Parsing

Eine Einführung in die Syntaxanalyse

von
Michael Dobrovnik und Klaus Preschern

Abstract

Parsing beschäftigt sich mit der syntaktischen Analyse von Programmiersprachen, einem zentralen Bestandteil jedes Compilers. Diese Arbeit gibt einen einführenden Überblick dieses Gebiets. Neben der Darstellung einiger theoretischen Grundlagen der Syntaxanalyse werden die Methoden des Top-Down und des Bottom-Up-Parsings näher erläutert.

Inhaltsverzeichnis

1 Grundlagen der Formalen Sprachen

1.1 Alphabete, Worte, Sprachen

Ein *Alphabet* Σ ist eine endliche, nichtleere Menge von Symbolen (Zeichen). Es bildet zusammen mit der assoziativen Verknüpfung $\circ$ in Σ ("Konkatenation") und dem neutralen Element ϵ bezüglich $\circ$ ein Monoid.

$$\Sigma = \{x_i | 1 \leq i \leq n, n \text{ endlich}\}$$

Alphabete sind auch im Alltag wohlbekannt: $\Sigma_{latin} = \{a, b, c, \ldots, x, y, z\}$ ist das Alphabet der lateinischen Kleinbuchstaben; $\Sigma_{arab} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ stellt die Menge der arabischen Ziffern dar.

Die Operation $\circ$, angewendet auf zwei Symbole $x, y \in \Sigma$ ergibt ein *Wort* (einen *String*, einen *Satz*, eine *Zeichenkette*) $s = x \circ y = (x, y) = xy$ als Sequenz der beiden Symbole. Allgemein gesagt ist ein Wort über dem Alphabet Σ eine Konkatenation von n Symbolen aus Σ mit $n \geq 0$. Insbesondere bildet ϵ das *leere Wort* über Σ. Als die Länge $l(s)$ eines Strings s bezeichnet man die Anzahl der den String bildenden Symbole.

Mit der Konkatenation kann man die *Exponentiation* auf der Menge Σ definieren: $\Sigma^0 = \{\epsilon\}$ und mit $n \geq 1 : \Sigma^n = \Sigma^{n-1} \circ \Sigma$. Σ^n ist also die Menge der Worte über Σ, die aus n Symbolen zusammengesetzt wurden. So ist beispielsweise Σ^3_{arab} die Menge aller Ziffernfolgen der Länge drei, wärend Σ^5_{latin} alle Folgen von fünf Kleinbuchstaben darstellt.

Aus der Exponentiation ergeben sich zwei weitere Operationen:

Sternbildung:

$$\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$$

Σ^* ist die Menge aller Worte über Σ, wobei insbesonders auch $\epsilon \in \Sigma^*$. Man nennt Σ^* die *Kleensche Hülle* von Σ. Σ^*_{latin} beinhaltet alle Strings, die aus einer Konkatenation von 0 oder mehr Kleinbuchstaben hervorgehen können.

Plusbildung:

$$\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n$$

Die *transitive Hülle* Σ^+ von Σ ist die Menge aller vom leeren Wort verschiedenen Worte über Σ. Σ^+_{arab} ist die Menge aller Ziffernfolgen der Länge > 0.

Ein Wort s über Σ kann man also auch als Element aus Σ^* auffassen. Ausgehend von einem Wort $x = uvw$, mit $u, v, w \in \Sigma^*$ nennt man u einen *Präfix*, w einen *Suffix* von x. v wird als *Substring* oder *Teilwort* bezeichnet. Sind $u, v, w \in \Sigma^+$, so spricht man von einem *echten* Präfix bzw. Substring oder Suffix).

Jede Teilmenge $L \subset \Sigma^*$ heißt *formale Sprache* $L(\Sigma)$ über dem Alphabet Σ (wir werden der Kürze halber oft den Begriff "Sprache" als Synonym für "formale Sprache" verwenden ohne uns damit etwa auf natürliche Sprachen zu beziehen). Die

Beschreibung einer formalen Sprache muß festlegen, welche Folgen von Zeichen korrekte Wörter der Sprache sind. Eine solche Beschreibung wird *Syntax* der Sprache genannt.

Es sind prinzipiell vier verschiedene Verfahren zur Definition einer Sprache $L \subset \Sigma^*$ bekannt:

Aufzählung: Darunter versteht man die explizite Auflistung aller Worte in L. Diese Methode ist nicht unbedingt die praktikabelste und wird in weiterer Folge nicht mehr erwähnt.

Generierung: Man postuliert einer Reihe von Regeln, die eine Menge von Worten $W \subset \Sigma^*$ erzeugen und definiert, daß $L(\Sigma) = W$. Solche Regeln sind ein zentraler Bestandteil von *Grammatiken*, auf die später ausführlicher eingegangen wird. Dieses Verfahren ist gut für die Praxis geeignet und findet u.a. bei der Definition von Programmiersprachen breite Anwendung.

Erkennung: Ein Algorithmus wird angegeben, der feststellt, ob für ein Wort $x \in \Sigma^*$ gilt, daß $x \in L$ ist (für $x \notin L$ wird die Terminierung des Algorithmus nicht gefordert). Der Algorithmus akzeptiert also gültige Strings und wird daher auch als *Akzeptor* bezeichnet. In der Informatik allerdings sind derartige Algorithmen als *Parser* bekannt.

Konstruktion: Hier wird ein Ausdruck angegeben, mit Hilfe dessen genau die Wörter aus L ermittelt werden können. Zum Beispiel wird durch den Ausdruck $L = \{(^n)^n | n \geq 1\}$ über dem Alphabet $\Sigma_{bracket} = \{(,)\}$ eine Sprache erzeugt, die alle Wörter umfaßt, in denen auf n öffnende Klammern genau n schließende Klammern folgen.

1.2 Grammatiken

In diesem Abschnitt wollen wir uns mit den Generationsverfahren zur Definition von formalen Sprachen beschäftigen. Diese Verfahren basieren auf einer Menge von Regeln, die es ermöglichen, aus gegebenen Strings neue, syntaktisch korrekte Strings zu erzeugen. Neben diesen Regeln besitzt eine Grammatik noch drei weitere Komponenten. Die Bestandteile einer Grammatik sind also:

Terminalsymbole: Sind Zeichen oder Gruppen von Zeichen. Wörter der erzeugten Sprache L bestehen ausschließlich aus Terminalsymbolen. Deshalb bezeichnet man diese auch als *Basiszeichen* der Sprache L. Eine andere gebräuchliche Bezeichnung lautet *Token*. T ist die endliche, nichtleere Menge der Terminalsymbole (das Terminalalphabet).

Nichtterminalsymbole: Sind Symbole, die syntaktische Klassen (grammatikalische Konstrukte) darstellen. Jedes Nichtterminalsymbol läßt sich (durch die schon erwähnten Regeln) in andere Nichtterminalsymbole oder Terminalsymbole zerlegen; man nennt die Nichtterminalsymbole daher auch die *Variablen* der Sprache L. Diese Nonterminals bilden eine Metasprache der syntaktischen Konstrukte, die zur Definition von L verwendet werden und diese Sprache strukturieren (in der deutschen Sprache sind etwa Satz, Subjekt, Objekt, Prädikat solche syntaktischen Klassen). Mit N bezeichen wir die Menge der Nonterminalsymbole. Es wird vorausgesetzt, daß $N \cap T = \emptyset$.

Startsymbol: Das Startsymbol S ist ein besonderes Element aus N. Es stellt das allgemeinste Konstrukt der Sprache dar und wird auch als *Axiom* bezeichnet. In Programmiersprachen wird das Startsymbol *Programm* genannt.

Produktionsregeln: Diese Regeln sind Ersetzungsvorschriften, die beschreiben, wie aus schon bestehenden Folgen von Elementen aus $N \cup T$ neue Folgen gebildet werden können. Die Menge P der Produktionsregeln ist also eine endliche Relation über $(N \cup T)^*$. Eine Produktionsregel oder kurz Produktion hat die Form $\alpha \rightarrow \beta$ oder auch (α, β) und bedeutet, daß aus jedem Wort mit der Struktur α ein Wort der Struktur β gebildet werden kann. α wird als *linke Seite*, β als *rechte Seite* der Produktion bezeichnet. Die linke Seite jeder Produktion muß mindestens ein Nonterminal enthalten, α ist also eigentlich aus $(N \cup T)^* N (N \cup T)^*$.

Eine Grammatik ist also ein Quadrupel $G = \{N, T, S, P\}$ mit der obengenannten Bedeutung. In der Literatur wird auch oft eine abweichende, aber zur hier verwendeten Form äquivalente Darstellung angegeben: $G = \{V, T, S, P\}$, wobei $V = N \cup T$ als *Vokabular* bezeichnet wird.

Manchmal werden wir in der Folge auch von Grammatiken sprechen, explizit aber nur die Produktionen angeben. In theoretischem Kontext verwenden wir zur Unterscheidung von Terminals und Nonterminals Kleinbuchstaben aus dem Beginn des Alphabets $(a, b, c, \ldots)$ für einzelne Terminals, Kleinbuchstaben am Ende des Alphabets $(\ldots, w, x, y, z)$ stehen für Strings aus Terminals. Nonterminals werden durch Großbuchstaben zu Beginn des Alphabets $(A, B, C, \ldots)$ dargestellt (insbesondere ist S dem Startsymbol vorbehalten), das Ende des Alphabets der Großbuchstaben $(\ldots, W, X, Y, Z)$ ist zur Darstellung von Grammatiksymbolen (Terminals oder Nonterminals) bestimmt. Griechische Kleinbuchstaben $(\alpha, \beta, \gamma, \ldots)$ schließlich repräsentieren Strings von Grammatiksymbolen aus $N \cup T$. Die folgende Zusammenfassung stellt Notation und korrespondierende Basismengen gegenüber:

Notation	Grundmenge
$a, b, c, \ldots$	$\in T$
$\ldots, x, y, z$	$\in T^*$
$A, B, C, \ldots$	$\in N$
$\ldots, X, Y, Z$	$\in (N \cup T)$
$\alpha, \beta, \gamma, \ldots$	$\in (N \cup T)^*$

Wenn wir Beispiele angeben, stellen wir Nonterminals *kursiv* dar. Terminals werden durch diesen `Font` identifiziert. Auch alle Operatoren $(+, -, *, \ldots)$ und Ziffern sind als Terminals aufzufassen. Die linke Seite der ersten Produktion ist gleichzeitig immer das Startsymbol der Grammatik. Gibt es Produktionen mit identischen linken Seiten, also etwa $\alpha \to \beta_1, \alpha \to \beta_2, \ldots, \alpha \to \beta_n$, so bezeichnen wir diese Produktionen nach der gemeinsamen linken Seite als α-Produktionen und notieren in Kurzform $\alpha \to \beta_1 | \beta_2 | \ldots | \beta_n$.

Wenden wir uns nun einem beliebten Beispiel zu und geben wir eine Grammatik an, die ein in fast jeder Programmiersprache vorhandenes Konstrukt, nämlich arithmetische Ausdrücke (in einfacher Form) generiert:

$$
\begin{aligned}
G^1_{expr} &= (N, T, S, P) \text{ mit} \\
N &= \{expr, number, identifier, digit, letter\}, \\
T &= \{+, -, *, /, (,), 0, 1, \ldots, 8, 9, \texttt{a}, \texttt{b}, \ldots, \texttt{y}, \texttt{z}\}, \\
S &= expr, \\
P &= \{ \\
expr &\to expr + expr \\
&\quad | expr - expr \\
&\quad | expr * expr \\
&\quad | expr / expr \\
&\quad | (\ expr\ ) \\
&\quad | number \\
&\quad | identifier \\
number &\to digit \\
&\quad | number\ digit \\
identifier &\to letter \\
&\quad | identifier\ digit \\
&\quad | identifier\ letter \\
digit &\to 0 | 1 | \ldots | 8 | 9 \\
letter &\to \texttt{a} | \texttt{b} | \ldots | \texttt{y} | \texttt{z} \\
&\quad \}
\end{aligned}
$$

Die Terminalsymbole dieser Grammatik umfassen arithmetische Operatoren, Klammern, Ziffern und Buchstaben. Die syntaktischen Klassen, die sich durch die Nonterminals manifestieren, sind *expr*, die Ausdrücke selbst, Zahlen (*number*) und Variablenbezeichner ($identifier$).

