Theoretische Informatik 2

Prof. Dr. Hans-Jörg Kreowski

Studiengang Informatik

Wintersemester 2000/01

MZH 3260
Tel.: 2956, 3697 (Sekr.), Fax: 4322

E-Mail: kreo@informatik.uni-bremen.de
www.informatik.uni-bremen.de/theorie

Inhaltsverzeichnis

1 Formale Sprachen – Wieso? Weshalb? Warum? 1

1.1 Syntax von Programmiersprachen und Syntaxanalyse . . . 1

1.2 Wortproblem und Berechenbarkeit . . . 3

1.2.1 Wortproblem einer beliebigen Sprache . . . 3

1.2.2 Entscheidungsprobleme als Wortprobleme . . . 4

1.2.3 Berechenbare Funktionen und das Wortproblem . . . 4

1.2.4 Wortproblem selten lösbar . . . 5

2 Chomsky-Grammatiken 5

2.1 Grammatik allgemein . . . 5

2.2 Beispiele . . . 7

3 Immerhin aufzählbar 8

3.1 Aufzählbarkeit erzeugter Sprachen . . . 8

3.2 Die halbe Miete . . . 9

3.3 Erzeugbarkeit aufzählbarer Sprachen . . . 10

3.4 Unlösbarkeit des Wortproblems . . . 10

3.5 Ausschöpfende Suche in die Breite mit roher Gewalt . . . 10

4 Lösbarkeit des Wortproblems für monotone Grammatiken 11

4.1 Monotone Grammatiken . . . 11

4.2 Lösung des Wortproblems für monotone Grammatiken . . . 12

4.3 So ein Aufwand . . . 13

5 Die Chomsky-Hierarchie 13

6 Endliche Automaten 14

6.1 Definition endlicher Automaten . . . 15

6.2 Der Potenzautomat . . . 16

6.3 Pumping-Lemma für erkannte Sprachen . . . 18

7 Rechtslineare Grammatiken und endliche Automaten 19

7.1 Übersetzung endlicher Automaten in Chomsky-Grammatiken . . . 20

7.2 Übersetzung rechtslinearer Grammatiken ohne Abweichung in endliche Automaten . . . 21

7.3 Verkleinern der Abweichung . . . 22

7.4 Übersetzung rechtslinearer Grammatiken in endliche Automaten . . . 23

8 Reguläre Sprachen und reguläre Ausdrücke 24

8.1 Reguläre Operationen . . . 24

8.2 Endliche Automaten erkennen reguläre Sprachen . . . 25

8.3 Reguläre Ausdrücke . . . 26

Literatur 28

1 Formale Sprachen – Wieso? Weshalb? Warum?

Die Lehrveranstaltung *Theoretische Informatik 2* führt in die Theorie formaler Sprachen ein, die eines der ältesten und am weitesten entwickelten Gebiete der Theoretischen Informatik darstellt. Die Bedeutung dieser Theorie rührt daher, dass ihre Konzepte und Methoden die Grundlage für die Definition der Syntax von Programmiersprachen und für den Bau ihrer Compiler bilden, wobei insbesondere die Syntaxanalyse unterstützt wird (1.1). Ein zentraler Aspekt der Syntaxanalyse ist die Lösung des sogenannten Wortproblems, die sehr viel mit Berechenbarkeit zu tun hat, wie sie im vorigen Semester behandelt wurde (1.2).

1.1 Syntax von Programmiersprachen und Syntaxanalyse

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. Dies ist in Abbildung 1 skizziert.

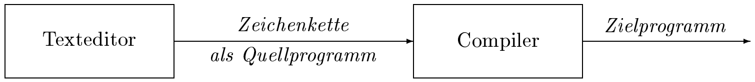

Abbildung 1: Erstellung eines Programms

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 Form 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.:

$$
\begin{array}{lll}
<\!\mathit{cesspec}\!> & \longrightarrow & <\!\mathit{name}\!>\ \text{opns: } <\!\mathit{decllist}\!>\ \text{eqns: } <\!\mathit{celist}\!> \\
 & \longrightarrow & <\!\mathit{name}\!>\ \text{opns: } <\!\mathit{decl}\!>\ \text{eqns: } <\!\mathit{celist}\!> \\
 & \longrightarrow & <\!\mathit{name}\!>\ \text{opns: } <\!\mathit{decl}\!>\ \text{eqns: } <\!\mathit{ce}\!>, <\!\mathit{celist}\!> \\
 & \longrightarrow & <\!\mathit{name}\!>\ \text{opns: } <\!\mathit{decl}\!>\ \text{eqns: } <\!\mathit{ce}\!>, <\!\mathit{ce}\!> \\
 & \longrightarrow & <\!\mathit{name}\!>\ \text{opns: } <\!\mathit{name}\!>:\ <\!\mathit{typelist}\!>\!\rightarrow\!<\!\mathit{type}\!>\ \text{eqns: } <\!\mathit{ce}\!>, <\!\mathit{ce}\!> \\
 & \vdots & \\
 & \longrightarrow & \textsf{\textbf{sort}} \\
 & & \quad \text{opns:}\quad \mathit{sort}\colon A^* \to A^* \\
 & & \quad \text{eqns:}\quad \mathit{sort}(\lambda) = \lambda, \\
 & & \qquad\qquad \mathit{sort}(xu) = \mathit{insort}(x, \mathit{sort}(u)) \text{ für } x \in A, u \in A^*
\end{array}
$$

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.

1.2 Wortproblem und Berechenbarkeit

Wie in Abschnitt 1.1 erläutert, ist die Syntaxanalyse eine zentrale Aufgabe bei der Übersetzung von Programmen einer Programmiersprache. Den Kern bildet dabei die Lösung des Wortproblems für die Menge aller syntaktisch korrekten Programme, das darin besteht, algorithmisch festzustellen, ob ein beliebiges eingegebenes Wort ein Programm ist oder nicht.

Aber das Wortproblem ist auch für andere Sprachen signifikant, weil ein enger Zusammenhang zur Berechenbarkeit besteht. Das soll im folgenden näher erläutert werden.

