```
        z = getch();
        printf("%c %d\n", z, state);
        switch( state ) {
        case 0:
            if( z == '+' ) state = 1;
            else if( z == '-' ) state = 2;
            else if( isdigit( z ) && z != '0' ) state = 3;
            else state = -1;
            break;
        case 1:
            if( isdigit( z ) && z != '0' ) state = 3;
            else state = -1;
            break;
        case 2:
            if( isdigit( z ) && z != '0' ) state = 3;
            else state = -1;
            break;

        case 3:
            if( ! isdigit( z )  ) state = -1;
            break;
        }

        printf("new state: %d\n", state);
        if( state == -1 ) {
            printf("Fehler\n");
            return 0;
        }

    }
}
```

14.3 Grammatiken

Ausgangspunkt unserer Betrachtungen ist ein Vorrat von Buchstaben. Für die Menge der Buchstaben führen wir die Bezeichnung Σ gemäß

$$\Sigma = \{a, b, c, \ldots, n\} \tag{14.1}$$

ein. Aus diesen Buchstaben werden Wörter gebildet. Ein einfaches Beispiel ist die Menge mit nur zwei Elementen

$$\Sigma_d = \{0, 1\} \tag{14.2}$$

Indem man die beiden Buchstaben aneinander reiht, lassen sich mit diesem Vokabular alle möglichen Binärzahlen darstellen:

$$\begin{array}{l} 0 \\ 1 \\ 01 \\ 10 \\ 001 \\ \ldots \end{array} \tag{14.3}$$

Alle diese Folgen oder Wörter zusammen bilden wiederum eine Menge, die als Σ^+ bezeichnet wird. Nimmt man noch die leere Folge ε hinzu, so erhält man die Menge Σ^* (Kleene-Stern-Produkt, benannt nach Stephen C. Kleene, 1909-1998). Diese unendlich große Menge enthält alle möglichen Folgen. Jede Einschränkung der Abfolge wählt eine Teilmenge von Σ^* aus. Ein Beispiel ist die Menge B der Bytes, das heißt alle Folgen von 8 Symbolen aus unserer Menge Σ_d:

$$B = \left\{ \begin{array}{l} 00000000 \\ 10000000 \\ \ldots \end{array} \right\} \tag{14.4}$$

Eine solche Menge von Wörtern definiert eine Sprache. Weitere Beispiele sind die Bytes mit gerader Parität oder die Sprache

$$B_n = \{0^n\ 1^n\} \tag{14.5}$$

mit jeweils zwei gleich langen Blöcken von Nullen und Einsen. Die Schreibweise a^n ist ein Abkürzung für n aufeinander folgende Buchstaben a.

In den allermeisten Fällen ist die vollständige Aufzählung aller Wörter einer Sprache nicht praktikabel. Gesucht sind daher andere Möglichkeiten, die Sprache zu beschreiben. Noam Chomsky (amerikanischer Linguist, geb. 1928) entwickelte das Konzept der generativen Grammatiken. Dabei beschreibt für eine Sprache ein Regelwerk, wie alle möglichen Wörter konstruiert werden können. Die verschiedenen Typen von Grammatiken wurden von ihm nach der Komplexität der Regeln klassifizieren.

Eine Grammatik ist eine gut geeignete Darstellungsform, um Folgen zu generieren. Für die umgekehrte Fragestellung, ob eine gegebene Folge von der Grammatik abgedeckt ist, ist diese Darstellung weniger nutzbar. Eine solche Prüfung lässt sich sehr viel einfacher mit Automaten durchführen. Es erweist sich, dass für die von Chomsky eingeführte Hierarchie von Grammatiken eine Äquivalenz zu einer Hierarchie von Automaten besteht. Für jede Grammatik lässt sich ein entsprechender Automat konstruieren, der nur die von dieser Grammatik erzeugten Folgen akzeptiert. Welcher Typ von Automat dazu im Einzelfall notwendig ist, hängt von der Einordnung der Grammatik in die Hierarchie ab.