1.3 Ableitungen

Nun wollen wir uns mit der konkreten Anwendung der Produktionsregeln beschäftigen. Sei $G = \{N, T, S, P\}$ eine Grammatik mit dem Vokabular $V = N \cup T$ und seien $\lambda, \rho \in V^*$, so bezeichnet man ρ als *unmittelbare (direkte) Ableitung* von λ, in Zeichen $\lambda \Rightarrow \rho$, wenn es Worte γ_1 und $\gamma_2 \in V^*$ gibt, sodaß $\lambda = \gamma_1\alpha\gamma_2$ und $\rho = \gamma_1\beta\gamma_2$ und $\alpha \rightarrow \beta \in P$. D.h. ein Substring α aus λ stimmt mit der linken Seite mindestens einer Produktion überein und die Regel kann daher zur Anwendung kommen. Der passende Substring α wird durch die rechte Seite β einer korrespondierenden Regel ersetzt.

Man sagt, daß ρ von λ *direkt produziert*, bzw. daß ρ zu λ *direkt reduzierbar* ist. Erweitert man das Konzept, so wird ρ von λ *produziert* (bzw. ρ ist aus λ *ableitbar*, in Symbolen $\lambda \overset{+}{\Rightarrow} \rho$, wenn es Worte $\gamma_0, \gamma_1, \ldots, \gamma_n (n > 0)$ gibt, sodaß $\lambda = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = \rho$. Die Folge der γ_i bezeichnet man als *Ableitung der Lä*nge n. Ist auch $n = 0$ zulässig, so notiert man $\lambda \overset{*}{\Rightarrow} \rho$ (d.h. $\lambda \overset{+}{\Rightarrow} \rho \vee \lambda = \rho$) und sagt ρ ist aus λ *ableitbar* oder auch λ führt zu ρ (in Null oder mehr Schritten).

Als *Satzform* σ bezeichnet man jede Ableitung $S \overset{*}{\Rightarrow} \sigma$ aus dem Startsymbol S der Grammatik G. Besteht σ nur aus Terminalsymbolen, gilt also, daß $\sigma \in T^*$, so nennt man σ einen *Satz* oder ein *Wort* von G. Die Menge aller Sätze einer Grammatik wird als die von der Grammatik *generierte Sprache* bezeichnet.

$$L(G) = \{\sigma | S \overset{*}{\Rightarrow} \sigma \wedge \sigma \in T^*\}$$

Generieren zwei Grammatiken G_1 und G_2 die selbe Sprache L, so bezeichnet man G_1 und G_2 als *äquivalent*, in Symbolen $G_1 \sim G_2$.

Führen wir uns nun anhand der oben definierten Grammatik G^1_{expr} einige der eben definierten Begriffe vor Augen:

Da $expr \rightarrow number \in P$, ist *number* eine direkte Ableitung von *expr*. Weiters ist z.B. $(number + identifier)$ eine Ableitung von *expr*, da $expr \Rightarrow (expr) \Rightarrow (expr + expr) \Rightarrow (number + expr) \Rightarrow (number + identifier)$ ableitet. Gleichzeitig ist $(number + identifier)$ eine Satzform der Grammatik, da dieser Ausdruck aus dem Startsymbol *expr* ableitbar ist. Leitet man weiter ab : $(number + identifier) \Rightarrow (digit\ number + identifier) \Rightarrow (digit\ digit + identifier) \Rightarrow (1\ digit + identifier) \Rightarrow (13 + identifier) \Rightarrow (13 + identifier\ digit) \Rightarrow (13 + letter\ digit) \Rightarrow (13 + \texttt{x}\ digit) \Rightarrow (13 + \texttt{x4})$, so erhält man einen Satz aus $L(G^1_{expr})$. Die Länge der Ableitung dieses Satzes beträgt 12, da $expr \overset{12}{\Rightarrow} (13 + x4)$.

2 Die Chomsky-Hierarchie

Die Hierarchie der Sprachklassen wurde 1959 von dem amerikanischen Sprachwissenschaftler Noam Chomsky als mögliches Modell für natürliche Sprachen aufgestellt. Die Chomsky-Grammatiken unterscheiden sich in ihrer generativen Mächtigkeit. Im folgenden Abschnitt werden die vier Hauptklassen der Chomsky-Hierarchie behandelt. Dies sind (mit aufsteigender Mächtigkeit) die regulären Sprachen (Typ-3 Grammatiken oder reguläre Mengen), die kontextfreien Sprachen (Typ-2 Grammatiken), die kontextsensitiven Sprachen (Typ-1 Grammatiken) und die nicht eingeschränkten Sprachen (Typ-0 Grammatiken oder rekusiv aufzählbare Mengen). Hierbei gilt:

a) Die regulären Sprachen sind eine echte Teilmenge der kontextfreien Sprachen.

b) Die kontextfreien Sprachen (ohne die leere Zeichenkette) sind eine echte Teilmenge der kontextsensitiven Sprachen.

c) Die kontextsensitiven Sprachen sind eine echte Teilmenge der aufzählbaren Mengen.

Die Chomsky-Hierarchie läßt sich durch die Einordnung von verschiedenen Modellen von erkennenden Automaten vervollständigen. In den folgenden Abschnitten, werden jeweils die entsprechenden Automaten der verschiedenen Sprachklassen und die zu den jeweiligen Automaten äquivalenten Grammatiken beschrieben [HoUl88].

2.1 Reguläre Sprachen

In diesem Abschnitt wird die Sprachklasse der regulären Mengen beschrieben und das Modell des endlichen Automaten, der diese Sprachklasse akzeptiert. In der Praxis wird die Theorie der endlichen Automaten beim Erstellen von Schaltkreisen, für die Erstellung von lexikalischen Analysern im Compilerbau oder in Texteditoren bzw. in Textverarbeitungsprogrammen verwendet.

2.1.1 Endliche Automaten

Ein *endlicher Automat* (EA) besteht aus einer endlichen Menge von Zuständen und einer Menge von Transitionen (Zustandsübergängen), die auf einem Eingabesymbol aus einem Alphabet Σ arbeiten und einen Zustand in einen anderen überführen (Abbildung 1). Für jedes Eingabe-Symbol existiert genau ein Zustandsübergang. Ein Zustand, der gewöhnlich mit q_0 bezeichnet wird, ist der Anfanszustand, in dem der Automat startet. Einige Zustände sind ausgezeichnet als akzeptierende bzw. Endzustände in denen der Automat stehenbleibt.

Formal wird ein *(deterministischer) endlicher Automat* als ein Quintupel der Form $(Q, \Sigma, \delta, q_0, F)$ defininiert, wobei Q eine endliche Menge von Zuständen, Σ ein endliches Eingabealphabet und δ die Übergangsfunktion ist, die Q$\times\Sigma$ auf Q abbildet, d.h. $\delta(q, a)$ ist eine Funktion, die einem Zustand q des Automaten und einem Eingabesybol a einen neuen Zustand zuordnet. F ist eine Menge von Endzuständen mit $F \subset Q$.

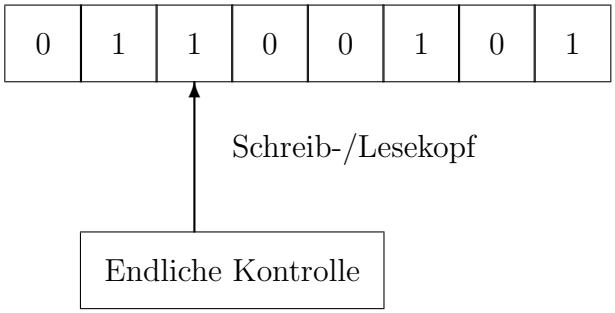

Abbildung 1: Modell eines endlichen Automaten.

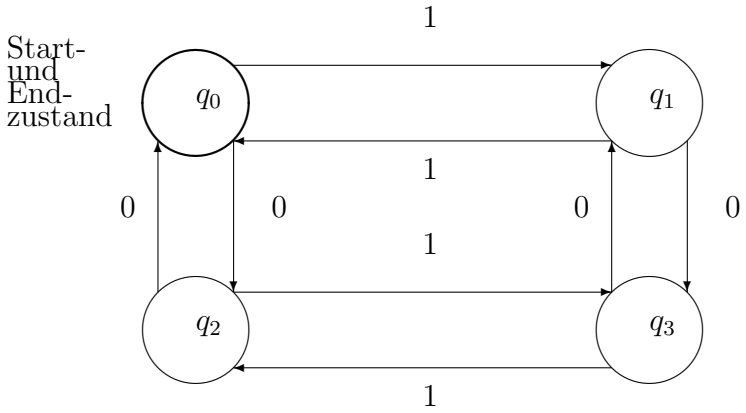

Abbildung 2: Transitionsdiagramm für einen endlichen Automaten.

In einem Bearbeitunsschritt geht der EA, der sich im Zustand q befindet und das Symbol a bearbeitet, in den Zustand $\delta(q, a)$ über und bewegt seinen Kopf um ein Symbol nach rechts. Ist $\delta(q, a)$ ein akzeptierender Zustand, so hat der EA die Zeichenkette akzeptiert, die sich vom Bandbeginn bis vor die aktuelle Position des Kopfes erstreckt.

Ein gerichteter Graph (= Transitionsdiagramm) kann wie folgt mit einem EA assoziiert werden: Die Knoten im Graph entsprechen den Zuständen des EA. Gibt es bei der Eingabe von a einen Übergang vom Zustand q in den Zustand p, dann existiert ein mit a markierter Pfeil vom Zustand q in den Zustand p im Transitionsdiagramm (Abbildung 2). Der EA akzeptiert eine Zeichenkette x, wenn die Transitionsfolge, die den Symbolen der Eingabe entspricht, den Anfangszustand in einen akzeptierenden Zustand überführt.

Um das Verhalten eines EA auf einer Zeichenkette formal zu beschreiben, wird im folgenden die Übergangsfunktion δ so erweitert, daß sie auf einen Zustand und eine Zeichenkette anstatt nur auf einen Zustand und ein Symbol angewendet werden kann. Wir definieren also eine Funktion $\hat{\delta}$ von $Q \times \Sigma^*$ nach Q.

Also ist $\hat{\delta}(q, \omega)$ genau der Zustand p, für den es einen Pfad im Transitionsdia-

gramm von q nach p gibt, der mit ω markiert ist. Formal läßt sich $\hat{\delta}$ folgendermaßen definieren:

1. $\hat{\delta}(q, \epsilon) = q$
2. Für alle Zeichenketten ω und alle Eingabe-Sybmole a gilt:

$$\hat{\delta}(q, \omega a) = \delta(\hat{\delta}(q, \omega), a).$$

Falls in Regel (2) $\omega = \epsilon$ gesetzt wird, so gilt $\hat{\delta}(q, a) = \delta(\hat{\delta}(q, \epsilon), a) = \delta(q, a)$, womit die Argumente für δ und $\hat{\delta}$ übereinstimmen. Daher werden wir wird im folgenden aus Gründen der Bequemlichkeit δ statt $\hat{\delta}$ benutzen.

Eine Zeichenkette x wird von einem endlichen Automaten $M = (Q, \Sigma, \delta, q_0, F)$ *akzeptiert*, wenn $\delta(q_0, x) = p$ für einen Zustand p aus F gilt. Die von M *akzeptierte Sprache*, bezeichnet mit $L(M)$, ist die Menge $\{x \mid \delta(q_0, x) \in F\}$. Eine Sprache ist eine *reguläre Menge* bzw. heißt *regulär*, wenn sie die Menge ist, die durch einen endlichen Automaten akzeptiert wird.

Sind für einen Zustand bei demselben Eingabesymbol Null, eine oder mehr Transitionen erlaubt, so spricht man von einem *nichtdeterministischen* EA (NEA). Gibt es in der Übergangsfunktion je Zustand maximal eine einzige Transition für jedes Symbol, so handelt es sich um einen *deterministischen* EA (DEA). Ein DEA ist also ein Spezialfall eines NEA. Um zu bestimmen, ob eine Zeichenkette ω von einem DEA akzeptiert wird, genügt es also einen Pfad im Transitionsdiagramm zu überprüfen, während es in einem NEA viele Pfade geben kann, die mit ω markiert sind. Das Konzept des Nichtdeterminismus spielt sowohl in der Sprachentheorie, als auch in der Berechenbarkeitstheorie eine wichtige Rolle.

Formal ist ein *nichtdeterministischer* EA (NEA) ein Quintupel $(Q, \Sigma, \delta, q_0, F)$, wobei Q, Σ, q_0, und F (Zustände, Eingabealphabet, Anfangszustand und Endzustände) die gleiche Bedeutung, wie bei einem DEA haben. δ ist jedoch eine Abbildung von $Q \times \Sigma$ nach 2^Q, also in die Menge aller Teilmengen (Potenzmenge) von Q.