1.2.1 Wortproblem einer beliebigen Sprache

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

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

Die Frage ist also, ob die sogenannte *charakteristische Funktion* $\chi_L\colon A^* \to \{\mathsf{T}, \mathsf{F}\}$ von $L \subseteq A^*$, die für alle $x \in A^*$ gegeben ist durch $\chi_L(x) = \mathsf{T}$, falls $x \in L$, und $\chi_L(x) = \mathsf{F}$ sonst, berechenbar ist. Die Lösbarkeit des Wortproblems von $L \subseteq A^*$ entspricht der Berechenbarkeit der zugehörigen charakteristischen Funktion.

Als Beispiel betrachte die Menge aller Palindrome aus A^*. Das Wortproblem ist in diesem Fall gerade die Frage, ob ein gegebenes Wort ein Palindrom ist oder nicht. Dass dieser Test berechenbar ist und damit das Wortproblem lösbar, wurde in einer Aufgabe des letzten Semesters gezeigt.

1.2.2 Entscheidungsprobleme als Wortprobleme

Aber man kann sich der Situation auch umgekehrt nähern, indem man von Entscheidungsproblemen ausgeht. Ein *Entscheidungsproblem* für einen Datenbereich D (wie beispielsweise A^*, $\mathbb{N}$, $\mathbb{Z}$ oder deren kartesische Produkte) ist durch eine totale Funktion $f\colon D \to \{\mathsf{T}, \mathsf{F}\}$ beschrieben. Es ist *lösbar*, wenn f berechenbar ist. Die Funktion f kann als charakteristische Funktion der Menge aller Eingaben, die T liefern, aufgefasst werden. Die Berechnung von $f(x)$ für $x \in D$ entspricht also gerade der Frage, ob $x \in f^{-1}(\mathsf{T})$ oder $x \notin f^{-1}(\mathsf{T})$ gilt. Falls $D = A^*$, ist das das Wortproblem von $f^{-1}(\mathsf{T})$.

Aber auch wenn die Daten in D durch Wörter repräsentiert sind, d.h. $D \subseteq A^*$ für ein geeignetes Alphabet A, kann f dadurch berechnet werden, dass man das Wortproblem für $f^{-1}(\mathsf{T})$ löst. Denn es gilt für alle $x \in A^*$:

(i) $f(x) = \mathsf{T}$, falls $x \in f^{-1}(\mathsf{T})$, und
(ii) $f(x) = \mathsf{F}$, falls $x \in D$, aber $x \notin f^{-1}(\mathsf{T})$.

Offensichtlich muss im Falle des negativen Ergebnisses auch das Wortproblem von D lösbar sein.

Ein Beispiel dieser Art ist ein Primzahltest $\mathit{prim}\colon \mathbb{N} \to \{\mathsf{T}, \mathsf{F}\}$ mit $\mathit{prim}(n) = \mathsf{T}$, falls n eine Primzahl ist, und $\mathit{prim}(n) = \mathsf{F}$ sonst. Für die natürlichen Zahlen in Dezimaldarstellung gilt $\mathbb{N} \subseteq \{0, \ldots, 9\}^*$. Ob eine Ziffernfolge eine Zahldarstellung ist, lässt sich leicht ermitteln, denn jede Zahl größer 0 hat keine führenden Nullen. Deshalb liefert die Lösung des Wortproblems für die Menge aller Primzahlen einen Primzahltest und umgekehrt.

1.2.3 Berechenbare Funktionen und das Wortproblem

Ähnliches gilt auch für beliebige berechenbare Funktionen $g\colon X \to Y$. Wenn $X \subseteq A^*$ gilt, dann ist für jedes $y \in Y$ die Menge der Urbilder $g^{-1}(y) \subseteq X \subseteq A^*$ eine formale Sprache. Und die Frage, ob g für eine Eingabe $x \in X$ den Wert y liefert, wird gerade durch die Lösung des Wortproblems von $g^{-1}(y)$ beantwortet.

Ein Beispiel ist die modulo-Funktion $\bmod k\colon \mathbb{N} \to \mathbb{N}$ für $k > 0$, die für jedes $n \in \mathbb{N}$ $n \bmod k$ als Wert liefert. Statt für n den Rest beim Teilen durch k zu berechnen, kann man auch nachsehen, in welcher Restklasse $[i] = \{m \in \mathbb{N} \mid m \bmod k = i\}$ für $i = 0, \ldots, k-1$ die Zahl n liegt.

1.2.4 Wortproblem selten lösbar

Wenn das unterliegende Alphabet A nicht leer ist, gibt es unendlich viele Wörter, so dass nach einer bekannten Überlegung aus der Mathematik die Menge aller Teilmengen von A^* überabzählbar unendlich ist. Da es aber nur abzählbar viele Algorithmen gibt, müssen die Wortprobleme der meisten Sprachen unlösbar sein (vgl. Abschnitt 7.3 des Skripts *Theoretische Informatik 1* [Kre00] mit der analogen Argumentation zu berechenbaren Funktionen). Aber obwohl die Lösbarkeit des Wortproblems grundsätzlich in den seltensten Fällen gegeben ist, sind doch die meisten Sprachen, denen man üblicherweise begegnet, ziemlich gutartig. So haben alle Sprachen, die in diesem Abschnitt vorkommen (die Menge der Primzahlen, die Menge der Palindrome, die Menge der durch k mit Rest i teilbaren Zahlen), ein lösbares Wortproblem. Es ist sogar relativ schwierig, Sprachen zu konstruieren, deren Wortproblem nicht lösbar ist (siehe ein derartiges Beispiel in [HU69, Abschnitt 8.3] oder auch Abschnitt 3.4 in diesem Skript). Dem Wortproblem wird dennoch in den nächsten Abschnitten viel Aufmerksamkeit gewidmet, weil man nicht nur an Lösbarkeit interessiert ist, sondern auch an praktisch benutzbaren Lösungen. So muss ein Algorithmus, der in einen Compiler eingebaut werden soll, insbesondere auch schnell sein.