Tabelle 14.3: Einfache Grammatik

Σ	$+, -, 0, 1$
N	$W\ B\ Z\ V$
P	$W \rightarrow B$
	$W \rightarrow V\ B$
	$B \rightarrow Z$
	$B \rightarrow Z\ B$
	$Z \rightarrow 0$
	$Z \rightarrow 1$
	$V \rightarrow -$
	$V \rightarrow +$
S	W

14.3.1 Produktionsregeln und Ableitungsbäume

Eine Grammatik enthält eine Reihe von Produktionsregeln. Die Regeln haben die Form

$$l \rightarrow r \tag{14.6}$$

mit der Bedeutung „der Ausdruck l auf der linken Seite wird durch r ersetzt“. Eine solche Regel ist

$$negZahl \rightarrow - \ Betrag$$

d.h. eine negative Zahl wird durch das Vorzeichen $-$ und den Betrag gebildet. In diesem Fall ist $-$ ein Symbol aus der Eingangsmenge. Da bei einem solchen Symbol keine weitere Regel mehr anzuwenden ist, wird es als Terminalsymbol bezeichnet. Das Symbole *Betrag* dagegen ist eine Kategorie, die durch Anwendung von Produktionsregeln weiter aufgelöst wird. Daher wird dieser Typ von Symbolen als Nichtterminalsymbolen oder einfach Nichtterminals bezeichnet. Die Nichtterminals und die Regeln können wieder als Mengen betrachtet werden. Dann ist eine Grammatik G bestimmt durch die drei Mengen Σ, N und P für die Terminals, Nichtterminals und Produktionsregeln. Zusätzlich benötigt man noch einen Startpunkt S für die Anwendung der Regeln. Zusammenfassend gilt dann

$$G = (\Sigma, N, P, S) \tag{14.7}$$

Mit $L(G)$ bezeichnet man dann die durch die Grammatik definierte Sprache. Betrachten wir das einfache Beispiel in Tabelle 14.3. Die Grammatik beschreibt, wie aus den Symbolen $\{+, -, 0, 1\}$ positive und negative ganze Binärzahlen gebildet werden. Ausgangspunkt für die Produktionsregeln ist das Nichtterminalsymbol W (Wert). Darauf können zwei Regeln angewandt werden:

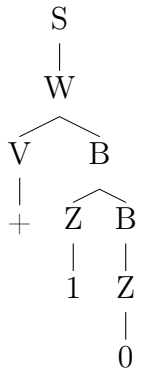
$$W \rightarrow VB$$

und

$$W \rightarrow B$$

Der Wert besteht entweder aus Vorzeichen plus Betrag oder nur einem Betrag. Wählen wir die erste Alternative. Auch für V stehen zwei Regeln zur Verfügung. Das Vorzeichen kann + oder − sein. Der Betrag wird als Folge von Ziffern gebildet. Durch wiederholte Anwendung der Regel $B \rightarrow Z\ B$ können beliebig lange Wörter erzeugt werden.

Der Ableitungsprozess endet, wenn der Ausdruck nur noch Terminals enthält. Jedes so generierte Wort gehört zur Grammatik. Im allgemeinen ist nicht garantiert, dass jede Ableitung zu einem legalen Ergebnis führt. Es können bei der Ableitung Nichtterminals übrig bleiben, für die keine passenden Regeln spezifiziert sind. Der Ableitungsprozess lässt sich als Baum darstellen. Das Startsymbol bildet die Wurzel. Die Nichtterminals sind die Knoten und die Terminals schließlich sind die Blätter. Als Beispiel erhält man aus der Folge +10 folgenden Baum:



Die Definition 14.7 gilt für alle Grammatiken der Chomsky Hierarchie. Die einzelnen Typen unterscheiden sich in den Einschränkungen für die Produktionsregeln.

Übung 14.5 *Radio-Grammatik*

1. *Geben Sie 3 Sätze an, die mit der folgenden regulären Grammatik generiert werden können.*

2. *Zeichnen Sie für einen der Beispielssätze den Ableitungsbaum.*

3. *Erweitern Sie die Grammatik so, dass optional Sätze auch mit dem Wort* Bitte *eingeleitet werden können.*

4. *Wie sieht der zugehörige endliche Automat aus?*

Σ	*Radio*, *CD*, *ein*, *aus*, *nächstes*, *Lied*, *Stück*
N	BEFEHL, SCHALTEN, LIED, GERÄT
P	BEFEHL → GERÄT BEFEHL → GERÄT SCHALTEN BEFEHL → *nächstes* LIED GERÄT → *CD* GERÄT → *Radio* LIED → *Lied* LIED → *Stück* SCHALTEN → *ein* SCHALTEN → *aus*
S	BEFEHL