Da jeder DEA ein NEA ist, ist unmittelbar klar, daß die Sprachklasse der regulären Mengen auch von NEA akzeptiert werden. Interessant ist jedoch, daß dies die einzigen Mengen sind, die von NEA akzeptiert werden, d.h. für jeden NEA kann ein äquivalenter DEA konstruiert werden, der dieselbe Sprache akzeptiert.

2.1.2 Reguläre Grammatiken

Wenn die Produktionen einer Grammatik von der Form $A \to \omega B$ oder $A \to \omega$ sind, wobei A und B Variablen und ω eine (möglicherweise leere) Zeichenkette von Terminalen ist, so nennt man die Grammatik *rechts-linear*. Sind alle Produktionen von der Form $A \to B\omega$ oder $A \to \omega$, so wird die Grammatik *links-linear* genannt. Eine rechts- oder links-lineare Grammatik wird *reguläre Grammatik* genannt. Reguläre Grammatiken, endliche Automaten und reguläre Mengen sind äquivalent.

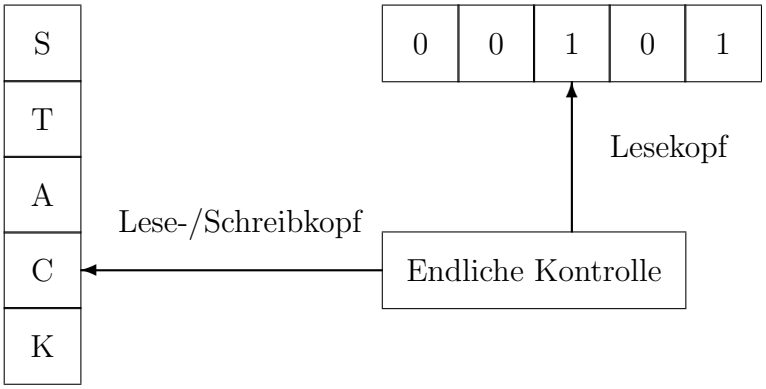

Abbildung 3: Ein Kellerautomat.

2.2 Kontextfreie Sprachen

In diesem Abschnitt werden die kontextfreien Sprachen (kfS), die zugehörige Klasse der diese Sprache akzeptierenden Automaten sowie die kontextfreien Grammatiken (kfG) beschrieben. Die kontextfreien Sprachen sind von großer praktischer Bedeutung, vor allem bei der Definition von Programmiersprachen, bei der Formalisierung der Syntaxanalyse im Compilerbau und in anderen Anwendungen, bei denen Zeichenketten verarbeitet werden.

2.2.1 Kellerautomaten

Im wesentlichen ist ein *Kellerautomat* (KA) ein endlicher Automat, der über eine Kontrolle für das Eingabeband und für einen Keller (Stack) verfügt (Abbildung 3). Wenn im folgenden der Begriff des KA gebraucht wird, so ist stets der nichtdeterministische KA gemeint, sonst wird explizit darauf hingewiesen, daß die deterministische Variante gemeint ist.

Formal ist ein Kellerautomat M ein System $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ mit:

1) Q ist eine endliche Menge von Zuständen,

2) Σ ist das Eingabealphabet,

3) Γ ist das Kelleralphabet,

4) q_0 aus Q ist der Anfangszustand,

5) Z_0 aus Γ ist das Anfangssymbol,

6) $F \subseteq Q$ ist eine Menge von Endzuständen und

7) δ ist eine Abbildung von $Q \times (\Sigma \cup \{\epsilon\} \times \Gamma)$ in endliche Teilmengen von $Q \times \Gamma^*$.

Es gibt zwei Arten von Bewegungen: Bei der ersten wird mit einem Eingabe-Symbol, dem obersten Kellersymbol und dem Zustand der endlichen Kontrolle entschieden in welchen Folgezustand überzugehen ist, wobei das oberste Kellersymbol durch eine - möglicherweise leere - Zeichenkette ersetzt wird. Danach wird der Eingabe-Kopf um ein Zeichen nach rechts geschoben. Die zweite Möglichkeit (auch als ϵ-Bewegung bezeichnet) ist, daß der Kellerautomat das Eingabe-Symbol nicht benutzt und die Stellung des Eingabe-Kopfes unverändert bleibt. Dieser Bewegungstyp erlaubt dem KA den Keller ohne das Lesen von Eingabe-Symbolen zu verändern.

Für einen Kellerautomaten gibt es zwei Möglichkeiten eine Sprache zu akzeptieren. Die erste Möglichkeit ist das Akzeptieren einer Eingabe durch das Übergehen in einen Endzustand (genauso wie der EA). Die zweite Möglichkeit ist das Akzeptieren durch den leeren Keller, d.h. eine Eingabe die den KA veranlaßt seinen Keller zu leeren wird akzeptiert. Beide Möglichkeiten sind äquivalent.

Für Kellerautomaten gilt, daß die nichtdeterministische Variante nicht gleich der deterministischen Variante ist. D.h. die Klasse der Sprachen die von deterministischen Kellerautomaten erkannt werden ist eine echte Teilmenge der Sprachen die von nichtdeterministischen Kellerautomaten erkannt werden.

2.2.2 Kontextfreie Grammatiken

Formal ist eine *kontextfreie Grammatik* (kfG) G ein Quadrupel (N, T, P, S), wobei N und T endliche disjunkte Mengen von Nichtterminalsymbolen (Variablen) und Terminalsymbolen sind. P ist eine endliche Menge von Produktionen, wobei jede Produktion von der Form $A \rightarrow \alpha$ ist, in der A für eine Variable steht und α eine Zeichenkette von Symbolen aus $(N \cup T)^*$ ist. S ist eine spezielle Variable, die als Startsymbol bezeichnet wird.

Wie man sieht besteht bei einer kfG die linke Regelseite nur aus einzelnen Nichtterminalsymbolen. Daher stammt auch die Bezeichnung "kontextfrei", da dadurch ein Nichtterminalsymbol immer unabhängig vom Kontext, also unabhängig von den benachbart stehenden Zeichen ersetzt wird. Dadurch kann man die Ableitungen in kontextfreien Grammatiken stets als Bäume (= Ableitungsbäume) darstellen.

2.3 Kontextsensitive Sprachen

Die Klasse der Kontextsensitiven Sprachen ist so allgemein, daß fast jede vorstellbare Sprache kontextsensitiv ist. Im folgenden Abschnitt wird zuerst der diese Sprache akzeptierende Automat vorgestellt, gefolgt von der Beschreibung der kontextsensitiven Grammatik.

2.3.1 Linear beschränkte Automaten

Der Automat, der die Klasse der kontextsensitiven Sprachen akzeptiert ist der *nichtdeterministische linear beschränkte Automat* (LBA). Da dieses Modell eine starke Ähnlichkeit zu einer anderen Klasse von Automaten, nämlich den *Turing-Maschinen* (TM) aufweist, wird in diesem Abschnitt einerseits das Modell der TM vorgestellt

und andererseits der LBA beschrieben. Ein nichtdeterministischer *linear beschränkter Automat* (LBA) ist nämlich vom Prinzip her eine nichtdeterministische Turing-Maschine, deren Arbeitsband durch die Länge des Eingabewortes beschränkt ist. Dazu verwendet man zwei spezielle Bandsymbole ϕ und \$, die das linke bzw. rechte Ende des Eingabewortes markieren und die während der Verarbeitung nicht überschrieben und auch nicht übersprungen werden dürfen.

Formal wird eine *Turing-Maschine* (TM) durch $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ beschrieben, wobei folgendes gilt:

1) Q ist eine endliche Menge von Zuständen,

2) Γ ist eine endliche Menge von erlaubten Bandsymbolen,

3) B ist das Blank, ein spezielles Symbol aus Γ,

4) Σ ist die Menge der Eingabesymbole (= eine Teilmenge von Γ, die B nicht einschließt),

5) δ ist die Übergangsfunktion und eine Abbildung von $Q \times \Gamma$ nach $Q \times \Gamma \times \{L, R\}$ (δ kann für einige Argumente undefiniert sein; L bzw. R bedeutet eine Bewegung des Schreib-/Lesekopfes nach links bzw. rechts),

6) $q_0 \in Q$ ist der Anfangszustand,

7) $F \subseteq Q$ ist die Menge von Endzuständen.

Mit einer Bewegung vollbringt die TM - abhängig von dem gerade durch den Bandkopf gelesenen Bandsymbol und dem Zustand der endlichen Kontrolle - folgendes:

1) Sie ändert ihren Zustand.

2) Sie schreibt ein Symbol in das gelesene Bandfeld und überschreibt dabei das gespeicherte Symbol.

3) Sie bewegt den Kopf um ein Feld nach rechts oder links.

Während bei der TM das Band möglicherweise unendlich lang ist, ist es beim LBA durch die beiden Ende-Markierungen beschränkt. Formal wird ein *linear beschränkter Automat* (LBA) durch $M = (Q, \Sigma, \Gamma, \delta, q_0, \phi, \$, F)$ bezeichnet, wobei $Q, \Sigma, \Gamma, \delta, q_0$ und F wie für die nichtdeterministische TM definiert sind und die beiden Symbole ϕ und \$ aus Σ (sie gehören jedoch nicht zum Eingabewort) die linke und rechte Ende-Markierung sind.

2.3.2 Kontextsensitive Grammatiken

Bei kontextsensitiven Grammatiken (ksG) werden die Produktionen von der Form $\alpha \rightarrow \beta$ so eingeschränkt, daß β mindestens so lang wie α sein muß. Manche Eigenschaften von Programmiersprachen lassen sich mit kontextfreien Grammatiken nicht beschreiben, wie z.B. die Forderung daß eine Variable vor ihrer Verwendung deklariert werden muß. Mit kontextsensitiven Grammatiken ist dies möglich. Allerdings muß einschränkend gesagt werden, daß die Menge der Produktionsregeln hierdurch so unübersichtlich wird, daß man in der Praxis lieber eine kontextfreie Grammatik verwendet und die zusätzlichen Einschränkungen in der Umgangssprache hinzufügt.

2.4 Nicht eingeschränkte Sprachen

Nicht eingeschränkte Sprachen (rekursiv aufzählbare Sprachen) sind die größte Familie von Sprachen innerhalb der Chomsky-Hierarchie. Der Term äufzählbarleitet sich von der Tatsache ab, daß es genau diese Sprachen sind, deren Zeichenketten durch eine Turing-Maschine (TM) aufgezählt werden können.

2.4.1 Turing-Maschinen

Das Modell der Turing-Maschine wurde bereits im Abschnitt über linear beschränkte Automaten formal beschrieben. Die Turing-Maschine ist ein anerkannter Formalismus für Algorithmen und ist, was die Berechenbarkeit betrifft, äquivalent zu dem uns heute bekannten digitalen Computer. Die von einer TM akzeptierte Sprache ist die Menge der Wörter aus Σ^*, die die TM veranlassen in einen Endzustand überzugehen. Ist eine TM gegeben, die eine Sprache L erkennt, so wird angenommen, daß die TM immer dann anhält - d.h. keine nächste Bewegung hat - wenn die Eingabe akzeptiert wird. Für Wörter die nicht erkannt werden, ist es möglich, daß die TM nie anhält. Ebenso wie beim endlichen Automaten werden auch bei Erweiterung der TM durch Nichtdeterminismus nicht mehr Sprachen erkannt. Dies bedeutet, daß die deterministische und die nichtdeterministische TM äquivalent sind. Interessant ist, daß diese Eigenschaft nur für EA und TM in den Randbereichen der Chomsky-Hierarchie gilt, während diese Eigenschaft nicht für KA gilt. Für LBA ist es bisher noch ungeklärt, ob es zu jedem nichtdeterministischen LBA einen deterministischen LBA gibt, der genau dieselbe Sprache akzeptiert.

2.4.2 Nicht eingeschränkte Grammatiken

Diese Form der Grammatiken erlaubt Produktionen der Form $\alpha \rightarrow \beta$, wobei $\alpha \in (N \cup T)^* N (N \cup T)^*$ und $\beta \in (N \cup T)^*$ beliebige Zeichenketten von Grammatiksymbolen sind (mit $\alpha \neq \epsilon$). Diese Grammatiken sind auch bekannt als Semi-Thue-Systeme, Typ-0 Grammatiken oder allgemeine Regelgrammatiken.