2 Chomsky-Grammatiken

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

2.1 Grammatik allgemein

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

1. Für ein gegebenes Alphabet A ist eine *Produktion* (*Regel*) ein Paar $p = (u, v) \in A^* \times A^*$, das meist als $u ::= v$ geschrieben wird. Die Zeichenkette u wird *linke Seite*, v *rechte Seite* von p genannt.

   Zur Abkürzung können mehrere Produktionen $u ::= v_1, \ldots, u ::= v_k$ $(k \geq 2)$ mit derselben linken Seite zu $u ::= v_1 \mid \cdots \mid v_k$ zusammengefasst werden.

2. Eine *Chomsky-Grammatik* ist ein System $G = (N, T, P, S)$, wobei N eine Menge *nichtterminaler Zeichen*, T eine Menge *terminaler Zeichen*, P eine endliche Menge von Produktionen und $S \in N$ ein *Startsymbol* ist.

   Soweit nichts anderes gesagt wird, nimmt man an, dass kein nichtterminales Zeichen gleichzeitig terminal ist, d.h. $N \cap T = \emptyset$, dass alle in Produktionen vorkommenden Zeichen terminal oder nichtterminal sind, d.h. $p \in (N \cup T)^* \times (N \cup T)^*$ für alle $p \in P$, und dass in jeder linken Seite mindestens ein nichtterminales Zeichen vorkommt, d.h. $u \in (N \cup T)^* N (N \cup T)^*$ für jede Produktion $u ::= v \in P$. In einzelnen Fällen wird die Forderung eines Startsymbols 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 $w_0 \xrightarrow[P]{n} w_n$. Für manche Zwecke ist es sinnvoll, auch *Nullableitungen* zuzulassen: $w \xrightarrow[P]{0} w$ für alle $w \in A^*$. Statt $w \xrightarrow[P]{n} w'$ für $n \in \mathbb{N}$ darf auch $w \xrightarrow[P]{*} w'$ 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:

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

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.

2.2 Beispiele

1. Mit der Produktion $S ::= aS$ lässt sich das Zeichen a hochzählen:

$$S \to aS \to a^2 S \to \cdots \to a^n S.$$

   Entsprechend kann man mit $S ::= a^k S$ $(k \in \mathbb{N})$ ein Vielfaches von k hochzählen. Terminieren lässt sich dieser Vorgang mit $S ::= \lambda$, so dass

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

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 $ab^nc ::= a^2b^{n+1}c^2$ für $n \geq 1$ aus der Zeichenkette abc alle Zeichenketten der Form $a^nb^nc^n$ abgeleitet werden können; aber das sind unendlich viele Regeln. Um dasselbe mit endlich vielen Regeln zu erreichen, bedarf es der bisher kompliziertesten Grammatik im nächsten Punkt. Oder geht es einfacher?

6. Folgende Produktionen erlauben, die Sprache $\{a^nb^nc^n \mid n \geq 1\}$ zu erzeugen, wenn man mit S startet und $\{a, b, c\}$ als terminales Alphabet wählt:

$$\begin{array}{cr@{\ ::=\ }l}
(1) & S & aAYZ \\
(2)\&(3) & A & aAY \mid X \\
(4) & Y & BC \\
(5) & CB & BC \\
(6)\&(7) & XB & bX \mid b \\
(8)\&(9) & CZ & Zc \mid c
\end{array}$$

   Mit (1) bis (4) erhält man $S \stackrel{*}{\longrightarrow} a^nX(BC)^nZ$.

   Mit (5) wird daraus: $a^nXB^nC^nZ$.

   Mit (6) bis (8) erhält man: $a^nb^nc^n$.

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

3 Immerhin aufzählbar

In diesem Kapitel wird der Zusammenhang zwischen Chomsky-Grammatiken und dem Konzept der Aufzählbarkeit (vgl. [Kre00, Abschnitt 7.3]) beschrieben. Der Ableitungsprozess zusammen mit einem Terminalitätstest für Zeichenketten zählt die erzeugte Sprache einer Chomsky-Grammatik auf (3.1), deren Wortproblem sich damit auch als "halb" lösbar erweist (3.2). Es wird außerdem plausibel gemacht, dass effektiv aufzählbare Mengen von Zeichenketten von Chomsky-Grammatiken erzeugt werden (3.3), allerdings ist das Wortproblem nicht immer "ganz" lösbar (3.4). In Abschnitt 3.5 schließlich wird auf Zusammenhänge zwischen dem hier angegebenen Aufzählungsverfahren und anderen bekannten Algorithmen hingewiesen.

3.1 Aufzählbarkeit erzeugter Sprachen

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

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

Es gilt offenbar $S(G)_k \subseteq S(G)_m$ für $k \leq m$. Es gilt genauer: $S(G)_0 = \{S\}$ und $S(G)_{k+1} = S(G)_k \cup \{w \mid v \underset{P}{\longrightarrow} w,\ v \in S(G)_k\}$. 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 jedes Wort aus $L(G)$ entsteht. Dieses Verfahren ist in Abbildung 2 dargestellt.

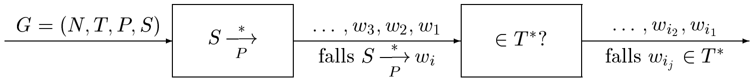

Abbildung 2: Aufzählung der von einer Grammatik erzeugten Sprache

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

3.2 Die halbe Miete

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

3.3 Erzeugbarkeit aufzählbarer Sprachen

Dass auch die Umkehrung gilt, dass also jede rekursiv aufzählbare 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 Turingmaschine aufzählt. Details zu der Verbindung zwischen Turingmaschinen und Chomsky-Grammatiken finden sich in vielen Büchern über formale Sprachen.

3.4 Unlösbarkeit des Wortproblems

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

3.5 Ausschöpfende Suche in die Breite mit roher Gewalt

Das der Aufzählung der erzeugten Sprache in Punkt 1 zugrundeliegende Verfahren, das in Abbildung 3 skizziert ist, lässt sich bei vielen Arten regelbasierter Systeme anwenden: Man beginnt mit einem Anfangszustand oder einem Eingabezustand, wendet wiederholt auf alle entstehenden Zwischenzustände alle möglichen Regeln an, wie und wo immer es geht, und filtert dann nach einem bestimmten Kriterium Ausgabe- oder Endzustände aus den erreichten und produzierten Zwischenzuständen heraus. 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.