14.3.2 Einseitig lineare Grammatiken

Die einfachsten Grammatiken - Typ 3 - sind dadurch gekennzeichnet, dass die Ableitungen nur in eine Richtung erfolgen. Alle Regeln haben bei einer rechtslinearen Grammatik die Form

$$\begin{aligned} A &\rightarrow w \\ A &\rightarrow wB \end{aligned} \tag{14.8}$$

mit $A, B \in N$ und $w \in \Sigma^*$. In jeder Regel steht auf der linken Seite ein Nichtterminal und auf der rechten Seite entweder ein Ausdruck mit Terminals alleine oder ein solcher Ausdruck gefolgt von einem Nichtterminal. Zu jeder derartigen Grammatik existiert ein endlicher Automat, der die mit der Grammatik verträglichen Folgen akzeptiert. Die möglichen Folgen sind damit auch durch reguläre Ausdrücke beschrieben und man spricht daher auch von regulären Sprachen. Äquivalent zu den rechts-linearen Grammatiken sind die links-linearen Grammatiken, bei denen die zweite Grundform der Regeln $A \rightarrow Bw$ ist.

14.3.3 Möglichkeiten und Grenzen endlicher Automaten und regulärer Sprachen

Reale Rechner kann man als endliche Automaten interpretieren. Zu einem Zeitpunkt befindet sich der Rechner in einem bestimmten Zustand, charakterisiert durch die Inhalte in all seinen Speicherzellen und Registern. Indem der nächste Befehl ausgeführt wird, ändern sich einige Inhalte. Die neuen Inhalte bestimmen wiederum den Folgezustand.

Übung 14.6 *Betrachten Sie einen Rechner mit 256 MByte Speicher. Dieser Rechner soll als endlicher Automat modelliert werden. Wie viele Zustände werden dazu benötigt?*

Von großer Bedeutung ist das Wortproblem:

- Kann man für eine gegebene Symbolfolge entscheiden, ob sie ein Wort der regulären Grammatik ist? Formal ausgedrückt: gilt für eine Folge $w \in \Sigma^*$ und eine Grammatik G $w \in L(G)$ oder $w \notin L(G)$?

Anwendungen dazu sind:

- Erfüllt ein Text die Suchabfrage?
- Ist ein Programm syntaktisch korrekt?

Diese Frage ist entscheidend für die praktische Verwendbarkeit. Eine Sprache, bei der nicht entscheidbar ist, ob eine Symbolfolge dazu gehört oder nicht, ist als Programmiersprache ungeeignet, da man keinen Compiler dazu konstruieren kann. Glücklicherweise ist bei regulären Grammatiken das Wortproblem entscheidbar. Es kann in der Tat für jede beliebige Symbolfolge entschieden werden, ob sie zu entsprechenden Sprache $L(G)$ gehört oder nicht. Darüber hinaus lässt sich auch eine Aussage über den dazu notwendigen Rechenaufwand treffen.

Typische Aussagen über die Komplexität haben die Form $O(f(N))$, wobei N die Anzahl der zu verarbeitenden Werte bezeichnet. Der Ausdruck $O(f(N))$ besagt, dass die Komplexität proportional ist zu einer Funktion $f(N)$. Mit anderen Worten, es gibt zwei Konstanten a und b, so dass für die Anzahl der Rechenschritte die Formel

$$a \cdot f(N) + b \tag{14.9}$$

gilt. Bei einem Algorithmus mit $O(N)$ etwa steigt der Aufwand proportional zu Anzahl der Daten. Demgegenüber würde bei einem Algorithmus mit $O(N^2)$ der Aufwand quadratisch ansteigen.

Das Wortproblem bei regulären Grammatiken hat die Komplexität $O(N)$, wobei N die Länge der Symbolfolge bezeichnet. Mit wachsender Länge nimmt der Aufwand proportional zu. Dies ist ein sehr erfreuliches Ergebnis. Probleme mit $O(N)$ sind gut lösbar. Selbst bei langen Eingangsfolgen – beispielsweise ein Programm mit vielen tausend Zeilen – ist der Compiler in überschaubarer Zeit fertig.

