihrer allgemeinen Form sind Chomsky-Grammatiken allerdings nur bedingt für die Definition der Syntax von Programmiersprachen geeignet, weil nicht zu jeder definierbaren Syntax auch eine Syntaxanalyse möglich ist. Auch das soll nicht formal bewiesen, sondern nur plausibel gemacht werden.

Betrachte etwa folgenden Algorithmus: Generiere nach und nach alle PASCALCHEN-Programme und alle ihre initialen Berechnungszustände; interpretiere jedes dieser Programme für jeden dieser Berechnungszustände; terminiert die Berechnung, gilt das Paar aus Programm und initialem Berechnungszustand als aufgezählt. Die so entstehende Menge ist also rekursiv aufzählbar und läßt sich nach der vorangegangenen Überlegung von einer Chomsky-Grammatik erzeugen. Das zugehörige Wortproblem kann aber nicht lösbar sein, weil es sonst das Halteproblem löste, was bekanntlich unmöglich ist.

4.5 Ausschöpfende Suche in die Breite mit roher Gewalt

Das der Aufzählung der erzeugten Sprache in Punkt 1 zugrundliegende Verfahren 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. Die Vorwärtsinterpretation von CE-S-Spezifikationen, die Berechnung von PASCALCHEN-Programmen und die Turingmaschinen funktionieren nach diesem Prinzip. Man mache sich in diesen drei Fällen klar, was die Systemzustände, was die Regeln sind und nach welchem Kriterium ausgefiltert wird.



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 Punkt 1 auf, d.h. nach wachsender Ableitungslänge bzw. wachsender Zahl von Regelanwendungen, so spricht man häufig auch von *Breitensuche* (*breadth first search*).

5 Lösbarkeit des Wortproblems für monotone Grammatiken

Nach den Überlegungen des vorigen Kapitels 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.

5.1 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:

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

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

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:

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

5.2 Lösung des Wortproblems für monotone Grammatiken

Theorem

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 $m \geq 0$ sei S_m die Menge aller in höchstens m Schritten aus S ableitbaren Wörter, deren Länge n nicht überschreitet; d.h.

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

Offenbar gilt $S_0 = \{S\}$, und S_{m+1} läßt sich folgendermaßen aus S_m konstruieren

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

Damit gilt offenbar $S_m \subseteq S_{m+1}$, und wenn $S_m = S_{m+1}$ gilt, 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_m rekursiv konstruiert.
Es soll nun gezeigt werden, daß nach endlich vielen Schritten $S_m = S_{m+1}$ gilt.
Damit 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 $w \in S_k$, wobei k die Länge der Ableitungskette ist. Für $k \leq m$ gilt aber $S_k \subseteq S_m$ und für $k > m$ $S_m = S_k$ nach Voraussetzung, also $w \in S_m$. 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 $w \in S_k$ $|w| \leq n$ und damit

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

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

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

5.3 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, 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.

6 Endliche Automaten

Unter einem endlichen Automaten hat man sich ein taktweise arbeitendes System vorzustellen, das sich in einem bestimmten Zustand (von endlich vielen verfügbaren Zuständen) befindet und das dann innerhalb eines Taktes einen Buchstaben des Eingabewortes einliest, daraufhin seinen Zustand ändert und den Lesekopf auf den nächsten Eingabebuchstaben einstellt usw. Ein Eingabewort gilt dann als erkannt, wenn vom Anfangszustand aus nach Verarbeitung des Wortes ein Endzustand erreicht wird.

6.1 Definition endlicher Automaten

Die Veranschaulichung führt zu folgender

Definition

1. Ein *endlicher relationeller erkennender Automat – kurz endlicher Automat* – ist ein System $A = (Z, I, d, s_0, F)$, wobei
   - Z eine endliche Menge von *Zuständen* ist,
   - I ein endliches *Eingabealphabet*,
   - $F \subseteq Z$ eine Menge von *Endzuständen*,
   - $s_0 \in Z$ der *Anfangszustand* und
   - $d \subseteq Z \times I \times Z$ eine Relation ist, geschrieben $d : Z \times I \rightsquigarrow Z$, die *Zustandsüberführung* genannt wird und jedem Zustand und jeder Eingabe eine Menge von Folgezuständen zuordnet.
2. Die *Zustandüberführung* d läßt sich rekursiv fortsetzen zu einer Relation $d^* : Z \times I^* \rightsquigarrow Z$:

(i) $d^*(s,\lambda) = \{s\}$ für alle $s \in Z$

(ii) $d^*(s,wx) = \bigcup_{t \in d^*(s,w)} d(t,x)$ für alle $s \in Z, x \in I, w \in I^*$.

3. Die von einem endlichen Automaten A *erkannte Sprache* $L(A)$ ist dann definiert durch:

$$L(A) := \{w \in I^* \mid d^*(s_0,w) \cap F \neq \emptyset\}.$$

4. Ein Automat $A = (Z, I, d, s_0, F)$ heißt *deterministisch*, wenn d eine Abbildung ist; d.h. für alle $s \in Z$ und $x \in I$ existiert genau ein $s' \in Z$, derart daß (s,x) und s' bzgl. d in Relation stehen.

Bemerkungen:

1. Für deterministische Automaten ist mit d auch d^* eine Abbildung. Deshalb können (i) und (ii) aus Punkt 2 in diesem Fall auch ausgedrückt werden durch:

(i') $d^*(s,\lambda) = s$

(ii') $d^*(s,wx) = d(d^*(s,w),x)$.

Ferner ist die von deterministischen Automaten erkannte Sprache bestimmt durch:

$$L(A) = \{w \in I^* \mid d^*(s_0,w) \in F\}.$$

2. Jeder endliche Automat A besitzt eine graphische Darstellung in Form eines *Zustandsgraphen*. Dabei werden die Zustände zu Knoten, und von einem Zustand(sknoten) s zu einem Zustand(sknoten) s' wird eine mit $x \in I$ markierte Kante gezogen, falls $s' \in d(s,x)$. Der Anfangszustand wird durch einen hineingehenden Pfeil, die Endzustände werden entsprechend durch herausgehende Pfeile gekennzeichnet.

3. Ist $R : A \rightsquigarrow B$ eine Relation (d.h. $R \subseteq A \times B$), dann wird statt $(a,b) \in R$ oft auch aRb geschrieben oder, wenn R eine Abbildung ist, $R(a) = b$. Ansonsten bezeichnet $R(a)$ die Menge aller Elemente, die bzgl. R zu a in Relation stehen: $R(a) = \{b \in B \mid aRb\}$.

6.2 Der Potenzautomat

Theorem

Zu jedem endlichen Automaten A läßt sich ein deterministischer Automat $P(A)$ konstruieren, der dieselbe Sprache erkennt; d.h.

$$L(A) = L(P(A)).$$

Beweis: Sei $A = (Z, I, d, s_0, F)$ ein endlicher Automat.
Daraus läßt sich der *Potenzautomat* konstruieren:

$$P(A) = (P(Z), I, D, \{s_0\}, F_P := \{S \subseteq Z | S \cap F \neq \emptyset\}),$$

wobei $P(Z)$ die Potenzmenge von Z ist und die Abbildung $D : P(Z) \times I \to P(Z)$ definiert ist durch $D(S,x) := \bigcup_{s \in S} d(s,x)$ für alle $S \in P(Z)$ und $x \in I$.
Dann stehen die fortgesetzten Zustandsüberführungen d^* und D^* in der Beziehung

$$d^*(s,w) = D^*(\{s\},w).$$

Denn für $w = \lambda$ gilt:

$$d^*(s,\lambda) = \{s\} = d^*(\{s\},\lambda),$$

und für Wörter wx erhält man, vorausgesetzt, die Behauptung gilt bereits für w:

$$\begin{aligned} d^*(s,wx) &= \bigcup_{t \in d^*(s,w)} d(t,x) \\ &= \bigcup_{t \in D^*(\{s\},w)} d(t,x) \\ &= D(D^*(\{s\},w),x) = D^*(\{s\},wx). \end{aligned}$$

Für die Sprachen $L(A)$ und $L(P(A))$ folgt somit:

$$\begin{aligned} w \in L(A) \quad &\text{gdw. } d^*(s_0,w) \cap F \neq \emptyset \\ &\text{gdw. } D^*(\{s_0\},w)(= d^*(s_0,w)) \in F_P \\ &\text{gdw. } w \in L(P(A)); \end{aligned}$$

$L(A)$ und $L(P(A))$ sind also gleich.

6.3 Pumping-Lemma für erkannte Sprachen

Theorem

Sei L eine von einem endlichen Automaten erkannte Sprache.
Dann existiert eine natürliche Zahl $p \in \mathbb{N}$, derart daß jedes Wort $w \in L$ mit $|w| \geq p$ zerlegt werden kann in drei Teilwörter $w = xyz$ mit $p \geq |y| > 0$ und daß $xy^iz \in L$ ist für alle $i \geq 0$.

Beweis: Nach Voraussetzung existiert ein Automat $A = (Z, I, d, s_0, F)$, der L erkennt. Wähle dann p als Anzahl der Zustände von A ($p = \sharp Z$). Gibt man nun ein Wort $w = x_1 \cdots x_n \in L$ ($x_i \in I$) mit $n \geq p$ in A ein, so erhält man durch buchstabenweises Abarbeiten eine Folge $\{s_i\}_{i=0}^n$ von Zuständen mit der Eigenschaft

$$s_i \in d(s_{i-1}, x_i) \text{ für } i = 1, \ldots, n.$$

Wegen $n \geq p$ gibt es unter den $n+1$ Zuständen $s_0, \ldots, s_n$ zwei gleiche, etwa $s_j = s_k$ mit $0 \leq j < k \leq n$. Das liefert die gewünschte Zerlegung von w, denn mit $x = x_1 \cdots x_j$ kommt man von s_0 nach s_j, mit $y = x_{j+1} \cdots x_k$ von s_j nach $s_k = s_j$, was man dann aber auch immer wiederholen oder auslassen kann, und von s_k gelangt man mit $z = x_{k+1} \cdots x_n$ nach $s_n \in F$. Insgesamt erreicht man also mit den Wörtern xy^iz für $i \geq 0$ von s_0 den Endzustand s_n. Das aber bedeutet gerade $xy^iz \in L$, wie behauptet.
Wie müssen j und k gewählt werden, damit $|y| \leq p$ ist?