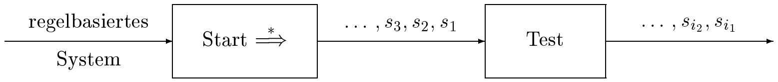

Abbildung 3: Allgemeine Funktionsweise regelbasierter Systeme

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

4 Lösbarkeit des Wortproblems für monotone Grammatiken

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.

4.1 Monotone Grammatiken

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

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

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

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

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

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

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

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

4.2 Lösung des Wortproblems für monotone Grammatiken

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

Beweis.
Sei $w_0 \in T^*$ mit $|w_0| = n$. Ohne Einschränkung kann man $n \geq 1$ annehmen, weil das leere Wort niemals von einer monotonen Grammatik erzeugt wird. Für jedes $m \geq 0$ 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:

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

Damit ist nach Definition $S_m \subseteq S_{m+1}$, 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 $w_0 \in S_k$, wobei k die Länge der Ableitungskette ist. Für $k \leq m$ gilt aber $S_k \subseteq S_m$ und für $k > m$ gilt $S_m = S_k$ nach Voraussetzung, also $w_0 \in S_m$. Die Umkehrung ist offenbar.

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

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

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

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

4.3 So ein Aufwand

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

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

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

5 Die Chomsky-Hierarchie

Wie in Abschnitt 3.4 diskutiert wurde, haben Chomsky-Grammatiken die ungünstige Eigenschaft, dass das Wortproblem für die erzeugten Sprachen im allgemeinen unentscheidbar ist. Grammatiken, die solche Sprachen erzeugen, sind z.B. für die Definition der Syntax von Programmiersprachen offenbar ungeeignet. Es stellt sich daher die Frage, ob man durch geeignete zusätzliche Bedingungen – insbesondere an die Form der Regeln – zu eingeschränkten Klassen von Chomsky-Grammatiken gelangen kann, die bessere Eigenschaften haben, aber trotzdem noch genügend Allgemeinheit besitzen. Dies führt zur sogenannten *Chomsky-Hierarchie*. Wie der Name andeutet, handelt es sich dabei um eine Hierarchie mehr oder weniger eingeschränkter Typen von Chomsky-Grammatiken. Praktisch der gesamte weitere Inhalt der Lehrveranstaltung *Theoretische Informatik 2* beschäftigt sich mit der Untersuchung dieser Grammatik-Typen bzw. mit den von ihnen definierten Sprachklassen.

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

(1) *kontext-sensitiv*, falls alle Regeln in P die Form $u_1Au_2 ::= u_1vu_2$ haben, wobei $u_1, u_2, v \in (N \cup T)^*$, $v \neq \lambda$ und $A \in N$;

(2) *kontextfrei*, falls $u \in N$ für alle Regeln $u ::= v \in P$;

(3) *rechtslinear* (auch *regulär* genannt), falls für alle Regeln $u ::= v \in P$ gilt, dass $u \in N$ und entweder $v \in T^*$ oder $v = v'B$ mit $v' \in T^+$ und $B \in N$. (Hierbei bezeichnet T^+ die Menge $T^* \setminus \{\lambda\}$.)

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

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

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

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

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

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

1. $\mathcal{L}_1 \subsetneqq \mathcal{L}_0$,
   d.h. Monotonie bzw. Kontext-Sensitivität ist eine echte Einschränkung.
2. $\{L \setminus \{\lambda\} \mid L \in \mathcal{L}_2\} \subsetneqq \mathcal{L}_1$,
   d.h. bis auf die bei monotonen Grammatiken offenbar nicht bestehende Möglichkeit, das leere Wort zu erzeugen, ist Kontextfreiheit eine echte Einschränkung gegenüber Kontext-Sensitivität.
3. $\mathcal{L}_3 \subsetneqq \mathcal{L}_2$,
   d.h. Rechtslinearität ist eine echte Einschränkung gegenüber Kontextfreiheit.

Aus der in Kapitel 4 behandelten Entscheidbarkeit des Wortproblems für Typ-1-Sprachen lässt sich die erste Aussage des obigen Satzes – $L_1 \subsetneqq L_0$ – als Folgerung ableiten. Zusammen mit der Unentscheidbarkeit des Wortproblems im allgemeinen Fall ergibt sich nämlich aus der Entscheidbarkeit für L_1 die Ungleichheit $L_1 \neq L_0$ (und $L_1 \subseteq L_0$ gilt ohnehin per Definition).

Der Nachweis, dass die beiden anderen Inklusionen echt sind, benötigt Werkzeuge, die erst später im Skript bereitgestellt werden.

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 $d\colon Z \times I \rightsquigarrow Z$, die *Zustandsüberführung* genannt wird und jedem Zustand und jeder Eingabe eine Menge von Folgezuständen zuordnet.
2. Die *Zustandsüberführung* d lässt sich rekursiv zu einer Relation $d^*\colon Z \times I^* \rightsquigarrow Z$ fortsetzen:

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

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

   (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 s zu einem Zustand s' wird eine mit $x \in I$ markierte Kante gezogen, falls $s' \in d(s, x)$. Falls es mehrere Zeichen $x_1, \ldots, x_k \in I$ gibt mit $s' \in d(s, x_i)$ für alle $i \in \{1, \ldots, n\}$, so wird der Übersichtlichkeit halber nur eine Kante gezogen, an der dann $x_1, \ldots, x_k$ steht. Der Anfangszustand wird durch einen hineingehenden Pfeil, die Endzustände werden entsprechend durch herausgehende Pfeile gekennzeichnet.
3. Ist $R\colon A \rightsquigarrow B$ 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: $R(a) = \{b \in B \mid aRb\}$.
4. Mit jeder Überführung des Zustandes eines deterministischen Automaten in den (eindeutigen) Nachfolgezustand wird ein Buchstabe des Eingabewortes abgearbeitet. Ist das Eingabewort vollständig durchlaufen, so bleibt der Automat stehen. Es sind also höchstens n Schritte erforderlich, um zu entscheiden, ob ein Eingabewort der Länge n zur erkannten Sprache gehört. Damit ist das Wortproblem für jede von einem deterministischen Automaten erkannte Sprache linear.