3 Kontextfreie Grammatiken

3.1 Die Bedeutung kontextfreier Grammatiken

Obwohl kontextfreie Grammatiken nicht mächtig genug sind, um alle syntaktischen Aspekte von Programmiersprachen zu beschreiben, spielen solche Typ-2 Grammatiken eine herausragende Rolle bei der Definition der Syntax von Programmiersprachen.

Dies hat seine Ursache einerseits darin, daß keine effizienten Parsing-Algorithmen existieren, die es ermöglichen würden, mächtigere Grammatiken zur Sprachdefinition heranzuziehen (während solche Algorithmen für kontextfreie Grammatiken wohlbekannt sind), andererseits aber können die wichtigsten Aspekte von Programmiersprachen durchaus mit Hilfe kontextfreier Grammatiken formalisiert werden. Die restlichen, kontextsensitiven syntaktischen Vorschriften müssen (wie übrigens auch die Semantik der Sprache) anders, nämlich informal ausgedrückt werden. Solche Vorschriften regeln z.B., daß Variable vor deren Verwendung deklariert werden müssen oder daß Prozedurdeklaration und Prozeduraufrufe hinsichtlich Anzahl und Datentyp der Parameter übereinzustimmen haben. Einschränkungen dieser Art formal zu beschreiben, hieße zugunsten von bloßem Formalismus auf Prägnanz und Klarheit der Sprachdefinition, wie sie sich Menschen darstellt, zu verzichten.

Die in der Praxis angegebenen kontextfreien Beschreibungen einer Sprache stellen allein also eine Obermenge der syntaktisch korrekten Programme dar. Erst die o.g. syntaktischen Restriktionen komplettieren diese Definition, wobei die Semantik noch immer außer acht gelassen wird.

Wegen der hohen praktischen Bedeutung der kontextfreien Grammatiken bei der syntaktischen Definition von Programmiersprachen wollen wir uns nun etwas genauer mit ihnen befassen.

3.2 Minimalität von Grammatiken

Definiert man die Syntax von Programmiersprachen durch Grammatiken, so ist man sowohl an der Klarheit der Beschreibung für den menschlichen Leser, aber auch an der Minimalität der Grammatik interessiert. Dieser Abschnitt beleuchtet einige Gesichtspunkte der Minimalität.

3.2.1 Reduzierte Grammatiken

Wir haben festgestellt, daß in einer kontextfreien Grammatik $CFG = (N, T, S, P)$ haben alle Produktionen $p \in P$ die Form $A \to \alpha$ mit $A \in N$ und $\alpha \in (N \cup T)^*$. D.h. die linke Seite jeder Produktion besteht aus einem einzelnen Nonterminal, also ist die Anwendung einer Regel ausschließlich von diesem Nonterminal abhängig und von der Umgebung dieses Symbols (seinem Kontext) völlig unabhängig. Eine von einer kontextfreien Grammatik CFG erzeugte Sprache $CFL(CFG)$ wird als kontextfreie Sprache bezeichnet.

Eine kontextfreie Grammatik heißt *reduziert*, wenn für alle $A \in N$ eine Ableitung

$$S \stackrel{*}{\Rightarrow} \alpha A \beta \stackrel{*}{\Rightarrow} \alpha x \beta \text{ mit } \alpha, \beta \in (N \cup T)^* \text{ und } x \in T^*$$

existiert.

Dies bedeutet, daß jedes Nonterminal A *erreichbar* vom Startsymbol aus ist ($S \stackrel{*}{\Rightarrow} \alpha A \beta$) und daß jedes Nonterminal *aktiv* ist, also eine Sequenz von Terminalsymbolen produzieren kann ($A \stackrel{*}{\Rightarrow} x$). Ein Nonterminal, das diese beiden Eigenschaften (erreichbar und aktiv) erfüllt, wird als *nützlich* bezeichnet.

In der Praxis werden naturgemäß keine Grammatiken mit nutzlosen Nonterminals verwendet, da man ja nicht die Übersetzung überflüssigen Konstrukte implementieren will.

Ohne Beweis sei angeführt, daß zu jeder kontextfreien Grammatik eine äquivalente reduzierte kontextfreie Grammatik konstruiert werden kann (für den Beweis siehe [BuMa84]). Algorithmen dazu sind etwa in [Back79] oder auch in [TrSo85] angegeben.

3.2.2 ϵ-freie Grammatiken

Eine Grammatik heißt *ϵ-frei*, wenn sie keine Regel der Form $A \rightarrow \epsilon$ (ϵ-Regel) enthält. Aus jeder kontextfreien Grammatik lässt sich eine äquivalente ϵ-freie, kontextfreie Grammatik konstruieren (für den Beweis siehe [BuMa84]).

3.2.3 Kettenregel-freie Grammatiken

Produktionen der Form $L \rightarrow R$ mit $L, R \in N$ bezeichnet man als *Kettenregeln*. Eine Kettenregel gibt an, daß die syntaktische Klasse L direkt in die Klasse R übergeführt werden kann.

Eine Grammatik wird als kettenregel-frei bezeichnet, wenn sie keine Kettenregeln enthält. Jede ϵ-freie kontextfreie Grammatik kann in eine äquivalente ϵ-freie, kettenregel-freie, kontextfreie Grammatik verwandelt werden [BuMa84].

3.3 Linksrekursion

Sei $CFG = (N, T, S, P)$ eine kontextfreie Grammatik, so bezeichnet man ein Nonterminal $A \in N$ als *linksrekursiv*, wenn eine Ableitung

$$A \stackrel{+}{\Rightarrow} A\alpha, \text{ mit } \alpha \in (N \cup T)^*$$

existiert, wenn also eine Folge von Ableitungsschritten ausgehend von einem Nonterminal A auf eine Satzform führt, deren erstes Symbol wieder A selbst ist. Diese Struktur kann von einigen Parsing-Algorithmen, und zwar von den sog. Top-Down-Methoden nicht verarbeitet werden. Man ist also daran interessiert, die Linksrekursion zu eliminieren.

Unmittelbare (direkte) Linksrekursion ist durch Produktionen der Form $A \rightarrow A\alpha$ gekennzeichnet. Sie ist klar erkennbar und auch relativ einfach zu beseitigen.

Sei $A \rightarrow A\alpha_1|A\alpha_2|\ldots|A\alpha_m|\beta_1|\beta_2|\ldots|\beta_n$ eine solche unmittelbar linksrekursive Produktion, wobei kein A Präfix der β_i ist. Die Produktionen

$$\begin{aligned} A &\rightarrow \beta_1 A'|\beta_2 A'|\ldots|\beta_n A' \\ A' &\rightarrow \alpha_1 A'|\alpha_2 A'|\ldots|\alpha_m A'|\epsilon \end{aligned}$$

ersetzen dann die linksrekursive Produktion, ohne die dadurch generierbaren Satzformen zu verändern.

Wir wollen nun die o.a. Grammatik G^1_{expr} von der Linksrekursion befreien. Die linksrekursiven Produktionen der Grammatik sind:

$$\begin{aligned} expr &\rightarrow expr + expr \\ &\quad |expr - expr \\ &\quad |expr * expr \\ &\quad |expr/expr \\ &\quad |(\ expr\ ) \\ &\quad |number \\ &\quad |identifier \\ number &\rightarrow digit \\ &\quad |number\ digit \\ identifier &\rightarrow letter \\ &\quad |identifier\ digit \\ &\quad |identifier\ letter \end{aligned}$$

Verfährt man nach der angegebenen Regel, so erhält man unmittelbar die folgenden, von Linksrekursion freien Produktionen:

$$\begin{aligned} expr &\rightarrow (\ expr\ )\ exprest \\ &\quad |number\ exprest \\ &\quad |identifier\ exprest \\ exprest &\rightarrow +expr\ exprest \\ &\quad | - expr\ exprest \\ &\quad | * expr\ exprest \\ &\quad |/expr\ exprest \\ &\quad |\epsilon \\ number &\rightarrow digit\ numrest \\ numrest &\rightarrow digit\ numrest \end{aligned}$$

$$
\begin{aligned}
& & & | \epsilon \\
& identifier & \rightarrow & \ letter\ idrest \\
& idrest & \rightarrow & \ digit\ idrest \\
& & & \ letter\ idrest \\
& & & | \epsilon
\end{aligned}
$$

Allgemeine Linksrekursion ist weit weniger augenscheinlich und auch schwerer in den Griff zu bekommen:

$$
\begin{aligned}
S &\rightarrow Aa|b \\
A &\rightarrow Ac|Sd|\epsilon
\end{aligned}
$$

Hier ist A unmittelbar linksrekursiv, da $A \stackrel{+}{\Rightarrow} Ac$. Aber auch S ist linksrekursiv, wie $S \Rightarrow Aa \Rightarrow Sda$ zeigt. Die Literatur bietet Algorithmen zur Beseitigung auch der allgemeinen Linksrekursion an [ASU86].

3.4 Normalformen

Normalformengrammatiken beschränken die Form der Produktionen und dienen der Standardisierung und Vereinfachung von kontextfreien Grammatiken.

3.4.1 Die Chomsky-Normalform

Eine kontextfreie Grammatik $CFG = (N, T, S, P)$ ist in *Chomsky-Normalform (CNF)*, wenn CFG ϵ-frei ist, und jede Produktion $p \in P$ die Form

$$A \rightarrow \alpha \text{ mit } \alpha \in (N^2 \cup T)$$

hat. Die rechten Seiten der Produktionen bestehen also aus zwei Nonterminals oder aber aus einem einzelnen Terminalsymbol. Zu jeder kontextfreien ϵ-freien Grammatik kann eine äquivalente Grammatik in CNF konstruiert werden [BuMa84].

3.4.2 Die Greibach-Normalform

Eine kontextfreie Grammatik $CFG = (N, T, S, P)$, die ϵ-frei ist, ist in *Greibach-Normalform (GNF)*, wenn jede Produktion $p \in P$ die Form

$$A \rightarrow a\alpha \text{ mit } a \in T \text{ und } \alpha \in N^*$$

besitzt. Der Präfix einer rechten Seite einer Produktion einer Grammatik in GNF ist also immer ein Terminalsymbol. Deshalb sind GNF-Grammatiken klarerweise auch frei von Linksrekursion. Zu jeder kontextfreien ϵ-freien Grammatik kann auch eine äquivalente Grammatik in GNF konstruiert werden [BuMa84].

3.5 Ableitungen in kontextfreien Grammatiken

3.5.1 Links- und Rechtsableitungen

In einer CFG treten in jedem Schritt einer Ableitung zwei Fragen auf:

- welches nichtterminale Symbol der aktuellen Satzform ist zu ersetzen?

- durch welche linke Seite wird es ersetzt?

Eine Ableitung $\gamma_1 A \gamma_2 \Rightarrow \gamma_1 \beta \gamma_2$ wird als *unmittelbare Linksableitung*, in Symbolen $\overset{L}{\Rightarrow}$ bezeichnet, wenn $A \rightarrow \beta \in P$ und $\gamma_1 \in T^*$, wenn also das am weitesten links stehende Nonterminal der Satzform ersetzt wird.

Besteht eine Ableitung aus mehreren unmittelbaren Linksableitungen, so notiert man $\overset{L*}{\Rightarrow}$ bzw. $\overset{L+}{\Rightarrow}$. Ist $S \overset{L*}{\Rightarrow} \alpha$, so wird α *Linkssatzform* genannt.

So ist z.B. $expr \Rightarrow (expr) \Rightarrow (expr + expr) \Rightarrow (number + expr) \Rightarrow (digit + expr) \Rightarrow (3 + expr) \Rightarrow (3 + identifier) \Rightarrow (3 + letter) \Rightarrow (3 + \mathtt{x})$ eine Linksableitung gemäss G^1_{expr}.

Ähnlich wie die Linksableitung kann man die Rechtsableitung $\overset{R}{\Rightarrow}$ definieren. Eine Ableitung $\gamma_1 A \gamma_2 \Rightarrow \gamma_1 \beta \gamma_2$ wird als *unmittelbare Rechtsableitung*, in Symbolen $\overset{R}{\Rightarrow}$ bezeichnet, wenn $A \rightarrow \beta \in P$ und $\gamma_2 \in T^*$. Hier wird also das am weitesten rechts stehende Nonterminal der Satzform ersetzt.

Eine Folge von unmittelbaren Rechtsableitungen wird als $\overset{R*}{\Rightarrow}$ bzw. $\overset{R+}{\Rightarrow}$ notiert. Wenn $S \overset{R*}{\Rightarrow} \alpha$, so spricht man von α als einer *Rechtssatzform*.