Begrenzt wird die Darstellungsmöglichkeit der endlichen Automaten durch das fehlende Gedächtnis. Ein endlicher Automat verfügt über keinen expliziten Speicher. Die Information liegt lediglich im aktuellen Zustand. In Folge dieser Beschränkung sind Ausdrücke, die eine Information über die bisherige Folge verwenden, im allgemeinen nicht abgedeckt.

Ein Musterbeispiel ist die Menge von Ausdrücken $\{a^n b c^n\}$, bei der die Symbole a und c genau gleich oft auftreten. Sofern es für n keine Beschränkung gibt, kann man keinen endlichen Automaten konstruieren, der diese Bedingung prüft. Es gibt keine Möglichkeit, die Anzahl der gesehenen Zeichen a zu speichern. In der Grammatik liegt die Beschränkung im einseitigen Wachstum der Regeln. Die

eigentlich benötigt Regel in der Art $A \rightarrow aBc$ widerspricht dem Konstruktionsprinzip von einseitig linearen Grammatiken.

Beispiele für derartige Fälle in Programmiersprachen sind Ausdrücke mit Klammern. Setzt man für a (und für c), so hat erhält man die Form $\{({}^n b)^n\}$. Es wird also verlangt, dass für jede öffnende Klammer wieder eine schließende Klammer gesetzt ist. Ohne Begrenzung von n lässt sich eine solche Sprache nicht durch eine reguläre Grammatik beschreiben.

Kapitel 15

Kellerautomaten und kontextfreie Grammatiken

15.1 Kellerautomaten

Erweitert man einen endlichen Automaten um einen Kellerspeicher (Stack), so erhält man einen Kellerautomaten. (engl. pushdown automaton PDA) Ein Kellerautomat hat unendlich viele Speicherzellen angeordnet als Stapel. Allerdings kann stets nur auf das oberste Ende zugegriffen werden. Die beiden Operationen `push` und `pop` zum Lesen und Schrieben hatten wir bereits bei der Diskussion des Stack-Konzeptes kennen gelernt. Mit dieser Erweiterung können auch Ausdrücke in der Form $\{a^n b^n\}$ erfasst werden.

In den Kellerspeicher können Symbole aus dem Kelleralphabet Γ geschrieben werden. Zur Vereinfachung führt man häufig ein spezielles Symbol $\perp$ ein, um das Ende des Kellerspeichers zu markieren. Der Folgezustand ergibt sich aus

- dem aktuellen Zustand
- dem Eingabesymbol
- dem obersten Kellersymbol

Mit jedem Zustandswechsel werden Symbole auf den Kellerspeicher gelegt. Das Verhalten des Automaten ist durch die Relation

$$f : E \times Z \times \Gamma \rightarrow \text{endliche Folge aus } Z \times \Gamma^+$$

bestimmt. Zu jedem möglichen Tripel von Zustand, Eingangssymbol und Kellersymbol ist einerseits der nächste Zustand und anderseits die Symbolfolge, die auf den Kellerspeicher gelegt wird, festgelegt. Die Möglichkeit, auch längere Folgen auf den Kellerspeicher zu legen, ist beispielsweise notwendig, wenn man den alten Zustand erhalten und ein zusätzliches Symbol darüber legen will. In jedem Schritt

wird automatisch das oberste Symbol gelesen. Man muss dann dieses Symbol und das neue Symbol zusammen wieder zurück schreiben.

Wir wollen jetzt einen Automaten angeben, der die Sprache $\{(^n)^n\}$ akzeptiert. Dazu definieren wir zunächst:

- Eingangsalphabet $E = \{(,), \varepsilon\}$
- Zustandsmenge $Z = \{S_0, S_1, S_2\}$
- Anfangszustand $z_a = S_0$
- Endzustand $F = \{S_2\}$
- Kelleralphabet $\Gamma = \{x, \perp\}$

Die drei Zustände sollen folgende Bedeutung haben:

- S_0: Startzustand. Solange das Eingangssymbol (ist, bleibt der Automat in diesem Zustand. Für jede weitere öffnende Klammer schreibt er ein zusätzliches x in den Kellerspeicher. Ist das Eingangssymbol), so wechselt er in den Zustand S_1.
- S_1: Für jede weitere schließende Klammer wird wieder ein x von Kellerspeicher entfernt. Ist die Eingangsfolge zu Ende und das oberste Kellersymbol $\perp$, so erfolgt der Übergang in den Endzustand S_2.
- S_2: Endzustand.