   Gilt das auch, wenn der erkennende Automat nichtdeterministisch ist?

Beispiel

In Abbildung 4 ist ein kleiner endlicher Automat dargestellt. Er erkennt gerade die Sprache der positiven ganzen Zahlen in Binärdarstellung ohne führende Nullen, d.h. die Menge $\{1w \mid w \in \{0,1\}^*\}$. Dieser Automat ist nichtdeterministisch; denn für die Zustandsüberführung d gilt: $d(s_0, 1) = \{s_0, s_1\}$ und $d(s_0, 0) = \emptyset$.

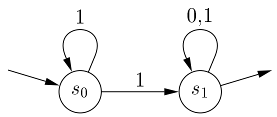

Abbildung 4: Graphische Darstellung eines endlichen Automaten

6.2 Der Potenzautomat

Offensichtlich ist Determinismus eine echte Einschränkung für endliche Automaten. Ist damit aber auch die Klasse der Sprachen, die von deterministischen Automaten erkannt werden, eine echte Teilmenge der von allgemeinen endlichen Automaten erkannten Sprachen? Das folgende Theorem verneint dies.

Theorem 4
Zu jedem endlichen Automaten A lässt sich effektiv ein deterministischer Automat $\mathcal{P}(A)$ konstruieren, der dieselbe Sprache erkennt; d.h.

$$L(A) = L(\mathcal{P}(A)).$$

Beweis.
Sei $A = (Z, I, d, s_0, F)$ ein endlicher Automat.

Daraus lässt sich der *Potenzautomat* konstruieren:

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

wobei $\mathcal{P}(Z)$ die Potenzmenge von Z ist und die Abbildung $D\colon \mathcal{P}(Z) \times I \to \mathcal{P}(Z)$ definiert ist durch $D(S, x) := \bigcup_{s \in S} d(s, x)$ für alle $S \in \mathcal{P}(Z)$ 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 $L(\mathcal{P}(A))$ folgt somit:

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

$L(A)$ und $L(\mathcal{P}(A))$ sind also gleich. □

Mit Theorem 4 folgt sofort, dass das Wortproblem für jede von einem beliebigen endlichen Automaten erkannte Sprache linear lösbar ist.

Beispiel

Der Potenzautomat zu dem endlichen Automaten aus Abbildung 4 ist in Abbildung 5 dargestellt. Offenbar kann der Zustand $\{s_1\}$ nie vom Anfangszustand $\{s_0\}$ aus erreicht werden. Er könnte also gestrichen werden, ohne die erkannte Sprache zu verändern.

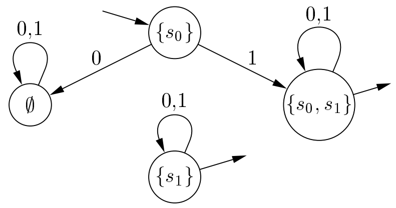

Abbildung 5: Der Potenzautomat zu dem endlichen Automaten aus Abbildung 4

6.3 Pumping-Lemma für erkannte Sprachen

Man kann einen endlichen Automaten als eine sehr eingeschränkte Turingmaschine (vgl. Kapitel 9 im Skript [Kre00]) auffassen, die in jedem Schritt ein Zeichen liest und den Kopf nach rechts bewegt; sobald das Eingabewort ganz abgearbeitet ist, hält die Maschine. Wieder stellt sich die Frage, ob eine solche Einschränkung eine Auswirkung auf die Klasse der erkannten Sprachen hat. Um eine Antwort zu finden, soll zunächst eine Eigenschaft aller von endlichen Automaten erkannten Sprachen bewiesen werden. Wenn es eine Sprache gibt, die diese Eigenschaft *nicht* hat, so gibt es demnach keinen endlichen Automaten, der sie erkennt.

Theorem 5 (Erstes Pumping-Lemma)
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 ($p = \#Z$). 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 $s_0 \cdots s_n$ 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? □

Beispiel: Anwendung des Pumping-Lemmas

1. Das Pumping-Lemma kann benutzt werden, um zu zeigen, dass eine Sprache von keinem endlichen Automaten erkannt wird.

   Betrachte die Sprache $L_{balance} = \{a^n b^n \mid n \in \mathbb{N}\}$. Angenommen, sie wird von einem endlichen Automaten erkannt. Sei dann $p \in \mathbb{N}$ die Konstante aus dem Pumping-Lemma, und wähle $w = a^p b^p \in L_{balance}$. Nach dem Pumping-Lemma kann w in drei Teilwörter $w = xyz$ mit $y \neq \lambda$ zerlegt werden, so dass $xy^iz \in L_{balance}$ für alle $i \in \mathbb{N}$. Es gibt drei Fälle, wie y in w liegen kann:

   (i) y liegt in der ersten Hälfte von w, d.h. $y = a^k$ für ein $k > 0$.
   Dies kann nicht sein, denn $xy^0z = xz = a^{p-k}b^p \notin L_{balance}$ für $k \neq 0$.

   (ii) y liegt in der Mitte von w, d.h. $y = a^k b^l$ für $k, l > 0$.
   Auch das ist nicht möglich, denn $xy^2z = a^p b^l a^k b^p \notin L_{balance}$ für $k \neq 0 \neq l$.

   (iii) y liegt in der zweiten Hälfte von w, d.h. $y = b^k$ für ein $k > 0$.
   Dies ist analog zu Fall (i) ausgeschlossen.

   Also gibt es keine Zerlegung von w mit den geforderten Eigenschaften, was bedeutet, dass die Annahme falsch gewesen sein muss.

2. Man kann das Pumping-Lemma *nicht* verwenden, um zu zeigen, dass es für eine gegebene Sprache einen erkennenden Automaten gibt.