7 Rechtslineare Grammatiken und endliche Automaten

In diesem Abschnitt wird ein Zusammenhang zwischen den von endlichen Automaten erkannten Sprachen und den von einer Klasse von Chomsky-Grammatiken erzeugten Sprachen hergestellt.

Es ist recht einfach, einen endlichen Automaten so in eine Chomsky-Grammatik umzuwandeln, daß die Grammatik die Sprache erzeugt, die der Automat erkennt. Dazu wird das Lesen und Verarbeiten eines Zeichens gemäß der Zustandsüberführung als Erzeugen des Zeichens gedeutet und das Erreichen eines Endzustands als Erzeugen des leeren Wortes.

Umgekehrt läßt sich eine Klasse von Chomsky-Grammatiken identifizieren, deren Mitglieder in endliche Automaten übersetzt werden können, so daß sich auch ihr Wortproblem als linear lösbar erweist. Das Charakteristikum dieser Grammatiken ist die sehr eingeschränkte Form der Regeln. Jede linke Seite ist ein nichtterminales Zeichen, jede rechte Seite enthält höchstens ein nichtterminales Zeichen, das immer ganz rechts steht, aber nie allein.

7.1 Übersetzung endlicher Automaten in Chomsky-Grammatiken

In diesem Abschnitt wird ein Übersetzer von endlichen Automaten in Chomsky-Grammatiken angegeben, bei dem im Prinzip nur die Zustandsüberführung in Regelform umgewandelt wird. Es wird gezeigt, daß der Übersetzer korrekt ist, d.h. daß der eingegebene Automat die Sprache erkennt, die die ausgegebene Grammatik erzeugt.

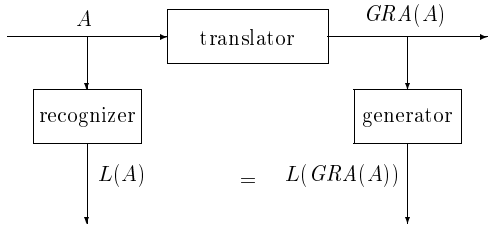
Theorem

Sei $A = (Z, I, d, s_0, F)$ ein endlicher Automat. Dann wird durch $GRA(A) = (Z, I, P_A, s_0)$ mit

$$P_A = \{(s ::= xs') \mid s' \in d(s,x)\} \cup \{(s'' ::= \lambda) \mid s'' \in F\}$$

eine Chomsky-Grammatik konstruiert, so daß gilt: $L(A) = L(GRA(A))$.

Diese Übersetzung (einschließlich der semantischen Verträglichkeit) läßt sich als Diagramm illustrieren:



Beweis: Da $GRA(A)$ offensichtlich eine Chomsky-Grammatik ist, muß nur die Sprachgleichheit gezeigt werden. Dazu wird ein Zusammenhang zwischen der fortgesetzten Zustandsüberführung von A und den Ableitungen von $GRA(A)$ im anschließenden Lemma hergestellt. Die behauptete Sprachgleichheit folgt dann vergleichsweise einfach: $w \in L(A)$ bedeutet $d^*(s_0, w) \cap F \neq \emptyset$, d.h. es gibt einen Zustand $s' \in F$ und $s' \in d^*(s_0, w)$. Nach Definition der Regeln und dem obigen Lemma ist das gleichbedeutend zu $(s' ::= \lambda) \in P_A$ und $s_0 \xrightarrow[P_A]{*} ws'$. Daraus läßt sich die Ableitung $s_0 \xrightarrow[P_A]{*} ws' \underset{s'::=\lambda}{\longrightarrow} w\lambda = w$ zusammensetzen. Die Ableitung $s_0 \xrightarrow[P_A]{*} w$ liefert aber gerade $w \in L(GRA(A))$. Umgekehrt zerfällt eine Ableitung der Form $s_0 \xrightarrow[P_A]{*} w$ immer in die Ableitungen $s_0 \xrightarrow[P_A]{*} ws'$ und $ws' \underset{s'::=\lambda}{\longrightarrow} w$, weil das die einzige Möglichkeit zum Terminieren ist. Damit läßt sich die gesamte Überlegung auch umdrehen. □

Lemma: Für alle $s, s' \in Z$ und $w \in I^*$ gilt: $s \xrightarrow[P_A]{*} ws'$ gdw. $s' \in d^*(s, w)$.

Beweis (mit Induktion über die Länge von w):

IA (für $w = \lambda$): $s \xrightarrow[P_A]{*} \lambda s' = s'$ gdw. $s = s'$ gdw. $s' \in \{s\} = d^*(s, \lambda)$, wobei in der zweiten Äquivalenz die Definition von d^* ausgenutzt wird und bei der ersten die Tatsache, daß jede Regelanwendung in $GRA(A)$ ein terminales Zeichen erzeugt oder das nichtterminale löscht.

IV: Die Behauptung gelte für $w \in I^*$.

IS (für wx mit $w \in I^*$ und $x \in I$): Eine Ableitung der Form $s \xrightarrow[P_A]{*} wxs'$ zerfällt immer in zwei Ableitungen der Form $s \xrightarrow[P_A]{*} w\overline{s}$ und $w\overline{s} \underset{\overline{s}::=xs'}{\longrightarrow} wxs'$, weil nur Regeln der Form $\overline{s} ::= xs'$ Wörter verlängern und weil das nur am rechten Ende geschehen kann. Nach der Induktionsvoraussetzung korrespondiert die Ableitung $s \xrightarrow[P_A]{*} w\overline{s}$ zu $\overline{s} \in d^*(s, w)$. Die im Ableitungsschritt angewendete Regel korrespondiert zu $s' \in d(\overline{s}, x)$. Beides zusammen bedeutet nach Definition von d^* gerade $s' \in \bigcup_{t \in d^*(s,w)} d(t, x) = d^*(s, wx)$. □

7.2 Rechtlineare Grammatiken

Eine Chomsky-Grammatik $G = (N, T, P, S)$ wird *rechtslinear* genannt, wenn jede Produktion die Form $A ::= u$ mit $A \in N$ und $u \in T^*$ oder $A ::= uB$ mit $A, B \in N$ und $u \in T^+ (= T^* - \{\lambda\})$ hat.

Offenbar ist die zu einem endlichen Automaten A konstruierte Grammatik $GRA(A)$ rechtslinear. Eine rechtslineare Grammatik muß jedoch nicht von einem endlichen Automaten stammen, denn ihre terminalen rechten Seiten müssen nicht unbedingt leer sein, und ihre weiteren rechten Seiten müssen neben dem nichtterminalen Zeichen nicht nur ein terminales Zeichen enthalten. Der Unterschied läßt sich aber offenbar quantitativ durch Längendifferenzen ausdrücken:

Für $p = (A ::= u)$ mit $A \in N$ und $u \in T^*$ wird die Länge $|u|$ *Abweichung* genannt und als *diff*(p) bezeichnet. Für $p = (A ::= uB)$ mit $A, B \in N$ und $u \in T^+$ wird $|u| - 1$ *Abweichung* genannt und als *diff*(p) bezeichnet. Für eine rechtslineare Grammatik $G = (N, T, P, S)$ wird die Summe aller Abweichungen von Produktionen *Abweichung* von G genannt und mit *diff*(G) bezeichnet, d.h.

$$\mathit{diff}(G) = \sum_{p \in P} \mathit{diff}(p).$$

7.3 Übersetzung rechtslinearer Grammatiken ohne Abweichung in endliche Automaten

Es ist ziemlich einfach, rechtslineare Grammatiken ohne Abweichung in endliche Automaten zu übersetzen, indem die Regeln als Zustandsüberführung aufgefaßt werden. Diese Übersetzung erweist sich als Umkehrung der Konstruktion in Punkt 1, woraus unmittelbar ihre Korrektheit folgt.

Theorem

Sei $G = (N, T, P, S)$ eine rechtslineare Grammatik mit *diff*$(G) = 0$. Dann kann durch $AUT(G) = (N, T, d_G, S, F_G = \{B \in N \mid (B ::= \lambda) \in P\})$ mit

$$d_G(A, x) = \{B \in N \mid (A ::= xB) \in P\} \text{ für alle } A \in N, x \in T$$

ein endlicher Automat konstruiert werden, dessen Grammatik gerade G ist, d.h. es gilt $GRA(AUT(G)) = G$.
Insbesondere gilt: $L(AUT(G)) = L(G)$.

Beweis: $AUT(G)$ ist offensichtlich ein endlicher Automat, dessen zugehörige Grammatik $GRA(AUT(G)) = (N, T, P_{AUT(G)}, S)$ ist mit

$$\begin{aligned} P_{AUT(G)} &= \{(A ::= xB) \mid B \in d_G(A,x)\} \cup \{(B ::= \lambda) \mid B \in F_G\} \\ &= \{(A ::= xB) \mid (A ::= xB) \in P\} \cup \{(B ::= \lambda) \mid (B ::= \lambda) \in P\}, \end{aligned}$$

wobei bei der zweiten Gleichung die Definition von d_G und F_G ausgenutzt wurde. Mit anderen Worten liegen in $P_{AUT(G)}$ gerade die Produktionen aus P, deren Abweichung 0 ist. Da G insgesamt die Abweichung 0 hat, gibt es keine anderen Produktionen, so daß P und $P_{AUT(G)}$ und damit G und $GRA(AUT(G))$ übereinstimmen.
Mit Hilfe von Theorem 7.2 folgt dann: $L(AUT(G)) = L(GRA(AUT(G)) = L(G)$. □

7.4 Verkleinern der Abweichung

Das folgende Ergebnis zeigt, daß für jede rechtslineare Grammatik mit einer positiven Abweichung eine konstruiert werden kann mit kleinerer Abweichung, die dieselbe Sprache erzeugt. Dabei wird eine Produktion mit positiver Abweichung durch zwei Produktionen ersetzt. Die Konstruktionen unterscheiden sich für terminierende und nichtterminierende Produktionen allerdings leicht voneinander.