Zusammengefasst als Tabelle ergibt sich folgendes Bild:

Zustand	Eingabe	Keller (pop)	Zustand	Keller (push)
S_0	(	$\perp$	S_0	$x\perp$
S_0	(	x	S_0	xx
S_0	)	x	S_1	ε
S_1	)	x	S_1	ε
S_1	ε	$\perp$	S_2	ε

15.2 Kontextfreie Grammatiken

Zu den Kellerautomaten äquivalent sind die kontextfreien Grammatiken (Typ 2). Es lässt sich zeigen, dass sowohl aus jedem Kellerautomaten eine kontextfreie Grammatik als auch umgekehrt aus jeder kontextfreien Grammatik ein Kellerautomat konstruiert werden kann.

In einer kontextfreien Grammatik steht auf der linken Seite genau ein Nichtterminal. Auf der rechten Seite kann im Allgemeinen ein beliebiger Ausdruck aus Terminalen und Nichtterminalen stehen:

$$A \to w,\ A \in N,\ w \in (\Sigma \cup N)^* \tag{15.1}$$

Tabelle 15.1: Kontextfreie Grammatik G

Σ	0, 1, +, $*$
N	AUSDRUCK, CONST, DIGIT, OP
P	AUSDRUCK $\rightarrow$ CONST AUSDRUCK $\rightarrow$ AUSDRUCK OP AUSDRUCK CONST $\rightarrow$ DIGIT CONST CONST $\rightarrow$ DIGIT OP $\rightarrow$ + OP $\rightarrow$ $*$ DIGIT $\rightarrow$ 1 DIGIT $\rightarrow$ 0
S	AUSDRUCK

Wie der Name bereits besagt, spielt bei der Anwendung der Regeln der jeweilige Kontext keine Rolle. Bei der Ersetzung eines Nichtterminals A wird keine Information über dessen Vorgänger oder Nachfolger einbezogen.

Kontextfreie Grammatiken spielen im Kompilerbau eine wesentliche Rolle. Sprachen wie C lassen sich durch kontextfreie Grammatiken beschreiben. Der zugehörige Kellerautomat bildet dann die Grundlage für den Compiler. Eine einfache kontextfreie Grammatik G für Ausdrücke mit Binärzahlen ist in Tabelle 15.1 angegeben. Die Grammatik enthält als Nichtterminals die Kategorien

AUSDRUCK	Ein vollständiger Ausdruck
CONST	Eine Binärzahl
DIGIT	Die Ziffer 0 oder 1
OP	Der Operand + oder $*$

Die dadurch definierte Sprache D umfasst Ausdrücke wie

$$D = \{ \begin{array}{l} 1001, \\ 11011 + 11, \\ 11 * 100 + 11, \\ \ldots \} \end{array} \tag{15.2}$$

Aufgrund der großen Bedeutung von kontextfreien Grammatiken wurden eine Reihe von Tools entwickelt, um aus einer Grammatikdefinition automatisch lauffähigen Code zu generieren. Ein Beispiel ist das Tool `yacc` (yet another compiler compiler), das standardmäßig zu jeder UNIX-Installation gehört. Um Parser in der Sprache Java zu bauen, kann man das frei verfügbare Tool CUP (Constructor of Useful Parsers) [1] einsetzen. Die entsprechende Definition für die Beispielgrammatik hat dann die Form

[1] http://www.cs.princeton.edu/~appel/modern/java/CUP/

```
terminal      ZERO, ONE, PLUS, TIMES;
non terminal  CONST, DIGIT, AUSDRUCK, OP;

...

AUSDRUCK ::= CONST | AUSDRUCK OP AUSDRUCK
CONST    ::= DIGIT CONST  | DIGIT
OP       ::= PLUS  |  TIMES
DIGIT    ::= ZERO  |  ONE
```

Alternative Regeln können durch das Oder-Zeichen | kompakt geschrieben werden. Das Tool erzeugt aus einer solchen Spezifikation Java-Code für den Parser. Zusätzlich können Aktionen angegeben werden, die bei Auswertung einer Regel ausgeführt werden sollen. Das folgende Fragment zeigt, wie bei der Auswertung einer Konstanten ein Knoten in einem Baum angelegt und mit den beiden bereits aufgebauten Teilen verknüpft wird.

```
CONST   ::= DIGIT:n CONST:v
        {:
        DefaultMutableTreeNode sNode =
           new DefaultMutableTreeNode( "Const" );
            sNode.add( n );
            sNode.add( v );
            RESULT = sNode;
        :}
```

In Bild 15.1 ist der resultierende Baum für den Ausdruck $01 + 1 * 10$ dargestellt.