   Betrachte die Sprache $L' = \{w \in \{a,b\}^* \mid count_a(w) = count_b(w)\}$. In jedem nichtleeren Wort $w \in L'$ gibt es eine Stelle, in der ein a auf ein b trifft. Demnach gibt es eine Zerlegung $w = xyz$ mit $y = ab$ oder $y = ba$, und weiter gilt für alle $i \in \mathbb{N}$, dass $xy^iz \in L'$. Es wäre aber falsch, daraus zu schließen, dass es einen endlichen Automaten gibt, der L' erkennt.

   Tatsächlich gibt es keinen solchen Automaten. Intuitiv liegt das daran, dass beim Abarbeiten eines Eingabewortes eine beliebig große Differenz zwischen der Anzahl der gelesenen a's und der der b's entstehen kann, aber in den endlich vielen Zuständen eines endlichen Automaten kann maximal nur eine beschränkt große Differenz gespeichert werden. Der formale Beweis muss an dieser Stelle geschuldet bleiben.

7 Rechtslineare Grammatiken und endliche Automaten

In diesem Kapitel wird ein Zusammenhang zwischen den von endlichen Automaten erkannten Sprachen und den von rechtslinearen 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 – die rechtslinearen

Grammatiken – identifizieren, deren Mitglieder in endliche Automaten übersetzt werden können, so dass sich auch ihr Wortproblem als linear lösbar erweist.

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 6
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 rechtslineare Chomsky-Grammatik konstruiert, so dass gilt: $L(A) = L(GRA(A))$.

Diese Übersetzung (einschließlich der semantischen Verträglichkeit) lässt sich mit dem Diagramm in Abbildung 6 illustrieren.

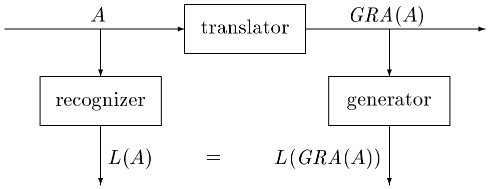

Abbildung 6: Korrekte Übersetzung endlicher Automaten in rechtslineare Grammatiken

Beweis.
Da $GRA(A)$ offensichtlich eine rechtslineare 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 7 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 $s' \in F$ mit $s' \in d^*(s_0, w)$. Nach Definition der Regeln und Lemma 7 ist das gleichbedeutend zu $s' ::= \lambda \in P_A$ und $s_0 \xrightarrow[P_A]{*} ws'$. Daraus lässt sich die Ableitung $s_0 \xrightarrow[P_A]{*} ws' \xrightarrow[s' ::= \lambda]{} w\lambda = w$ 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 Ableitungen $s_0 \xrightarrow[P_A]{*} ws'$ und $ws' \xrightarrow[s' ::= \lambda]{} w$, weil das die einzige Möglichkeit zum Terminieren ist. Damit lässt sich die gesamte Überlegung auch umdrehen. $\square$

Lemma 7
Seien A und $GRA(A)$ wie in Theorem 6.
Dann gilt für alle $s, s' \in Z$ und $w \in I^*$: $s \xrightarrow[P_A]{*} ws'$ 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 $s \xrightarrow[P_A]{*} w\bar{s}$ und $w\bar{s} \xrightarrow[\bar{s} ::= xs']{} wxs'$, weil nur Regeln der Form $\bar{s} ::= xs'$ Wörter verlängern und weil das nur am rechten Ende geschehen kann. Nach der Induktionsvoraussetzung korrespondiert die Ableitung $s \xrightarrow[P_A]{*} w\bar{s}$ zu $\bar{s} \in d^*(s, w)$. Die im Ableitungsschritt angewendete Regel korrespondiert zu $s' \in d(\bar{s}, x)$. Beides zusammen bedeutet nach Definition von d^* gerade $s' \in \bigcup_{t \in d^*(s,w)} d(t, x) = d^*(s, wx)$. □

7.2 Übersetzung rechtslinearer Grammatiken ohne Abweichung in endliche Automaten

Eine rechtslineare Grammatik muss 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:

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

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

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 Abschnitt 7.1, woraus unmittelbar ihre Korrektheit folgt.

Theorem 8
Sei $G = (N, T, P, S)$ eine rechtslineare Grammatik mit $\mathit{diff}(G) = 0$. 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 6 folgt dann: $L(AUT(G)) = L(GRA(AUT(G)) = L(G)$. □

7.3 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 9
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*

   $$G_p = (N \cup \{X\},\ T,\ (P \setminus \{p\}) \cup \{A ::= uX,\ X ::= xB\},\ S)$$

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

   $$G_p = (N \cup \{X\},\ T,\ (P \setminus \{p\}) \cup \{A ::= uX,\ X ::= \lambda)\},\ S)$$

   eine rechtslineare Grammatik mit $L(G) = L(G_p)$ und $\mathit{diff}(G) = \mathit{diff}(G_p) + 1$ konstruiert.

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 $S \xrightarrow[P]{*} vA \xrightarrow[p]{} vuxB \xrightarrow[P]{*} w$. Die Anwendung von p kann ersetzt werden durch die Anwendung der beiden neuen Produktionen, was $S \xrightarrow[P]{*} vA \xrightarrow[A ::= uX]{} vuX \xrightarrow[X ::= xB]{} vuxB \xrightarrow[P]{*} w$ 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 $S \xrightarrow[P']{*} vA \xrightarrow[A ::= uX]{} vuX \xrightarrow[X ::= xB]{} vuxB \xrightarrow[P']{*} w$. Die zwei Schritte in der Mitte lassen sich durch $vA \xrightarrow[p]{} vuxB$ 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. $\square$

7.4 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 Abschnitt 7.2 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.