Eine Rechtsableitung unter G^1_{expr} wäre beispielsweise: $expr \Rightarrow (expr) \Rightarrow (expr + expr) \Rightarrow (expr + identifier) \Rightarrow (expr + letter) \Rightarrow (expr + \mathtt{x}) \Rightarrow (number + \mathtt{x}) \Rightarrow (digit + \mathtt{x}) \Rightarrow (3 + \mathtt{x})$.

3.5.2 Ableitungsbäume

Ableitungsbäume sind ein Mittel zur graphischen Visualisierung von Ableitungen von kontextfreien Grammatiken. Sie stellen die Entscheidungen bezüglich der Reihenfolge der Ersetzungen in kompakter, anschaulicher Form dar.

Ein Baum ist ein (vollständiger) Ableitungsbaum für eine kontextfreie Grammatik $CFG = (N, T, P, S)$, wenn

1) jeder Knoten des Baumes mit einem Symbol aus $(N \cup T \cup \epsilon)$ markiert ist,

2) die Wurzel des Baumes mit dem Startsymbol S markiert ist,

3) jedem inneren Knoten ein Nonterminal, jedem Blattknoten ein Terminalsymbol bzw. ϵ zugeordnet wird,

4) einem Knoten, dem ein Nonterminal A zugeordnet wurde und die Nachfolger dieses Knotens von rechts nach links mit den Symbolen $\alpha_1, \ldots, \alpha_n$ bezeichnet werden, wobei die $\alpha_1, \ldots, \alpha_n$ die rechte Seite einer Produktion $A \rightarrow \alpha_1 \ldots \alpha_n \in P$ darstellen,

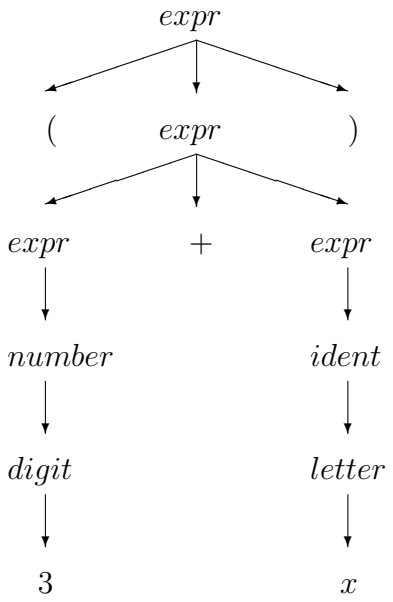

Abbildung 4: Ableitungsbaum für $(3 + x)$

5) Blätter, die mit ϵ markiert wurden, keine Geschwister besitzen.

Die Blätter eines Ableitungsbaumes ergeben, von links nach rechts gelesen, einen Satz aus $L(CFG)$. Läßt man auch Nonterminals als Blattknoten zu, so ergeben sich Bäume, die unvollständige Ableitungen, also Satzformen beschreiben.

3.5.3 Mehrdeutigkeit in Grammatiken

Es ist nun aber keineswegs so, daß zu jeden Satz einer Sprache genau ein korrespondierender Ableitungsbaum existiert. Betrachten wir den Satz $ident * ident + ident$ aus $L(G^1_{expr})$ (unter der Annahme, daß $ident \in T$, da wir uns hier nicht für die Bezeichner interessieren), so finden wir zwei verschiedene Linksableitungen:

a) $expr \Rightarrow expr{*}expr \Rightarrow ident{*}expr \Rightarrow ident{*}expr{+}expr \Rightarrow ident{*}ident{+}expr \Rightarrow ident * ident + ident$

b) $expr \Rightarrow expr + expr \Rightarrow expr * expr + expr \Rightarrow ident * expr + expr \Rightarrow ident * ident + expr \Rightarrow ident * ident + ident$

Dementsprechend existieren zwei voneinander verschiedene Ableitungsbäume, die in Abbildung 5 dargestellt sind.

Ein durch eine Grammatik generierter Satz ist *mehrdeutig (engl. ambiguous)*, wenn mehr als ein Ableitungsbaum für ihn existiert. Eine Grammtik, die mindestens einen mehrdeutigen Satz generiert, heißt *mehrdeutig*, ansonsten *eindeutig*.

Die Mehrdeutigkeit der Grammatik G^1_{expr} hat ihre Ursache darin, daß es keine Regel gibt, die den Vorrang zwischen den Operatoren beschreibt. Im Bereich des

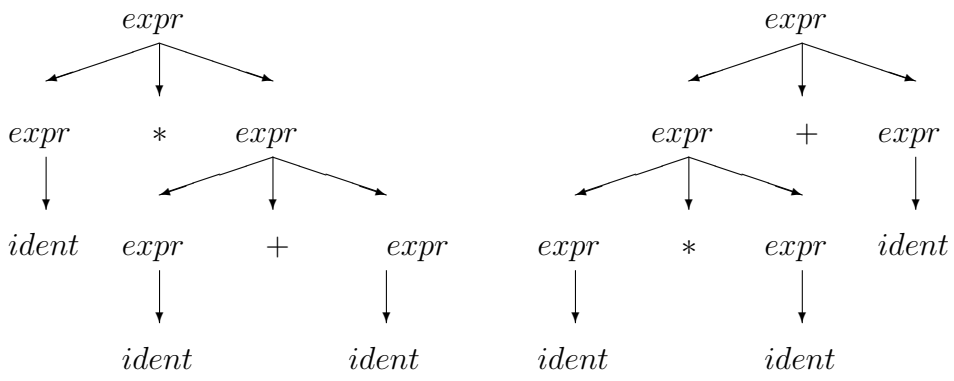

Abbildung 5: Ableitungsbäume für $ident * ident + ident$

Übersetzerbaus ist Mehrdeutigkeit äußerst unangenehm, da der Übersetzer in einem solchen Fall für ein und denselben Satz der Quellsprache verschiedene Sätze der Zielsprache generieren kann und damit Nichtdeterminismus Platz greift.

Ein beliebtes Beispiel im Zusammenhang mit Mehrdeutigkeit ist die sogenannte *dangling-else-Problematik* in blockstrukturierten Programmiersprachen. Darauf soll hier nicht näher eingegangen werden, die Literatur beschäftigt sich eingehend damit [ASU86,John86]

3.6 Praktische Darstellung kontextfreier Grammatiken

In diesem Abschnitt werden zwei weitverbreitete Darstellungsformen von kontextfreien Grammatiken erläutert, die textuelle Backus-Naur-Form und die graphische Notation mittels Syntaxdiagrammen.

3.6.1 Die Backus-Naur-Form

Diese Beschreibungsform kontextfreier Grammatiken lehnt sich strukturell stark an die Notation der Produktionsregeln der Grammatik an, ist aber, besonders in der erweiterten Form, kompakter und einfacher zu lesen. Die Backus-Naur-Form (BNF) ist eine Metasprache, eine Sprache zur Formulierung der syntaktischen Aspekte kontextfreier, formaler Sprachen.

Die Produktionsregeln der Grammatik werden mehr oder weniger direkt übernommen. Rechte und linke Seite der Produktion werden nicht durch $\rightarrow$, sondern durch ::= separiert. Existieren für ein Nonterminalsymbol mehrere rechte Seiten, so werden die Alternativen jeweils durch einen senkrechten Strich | voneinander getrennt, die linke Seite wird dabei lediglich einmal angeschrieben. Terminalsymbole werden in "" eingeschlossen.

Die Backus-Naur-Form ist selbstbezüglich, d.h. BNF kann durch BNF definiert werden:

```
syntax       ::=  rule syntax
                |  rule
rule         ::=  nonterminal "::=" expression
                |  nonterminal "::="
expression   ::=  term "|" expression
                |  term
term         ::=  factor term
                |  factor
factor       ::=  nonterminal
                |  terminal
nonterminal  ::=  string
terminal     ::=  " " " string " " "
```

Eine erweiterte Backus-Naur-Form (EBNF) wird um einige weitere Metasymbole "{", "}", "[", "]" angereichert. Die eckigen Klammern schließen optionale Teile der rechten Produktionsseiten ein, die entweder angegeben oder weggelassen werden können. In geschwungene Klammern werden Symbolfolgen eingeschlossen, die beliebig oft wiederholt werden können (auch Null mal). Diese Erweiterungen finden breite Anwendung, da sie die Formulierung von Wiederholungen auf natürlichere Weise zulassen, als das mit der Rekursion in der Grundform von BNF möglich ist. Das wird auch durch die folgende Definition von EBNF in EBNF bestätigt, die deutlich kürzer, aber nicht unklarer ist, als die Beschreibung der vom Umfang kleineren BNF in BNF.

```
syntax       ::=  rule { rule }
rule         ::=  nonterminal "::=" [ expression ]
expression   ::=  term { "|" term }
term         ::=  factor { factor }
factor       ::=  nonterminal
                |  terminal
                |  "[" expression "]"
                |  "{" expression "}"
nonterminal  ::=  string
terminal     ::=  " " " string " " "
```

3.6.2 Syntaxdiagramme

Diese Diagramme sind eine Methode, kontextfreie Grammatiken in klarer und übersichtlicher Weise darzustellen. Dabei wird für jedes Nichtterminalsymbol der Grammatik ein gerichteter Graph konstruiert. Dieser Graph hat genau eine Eingangskante und eine Ausgangskante. Die Knoten des Graphen repräsentieren die Grammatiksymbole auf den rechten Seiten der Produktionen des dem Graphen zugeordneten Nonterminals. Es gibt zwei Arten von Knoten: Kreise oder Ellipsen stellen Terminalsymbole dar, während Nonterminals mit Rechtecken symbolisiert werden. Die Knoten werden mit den Symbolen bzw. deren Namen beschriftet. Die Kanten zwischen den Knoten beschreiben die syntaktisch korrekten Wege durch die von den

Diagrammen repräsentierte Grammatik. In Abbildung 6 sind die der Grammatik G^1_{expr} entsprechenden Syntaxdiagramme angegeben.

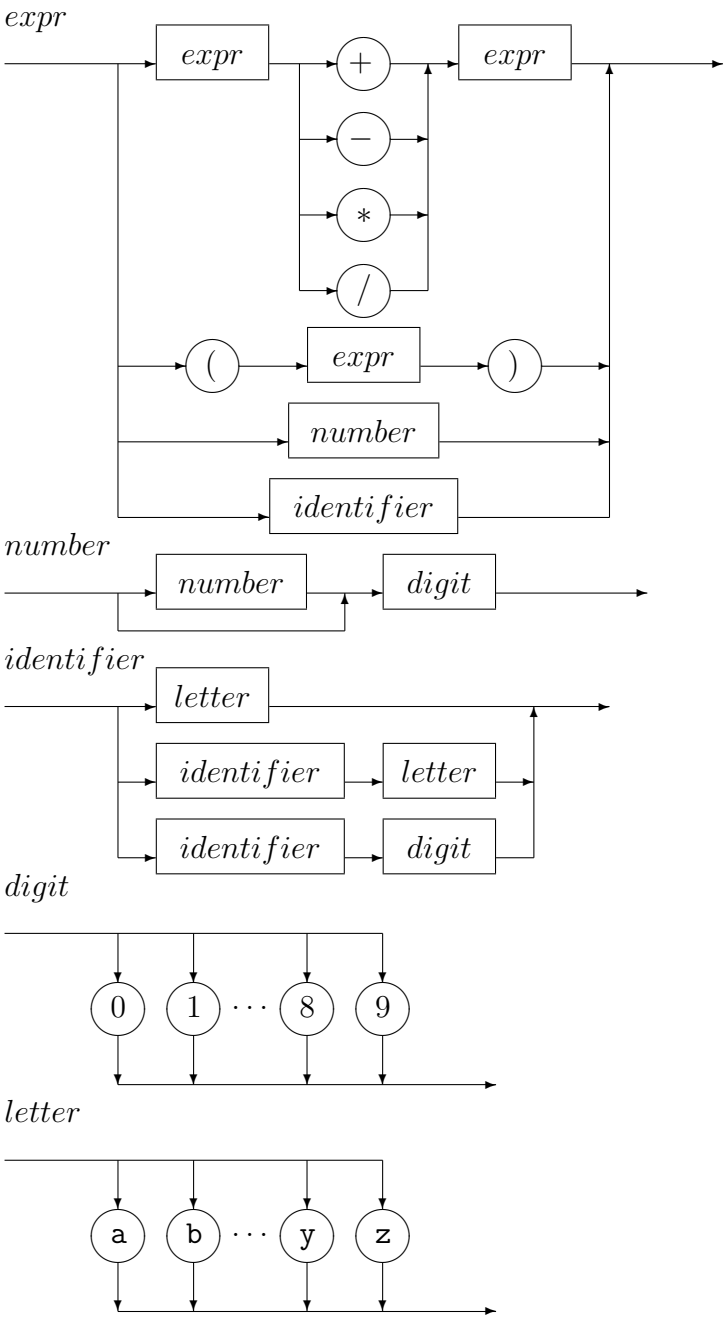

Abbildung 6: Syntaxdiagramme für G^1_{expr}

4 Compiler

Ein *Compiler* ist ein Algorithmus, der eine Sprache A in eine Sprache B übersetzt. In der Praxis handelt es sich bei A meist um eine (höhere) Programmiersprache (z.B. PASCAL, MODULA-2) und bei B um eine maschinennahe Sprache (Assembler, Maschinensprache). Die Sprache A wird auch als Quellsprache und B als Zielsprache des Compilers bezeichnet. Ein Compiler realisiert also eine Abbildung

$$\text{Quellsprache A} \rightarrow \text{Zielsprache B}$$

wobei jedem $a \in A$ (Quellprogramm) genau ein $b \in B$ (Zielprogramm) zugeordnet wird.