Theorem
Sei $G = (N, T, P, S)$ eine rechtslineare Grammatik. Sei ferner X ein Extrazeichen mit $X \notin N \cup T$.

1. Für $p = (A ::= uxB) \in P$ mit $A, B \in N, x \in T, u \in T^+$ wird dann durch $G_p = (N \cup \{X\}, T, (P - \{p\}) \cup \{(A ::= uX), (X ::= xB)\}, S)$ eine rechtslineare Grammatik konstruiert mit $L(G) = L(G_p)$ und $\mathit{diff}(G) = \mathit{diff}(G_p) + 1$.

2. Für $p = (A ::= u) \in P$ mit $A \in N, u \in T^+$ wird dann durch $G_p = (N \cup \{X\}, T, (P - \{p\}) \cup \{(A ::= uX), (X ::= \lambda)\}, S)$ eine rechtslineare Grammatik konstruiert mit $L(G) = L(G_p)$ und $\mathit{diff}(G) = \mathit{diff}(G_p) + 1$.

Beweis: Daß G_p in beiden Fällen eine rechtslineare Grammatik ergibt, ist offensichtlich. In beiden Fällen sinkt auch die Abweichung um 1, weil jeweils die erste neue Produktion eine um 1 kleinere Abweichung hat als die herausgenommene und die jeweils zweite neue Produktion die Abweichung 0 besitzt.
Bleibt die Sprachgleichheit zu zeigen. Zuerst wird die erste Konstruktion betrachtet.
Sei $w \in L(G)$, d.h. $S \xrightarrow[P]{*} w$. Kommt p darin nicht vor, so ist das bereits eine Ableitung in G_p und damit $w \in L(G_p)$. Kommt p vor, so läßt sich die Ableitung zerlegen in $S \xrightarrow[P]{*} vA \underset{p}{\longrightarrow} vuxB \xrightarrow[P]{*} w$. Die Anwendung von p kann ersetzt werden durch die Anwendung der beiden neuen Produktionen, was $S \xrightarrow[P]{*} vA \underset{A::=uX}{\longrightarrow} vuX \underset{X::=xB}{\longrightarrow} vuxB \xrightarrow[P]{*} w$ ergibt. So lassen sich mit einfacher Induktion alle Vorkommen von p beseitigen, und es entsteht eine Ableitung von w aus S in G_p, so daß $w \in L(G_p)$.

Sei umgekehrt $w \in L(G_p)$, d.h. $S \xrightarrow[P']{*} w$, wenn P' die Produktionsmenge von G_p bezeichnet. Kommt dabei zwischendurch nirgends X vor, so ist diese Ableitung bereits in G, so daß $w \in L(G)$. Kommt X vor, so muß es durch die erste neue Produktion entstanden sein und unmittelbar danach von der zweiten wieder ersetzt werden, weil die Produktionen aus P weder links noch rechts X enthalten. Also zerfällt die Ableitung in $S \xrightarrow[P']{*} vA \underset{A::=uX}{\longrightarrow} vuX \underset{X::=xB}{\longrightarrow} vuxB \xrightarrow[P']{*} w$. Die zwei Schritte in der Mitte lassen sich durch $vA \underset{p}{\longrightarrow} vuxB$ ersetzen. Induktiv lassen sich so alle Vorkommen der neuen Produktionen beseitigen, so daß eine Ableitung $S \xrightarrow[P]{*} w$ entsteht und damit $w \in L(G)$.

Für die zweite Konstruktion geht der Beweis analog, wobei allerdings keine Induktion nötig ist, weil die terminierenden Regeln höchstens einmal angewendet werden. □

7.5 Übersetzung rechtslinearer Grammatiken in endliche Automaten

Eine rechtslineare Grammatik mit positiver Abweichung hat mindestens eine Produktion mit positiver Abweichung. Deshalb kann die Konstruktion für eine Grammatik G mit Abweichung k k-mal wiederholt werden. Dabei entsteht eine rechtslineare Grammatik $\overline{G}$ ohne Abweichung, für die die Konstruktion aus Punkt 3 durchgeführt werden kann.

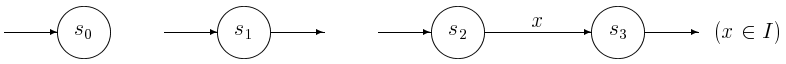
Insgesamt wird also G in $AUT(\overline{G})$ übersetzt mit $L(G) = L(\overline{G}) = L(AUT(\overline{G}))$. Die lineare Lösung des Wortproblems für endliche Automaten ist somit auch für rechtslineare Grammatiken anwendbar.

8 Reguläre Sprachen

Neben dem Erzeugen und Erkennen von Sprachen ist es interessant, ihre Kompositionalität oder Modularität zu untersuchen. Dabei geht es darum, wie sich umfangreiche und komplizierte Sprachen aus einfachen Bausteinen aufbauen lassen. Diese Frage ist auch für die Entwicklung großer Spezifikationen und informationstechnischer Systeme von zentraler Bedeutung. Es zeigt sich, daß sich die von endlichen Automaten erkannten Sprachen und damit die von rechtslinearen Grammatiken erzeugten in besonders einfacher Weise modular aufbauen lassen.

8.1 Reguläre Operationen

Die folgenden drei endlichen Automaten



erkennen die leere Menge $\emptyset$, die einelementige Sprache $\{\lambda\}$, die das leere Wort enthält, bzw. die einelementige Sprache $\{x\}$, die das Wort x der Länge 1 enthält. Diese Sprachen dienen als kleinste, unzerlegbare Sprachmoduln.

Sind L, L_1 und L_2 Sprachen zum Alphabet I, die von endlichen Automaten erkannt werden, so werden auch die zusammengesetzten Sprachen $L_1 \cup L_2, L_1 L_2$ und L^* von endlichen Automaten erkannt (vgl. Übungsblatt 4). Die dabei verwendeten Sprachoperationen sind

die Vereinigung, die Konkatenation ($L_1L_2 = \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}$) und die Kleene-Hülle ($L^* = \bigcup_{i\in\mathbb{N}} L^i$ mit $L^0 = \{\lambda\}$ und $L^{i+1} = L^iL$).

Bezeichne $\mathcal{L}_{AUT(I)}$ die Klasse aller Sprachen über dem Alphabet I, die von endlichen Automaten erkannt werden, so hat $\mathcal{L}_{AUT(I)}$ nach den obigen Überlegungen folgende Eigenschaften:

(i) $\emptyset, \{\lambda\}, \{x\} \in \mathcal{L}_{AUT(I)}$ für $x \in I$,

(ii) $L, L_1, L_2 \in \mathcal{L}_{AUT(I)}$ impliziert $L_1 \cup L_2, L_1L_2, L^* \in \mathcal{L}_{AUT(I)}$.

Die kleinste Klasse $\mathcal{L}_{REG(I)}$ von Sprachen mit diesen Eigenschaften sind die sogenannten *regulären Sprachen*, die durch endlich maliges Anwenden der Sprachoperationen in (ii), beginnend mit den Sprachen in (i), entstehen. Die regulären Sprachen sind also vollständig modular aufgebaut. Insbesondere gilt: $\mathcal{L}_{REG(I)} \subseteq \mathcal{L}_{AUT(I)}$.

8.2 Endliche Automaten erkennen reguläre Sprachen

Bemerkenswerterweise kann auch die Umkehrung nachgewiesen werden, so daß sich die von endlichen Automaten erkannten und von rechtslinearen Grammatiken erzeugten Sprachen als regulär erweisen und somit aus den atomaren Sprachmoduln allein durch endlich viele Vereinigungen, Konkatenationen und Kleene-Hüllen-Bildungen aufgebaut werden können.

Theorem

Sei $A = (Z, I, d, s_0, F)$ ein endlicher Automat. Dann ist $L(A)$ regulär.

Beweis: Ohne Beschränkung der Allgemeinheit kann $Z = \{1, \ldots, n\}$ und $s_0 = 1$ angenommen werden. Dann lassen sich für $i, j \in Z$ und $k \in \mathbb{N}$ folgende Sprachen betrachten:

$$L_{ij}^k = \{w \in I^* \mid j \in d^*(i, w), d^*(i, u) \subseteq \{1, \ldots, k\} \text{ für alle } u \text{ mit } w = uv, w \neq u \neq \lambda\}.$$

Das sind alle Wörter, die im Zustandsgraphen von i nach j führen, ohne zwischendurch Zustände jenseits von k zu besuchen.

Für $k = 0$ und $i \neq j$ sind das die Eingaben, die direkt von i nach j führen, weil alle Zustände jenseits von 0 liegen und deshalb nicht besucht werden dürfen. D.h. $L_{ij}^0 = \{x \in I \mid j \in d(i, x)\}$. Diese Sprache ist entweder leer oder die endliche Vereinigung von atomaren regulären Sprachen und deshalb selbst regulär. Für $k = 0$ und $i = j$ gehört in jedem Fall noch λ zur Sprache, d.h. $L_{ii}^0 = \{\lambda\} \cup \{x \in I \mid i \in d(i, x)\}$, was sich analog als regulär erweist, weil $\{\lambda\}$ regulär ist. Beides zusammen wird als *Induktionsanfang* verwendet.

Als *Induktionsvoraussetzung* wird von der Regularität von L_{ij}^k für alle i und j ausgegangen.

