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?

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.

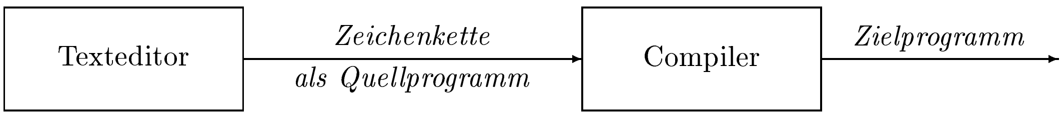

Abbildung 1: Erstellung eines Programms

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.:

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

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 $w_0 \overset{n}{\underset{P}{\longrightarrow}} w_n$. 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.

2.2 Beispiele

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

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

3. Auch das getrennte Zählen zweier (bzw. mehrerer) Größen ist kein Problem. Sei $T_l = \{a_1, \dots, a_l\}$ ($l \geq 1$), $N_l = \{S\} \cup \{A_1, \dots, A_l\}$ und $P_l = \{S ::= A_1 \cdots A_l\} \cup \{A_i ::= a_iA_i \mid i = 1, \dots, l\} \cup \{A_i ::= \lambda \mid i = 1, \dots, l\}$.
Dann gilt:
$$L((N_l, T_l, P_l, S)) = \{a_1^{n_1} \cdots a_l^{n_l} \mid n_i \geq 0,\ i = 1, \dots, l\}.$$

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 $\{a^n b^n c^n \mid n \geq 1\}$ 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 $S \xrightarrow{*} a^n X(BC)^n Z$.

Mit (5) wird daraus: $a^n X B^n C^n Z$.

Mit (6) bis (8) erhält man: $a^n b^n c^n$.

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

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

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

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

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

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

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

Bildet man also die Mengen $S(G)_k$ für wachsende k, beginnend mit $S(G)_0$, und filtert die terminalen Wörter heraus, so erhält man einen algorithmischen Vorgang, bei dem nach und nach jedes Wort aus $L(G)$ entsteht. Dieses Verfahren ist in Abbildung 2 dargestellt.

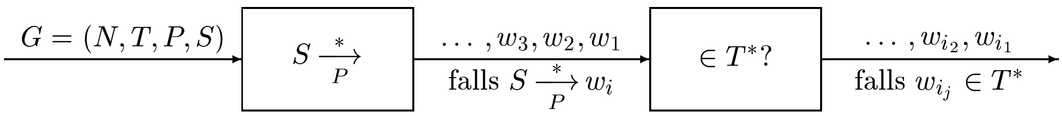

Abbildung 2: Aufzählung der von einer Grammatik erzeugten Sprache

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.

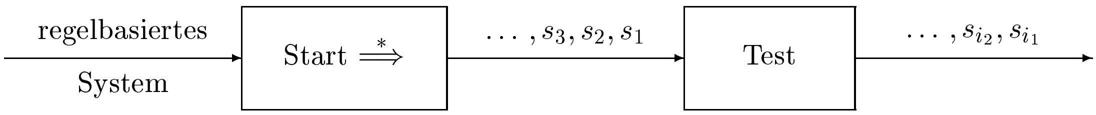

Abbildung 3: Allgemeine Funktionsweise regelbasierter Systeme

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 Gammatiken 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 in [MS97, Theorem 2.1] 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.

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.

[MS97] Alexandru Mateescu und Arto Salomaa. Aspects of classical language theory. In [RS97], Vol. 1, S. 175–251. Springer, 1997.

[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.