15.2.1 Wortproblem

Das Wortproblem ist für kontextfreie Sprachen lösbar. Allerdings ist die Komplexität von der Ordnung $O(N^3)$. Dies ist bereits ein beträchtlicher Aufwand. Ein Compiler benötigt dann beispielsweise für ein Programm mit 1000 Zeichen 1000^3 Operationen. Glücklicherweise kann man mit einer Einschränkung der Kellerautomaten eine wesentliche Reduktion erreichen. Dazu wird verlangt, dass zu jeder Kombination von Zustand, Eingang- und Kellersymbol genau eine Folgeaktion gehört. Man spricht dann von deterministischen Kellerautomaten (engl. Deterministic Pushdown Automaton, DPDA). Dann hat die Komplexität die Form $O(N)$.

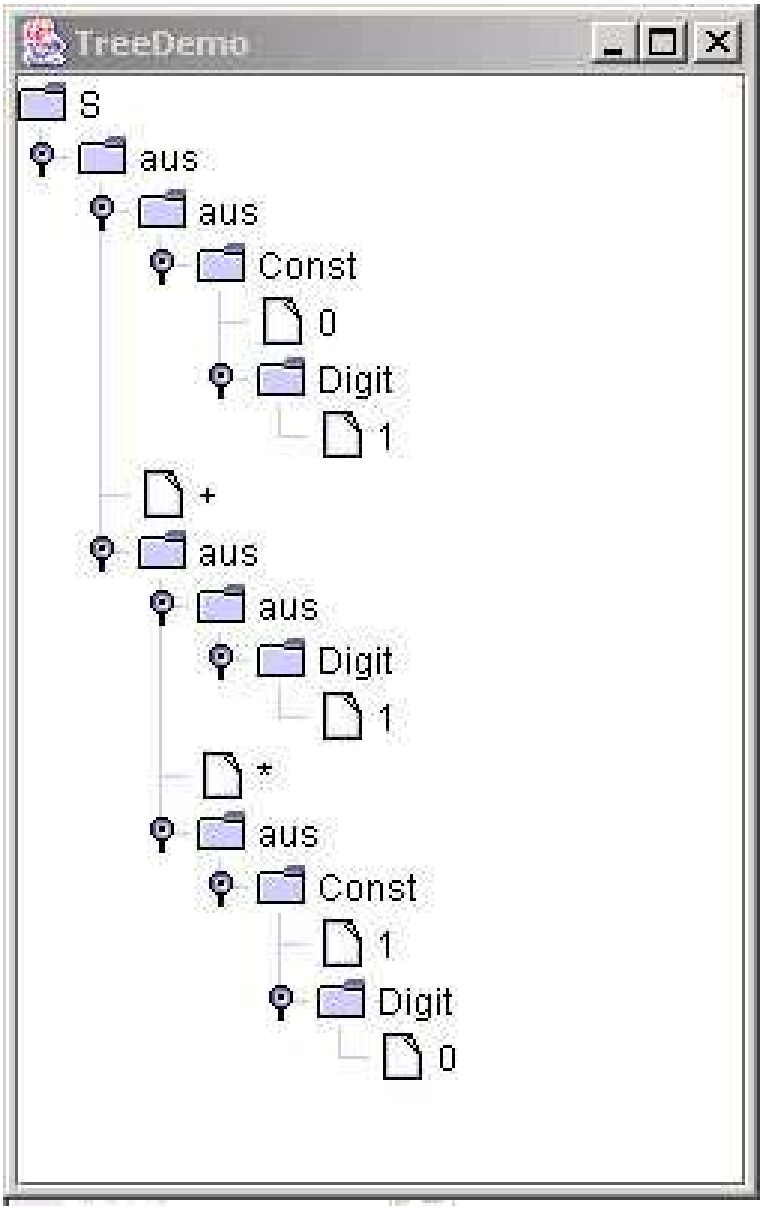


Abbildung 15.1: Ableitungsbaum für den Ausdruck $01 + 1 * 10$

Kapitel 16

Turingautomaten

Turingautomaten haben im Gegensatz zu Kellerautomaten eine wahlfreien Zugriff auf den Arbeitsspeicher. Als Speicher dient dabei ein unbegrenzt langes Band mit einem Lese- und Schreibkopf. Zu Beginn steht auf diesem Band die Eingangsfolge und der Lesekopf zeigt auf den Beginn der Folge. Leere Zellen sind durch ein spezielles Symbol (üblicherweise #) markiert.

- Endliche Zustandsmenge
- Unbegrenztes Band mit Speicherzellen
- Lese- und Schreibkopf

Verarbeitungsschritt:

- Lesen des Zeichens an der aktuellen Position
- In Abhängigkeit von diesem Zeichen und dem aktuellen Zustand:
  - Zeichen auf aktuelle Position schreiben
  - Kopf bewegen (nach links L, nach rechts R, hier bleiben H)
  - zu nächsten Zustand gehen

Überführungsfunktion δ:

$$\delta : E \times Z \rightarrow E \times Z \times \{L, R, H\}$$

16.0.2 Linear beschränkten Automaten

Eine Unterklasse der Turingautomaten sind die linear beschränkten Automaten (engl. Linear Bounded Automaton, LBA). Dabei darf der Schreiblesekopf den Bereich der Eingangsfolge nicht verlassen.

Tabelle 16.1: Grammatiken gemäß der Chomsky-Hierarchie

Typ-0	rekursiv-aufzählbare Grammatiken	Turingautomaten