Betrachte nun im *Induktionsschluß* L_{ij}^{k+1} bzw. ein Element w daraus. Dieses Wort induziert einen Weg von i nach j im Zustandsgraph. Sei $l_1 \cdots l_p$ die durchlaufene Sequenz von Zuständen mit $i = l_1$ und $j = l_p$. Kommt zwischen i und j der Zustand $k + 1$ gar nicht vor, so liegt w in L_{ij}^k. Ansonsten kommt der Zustand $k+1$ zwischen i und j m-mal vor mit $0 < m < p$, so daß die Sequenz folgende Form hat

$$l_1 \cdots l_{p_1-1}(k+1)l_{p_1+1} \cdots l_{p_2-1}(k+1) \cdots (k+1)l_{p_m+1} \cdots l_p,$$

wobei $p_1 < p_2 < \cdots < p_m$ die Stellen sind, wo $k + 1$ auftritt. Insbesondere kommt in den Abschnitten $l_2 \cdots l_{p_1-1}, l_{p_1+1} \cdots l_{p_2-1}, \ldots, l_{p_m+1} \cdots l_{p-1}$ der Zustand $k+1$ nicht vor, sondern

nur die Zustände $1, \ldots, k$. Es stellt sich also heraus, daß $w \in L_{i(k+1)}^k (L_{(k+1)(k+1)}^k)^* L_{(k+1)j}^k$. Offensichtlich gilt auch die Umkehrung, so daß insgesamt

$$L_{ij}^{k+1} = L_{ij}^k \cup L_{i(k+1)}^k (L_{(k+1)(k+1)}^k)^* L_{(k+1)j}^k$$

folgt. Nach Induktionsvoraussetzung sind $L_{ij}^k, L_{i(k+1)}^k, L_{(k+1)(k+1)}^k$ und $L_{(k+1)j}^k$ regulär, so daß auch die Kleene-Hülle der dritten Sprache und die Konkatenation mit den anderen beiden Sprachen sowie die Vereinigung mit der ersten Sprache regulär sind. Also ist L_{ij}^{k+1} regulär, was zu zeigen war.

Die Sprachen L_{ij}^k sind also für alle $i, j \in Z$ und $k \in \mathbb{N}$ regulär. Die von A erkannte Sprache ist eine Vereinigung endlich vieler dieser Sprachen. Denn es gilt $L(A) = \bigcup_{j\in F} L_{1j}^n$. Somit ist auch $L(A)$ regulär, wie behauptet. □

9 Kellerautomaten

Die Erkennung von Sprachen durch endliche Automaten ist angemessen schnell, aber in ihrem Anwendungsspektrum ziemlich eingeschränkt, weil ein endlicher Automat während des Abarbeitens eines Eingabewortes nur eine beschränkte Zahl von Informationen speichern kann, die durch die Zustände vorgegeben ist. Insbesondere kann nur bis zu einer Schranke gezählt werden, und nur beschränkte Abschnitte des Eingabewortes können für spätere Vergleiche aufgehoben werden. Um dagegen eine Sprache wie

$$L_{balance} = \{a^n b^n \mid n \in \mathbb{N}\}$$

zu erkennen, muß man unbeschränkt zählen können. Oder um das Wortproblem von

$$L_{palindrom} = \{w \in T^* \mid w = trans(w)\}$$

zu lösen, ist es nötig, die erste Hälfte des Wortes zwischenzuspeichern, damit sie mit der zweiten Hälfte des Wortes verglichen werden kann.[3]

Um die Technik des Erkennens endlicher Automaten beibehalten zu können, aber gleichzeitig auch in der Lage zu sein, mitzuzählen und sich beliebige Teile des gelesenen Wortes zu merken, werden die endlichen Automaten um einen Keller (Stapel, stack) als zweites Speichermedium erweitert. Ein Zustandsübergang wird dann auch vom obersten Kellersymbol abhängig gemacht, das dabei durch eine Sequenz von Kellersymbolen ersetzbar ist. Auf dem Keller werden in jedem Schritt also eine POP-Operation sowie eine Folge von PUSH-Operationen ausgeführt, die auch leer sein kann. Außerdem werden noch – anders als bei endlichen Automaten – Zustandsübergänge erlaubt, bei denen kein Eingabesymbol gelesen wird.

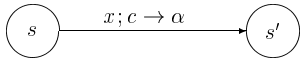
9.1 Konzept des Kellerautomaten

Das Modell des Kellerautomaten formalisiert diese Beschreibung. Die Arbeitsweise des Kellerautomaten wird ähnlich wie bei Turingmaschinen durch fortgesetzte Übergänge zwischen Konfigurationen beschrieben, wobei eine Konfiguration den aktuellen Zustand, das

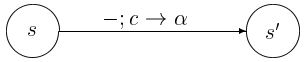
[3] Dabei wird vorausgesetzt, daß das Eingabewort nur einmal von links nach rechts gelesen werden kann.

noch zu lesende Eingabewort und ein aktuelles Kellerwort umfaßt. Einzelne Übergänge sind durch die Zustandsüberführung festgelegt. Ziel ist es, eine Anfangskonfiguration mit Anfangszustand, dem vollständigen Eingabewort und dem initialen Kellersymbol als Startkeller in eine Endkonfiguration zu überführen, bei der der aktuelle Zustand ein Endzustand und die Eingabe vollständig gelesen ist. Wenn das gelingt, ist das Eingabewort vom Kellerautomaten erkannt.

1. Ein *Kellerautomat* ist ein System $K = (Z, I, C, d, s_0, F, c_0)$ mit einer endlichen Menge Z von *Zuständen*, einem endlichen Alphabet I von *Eingaben*, einem endlichen Alphabet C von *Kellersymbolen*, einer *Zustandsüberführung* d, die jedem Zustand s, jeder Eingabe x und jedem Kellersymbol c eine Menge von Paaren aus Zuständen und Kellerwörtern $d(s, x, c) \subseteq Z \times C^*$ sowie jedem Zustand s und jedem Kellersymbol c eine entsprechende Menge $d(s, -, c) \subseteq Z \times C^*$ zuordnet, einem *Anfangszustand* $s_0 \in Z$, einer Menge von *Endzuständen* $F \subseteq Z$ und einem *initialen Kellersymbol* c_0.
2. Ein Kellerautomat $K = (Z, I, C, d, s_0, F, c_0)$ läßt sich ähnlich einen endlichen Automaten graphisch darstellen. Unterschiedlich ist lediglich die Beschriftung der Kanten:



für $(s', \alpha) \in d(s, x, c)$,



für $(s', \alpha) \in d(s, -, c)$.

3. Eine *Konfiguration* (s, v, γ) besteht aus einem Zustand $s \in Z$, einem Eingabewort $v \in I^*$ und einem Kellerwort $\gamma \in C^*$. Für $w \in I^*$ ist (s_0, w, c_0) die *Anfangskonfiguration* von w. Und (s'', λ, γ) wird *Endkonfiguration* genannt, falls $s'' \in F$ (bei beliebigem Kellerwort γ).

   Falls $(s', \alpha) \in d(s, x, c)$, so hat die Konfiguration $(s, xv, c\gamma)$ die *Folgekonfiguration* $(s', v, \alpha\gamma)$; falls $(s', \alpha) \in d(s, -, c)$, so hat die Konfiguration $(s, v, c\gamma)$ die *Folgekonfiguration* $(s', v, \alpha\gamma)$.

   Ist con' eine Folgekonfiguration von con, so schreibt man dafür auch $con \vdash con'$. Eine Folge von n solchen direkten Übergängen $con = con_0 \vdash con_1 \vdash \cdots \vdash con_n = con'$ (für $n \in \mathbb{N}$) kann durch $con \vdash^n con'$ oder $con \vdash^* con'$ abgekürzt werden.
4. Ein Wort $w \in I^*$ wird von K *erkannt*, falls die Anfangskonfiguration von w in eine Endkonfiguration überführbar ist, d.h. es existieren $s'' \in F$ und $\gamma \in C^*$ mit $(s_0, w, c_0) \vdash^* (s'', \lambda, \gamma)$.

   Die Menge aller von K erkannten Wörter bildet die *erkannte Sprache* $L(K)$.

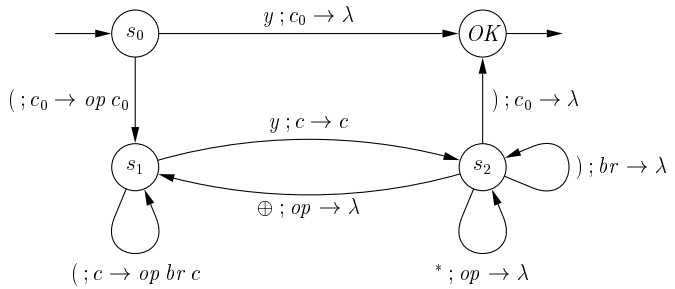
9.2 Deterministische Kellerautomaten

Bei einem Kellerautomaten kann eine Konfiguration mehrere Folgekonfigurationen haben, so daß das Erkennungsverfahren nichtdeterministisch ist. Es wird deterministisch, wenn es jeweils höchstens eine Folgekonfiguration gibt, was offenbar für folgende Kellerautomaten gilt:

Ein Kellerautomat $K = (Z, I, C, d, s_0, F, c_0)$ ist *deterministisch*, wenn für jedes $s \in Z$ und $c \in C$ die Mengen $d(s, x, c)$ für alle $x \in I$ und $d(s, -, c)$ leer oder einelementig sind und $d(s, -, c)$ höchstens dann nicht leer ist, wenn alle $d(s, x, c)$ leer sind.