In vielen Compilern wird die Übersetzung in den in Abbildung 7 dargestellten Phasen durchgeführt [ASU86].

4.1 Lexikalische Analyse

Damit der syntaktische Aufbau eines Programmes anlysiert werden kann, wird in der *lexikalischen Analyse* die textuelle Repräsentation des Quellprogrammes (= eine Folge von Zeichen) in eine Folge von Terminalsymbolen, die sogenannten *Token*, übergeführt. So könnte z.B. der folgende Ausdruck

$$way := speed * time/2$$

in die Tokenfolge

$$\textit{identifier assignop identifier times identifier divop number}$$

umgeformt werden.

Eine Sequenz von Zeichen, die ein syntaktisches Symbol textuell repräsentiert, wird *Lexem* dieses Tokens genannt (z.B. sind "way", "speed" oder "time" Lexeme für das Terminalsymbol *identifier*).

Der Lexikalische Analysator oder Scanner hat die Aufgabe Lexeme auf die entsprechenden Tokens abzubilden. Diese im Prinzip einfache Aufgabe löst man im allgemeinen mit einem deterministischen endlichen Automaten (DEA).

4.2 Syntaktische Analyse

Die *Syntaktische Analyse* (Parser) hat in einem Compiler die Funktion, die vom Scanner gelieferte Folge von Tokens in einen Ableitungsbaum zu übertragen um dadurch zu entscheiden, ob die vorliegende Tokenfolge ein syntaktische korrektes Wort der Quellsprache ist.

Der Ableitungsbaum für den obigen Ausdruck ist aus Abbildung 8 ersichtlich.

Algorithmen die die Umsetzung der Tokenfolge in einen Ableitungsbaum durchführen werden in den folgenden Abschnitten noch ausführlich dargestellt.

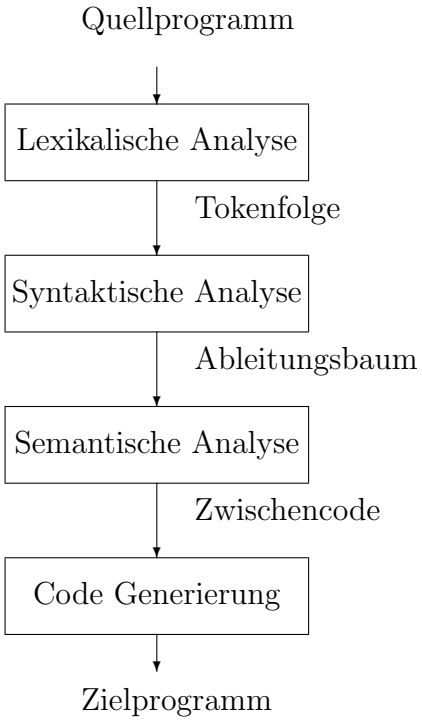

Abbildung 7: Phasen eines einfachen Compilers.

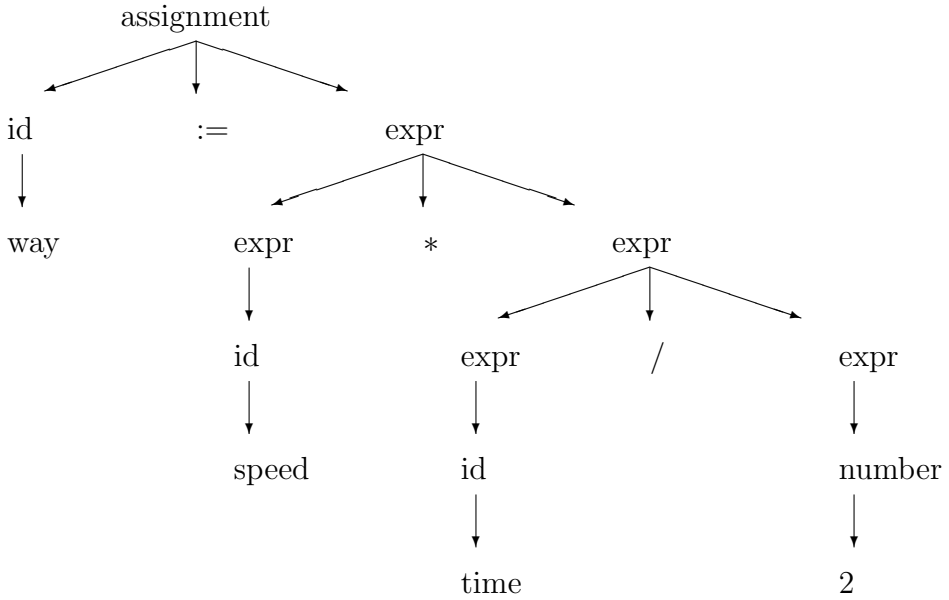

Abbildung 8: Ableitungsbaum für $way := speed * time / 2$.

4.3 Semantische Analyse

Die Syntax von Programmiersprachen wird meist durch kontextfreie Grammatiken beschreiben. Allerdings müssen in solchen Sprachen im allgemeinen eine Reihe von Bedingungen erfüllt werden, die durch solche Grammatiken nicht ausgedrückt werden können. Beispielsweise die Bedingung, daß ein Bezeichner (Identifier) zuerst deklariert werden muß bevor er verwendet werden darf oder, daß eine Operation nur auf bestimmte Datentypen angewandt werden darf (z.B. für Bezeichner vom Typ BOOLEAN sind nur die Operationen AND, OR oder NOT erlaubt).

Derartige Bedingungen werden während der *semantischen Analyse* überprüft, die dazu den Ableitungsbaum aus der syntaktischen Analyse verwendet. Am Ende diese Phase wird eventuell ein Zwischencode erzeugt, der für den als Beispiel verwendeten Ausdruck folgende Form haben kann:

$$\begin{aligned} temp1 &:= time/2 \\ temp2 &:= speed * temp1 \\ way &:= temp2 \end{aligned}$$

4.4 Code Generierung

Die letzte Phase eines Compilers besteht aus der *Code-Generierung* . Hier werden Speicherplätze bzw. Register für die Variablen reserviert und das Programm wird in die entsprechenden Maschineninstruktionen übersetzt. Eine entsprechende Sequenz von Instruktionen für den oben verwendeten Ausdruck könnte bei der Verwendung von zwei Registern R1 und R2 folgendermaßen aussehen:

$$\begin{aligned} MOV &\quad R2, &\quad time \\ DIV &\quad R2, &\quad 2 \\ MOV &\quad R1, &\quad speed \\ MUL &\quad R1, &\quad R2 \\ MOV &\quad way, &\quad R1 \end{aligned}$$

5 Top-Down Parsing

In diesem Abschnitt wird das Konzept des Top-Down Parsing erläutert und gezeigt, wie ein effizienter Top-Down Parser aus einer Grammatik konstruiert werden kann. *Top-Down Parsing* ist der Versuch, eine Linksableitung für eine gegebene Folge von Tokens (Input) zu finden oder mit anderen Worten die Konstruktion des Ableitungsbaumes in Preorder.

In den meisten Fällen haben Grammatiken für Programmiersprachen die Eigenschaft, daß der entsprechende Parser den Input nur ein einziges mal von links nach rechts lesen muß und dabei immer nur ein Token vorausschauen (lookahead) muß, um den korrekten Ableitungsbaum zu konstruieren. Solche Grammatiken werden auch als LL(1) Grammatiken bezeichnet, wobei das erste "L" für das Lesen des Inputs von links nach rechts steht und das zweite "L" bedeutet, daß man nach links ableitet, indem man in jedem Schritt des Parsingprozesses genau "1" Symbol vorausschaut. Parser für solche Grammatiken werden auch als Predictive Parser bezeichnet.

Hat eine Produktion $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ in einer solchen Grammatik mehrere Alternativen, so muß ein einzelnes Input-Symbol a ausreichen, um die richtige Alternative α_i auszuwählen, die eine Folge von Token ableitet, die mit a beginnt. In Programmiersprachen erfüllen die einzelnen Schlüsselwörter die Funktion die verschiedenen Alternativen unterscheidbar zu machen. Beispielsweise in der Produktion

stmt → IF *expr* THEN *stmtseq* ELSE *stmtseq* END
| WHILE *expr* DO *stmtseq* END
| REPEAT *stmtseq* UNTIL *expr*

zeigen die Schlüsselwörter IF, WHILE und REPEAT welche Alternative die (einzig) richtige ist.

Ermöglicht es eine Grammatik nicht sofort die richtige Alternative zu erkennen, so kann man einen Parser konstruieren, der versucht mit Hilfe von Backtracking (d.h. im Prinzip mehrfaches Lesen des Inputs) den richtigen Ableitungsbaum zu konstruieren. Parser die Backtracking benutzen sind jedoch eher selten, da sie einerseits für Programmiersprachen nicht gebraucht werden und weil andererseits Backtracking nicht sehr effizient ist. Es werden daher im folgenden nur solche Grammatiken untersucht, für die kein Backtracking erforderlicht ist.

5.1 Recursive-Descent Parsing

Recursive-Descent Parsing ist eine Top-Down Methode der Syntax-Analyse bei der rekursive Prozeduren verwendet werden, um den Input zu verarbeiten. Im Prinzip wird jedem Nichtterminal in der Grammatik eine Prozedur zugeordnet. Wie bereits erwähnt wird hier eine spezielle Form des Recursive-Descent Parsing erläutert, bei der jenes Symbol um das vorausgeschaut wird (lookahead symbol), eindeutig die Prozedur bestimmt, die aufgerufen werden muß. Somit bestimmt die Folge der Prozeduraufrufe implizit den Ableitungsbaum für den gegebenen Input.

Eine Grammatik für einen Recursive-Descent Parser darf keine Linksrekursion enthalten, da dies den Parser veranlassen könnte in eine Endlosschleife zu gehen. So ist beispielsweise die folgende Produktion

$$A \rightarrow A\alpha \mid \beta$$

linksrekursiv. Linksrekursion kann jedoch in einer Grammatik eliminiert werden, indem die Produktionen umgeformt werden. Dies würde bei obigem Beispiel folgendes ergeben:

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \epsilon \end{aligned}$$

Es wurde also ein neues Nichtterminal-Symbol R eingeführt und die Linksrekursion wurde in eine Rechtsrekursion umgewandelt.