Typ-1	kontextsensitive Grammatiken	linear beschränkte Automaten
Typ-2	kontextfreie Grammatiken	Kellerautomaten
Typ-3	reguläre Grammatiken	endlichen Automaten

16.1 Kontextsensitive Grammatiken und allgemeine Regelsprachen

Die weitere Verallgemeinerung der zulässigen Regeln führt zu den Typ-1 und Typ-0 Sprachen. Auf der linken Seite kann jetzt ein längerer Ausdruck gebildet aus mehreren Terminals und mindestens einem Nichtterminal stehen. Kontextsensitive Grammatiken (Typ-1) erfüllen die Monotoniebedingung, dass der Ausdruck auf der rechten Seite nicht kleiner ist als der Ausdruck auf der linken Seite. Die zugehörigen Automaten sind die Turingautomaten für Typ-0 und die linear beschränkten Automaten für Typ-1 Grammatiken.

Die 4 Grundtypen nach Chomsky sind in Tabelle 16.1 zusammen gefasst. Sie bilden eine Hierarchie zunehmender Freiheit in Bezug auf die Produktionsregeln. Die äquivalenten Automaten verfügen dementsprechend über zunehmende Speicherfähigkeit. Grammatiken höheren Typs sind Untermengen der übergeordneten Grammatiken. So ist jede reguläre Grammatik auch eine kontextfreie Grammatik. Neben den Grundtypen finden sich in der Literatur weitere Verfeinerungen.

16.1.1 Wortproblem

- Typ-1: lösbar, Komplexität $2^{O(N)}$
- Typ-0: unlösbar

16.2 Berechenbarkeit

Church[1]-Turing-These

> Jede intuitiv berechenbare Funktion kann mit einer Turing-Maschine berechnet werden.

Umkehrung:

> Wenn ein Funktion mit keiner Turing-Maschine berechnet werden kann, so ist sie nicht berechenbar.

[1] Alonzo Church 1903-1995, Amerikanische Mathematiker

16.3 Nicht berechenbare Funktionen

Wir betrachten eine einfache Turingmaschine mit n Zuständen, einem zusätzlichen Endzustand, 2 Bandsymbolen $\{0, 1\}$ und den beiden Kopfbewegungen $\{L, R\}$. Wir fragen nun:

- beginnend mit einem Band voller 0-Zeichen, welche Turing-Maschine wird die größte Anzahl von 1-Zeichen schrieben, bevor sie anhält?