9.3 Beispiel: Reguläre Ausdrücke

Reguläre Ausdrücke über I (vgl. 5. Aufgabenblatt) werden von folgendem deterministischen Automaten erkannt (wobei $y \in I \cup \{empty, lambda\}$, $c \in \{op, br, c_0\}$ und $\oplus \in \{+, \circ\}$):



Durch folgende Konfigurationsfolge wird der Ausdruck $(((x + empty)^*) \circ lambda)$ als regulär erkannt:

$$
\begin{array}{lllr}
 & (s_0, & (((x + empty)^*) \circ lambda), & c_0) \\
\vdash & (s_1, & ((x + empty)^*) \circ lambda), & op\, c_0) \\
\vdash & (s_1, & (x + empty)^*) \circ lambda), & op\, br\, op\, c_0) \\
\vdash & (s_1, & x + empty)^*) \circ lambda), & op\, br\, op\, br\, op\, c_0) \\
\vdash & (s_2, & +empty)^*) \circ lambda), & op\, br\, op\, br\, op\, c_0) \\
\vdash & (s_1, & empty)^*) \circ lambda), & br\, op\, br\, op\, c_0) \\
\vdash & (s_2, &)^*) \circ lambda), & br\, op\, br\, op\, c_0) \\
\vdash & (s_2, & ^*) \circ lambda), & op\, br\, op\, c_0) \\
\vdash & (s_2, &) \circ lambda), & br\, op\, c_0) \\
\vdash & (s_2, & \circ lambda), & op\, c_0) \\
\vdash & (s_1, & lambda), & c_0) \\
\vdash & (s_2, &), & c_0) \\
\vdash & (OK, & , &)
\end{array}
$$

Die folgende Konfigurationsfolge zeigt, daß der Ausdruck $((x + empty)^*) \circ lambda)$ nicht regulär ist:

$$
\begin{array}{lllr}
 & (s_0, & ((x + empty)^*) \circ lambda), & c_0) \\
\vdash & (s_1, & (x + empty)^*) \circ lambda), & op\, c_0) \\
\vdash & (s_1, & x + empty)^*) \circ lambda), & op\, br\, op\, c_0) \\
\vdash & (s_2, & +empty)^*) \circ lambda), & op\, br\, op\, c_0) \\
\vdash & (s_1, & empty)^*) \circ lambda), & br\, op\, c_0) \\
\vdash & (s_2, &)^*) \circ lambda), & br\, op\, c_0) \\
\vdash & (s_2, & ^*) \circ lambda), & op\, c_0) \\
\vdash & (s_2, &) \circ lambda), & c_0) \\
\vdash & (OK, & \circ lambda), &)
\end{array}
$$

Wobei die letzte Konfiguration keine Folgekonfiguration besitzt, aber auch keine Endkonfiguration ist.

10 Von kontextfreien Grammatiken zu Kellerautomaten

Kontextfreie Grammatiken haben nur Regeln, deren linke Seiten einzelne nichtterminale Zeichen sind. Da die Syntax von Programmiersprachen üblicherweise mit Hilfe solcher Regeln definiert wird, sind kontextfreie Grammatiken und die Lösung ihres Wortproblems besonders interessant. Es trifft sich deshalb gut, daß kontextfreie Grammatiken korrekt in Kellerautomaten übersetzt werden können. Dabei bedeutet Korrektheit, daß jede eingegebene Grammatik dieselbe Sprache erzeugt, wie der aus der Eingabe konstruierte Automat erkennt. Der konstruierte Automat löst also das Wortproblem der Eingabegrammatik. Da Kellerautomaten im allgemeinen nichtdeterministisch arbeiten, ist diese Lösung jedoch nicht polynomiell (wenn man den Nichtdeterminismus z.B. durch Breitensuche vermeidet). Leider läßt sich, wie das bei endlichen Automaten möglich ist, nicht jeder Kellerautomat in einen deterministischen umbauen, ohne daß sich die erkannte Sprache ändert.

10.1 Der Übersetzer

Zu jeder kontextfreien Grammatik wird ein Kellerautomat konstruiert, in dessen Keller praktisch der Ableitungsprozeß der Eingabegrammatik abläuft. Sonstige Zustandsübergänge dienen lediglich dem richtigen Starten und Halten.

Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik. Dann ist der zugehörige Kellerautomat $PDA(G)$ [4] gegeben durch:

$$PDA(G) = (\{s_0, OK\}, T, N \cup T \cup \{c_0, c_{OK}\}, d_G, s_0, \{OK\}, c_0)$$

mit

$$\begin{array}{l} d_G(s_0, -, c_0) = \{(s_0, Sc_{OK})\}, \\ d_G(s_0, -, A) = \{(s_0, u) \mid (A ::= u) \in P\}, \\ d_G(s_0, x, x) = \{(s_0, \lambda)\} \text{ für alle } x \in T, \text{und} \\ d_G(s_0, -, c_{OK}) = \{(OK, \lambda)\}. \end{array}$$

Theorem (Korrektheit der Übersetzung): $L(G) = L(PDA(G))$.

Beweis: Es werden zwei Beobachtungen über den engen Zusammenhang zwischen Ableiten in G und dem Bilden von Folgekonfigurationen in $PDA(G)$ verwendet:

(1) $S \xrightarrow{*} w$ impliziert $(s_0, uv, Sc_{OK}) \vdash^* (s_0, v, \overline{u}c_{OK})$ für beliebige $v \in T^*$ und $w \in (N \cup T^*)$, wobei $w = u\overline{u}$ gilt und u das längste terminale Anfangsstück von w ist.

(2) $(s_0, uv, Sc_{OK}) \vdash^* (s_0, v, \overline{u}c_{OK})$ impliziert $S \xrightarrow{*} u\overline{u}$ für beliebige $u, v \in T^*$ und $\overline{u} \in (N \cup T)^*$.

Beide Eigenschaften werden im nächsten Punkt gezeigt. Mit ihrer Hilfe kann dann folgendermaßen geschlossen werden.

Sei $w \in L(G)$, d.h. $S \xrightarrow{*} w$ und $w \in T^*$. Nach Beobachtung 1 existiert eine Konfigurationsfolge $(s_0, w, Sc_{OK}) \vdash^* (s_0, \lambda, \lambda c_{OK})$ für $v = \lambda$. Denn w ist das längste terminale Anfangsstück von w, so daß $\overline{u} = \lambda$ gewählt werden muß. Mit dem ersten und letzten

[4] *PDA* steht für die englische Bezeichnung von Kellerautomaten als "pushdown automata".

Zustandsübergang läßt sich diese Folgekonfigurationsbildung vorn und hinten verlängern zu:

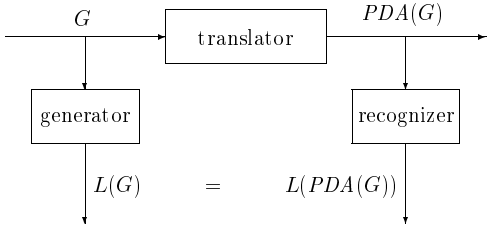
$$(*) \quad (s_0, w, c_0) \vdash (s_0, w, Sc_{OK}) \vdash^* (s_0, \lambda, c_{OK}) \vdash (OK, \lambda, \lambda).$$

Das bedeutet aber gerade $w \in L(PDA(G))$.

Sei umgekehrt $w \in L(PDA(G))$, d.h. $(s_0, w, c_0) \vdash^* (OK, \lambda, \gamma)$ für irgendein γ. Das läßt sich immer in die Form $(*)$ zerlegen, denn der erste und letzte Schritt können nicht anders aussehen. Der mittlere Teil impliziert nach Beobachtung 2 $S \xrightarrow{*} w\lambda = w$. Somit gilt $w \in L(G)$.

Insgesamt sind die beiden Sprachen also gleich. □

Die Situation der Übersetzung von kontextfreien Grammatiken in Kellerautomaten und ihre Korrektheit lassen sich durch folgendes Diagramm veranschaulichen:



Bemerkung: Der Nachweis für die beiden Eigenschaften oben wird nachgeliefert.

11 Linksableitungen

Wenn man in einer kontextfreien Grammatik beim Ableiten in jedem Schritt die jeweilige Regel möglichst weit links anwendet, also nichtterminale Zeichen links vom gerade ersetzten in der Folge nicht mehr ersetzt werden, spricht man von Linksableitungen. Ist das am Ende abgeleitete Wort terminal, wird in einer Linksableitung immer das am weitesten links stehende nichtterminale Zeichen ersetzt. Trennt man dabei die noch weiter links stehenden terminalen Zeichen vorher ab, so handelt es sich gerade um Schritte, die der aus einer kontextfreien Grammatik konstruierte Kellerautomat auf dem Keller vollführt. Es wird in diesem Abschnitt gezeigt, daß durch Linksvertauschungen jede Ableitung in eine Linksableitung umgebaut werden kann. Es stellt sich also heraus, daß das Ableiten in der Grammatik dieselbe Leistungsfähigkeit hat wie das Bilden von Folgekonfigurationen im zugehörigen Kellerautomaten.

11.1 Linksvertauschung

Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik. Dann läßt sich jede Ableitung der Form

$$w_1Aw_2Bw_3 \underset{B::=v}{\longrightarrow} w_1Aw_2vw_3 \underset{A::=u}{\longrightarrow} w_1uw_2vw_3$$

in eine Ableitung der Form

$$w_1Aw_2Bw_3 \underset{A::=u}{\longrightarrow} w_1uw_2Bw_3 \underset{B::=v}{\longrightarrow} w_1uw_2vw_3$$

umbauen (und umgekehrt).

Zwei aufeinander folgende Ableitungsschritte, bei denen das zweite ersetzte nichtterminale Zeichen nicht gerade erst erzeugt wurde, lassen sich also vertauschen, ohne daß sich die abgeleiteten Wörter ändern, sondern es entstehen höchstens zwischendurch unterschiedliche Wörter. Steht das als zweites ersetzte nichtterminale Zeichen links vom ersten, nennt man die Vertauschung der beiden Schritte *Linksvertauschung*.

11.2 Linksableitung

Eine Ableitung $w \xrightarrow{*} w'$ wird *Linksableitung* genannt und mit $w \xrightarrow[l]{*} w'$ bezeichnet, wenn keine Linksvertauschung zweier aufeinander folgender Schritte möglich ist.