Mit den Theoremen 5, 8 und 9 lässt sich außerdem die dritte Aussage in Theorem 3, d.h. $\mathcal{L}_3 \subsetneqq \mathcal{L}_2$, beweisen. Die Sprache $L_{balance} = \{a^n b^n \mid n \in \mathbb{N}\}$ wird nämlich von der kontextfreien Grammatik $G_{balance} = (\{S\}, \{a, b\}, \{S ::= aSb \mid \lambda\}, S)$ erzeugt, so dass $L_{balance} \in \mathcal{L}_2$ gilt. Zugleich ist aber $L_{balance} \notin \mathcal{L}_3$: Würde $L_{balance}$ von einer rechtslinearen Grammatik erzeugt, so auch von einer rechtslinearen Grammatik mit Abweichung 0 (Theorem 9), und damit gäbe es einen endlichen Automaten, der $L_{balance}$ erkennt (Theorem 8). In Beispiel 1

zur Anwendung des Pumping-Lemmas für erkannte Sprachen (Theorem 5) wurde aber gerade gezeigt, dass es einen solchen Automaten nicht geben kann. Also gibt es auch keine rechtslineare Grammatik, die $L_{balance}$ erzeugt. Damit ist $L_{balance} \in \mathcal{L}_2 \setminus \mathcal{L}_3$.

8 Reguläre Sprachen und reguläre Ausdrücke

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 in Abbildung 7 gezeigten drei endlichen Automaten erkennen (a) die leere Menge $\emptyset$, (b) die einelementige Sprache $\{\lambda\}$, die das leere Wort enthält, bzw. (c) die einelementige Sprache $\{x\}$, die das Wort x der Länge 1 enthält. Diese Sprachen dienen als kleinste, unzerlegbare Sprachmoduln.

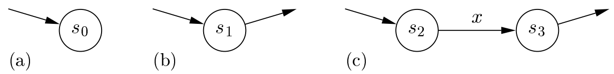

Abbildung 7: Endliche Automaten für atomare Sprachen: (a) $\emptyset$ (b) $\{\lambda\}$ (c) $\{x\}$ mit $x \in I$

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

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

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 10
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 ist für $i, j \in Z$ und $k \in \mathbb{N}$ die Sprache $L^k_{i,j}$ aller Wörter, die im Zustandsgraphen von i nach j führen, ohne zwischendurch Zustände jenseits von k zu besuchen, definiert als:

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

Mit Induktion über k kann gezeigt werden, dass $L^k_{i,j}$ für alle $i, j \in Z$ und $k \in \mathbb{N}$ regulär ist.

IA: Für $k = 0$ und $i \neq j$ enthält $L^k_{i,j}$ die Eingaben, die direkt von i nach j führen, weil alle Zustände jenseits von 0 liegen und deshalb nicht besucht werden dürfen. D.h. $L^0_{i,j} = \{x \in I \mid j \in d(i, x)\}$. 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. $L^0_{i,i} = \{\lambda\} \cup \{x \in I \mid i \in d(i, x)\}$, was sich analog als regulär erweist, weil $\{\lambda\}$ regulär ist.

IV: Als Induktionsvoraussetzung wird von der Regularität von $L^k_{i,j}$ für alle i und j ausgegangen.

IS: Betrachte nun im Induktionsschluss $L^{k+1}_{i,j}$ bzw. ein Element w daraus. Dieses Wort induziert einen Weg von i nach j im Zustandsgraph. Sei $l_0 \cdots l_p$ die durchlaufene Sequenz von Zuständen mit $i = l_0$ und $j = l_p$. Kommt der Zustand $k+1$ gar nicht in $l_1 \cdots l_{p-1}$ vor, so liegt w in $L^k_{i,j}$. Ansonsten kommt der Zustand $k+1$ m-mal darin vor mit $0 < m < p$, so dass die Sequenz folgende Form hat:

$$l_0 \cdots l_{p_1-1}(k+1)l_{p_1+1} \cdots l_{p_2-1}(k+1) \cdots (k+1)l_{p_m+1} \cdots l_p,$$

wobei $p_1 < p_2 < \cdots < p_m$ die Stellen sind, wo $k+1$ auftritt. Insbesondere kommt in den Abschnitten $l_1 \cdots l_{p_1-1}, l_{p_1+1} \cdots l_{p_2-1}, \ldots, l_{p_m+1} \cdots l_{p-1}$ der Zustand $k+1$ nicht vor, sondern nur die Zustände $1, \ldots, k$. Es stellt sich also heraus, dass $w = w_0 w_1 \ldots w_m$ für Wörter $w_0 \in L^k_{i,k+1}$, $w_1, \ldots, w_{m-1} \in L^k_{k+1,k+1}$ und $w_m \in L^k_{k+1,j}$, d.h. $w \in L^k_{i,k+1}(L^k_{k+1,k+1})^* L^k_{k+1,j}$. Damit ist gezeigt, dass

$$L^{k+1}_{i,j} \subseteq L^k_{i,j} \cup L^k_{i,k+1}(L^k_{k+1,k+1})^* L^k_{k+1,j}.$$

Nach Definition von $L^k_{i,j}$ gilt auch die umgekehrte Inklusion, so dass insgesamt

$$L^{k+1}_{i,j} = L^k_{i,j} \cup L^k_{i,k+1}(L^k_{k+1,k+1})^* L^k_{k+1,j}$$

folgt. Nach Induktionsvoraussetzung sind $L^k_{i,j}$, $L^k_{i,k+1}$, $L^k_{k+1,k+1}$ und $L^k_{k+1,j}$ 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 $L^{k+1}_{i,j}$ regulär, was zu zeigen war.

Die Sprachen $L^k_{i,j}$ 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 $L(A) = \bigcup_{j \in F} L^n_{1,j}$.
Somit ist auch $L(A)$ regulär, wie behauptet. □

8.3 Reguläre Ausdrücke

Reguläre Ausdrücke sind, analog zu arithmetischen Ausdrücken, aus Konstanten, Operationen und Klammern aufgebaute Zeichenketten, die einen Wert beschreiben. Der Wert soll in diesem Fall allerdings keine Zahl sein, sondern eine Sprache. Demzufolge müssen die Konstanten möglichst einfache Sprachen repräsentieren und die Operationen müssen es erlauben, aus diesen Sprachen komplexere zu bilden.

In der einen oder anderen Form dürften reguläre Ausdrücke den meisten schon einmal über den Weg gelaufen sein, da sie in Editoren mit komfortablen Suchfunktionen und in UNIX-Kommandos wie z.B. grep, find und sed Verwendung finden.