Im folgenden wird nun ein Recursive-Descent Parser konstruiert, der folgende Grammatik für einfache Ausdrücke erkennt:

```
Expression ::= Term { "+" Term | "−" Term }.
Term       ::= Factor { "∗" Factor | "/" Factor }.
Factor     ::= Number | Identifier | "(" Expression ")".
Identifier ::= "a" | "b" | ... | "z" | "A" | "B" | ... | "Z".
Number     ::= "0" | "1" | ... | "9".
```

```
MODULE RecursiveDescentParser;

FROM InOut IMPORT WriteString, WriteLn, Read, Write, EOL;

VAR
    Token : CHAR;

(**********************************************)

PROCEDURE Lookahead;
BEGIN (* Lookahead *)
  Read(Token);
  Write(Token);
  WHILE (Token = " ") DO (* skip blanks *)
    Read(Token);
    Write(Token);
  END (* while *);
END Lookahead;
(**********************************************)
```

```
PROCEDURE Factor;
BEGIN (* Factor *)
   CASE Token OF
      "a".."z" : Lookahead; (* identifier *)
   |  "A".."Z": Lookahead; (* identifier *)
   |  "0".."9" : Lookahead; (* number *)
   |  "("      : Lookahead; (* '(' Expression ')' *)
                 Expression;
                 IF(Token = ")") THEN
                   Lookahead;
                 ELSE (* error *)
                   WriteLn;
                   WriteString(" ')' expected."); WriteLn;
                 END (* if *);
   ELSE (* error *)
      WriteLn;
      WriteString("Error in expression."); WriteLn;
   END (* case *);
END Factor;
(**********************************************)


PROCEDURE Term;
BEGIN (* Term *)
   Factor;
   WHILE (Token = "*") OR (Token = "/") DO
      Lookahead;
      Factor;
   END (* while *);
END Term;
(**********************************************)


PROCEDURE Expression;
BEGIN (* Expression *)
   Term;
   WHILE (Token = "+") OR (Token = "-") DO
      Lookahead;
      Term;
   END (* while *);
END Expression;
(**********************************************)


BEGIN (* RecursiveDescentParser *)
   WriteString("> ");
```

```
    Lookahead;
    Expression;

    WriteLn;
    IF(Token = EOL) THEN
      WriteString("Expression accepted.");
    ELSE (* error *)
      WriteString("EOL expected.");
    END (* if *);
    WriteLn;
END RecursiveDescentParser.
```

Wie bereits erwähnt, wird einfach für jedes Nonterminalsymbol eine Prozedur geschrieben. Der konstruierte Parser besteht daher aus den Prozeduren *Expression, Term, Factor* und der zusätzlichen Prozedur *Lookahead* . Der Einfachheit halber wurde auf Prozeduren für *Identifier* und *Number* verzichtet und deren Funktion direkt in die Prozedur *Factor* integriert. Die Prozedur *Lookeahed* simuliert den Scanner und liefert bei jedem Aufruf das nächste Token. Da die Bezeichner, die Zahlen und die Operationszeichen in der Grammatik der Einfachheit halber alle nur aus 1 Zeichen bestehen dürfen, kann das jeweilige Zeichen direkt als Token verwendet werden (d.h. Lexem = Token).

Betrachten wir nun die Operationsweise des Parsers, wenn der Input aus dem folgenden String

$$a + b * c\$$$

besteht, wobei das Zeichen \$ das Ende des Inputs anzeigt. Im oben angegebenen Programm wird \$ durch die Konstante EOL (end of line) repräsentiert.

Der Parsingprozeß beginnt mit einem Aufruf der Prozedur Lookahead, die das erste Token "a" liefert. Daraufhin werden nacheinander die Prozeduren Expression, Term und Factor aufgerufen. Die Prozedur Factor stellt fest, ob das aktuelle Token a ein an dieser Stelle gültiges Terminalsymbol ist. Da dies der Fall ist wird die Prozedur Lookahead aufgerufen, welche als nächstes Token "+" liefert. Man beachte, daß nun dieses Token eindeutig die Prozedur bestimmt, die als nächstes aufgerufen werden muß. Es erfolgt daher die Rückkehr in die Prozedur Expression, wo festgestellt wird, daß hier das Token "+" paßt (*matched*). Danach wird das nächste Token "b" angefordert, woraufhin wiederum die Prozeduren Term und Factor aufgerufen werden. Nachdem Factor festgestellt hat, daß "b" matched wird als nächstes Token "$*$" verarbeitet. Man erkennt, daß "$*$" bereits in der Prozedur Term matched woraufhin wiederum die Prozedur Factor mit dem Token "c" aufgerufen wird. In Factor liefert nun der Aufruf von Lookahead das Token "\$". Da dieses Token in keiner Prozedur matched, erfolgt die Rückkehr ins Hauptprogramm, wo gemeldet wird, daß der Ausdruck akzeptiert wird.

Auf den ersten Blick scheint der oben angegenben Algorithmus nicht sehr viel Ähnlichkeit mit einem Kellerautomaten zu besitzen. Dennoch ist das angegebene

Programm im Prinzip ein Kellerautomat, wobei die Prozedur Lookahead den Lesekopf für das Eingabeband simuliert, die restlichen Prozeduren die "ausprogrammierte" Übergangsfunktion darstellen und der Stack implizit durch die rekursiven Prozeduraufrufe simuliert wird.

5.2 Nichtrekursives Top-Down Parsing

Im vorhergehenden Abschnitt wurde ein Parser vorgestellt, der die syntaktische Korrektheit von arithmetischen Ausdrücken mit Hilfe von rekursiven Prozeduren untersucht hat. Es ist allerdings auch möglich einen nichtrekursiven Parser zu konstruieren, der genau dieselbe Sprache akzeptiert, indem explizit ein Stack verwaltet wird. Das Hauptproblem ist dabei die Bestimmung der entsprechenden Produktion, die für ein jeweils vorhandenes Nichtterminalsymbol angewendet werden muß.

5.2.1 Die LL-Maschine

Ein nichtrekursiver Top-Down Parser oder Tabellengesteuerter Predictive Parser besteht aus einem Eingabeband, einem Stack und einer Tabelle, die die einzelnen Produktionen enthält (Abbildung 9). Auf dem Eingabeband steht der String der untersucht werden soll, gefolgt von dem Symbol \$, das das rechte Ende des Eingabebandes darstellt. Auf dem Stack werden Grammatiksymbole (Terminalsymbole und Nichtterminalsymbole) abgelegt, wobei das unterste Symbol ebenfalls das Zeichen \$ ist. Am Anfang enthält der Stack als oberstes Symbol das Startsymbol der Grammatik, gefolgt von \$. Die Tabelle ist ein zweidimensionales Feld M, wobei jeder Eintrag durch $M[A, a]$ angesprochen werden kann. Bei A handelt es sich dabei um ein Nichtterminalsymbol und a ist entweder ein Terminalsymbol oder das Bandendesymbol \$.

5.2.2 LL-Parsing

Der Parsing-Algorithmus arbeitet nun auf folgende Weise. Es wird einerseits das oberste Stacksymbol X und andererseits das aktuelle Token a verwendet, um zu bestimmen, welcher Verarbeitungsschritt durchzuführen ist. Dabei gibt es die folgenden drei Möglichkeiten:

1) Wenn $X = a = \$$ dann akzeptiert der Parser die Eingabe, d.h. es wurde ein gültiger arithmetischer Ausdruck erkannt.

2) Wenn $X = a \neq \$$, dann wird X vom Stack entfernt und das nächste Token angefordert (lookahead).

3) Wenn es sich bei X um ein Nichtterminalsymbol handelt, dann wird der Tabelleneintrag $M[X, a]$ verwendet, der entweder eine Produktion der Form $X \rightarrow \alpha$ enthält oder einen speziellen Eintrag, der einen Fehler anzeigt. Ist z.B. $M[X, a] = \{X \rightarrow UVW\}$ so ersetzt der Parser das Symbol X auf dem Stack durch WVU (mit U als neuem obersten Stacksymbol). Wenn $M[X, a] = error$, so wird eine Fehlermeldung ausgegeben.

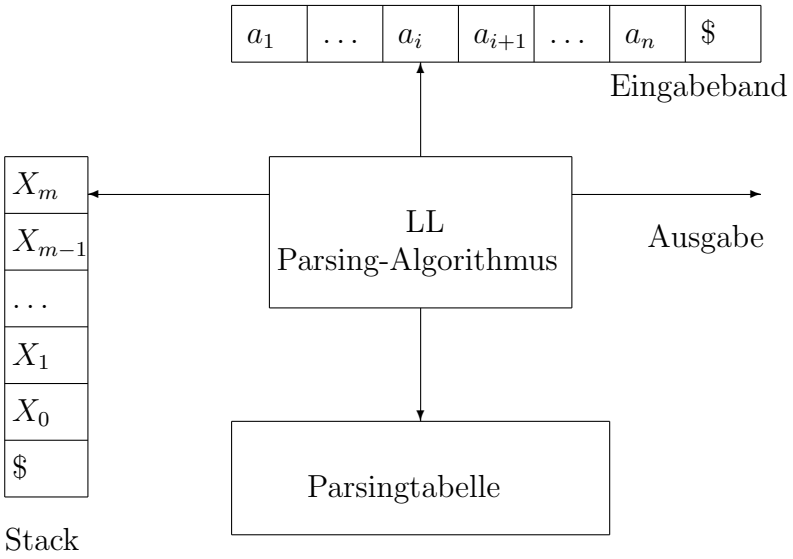

Abbildung 9: Der LL(1)-Automat

Das Verhalten des Parsers wird also durch die aktuelle *Konfiguration* bestimmt, welche durch die Symbole auf dem Stack und den noch abzuarbeitenden Input gegeben ist. Um ein Wort ω zu parsen, beginnt der Parser mit der Startkonfiguration, in der sich $\$S$ auf dem Stack befindet, wobei S das Startsymbol der Grammatik ist, und mit $\omega\$$ auf dem Eingabeband. Der Parsing Algorithmus [ASU86] arbeitet nun den Input unter Verwendung der Parsingtabelle M in folgender Art und Weise ab:

```
PROCEDURE NonrecursiveParser;
BEGIN (* NonrecursiveParser *)
    Setze den Lesekopf auf das erste Symbol von ω$;
    REPEAT
        Sei X das oberste Stacksymbol und a jenes Symbol auf dem der
        Lesekopf des Automaten steht;
        IF X ist ein Terminalsymbol oder $ THEN
          IF X = a THEN
            Entferne X vom Stack und bewege den Lesekopf um ein Symbol
            nach rechts;
          ELSE
            Gib eine Fehlermeldung aus;
          END (* if *);
        ELSE (* X ist ein Nichtterminalsymbol *)
          IF M[X, a] = X → Y1, Y2, ..., Yk THEN
            Entferne X vom Stack;
            Gib Yk, Yk−1, ..., Y1 auf den Stack, mit Y1 als
            oberstes Symbol;
```

```
                ELSE
                  Gib eine Fehlermeldung aus;
                END (* if *);
              END (* if *);
        UNTIL X = $ (* Der Stack ist leer *)
END NonrecursiveParser;
```

Betrachten wir nun nochmals die Grammatik für arithmetische Ausdrücke aus dem vorhergehenden Abschnitt. Da dort die einzelnen Produktionen in Prozeduren umgewandelt wurden, konnten Schleifen verwendet werden, um Wiederholungen in der Grammatik auszudrücken. Da hier die Produktionen in eine Tabellenform umgewandelt werden, muß auch die Grammatik entsprechend umgeformt werden. Dies erfolgt dadurch, daß die "Schleifen" in der BNF in Rekursionen umgewandelt werden. Es ergibt sich somit die folgende zum vorhergehenden Abschnitt äquivalente Grammatik für arithmetische Ausdrücke:

$$
\begin{aligned}
Expression &\rightarrow Term\ Expression' \\
Expression' &\rightarrow \text{"}+\text{"}\ Term\ Expression' \mid \\
&\quad\ \text{"}-\text{"}\ Term\ Expression' \mid \epsilon \\
Term &\rightarrow Factor\ Term' \\
Term' &\rightarrow \text{"}*\text{"}\ Factor\ Term' \mid \\
&\quad\ \text{"}/\text{"}\ Factor\ Term' \mid \epsilon \\
Factor &\rightarrow \text{"}(\text{"}\ Expression\ \text{"})\text{"} \mid \\
&\quad\ Identifier \mid Number \\
Identifier &\rightarrow \text{"}a\text{"} \mid \text{"}b\text{"} \mid \ldots \mid \text{"}z\text{"} \mid \\
&\quad\ \text{"}A\text{"} \mid \text{"}B\text{"} \mid \ldots \mid \text{"}Z\text{"} \\
Number &\rightarrow \text{"}0\text{"} \mid \text{"}1\text{"} \mid \ldots \mid \text{"}9\text{"}
\end{aligned}
$$

5.2.3 Konstruktion der Parsing-Tabellen

Im folgenden wollen wir uns nun mit der Frage beschäftigen, wie aus der vorliegenden Grammatik eine Tabelle M konstruiert werden kann, mit der dann der oben angegebene Algorithmus einen arithmetischen Ausdruck verarbeiten kann. Zuerst wollen wir eine intuitive Beschreibung der Konstruktionsmethode geben. Wie bereits erwähnt verwendet ein Predictive Parser das sogenannte lookahead symbol, um eindeutig jene Produktion zu bestimmen, die bei einer gegebenen Konfiguration anzuwenden ist. Die Idee bei der Konstruktion einer Parsingtabelle ist nun, alle jene Terminalsymbole zu bestimmen, mit der eine rechte Seite einer Produktion $A \rightarrow \alpha$ beginnen kann. Beginnt also α mit dem Terminalsymbol a, so wird die Produktion $A \rightarrow \alpha$ in $M[A, a]$ aufgenommen. Dies ist jedoch noch nicht ausreichend, da

$\alpha \stackrel{*}{\Rightarrow} \epsilon$ ableiten könnte. Ist dies der Fall, wird die Produktion $A \rightarrow \alpha$ in $M[A, b]$ aufgenommen, wenn das Terminal b in irgendeiner rechten Seite einer Produktion auf A folgen kann. Gibt es kein solches Terminal b, so wird $A \rightarrow \alpha$ in $M[A, \$]$ aufgenommen, wobei \$ das Endesymbol des Inputs ist. Alle noch undefinierten Tabelleneinträge werden dann mit einem Fehlereintrag versehen, d.h. sie beschreiben eine ungültige Konfiguration.