Ist $w_1 A w_2 \underset{A::=u}{\longrightarrow} w_1 u w_2$ ein Ableitungsschritt aus einer Linksableitung $w \xrightarrow[l]{*} w'$, bei der w' terminal ist, so muß w_1 terminal sein; in jedem Schritt wird also das am weitesten links stehende nichtterminale Zeichen ersetzt. Andernfalls wäre eine Linksvertauschung möglich, was der Definition von Linksableitungen widerspricht.

11.3 Transformation von Ableitungen in Linksableitungen

Durch wiederholte Linksvertauschungen, bis es nicht mehr geht, läßt sich jede Ableitung in eine Linksableitung transformieren.

Lemma: Sei $w \xrightarrow{*} w'$ eine Ableitung in einer kontextfreien Grammatik $G = (N, T, P, S)$. Dann gibt es eine Linksableitung $w \xrightarrow[l]{*} w'$.

Sie kann durch eine maximale Folge von Linksvertauschungen konstruiert werden.

Insbesondere gilt: $L(G) = \{w \in T^* \mid S \xrightarrow[l]{*} w\}$.

Beweis: Sei $w = w_0 \longrightarrow w_1 \longrightarrow \cdots \longrightarrow w_n = w'$ die gegebene Ableitung. Sei i der größte Index, so daß $w_0 \longrightarrow \cdots \longrightarrow w_i$ bereits Linksableitung ist. Im Falle $i = n$ liegt bereits eine Linksableitung vor. Sei also $i < n$. In diesem Falle ist $n > 0$ und damit $i > 0$, weil der erste Schritt eine Ableitung immer eine Linksableitung ist. Dann haben der i-te und $(i+1)$-te Schritt folgende Form

$$v_i A_{i+1} v_i' A_i v_i'' \underset{A_i::=u_i}{\longrightarrow} v_i A_{i+1} v_i' u_i v_i'' \underset{A_{i+1}::=u_{i+1}}{\longrightarrow} v_i u_{i+1} v_i' u_i v_i''.$$

Denn läge A_{i+1} rechts von A_i oder läge es in u_i (wäre also gerade erst erzeugt), dann wäre $w_0 \longrightarrow \cdots \longrightarrow w_{i+1}$ eine Linksableitung im Widerspruch zur Wahl von i. Man kann also eine Linksvertauschung zwischen dem i-ten und $(i+1)$-ten Schritt vornehmen. Der neue i-te Schritt ist eventuell noch mit weiteren vorausgehenden Schritten linksvertauschbar, aber höchstens bis zum ersten. Dann bilden die ersten $i+1$ Schritte eine Linksableitung. Das einzusehen erfordert einige zusätzliche Betrachtungen, auf die hier verzichtet wird.

Insgesamt läßt sich also das Anfangsstück einer Ableitung, das bereits Linksableitung ist, immer länger machen, bis die gesamte Ableitung Linksableitung ist (vollständige Induktion über die Länge des Reststücks der Ableitung, das noch nicht zur längsten Linksableitung am Anfang gehört). □

12 Kontextfreiheitslemma

Da die linke Seite einer kontextfreien Regel nur aus einem Zeichen besteht, ist die Stelle eines Wortes, auf die diese Regel angewendet wird, in einem sehr strengen Sinn lokalisierbar: Wenn das Wort zerteilt wird, ist das ersetzte Zeichen in genau einem der Teile. Auf Ableitungen übertragen, bedeutet das, daß die einzelnen Ableitungsschritte auf Teile des Ausgangswortes verteilbar sind, wobei eine horizontale Zerlegung von Ableitungen entsteht. Diese als Kontextfreiheitslemma bezeichnete Eigenschaft ist recht offensichtlich, hat aber viele interessante Konsequenzen. Beispielsweise erhält man eine kubische Lösung des Wortproblems kontextfreier Grammatiken (siehe Kapitel 13).

Theorem (Kontextfreiheitslemma)
Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik, $u \xrightarrow[P]{n} v$ eine Ableitung der Länge n und $u = u_1 u_2 \cdots u_k$ eine Zerlegung von u in k Teilwörter. Dann gibt es k Ableitungen $u_i \xrightarrow[P]{n_i} v_i$, so daß $v = v_1 \cdots v_k$ und $n = \sum_{i=1}^{k} n_i$.

Beweis (mit Induktion über n):

IA: $n = 0$. Dann wähle $v_i = u_i$ und $u_i \xrightarrow[P]{0} u_i$.

IV: Die Behauptung gelte für n.

IS: Betrachte $u \xrightarrow[P]{n+1} v$. Der erste Schritt hat dann die Form $u = u' A u'' \underset{A::=r}{\longrightarrow} u' r u'' = \overline{u}$. Da A ein einzelnes Zeichen ist, existiert ein i_0, so daß $u_{i_0} = u'_{i_0} A u''_{i_0}$ und $u' = u_1 \cdots u_{i_0-1} u'_{i_0}$ sowie $u'' = u''_{i_0} u_{i_0+1} \cdots u_k$. Wähle nun $\overline{u}_1 = u_i$ für $i \neq i_0$ und $\overline{u}_{i_0} = u'_{i_0} r u''_{i_0}$, so daß insbesondere $u_{i_0} \underset{A::=r}{\longrightarrow} \overline{u}_{i_0}$ gilt. Außerdem gilt nach Konstruktion:

$$\overline{u} = u' r u'' = u_1 \cdots u_{i_0-1} u'_{i_0} r u''_{i_0} u_{i_0+1} \cdots u_k = \overline{u}_1 \cdots \overline{u}_k.$$

Auf die restlichen n Ableitungsschritte $\overline{u} \xrightarrow[P]{n} v$ ist nun die Induktionsvoraussetzung anwendbar, was Ableitungen $\overline{u}_i \xrightarrow[P]{n_i} v_i$ mit $v = v_1 \cdots v_k$ und $\sum_{i=1}^{k} n_i = n$ ergibt. Für i_0 kann das zu $u_{i_0} \underset{A::=r}{\longrightarrow} u_{i_0} \xrightarrow[P]{n_{i_0}} v_{i_0}$, einer Ableitung der Länge $n_{i_0} + 1$, zusammengesetzt werden. Für $i \neq i_0$ können die Ableitungen wegen $u_i = \overline{u}_i$ bleiben, wie sie sind. Die Zerlegung von v bleibt unverändert, die Summe der Ableitungslängen ergibt $n + 1$. Damit ist alles gezeigt. □

Es sei noch angemerkt, daß die Umkehrung des Kontextfreiheitslemmas für alle Chomsky-Grammatiken gilt. Ableitungen $u_i \xrightarrow[P]{n_i} v_i$ für $i = 1, \ldots, k$ können immer zu einer Ableitung $u = u_1 \cdots u_k \xrightarrow[P]{n} v_1 \cdots v_k = v$ mit $n = \sum_{i=1}^{k} n_i$ zusammengesetzt werden.

13 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*, 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 $(A ::= w) \in P$.

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 es 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 $x_1 \cdots x_n (x_l \in T$ 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 $j > 1$ und $(A ::= BC) \in P$ sowie k mit $1 \leq k < j$ existieren, 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. Seien für $i = 1, \ldots, n$ und $j = 1, \ldots, n - i + 1$

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

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

> Für $i = 1, \ldots, n$: $CELL_{i1} = \{A \in N \mid (A ::= x_i) \in P\}$;
> für $j = 2, \ldots, n$; $i = 1, \ldots, n - j + 1$: $CELL_{ij} = \bigcup_{k=1}^{j-1} \{\ A \in N \mid (A ::= BC) \in P, B \in CELL_{ik}, C \in CELL_{i+k\,j-k}\}$.

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

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

Da es für jedes $j = 1, \ldots, n$ jeweils $n - j + 1$ Zellen mit j als zweitem Index gibt, müssen insgesamt $\sum_{j=1}^{n}(n - j + 1) = \frac{n \cdot (n+1)}{2}$ Zellen berechnet werden. Für die Zellen $CELL_{i1}$ geht das

in konstanter Zeit, weil nur die gegebenen terminierenden Produktionen inspiziert werden müssen (höchstens $\sharp N \cdot \sharp T$ viele). Um eine Zelle $CELL_{ij}$ mit $j > 1$ zu bilden, muß man für $k = 1, \ldots, j - 1$ auf die Zellenpaare $CELL_{ik}$ 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 $\sharp N$ viele). Aus den $j - 1$ Zellenpaaren sind die beschränkt vielen Elementpaare zu bilden (höchstens $\sharp N^2$ viele) und mit den rechten Seiten der nichtterminalen Produktionen (höchstens $\sharp N^3$ viele) zu vergleichen. Da $j \leq n$ ist, sind für die $O(n^2)$ vielen Zellen höchstens $O(n)$ viele konstante Aktionen erforderlich, was einen Gesamtaufwand von $O(n^3)$ ergibt.

14 Ableitungsbäume

Daß das Wortproblem kontextfreier Sprachen mit Hilfe von Kellerautomaten gelöst werden kann, liegt wesentlich am Konzept der Linksableitungen. Denn sie erlauben, Ableitbarkeit und Erzeugbarkeit von Wörtern durch Operationen auf einem Keller zu realisieren. Ableitungsbäume sind ein vergleichbares Konzept zur Repräsentation kontextfreier Ableitungen. Die Konstruktion von Ableitungsbäumen beruht auf dem Kontextfreiheitslemma und gestattet, bekannte Eigenschaften von Bäumen für die Untersuchung von Ableitungen zu nutzen.

Um das zu demonstrieren, werden einige Beispiele für die vorteilhafte Nutzung der Baumstruktur gegeben. Sie alle werden im Kapitel 15 zum Nachweis des Pumping-Lemmas für kontextfreie Sprachen verwendet. In Punkt 2 wird nachgewiesen, daß man aus der Länge des Resultats eines Ableitungsbaumes dessen Höhe abschätzen kann. In Punkt 3 wird zu jedem Ableitungsbaum ein Weg von der Wurzel zu einem Blatt konstruiert, der so lang ist, wie der Baum hoch ist. In Punkt 4 schließlich werden das An- und Abhängen von Ableitungsbäumen betrachtet.