Sei I ein Alphabet mit $\mathit{lambda}, \mathit{empty}, +, \circ, *, (,) \notin I$. Die Menge $\mathit{REX}(I)$ der *regulären Ausdrücke über* I ist rekursiv definiert durch:

(i) $\mathit{empty}, \mathit{lambda} \in \mathit{REX}(I)$,
(ii) $I \subseteq \mathit{REX}(I)$ und
(iii) für alle $r, r_1, r_2 \in \mathit{REX}(I)$ sind auch die Wörter $(r_1 + r_2)$, $(r_1 \circ r_2)$ und (r^*) in $\mathit{REX}(I)$.

Jedem regulären Ausdruck r über I wird wie folgt eine Sprache $L(r)$ zugeordnet:

(i) $L(\mathit{empty}) = \emptyset$, $L(\mathit{lambda}) = \{\lambda\}$ und $L(x) = \{x\}$ für alle $x \in I$,
(ii) für alle $r, r_1, r_2 \in \mathit{REX}(I)$ sei
  (1) $L((r_1 + r_2)) = L(r_1) \cup L(r_2)$,
  (2) $L((r_1 \circ r_2)) = L(r_1)L(r_2) = \{w_1 w_2 \mid w_1 \in L(r_1),\ w_2 \in L(r_2)\}$ und
  (3) $L((r^*)) = L(r)^* = \{w_1 \cdots w_n \mid w_1, \ldots, w_n \in L(r),\ n \in \mathbb{N}\}$.

Bemerkungen

1. Nach der Interpretation regulärer Ausdrücke steht $+$ für die Vereinigung zweier Sprachen, $\circ$ für deren Konkatenation und $*$ für die Iteration einer Sprache. Die

Verwendung der Symbole + und $\circ$ für Vereinigung und Konkatenation hat einen guten Grund: Abstrakt gesehen, verhalten sich diese Operationen wie Addition und Multiplikation (genauer gesagt, die Menge der Wörter bildet mit diesen Operationen einen Semiring). Dabei ist $\emptyset$ das neutrale Element der Addition (die "Null") und $\{\lambda\}$ ist das der Multiplikation (die "Eins"). Wie gewohnt ergibt die Multiplikation mit Null immer Null: $\emptyset L = \emptyset = L\emptyset$. Beide Operationen sind offenbar assoziativ, aber nur die Vereinigung ist auch kommutativ, denn L_1L_2 ist natürlich im allgemeinen nicht dasselbe wie L_2L_1. Wie man sich leicht überzeugen kann, gelten auch die Distributivgesetze: $L(L_1 \cup L_2) = LL_1 \cup LL_2$ und $(L_1 \cup L_2)L = L_1L \cup L_2L$. Damit sind alle Zutaten beisammen, die man für einen Semiring benötigt (zu einem "echten" Ring fehlen die inversen Elemente bezüglich der Addition). Übrigens hat auch der Kleene-Stern einige interessante Eigenschaften. Insbesondere ist er idempotent, $L^{**} = L^*$, was praktisch direkt aus der Definition folgt.

2. Um reguläre Ausdrücke übersichtlicher zu machen, wird üblicherweise von der Konvention Gebrauch gemacht, dass $*$ stärker bindet als $\circ$ und dies wiederum stärker als +. Klammern, die nach dieser Konvention (oder der Assoziativität von + und $\circ$) zur Vermeidung von Mehrdeutigkeiten unnötig sind, können weggelassen werden. Außerdem lässt man das Symbol $\circ$ oft weg, analog zur Schreibweise bei arithmetischen Ausdrücken, wo man ja auch oft xy für $x \cdot y$ schreibt. Demnach kann man beispielsweise statt $(((a \circ b)^*) \circ (c \circ (c^*)))$ auch kurz $(ab)^*cc^*$ schreiben.
3. Gelegentlich wird in der Literatur ein regulärer Ausdruck r angegeben, wenn eigentlich $L(r)$ gemeint ist, sofern dies aus dem Kontext genügend klar hervorgeht. Dies sollte aber nicht darüber hinwegtäuschen, dass r und $L(r)$ zwei grundverschiedene Dinge sind: r ist eine Zeichenkette, während $L(r)$ eine Sprache ist.

Aus der Definition der regulären Ausdrücke und ihrer Interpretation sowie der Definition der regulären Sprachen folgt unmittelbar, dass eine Sprache $L \subseteq I^*$ genau dann regulär ist, wenn es einen regulären Ausdruck $r \in REX(I)$ mit $L = L(r)$ gibt. Reguläre Ausdrücke stellen somit den dritten Formalismus neben rechtslinearen Grammatiken und endlichen Automaten dar, um reguläre Sprachen zu spezifizieren. Formal:

$$\mathcal{L}_{REG(I)} = \mathcal{L}_{3(I)} = \mathcal{L}_{AUT(I)} = \mathcal{L}_{REX(I)},$$

wobei $\mathcal{L}_{REX(I)} = \{L(r) \mid r \in REX(I)\}$ alle durch Interpretation der regulären Ausdrücke über I erhaltenen Sprachen und $\mathcal{L}_{3(I)}$ alle Typ-3-Sprachen über I enthält.

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, EP00] behandeln formale Sprachen ausführlich, wobei [Sal78] und [HU90] die deutschen Übersetzungen von [Sal73] und [HU79] sind. Wer noch weitergehende Informationen sucht, wird vermutlich im dreibändigen *Handbook of Formal Languages* [RS97] fündig. 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 und Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing.* Prentice-Hall, Englewood-Cliffs, New Jersey, 1972.

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

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

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

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

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

[Kre00] Hans-Jörg Kreowski. Theoretische Informatik 1. Universität Bremen, 2000. Skript zur Lehrveranstaltung.

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

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

[RS97] Grzegorz Rozenberg und Arto K. Salomaa (Hrsg.). *Handbook of Formal Languages.* Springer, 1997. Vol. 1: Word, Language, Grammar. Vol. 2: Linear Modeling. Vol. 3: Beyond Words.

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

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