Diese Maschine erhält den Titel **Fleißiger Biber**. Ausgeschlossen sind bei dem Wettbewerb alle Maschinen, die endlos lange arbeiten. Wir bezeichnen die Maximalzahl von geschriebenen Symbolen für eine gegebene Anzahl von Zuständen mit $\Sigma(n)$ (Rado-Funktion). Alternativ kann man die Anzahl von Schritten $S(n)$ als Kriterium verwenden. Für die Suche bietet sich folgendes Verfahren an:

- Man probiere nacheinander alle Maschinen mit der vorgegebenen Anzahl von Zuständen. Das sind zwar recht viele $((4 \cdot (n + 1))^{2n})$, aber nur endlich viele.

- Maschinen, die in eine Endlosschleife geraten, werden aus der Konkurrenz genommen.

- Ansonsten führt man Buch, welche Maschine die meisten 1-Zeichen produziert.

Dies ist eine klare Vorschrift, um einen fleißigen Biber zu finden. Allerdings gibt es dabei ein Problem: wie lässt sich feststellen, ob eine Maschine in eine Endlosschleife geraten ist? Dieses Problem tritt auch im praktischen Umgang mit Computern auf. Ein Programm reagiert nicht mehr. Dann stellt sich die Frage, wie viel Zeit man ihm gibt, bevor es abgebrochen wird. Schließlich ist man in der Regel nicht sicher, ob das Programm sich nicht doch noch einmal fängt und normal weiter arbeitet.

Diese Fragestellung ist als Halteproblem bekannt. Wünschenswert wäre es, wenn man automatisch testen könnte, ob ein Programm für eine gegebene Eingabe in endlicher Zeit zu einem Ergebnis kommt oder in eine Endlosschleife gerät. Mit anderen Worten, wir suchen eine Maschine H(Prog p, Eingabe e) mit

- H(Prog p, Eingabe e) = wahr: Die Maschine wird nach einer endlichen Anzahl von Schritten anhalten.

- H(Prog p, Eingabe e) = falsch: Die Maschine kommt zu keinem Ergebnis und bleibt in einer Endlosschleife hängen.

Mit einem solchen Programm wäre auch die Suche nach den fleißigen Bibern nach dem oben beschriebenen Verfahren möglich. Man würde dann mit dem Programm

Tabelle 16.2: Bekannte Ergebnisse für Fleißige Biber

n	$\Sigma(n)$	$S(n)$
1	1	1
2	4	6
3	6	21
4	13	107
5	≥ 4098	$\geq 47,176,870$
6	$> 1.29 \cdot 10^{865}$	$> 3 \cdot 10^{1730}$

H(Maschine$_n$, leeres Band) testen, ob die jeweilige Maschine überhaupt fertig wird. Alle anderen würden sofort vom Wettbewerb ausgeschlossen.

Leider lässt sich ziemlich leicht beweisen, dass es die Maschine H beziehungsweise ein entsprechendes Programm nicht geben kann. Dazu betrachten wir die Funktion

```
Seltsam( Prog p ) {
  if( H( p, p ) == wahr )  for(;;)          // Endlosschleife
  else                     print("fertig")  // Fertig
}
```

Falls H angibt, dass eine Kombination von Programm und Eingabe in endlicher Zeit zu Ende kommt, geht Seltsam in eine Endlosschleife. Jetzt kann man Seltsam auf sich selbst anwenden:

```
Seltsam( p=Seltsam )
```

Dann kann man zwei Fälle unterscheiden:

- H(Seltsam, Seltsam) = wahr: Seltsam(Seltsam) endet, dann führt aber die if-Abfrage in eine Endlosschleife, d. h. Seltsam(Seltsam) endet nie
- H(Seltsam, Seltsam) = falsch: Seltsam(Seltsam) endet nie, dann wird bei der if-Abfrage das Programm beendet, d. h. Seltsam(Seltsam) endet

In beiden Fällen führt der Selbstbezug zu einem Widerspruch. Damit kann es ein Programm H als Lösung des allgemeinen Halteproblems nicht geben. Die Funktion H(Prog p, Eingabe e) ist nicht berechenbar.

Allerdings könnte es durchaus sein, dass für die speziellen Maschinen im Fleißigen Biber Wettbewerb das Halteproblem lösbar ist. Aber man kann beweisen, dass auch die Funktion $\Sigma(n)$ nicht berechenbar ist. Einige Ergebnisse dazu sind in Tabelle 16.2 [MB90]