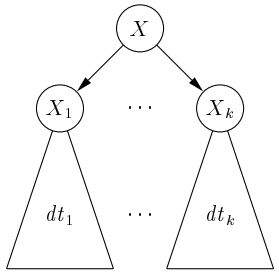
14.1 Konstruktion

Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik ohne λ-Produktionen, d.h. ohne Produktionen mit dem leeren Wort als rechter Seite.

Dann kann jeder Ableitung, die in einem einzelnen Symbol beginnt, folgendermaßen rekursiv ein *Ableitungsbaum* zugeordnet werden:

(i) Für $X \xrightarrow{0} X$ mit $X \in N \cup T$ besteht der zugeordnete *Ableitungsbaum* aus einem mit X markierten Knoten mit der *Höhe* 0 und X als *Resultat*. Der einzelne Knoten ist sowohl *Wurzel* als auch *Blatt*. Dieser Ableitungsbaum wird mit $dt(X \xrightarrow{0} X)$ bezeichnet.

(ii) Für $X \xrightarrow{n+1} U$ sei $X ::= X_1 \cdots X_k$ mit $X_i \in N \cup T$ die erste angewendete Regel, und seien $X_i \xrightarrow{n_i} U_i$ die korrespondierenden Ableitungen gemäß dem Kontextfreiheitslemma. Wegen $n_i \leq n$ kann angenommen werden, daß Ableitungsbäume $dt_i = dt(X_i \xrightarrow{n_i} U_i)$ mit einer jeweiligen Höhe, U_i als Resultat und einem Knoten X_i als Wurzel bereits existieren. Dann hat der Ableitungsbaum $dt(X \xrightarrow{n+1} U)$ die Form



mit einer Höhe, die um 1 größer als die größte Höhe der angehängten Ableitungsbäume dt_i ist, mit U als Resultat, dem obersten Knoten als *Wurzel* und den Blättern der angehängten Teilbäume als *Blätter*.

Ist dt ein Ableitungsbaum und bezeichnet man seine Höhe mit $height(dt)$ und sein Resultat mit $res(dt)$, so gilt nach Konstuktion:

(i) $height(dt(X \xrightarrow{0} X)) = 0$ und $res(dt(X \xrightarrow{0} X)) = X$,

(ii) $height(dt(X \xrightarrow{n+1} U)) = 1 + max\{height(dt_i) \mid i = 1, \ldots, k\}$
und $res(dt(X \xrightarrow{n+1} U)) = U = U_1 \cdots U_k = res(dt_1) \cdots res(dt_k)$.

14.2 Beziehung zwischen Baumhöhe und Resultatslänge

Gemäß der rekursiven Definition erhöht sich die Höhe um 1, wenn man k Bäume an eine Wurzel hängt, während sich die Länge des Resultats als Summe der Längen der Resultate der angehängten Bäume ergibt. Daraus ergibt sich folgende Beziehung zwischen Höhe und Resultatslänge.

Sei b die Länge der längsten rechten Seite von Produktionen aus P. Dann gilt für alle Ableitungsbäume dt:

$$|res(dt)| \leq b^{height(dt)}.$$

Beweis (mit Induktion über die Höhe):

IA: $height(dt) = 0$ bedeutet $dt = dt(X \xrightarrow{0} X)$ für ein geeignetes $X \in N \cup T$. Somit gilt:

$$|res(dt)| = |res(dt(X \xrightarrow{0} X))| = |X| = 1 \leq 1 = b^0 = b^{height(dt)}.$$

IV: Die Behauptung gelte für alle Höhen bis m.

IS: Betrachte nun $height(dt) = m+1$. Gemäß der rekursiven Definition korrespondiert dt zu einer Ableitung $X \xrightarrow{n+1} U$ (mit $n \geq m$), einer ersten Regelanwendung $X \longrightarrow X_1 \cdots X_k$ und induzierten Ableitungen $X_i \xrightarrow{n_i} U_i$ $(i = 1, \ldots, k)$. Bezeichne dt_i den korrespondierenden Ableitungsbaum $dt(X_i \xrightarrow{n_i} U_i)$. Wegen $height(dt) = 1 + max\{height(dt_i) \mid i = 1, \ldots, k\}$ ist dann $height(dt_i) \leq m$, so daß nach Induktionsvoraussetzung gilt:

$$|res(dt_i)| \leq b^{height(dt_i)}.$$

Daraus folgt wunschgemäß :

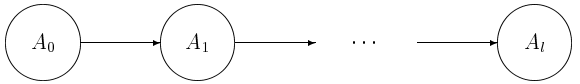
$$\begin{aligned}
|res(dt)| &= |res(dt_1) \cdots res(dt_k)| \\
&= \sum_{i=1}^{k} |res(dt_i)| \\
&\leq \sum_{i=1}^{k} b^{height(dt_i)} \\
&\leq \sum_{i=1}^{k} b^m \\
&= k \cdot b^m \\
&\leq b \cdot b^m \\
&= b^{height(dt)}.
\end{aligned}$$

□

14.3 Ein langer Weg von der Wurzel zu einem Blatt

In jedem Baum gibt es mindestens einen Weg von der Wurzel zu einem Blatt, der so lang ist, wie der Baum hoch ist.

Sei dt ein Ableitungsbaum. Dann gibt es in dt einen Weg



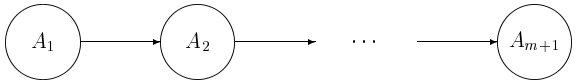
mit $l = height(dt)$, wobei der erste Knoten die Wurzel und der letzte ein Blatt ist.

Beweis (mit Induktion über die Höhe):

IA: $height(dt) = 0$ bedeutet $dt = dt(X \xrightarrow{0} X)$ für ein $X \in N \cup T$. Als Weg der Länge 0 kann (und muß) man dann den einzigen Knoten wählen, d.h. $A_0 = X$.

IV: Die Behauptung gelte für alle Ableitungsbäume der Höhe m.

IS: Betrachte einen Ableitungsbaum dt mit $height(dt) = m + 1$. Mit den Bezeichnungen aus dem Beweis von 14.2 gibt es dann ein i_0, so daß $height(dt_{i_0}) = m$ ist, sowie eine Kante von der Wurzel von dt zur Wurzel von dt_{i_0}. Nach Induktionsvoraussetzung gibt es in dt_{i_0} einen Weg



wobei der erste Knoten die Wurzel von dt_{i_0} ist, d.h. insbesondere $A_1 = X_{i_0}$, und der letzte Knoten ein Blatt von dt_{i_0} ist. Dieser Knoten ist auch ein Blatt von dt, so daß der gesuchte Weg entsteht, wenn man die eine Kante davorsetzt, wobei man $A_0 = X$ wählt. □

14.4 Anhängen und Abhängen von Ableitungsbäumen

Nach Konstruktion führt von der Wurzel eines Ableitungsbaumes genau eine Kante zu der Wurzel jedes angehängten Ableitungsbaumes, was rekursiv bedeutet, daß von der Wurzel zu jedem Knoten genau ein Weg führt und jeder Knoten Wurzel genau eines Teilbaumes ist. Wenn man den abhängt (bis auf seine Wurzel), bleibt wieder ein Ableitungsbaum übrig. Genauer gilt für $dt = dt(X \xrightarrow{*} U)$ und für einen Weg

von der Wurzel zu einem Knoten folgendes:

Es existieren Ableitungen $X \xrightarrow{*} U_1 A_l U_3$ und $A_l \xrightarrow{*} U_2$, so daß $dt_2 = dt(A_l \xrightarrow{*} U_2)$ der Teilbaum von dt ist, der A_l als Wurzel hat, und $dt_1 = dt(X \xrightarrow{*} U_1 A_l U_3)$ der Baum ist, der durch Löschen von dt_1 aus dt entsteht. Insbesondere liegt der obere Weg auch in dt_1 und führt dort zu einem Blatt.

Sind umgekehrt dt_1 und dt_2 mit dem obigen Weg in dt_1 von der Wurzel zu einem Blatt gegeben, kann man dt_2 an das Blatt A_l von dt_1 hängen, und man erhält einen Ableitungsbaum $dt_3 = dt(X \xrightarrow{*} U_1 A_l U_3 \xrightarrow{*} U_1 U_2 U_3)$. Sind dt_1 und dt_2 die beiden Bäume, die beim Abhängen aus dt entstehen, dann gilt $dt_3 = dt$ und damit insbesondere $U = U_1 U_2 U_3$.

Auf die vollständige Formalisierung dieser beiden Konstruktionen wird hier verzichtet.

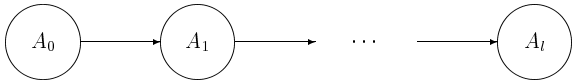
15 Pumping-Lemma für kontextfreie Sprachen

In diesem Abschnitt wird ein Pumping-Lemma für kontextfreie Sprachen formuliert und bewiesen. Analog zum Pumping-Lemma für reguläre Sprachen in Kapitel 6[5] eignet es sich zum Nachweis, daß bestimmte Sprachen nicht kontextfrei sind.

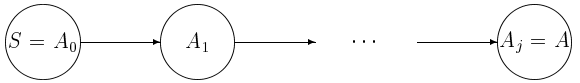
Das Pumping-Lemma für reguläre Sprachen beruht darauf, daß beim Erkennen eines genügend langen Wortes in einem endlichen Automaten eine Zustandsfolge durchlaufen wird, die einen Kreis enthält. Übertragen auf rechtslineare Grammatiken heißt das, daß es Ableitungen der Form $S \xrightarrow{*} xA, A \xrightarrow{*} yA$ und $A \xrightarrow{*} z$ mit $x, y, z \in T^*$ und $|y| > 0$ gibt, woraus sich Ableitungen der Form $S \xrightarrow{*} xy^iz$ für $i \in \mathbb{N}$ bilden lassen. Bei beliebigen kontextfreien Grammatiken kann man nicht erwarten, daß das nichtterminale Zeichen A immer ganz rechts steht, sondern irgendwo im abgeleiteten Wort. Aber auch Ableitungen der Form $S \xrightarrow{*} uAy, A \xrightarrow{*} vAx$ und $A \xrightarrow{*} w$ lassen sich zu Ableitungen $S \xrightarrow{*} uv^iwx^iy$ für $i \in \mathbb{N}$ zusammensetzen, was ebenfalls einen Pumpeffekt ergibt, bei dem allerdings zwei Teilwörter gleichzeitig iteriert werden. Es bleibt, die drei Ableitungen zu finden. Der schwierige Teil dabei ist, ein A zu finden, das in einem seiner abgeleiteten Wörter wieder auftaucht. Um das zu erreichen, werden die Ergebnisse über Ableitungsbäume herangezogen.

[5] In Kapitel 6 ist das Pumping-Lemma für die von endlichen Automaten erkannten Sprachen formuliert. Reguläre Sprachen sind genau die von endlichen Automaten erkannten; Regularität ist aber die viel griffigere Bezeichnung und wird deshalb hier verwendet.

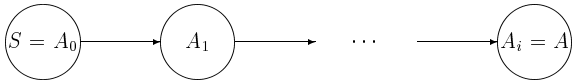
Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik ohne λ-Produktionen und Kettenregeln. Für $z \in L(G)$ gibt es dann einen Ableitungsbaum $dt = dt(S \xrightarrow{*} z)$. Dessen Höhe ist nach Punkt 14.2 größer als $\sharp N$, falls $|z| > b^{\sharp N}$, wobei b wieder die Länge der längsten rechten Seite einer Produktion von G ist. Nach Punkt 14.3 gibt es dann in dt einen Weg von der Wurzel zu einem Blatt



mit $l = height(dt)$ und $A_0 = S$. Die l Zeichen $A_0, \ldots, A_{l-1}$ sind nichtterminal, weil in einem Ableitungsbaum höchstens Blätter terminal sind. Wegen $l > \sharp N$ muß es Indizes $i < j$ geben mit $A_i = A_j$. Dieses nichtterminale Zeichen wird im folgenden auch mit A bezeichnet. Nach Punkt 14.4 induziert der Teilweg



zwei Ableitungen $S \xrightarrow{*} z_1 A z_2$ und $A \xrightarrow{*} w$, derart daß $z = z_1 w z_2$ und der obige Teilweg in $dt_1 = dt(S \xrightarrow{*} z_1 A z_2)$ liegt und dort von der Wurzel zu einem Blatt führt. Entsprechend induziert der Teilweg



zwei Ableitungen $S \xrightarrow{*} uAy$ und $A \xrightarrow{*} vAx$ mit $z_1 A z_2 = uvAxy$. Beachte dabei, daß z_1 und z_2 terminal sind, weil z terminal ist, und daß deshalb auch u und y terminal sein müssen. Das nichtterminale Zeichen A in $z_1 A z_2$ kann also nur in der zweiten Ableitung abgeleitet werden.

Damit sind die drei gesuchten Ableitungen gefunden. Daraus lassen sich induktiv Ableitungen der folgenden Form iterieren:
(i) $S \xrightarrow{*} uAy (= uv^0Ax^0y)$; und
(ii) wenn $S \xrightarrow{*} uv^iAx^iy$ bereits konstruiert ist, erhält man $S \xrightarrow{*} uv^iAx^iy \xrightarrow{*} uv^ivAxx^iy = uv^{i+1}Ax^{i+1}y$.

Schließlich läßt sich jede dieser Ableitungen terminieren: $S \xrightarrow{*} uv^iAx^iy \xrightarrow{*} uv^iwx^iy$, so daß $uv^iwx^iy \in L(G)$ für alle $i \in \mathbb{N}$ folgt.

Wegen $i < j$ ist $A \xrightarrow{*} vAx$ keine 0-Ableitung. Und da G keine Kettenregeln enthält, können v und x nicht beide leer sein; d.h. $vx \neq \lambda$.

Wählt man darüber hinaus den ursprünglichen Weg als den längst möglichen und i ebenfalls so groß wie möglich, dann muß $l - i \leq \sharp N + 1$ gelten, weil sonst zwei andere Knoten unterhalb von i gleich markiert wären, was der Wahl von i widerspricht. Außerdem kann in dem Ableitungsbaum $dt_2 = dt(A \xrightarrow{*} vAx \xrightarrow{*} vwx)$ kein Weg länger als $\sharp N + 1$ sein, weil sonst der ursprüngliche Weg nicht der längste gewesen wäre. Dann ist dt_2 aber auch nicht höher, und vwx hat höchstens die Länge $b^{\sharp N+1}$.

Die vorausgehenden Überlegungen ergeben das folgende Pumping-Lemma für kontextfreie Sprachen.

Theorem

Zu jeder kontextfreien Sprache L existieren zwei natürliche Zahlen $p, q \in \mathbb{N}$, so daß gilt: Ist $z \in L$ mit $|z| > p$, dann läßt sich z schreiben als $z = uvwxy$, wobei $|vwx| \leq q$ ist und $vx \neq \lambda$, und für alle $i \in \mathbb{N}$ gilt:

$$uv^i wx^i y \in L.$$

Beweis: Zu jeder kontextfreien Sprache L existiert eine kontextfreie Grammatik G ohne λ-Produktionen und Kettenregeln, derart daß $L(G) = L - \{\lambda\}$ (vgl. Zusatzaufgaben). Wählt man nun $p = b^{\sharp N}$ und $q = b^{\sharp N+1}$, dann bilden die Überlegungen vor dem Theorem den Rest des Beweises. Beachte, daß es keine Rolle spielt, ob $\lambda \in L$ gilt, weil ohnehin nur Wörter ab einer bestimmten Länge pumpbar sein müssen.

Folgerungen

1. Die Sprache $L_0 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ ist nicht kontextfrei.
2. Die Klasse der kontextfreien Sprachen ist nicht abgeschlossen gegenüber Durchschnitt, d.h. es gibt kontextfreie Sprachen, deren Durchschnitt nicht kontextfrei ist.

Beweis:

1. Angenommen L_0 wäre kontextfrei. Seien p und q die zu L_0 gehörenden Zahlen aus dem Pumping-Lemma. Dann gibt es für $n \in \mathbb{N}$ mit $3n > p$ eine Zerlegung $a^n b^n c^n = uvwxy$, derart daß $uv^i wx^i y \in L$ für alle $i \geq 0$, $|vwx| \leq q$ und $vx \neq \lambda$.

   Enthielte v oder x $a's$ und $b's$ oder $b's$ und $c's$, so käme in uv^2wx^2y im Widerspruch zur Definition von L_0 ein a bzw. ein b hinter einem b bzw. einem c vor. Also ist $v = a^r$(oder $v = b^r$ oder $v = c^r$) und $x = c^s$ (oder $x = b^s$ oder $x = a^s$) mit $r + s \geq 1$. Ferner gilt $u = a^j, w = a^k b^n c^l, y = c^m$ und $z = uvwxy = a^j a^r a^k b^n c^l c^s c^m$ mit $j + r + k = n = l + s + m$ (analog für die anderen fünf Fälle). Und man erhält $uv^0wx^0y = a^j \lambda a^k b^n c^1 \lambda c^m = a^{n-r} b^n c^{n-s} \in L_0$, was wegen $r + s \geq 1$ der Definition von L_0 widerspricht.
2. Die durch die Produktionen $S_1 ::= S_1 c \mid A$ und $A ::= aAb \mid \lambda$ sowie $S_2 ::= aS_2 \mid B$ und $B ::= bBc \mid \lambda$ definierten Grammatiken erzeugen die kontextfreien Sprachen $L_1 = \{a^m b^m c^n \mid m, n \in \mathbb{N}\}$ und $L_2 = \{a^m b^n c^n \mid m, n \in \mathbb{N}\}$, deren Durchschnitt gerade die nicht kontextfreie Sprache L_0 aus Punkt 1 ist.

16 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 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:

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

Offensichtlich lassen sich damit aus S die terminalen Wörter der Form

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

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

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

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

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 als unentscheidbar wie die Frage nach der Leerheit des Durchschnitts.

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.

17 Literatur

Die folgenden Literaturhinweise geben einen Einblick in die Theorie der formalen Sprachen und verschiedene andere Gebiete der Theoretischen Informatik. Die Bücher [HU69, Sal73, Sal78, HU79, MAK88, HU90] behandeln formale Sprachen ausführlich, wobei [Sal78] und [HU90] die deutschen Übersetzungen von [Sal73] und [HU79] sind. In [Woo87, AU92] werden vorrangig kontextfreie Sprachen untersucht. Die Verbindung von formalen Sprachen und der syntaktischen Behandlung von Programmiersprachen wird umfassend in [AU72] abgehandelt.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing.* Prentice-Hall, Englewood-Cliffs, New Jersey, 1972.

[AU92] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science.* Computer Science Press, New York, 1992.

[HU69] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata.* Addison-Wesley, Reading, Mass., 1969.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Compuation.* Addison-Wesley, Reading, Mass., 1979.

[HU90] John E. Hopcroft and Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Addison-Wesley (Deutschland), Bonn, 1990.

[LP81] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, Englewood Cliffs, New Jersey, 1981.

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

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

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

[Sch95] Uwe Schöning. *Theoretische Informatik – kurzgefaßt (2. Auflage).* Spektrum Akademischer Verlag, 1995.

[Weg93] Ingo Wegener. *Theoretische Informatik.* B.G. Teubner, Stuttgart, 1993.

[Woo87] Derick Wood. *Theory of Computation.* John Wiley & Sons, New York, 1987.