Um den Algorithmus für die Konstruktion der Parsing-Tabellen [ASU86] formal zu beschreiben, wollen wir uns zwei Funktionen FIRST und FOLLOW definieren. Wenn α eine Folge von Grammatiksymbolen ist, dann sei FIRST(α) jene Menge von Terminalsymbolen, mit denen Ableitungen von α beginnen können. Wenn $\alpha \stackrel{*}{\Rightarrow} \epsilon$, dann sei auch ϵ in FIRST(α). Wenn A ein Nichtterminalsymbol ist, dann sei FOLLOW(A) jene Menge von Terminalsymbolen a, für die es eine Ableitung der Form $S \stackrel{*}{\Rightarrow} \alpha A a \beta$ gibt. Man beachte, daß es während des Ableituntsvorganges sehr wohl Symbole zwischen A und a geben darf, die aber auf ϵ ableiten müssen. Wenn A das am weitesten rechts stehende Symbol sein kann, dann ist auch \$ in FOLLOW(A).

Um FIRST(X) für alle Grammatiksymbole X zu berechnen, müssen die folgenden Regeln angewendet werden:

1) Wenn X ein Terminalsymbol ist, dann FIRST(X) = $\{X\}$.

2) Wenn $X \rightarrow \epsilon$, dann ist auch ϵ in FIRST(X).

3) Wenn X ein Nichtterminalsymbol ist und es eine Produktion der Form $X \rightarrow Y_1 Y_2 \ldots Y_k$ gibt, dann ist a in FIRST(X), wenn es ein i gibt, für das a in FIRST(Y_i) ist und ϵ in allen FIRST(Y_1), ..., FIRST(Y_{i-1}) ist. Das heißt, daß $Y_1 \ldots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. Wenn ϵ in FIRST(Y_j) ist, für $j = 1, 2, \ldots, k$ dann ist ϵ auch in FIRST(X). Es wird also FIRST(Y_1) berechnet, wobei gilt, daß alle Elemente von FIRST(Y_1) auch in FIRST(X) sind, wenn Y_1 nicht auf ϵ ableitet. Gilt jedoch $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, dann wird FIRST($Y_2$) berechnet, usw.

Um FOLLOW(A) für alle Nichtterminalsymbole A zu berechnen, müssen die folgenden Regeln angewendet werden:

1) Das Symbol \$ ist in FOLLOW(S), wobei S das Startsymbol der Grammatik bezeichnet und \$ das Endesymbol des Inputs ist.

2) Wenn es eine Produktion der Form $A \rightarrow \alpha B \beta$ gibt, dann sind alle Elemente in FIRST(β), außer ϵ, auch in FOLLOW(B).

3) Wenn es einer Produktion der Form $A \rightarrow \alpha B$ oder $A \rightarrow \alpha B \beta$, wobei FIRST($\beta$) ϵ enthält (d.h. $\beta \stackrel{*}{\Rightarrow} \epsilon$), dann sind alle Elemente in FOLLOW(A) auch in FOLLOW(B).

Für die oben angegebene Grammatik für arithmetische Ausdrücke würden sich folgende Mengen ergeben, wobei der Einfachheit halber sämtliche Identifier auf das Symbol *id* und alle Zahlen auf das Symbol *no* abgebildet werden:

$$\text{FIRST}(Expression) = \text{FIRST}(Term) = \text{FIRST}(Factor) = \{(,\ id,\ no\}.$$

FIRST($Expression'$) = $\{+,\ -,\ \epsilon\}$.

FIRST($Term'$) = $\{*,\ /,\ \epsilon\}$.

FOLLOW($Expression$) = FOLLOW($Expression'$) = $\{),\ \$\}$.

FOLLOW($Term$) = FOLLOW($Term'$) = $\{+,\ -,\ ),\ \$\}$.

FOLLOW($Factor$) = $\{+,\ -,\ *,\ /,\ ),\ \$\}$.

Angenommen, es gibt eine Produktion der Form $A \rightarrow \alpha$, mit a in FIRST(α). Ist dies der Fall, so wird der LL-Parser A durch die rechte Seite α ersetzen, wenn das gegebene Input-Symbol a ist. Ein Problem ergibt sich nur dann, wenn $\alpha = \epsilon$ oder $\alpha \stackrel{*}{\Rightarrow} \epsilon$. In diesem Fall wird A durch α ersetzt, wenn das gegebene Input-Symbol in FOLLOW(A) ist, oder wenn das Ende des Inputs erreicht ist und auch \$ in FOLLOW(A) enthalten ist.

Um die Parsingtabelle M auszufüllen werden für jede Produktion $A \rightarrow \alpha$ in einer Grammatik die folgenden Schritte durchgeführt:

1) Für jedes Terminalsymbol a in FIRST(α) wird in $M[A,\ a]$ die Produktion $A \rightarrow \alpha$ eingetragen.

2) Wenn ϵ in FIRST(α) enthalten ist, dann wird die Produktion $A \rightarrow \alpha$ in $M[A,\ b]$ eingetragen, für alle Terminalsymbole b, die in FOLLOW(A) enthalten sind. Wenn ϵ in FIRST(α) und \$ in FOLLOW(A) enthalten ist, dann wird $A \rightarrow \alpha$ in $M[A,\ \$]$ eingetragen.

3) Jeder noch undefinierte Eintrag in M ergibt sich zu *error*.

Für die oben angegebene Grammatik ergibt sich somit folgende Parsingtabelle [ASU86], wobei die einzelnen Nichtterminalsymbole auf ihren Anfangsbuchstaben verkürzt wurden. Der Eintrag für Identifier wurde wieder in Factor (F) eingebunden, wobei jeder Identifier auf das Token *id* abgebildet wird. Auf Einträge für Number, "-" und "/" wurde verzichtet, da diese äquivalent zu den Einträgen für Identifier, "+" und "∗" sind.

Nicht-terminale	Token					
	id	"+"	"∗"	"("	")"	\$
E	E → TE'	error	error	E → TE'	error	error
E'	error	E' → "+"TE'	error	error	E' → ϵ	E' → ϵ
T	T → FT'	error	error	T → FT'	error	error
T'	error	T' → ϵ	T' → "*"FT'	error	T' → ϵ	T' → ϵ
F	F → id	error	error	F → "(" E ")"	error	error

Parsingtabelle für arithmetische Ausdrücke.

Im folgenden werden nun die Bewegungen des Parsers für den Input-String $a\ +\ b * c$ gezeigt [ASU86], wobei der Stack, das Eingabeband und die jeweils angewendete Produktion dargestellt werden. Am Eingabeband wird immer nur jener Teil des Inputs dargestellt, der noch abgearbeitet werden muß.

STACK	INPUT	PRODUKTION
$\$E$	$a + b * c\$$	
$\$E'T$	$a + b * c\$$	$E \rightarrow TE'$
$\$E'T'F$	$a + b * c\$$	$T \rightarrow FT'$
$\$E'T'id$	$a + b * c\$$	$F \rightarrow id$
$\$E'T'$	$+ b * c\$$	
$\$E'$	$+ b * c\$$	$T' \rightarrow \epsilon$
$\$E'T''+''$	$+ b * c\$$	$E' \rightarrow'' +''TE'$
$\$E'T$	$b * c\$$	
$\$E'T'F$	$b * c\$$	$T \rightarrow FT'$
$\$E'T'id$	$b * c\$$	$F \rightarrow id$
$\$E'T'$	$* c\$$	
$\$E'T'F''*''$	$* c\$$	$T' \rightarrow'' *''FT'$
$\$E'T'F$	$c\$$	
$\$E'T'id$	$c\$$	$F \rightarrow id$
$\$E'T'$	$\$$	
$\$E'$	$\$$	$T' \rightarrow \epsilon$
$\$$	$\$$	$E' \rightarrow \epsilon$

Man erkennt an Hand der verwendeten Produktionen, daß der Parser eine Linksableitung für den Input produziert.

Zum Abschluß sei nochmals erwähnt, daß in diesem Abschnitt nur LL(1) Grammatiken behandelt wurden. Bei Grammatiken die diese Eigenschaft nicht haben, also z.B. Grammatiken mit Linksrekursionen oder bei Grammatiken die keinen eindeutigen Ableitungsbaum erzeugen, ergeben sich mehrfach definierte Einträge in den Parsingtabellen. Solche Grammatiken können daher von einem Predictive Parser nicht bearbeitet werden. Die Hauptschwierigkeit bei der Verwendung eines Predictive Parsers ist daher die Konstruktion einer geeigneten Grammatik. Eine Grammatik G ist genau dann eine LL(1) Grammatik, wenn die Produktionen in G die folgenden Bedingungen erfüllen [Wirth84]:

1) Für jede Produktionsregel $A = \alpha_1|\alpha_2|\ldots|\alpha_n$ der Grammatik muß gelten: $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$ für alle $i \neq j$.

2) Wenn $A \stackrel{*}{\Rightarrow} \epsilon$, so muß $\text{FIRST}(A) \cap \text{FOLLOW}(A) = \emptyset$.

Recht einfach hingegen ist das Erzeugen eines Parsers, wenn ein Parser-Generator verfügbar ist. Solche Werkzeuge verwenden im allgemeinen LR-Grammatiken und erzeugen Parser, welche nach der Bottom-Up Methode versuchen den Ableitungsbaum zu erzeugen. Solche Parsingmethoden werden im nächsten Abschnitt behandelt.

6 Bottom-Up-Parsing

6.1 Prinzip der Bottom-Up-Methoden

Im Gegensatz zu den Top-Down-Methoden beginnen die Bottom-Up-Methoden des Parsings mit einem gegebenen String und versuchen diesen auf das Startsymbol der vorliegenden kontextfreien Grammatik zu reduzieren. In jedem Reduktionsschritt wird ein mit der rechten Seite einer Produktion übereinstimmender Substring zur linken Seite dieser Produktion reduziert. Bei geeigneter Wahl der zu ersetzenden Teilworte und der ersetzenden Nonterminalsymbole wird eine umgekehrte Rechtsableitung konstruiert.

6.2 Shift-Reduce-Parsing

6.2.1 Handles und Shift-Reduce-Parsing

Ein zentraler Begriff im Bottom-Up-Parsing ist der des *Handles* (engl. Henkel, Griff). Ein Handle einer Rechtssatzform γ der Gestalt αAw ist eine Produktionsregel $A \to \beta$ und eine Zahl, die die Position von β in γ angibt.

Wenn $S \overset{R*}{\Rightarrow} \alpha Aw \overset{R}{\Rightarrow} \alpha\beta w$ mit $w \in T^*$, dann ist β in der Position nach α ein Handle von $\alpha\beta w$.

Ist die zugrundeliegende Grammatik eindeutig, so besitzt jede Rechtssatzform genau ein Handle.

Ein Handle kann auch als der am weitesten links stehende vollständige Teilbaum eines Ableitungsbaumes angesehen werden.

Der Prozeß der Reduktion eines Handles wird als *Pruning* (engl. wegschneiden, stutzen) bezeichnet.

Bottom-Up-Parsing ist also eine Reihe von Pruning-Schritten, die die Umkehrung einer Rechtsableitung ergeben.

Bottom-Up-Parser lesen in jedem Schritt ein Symbol des Eingabestrings. Nun muß entschieden werden, ob der Suffix des bisher gelesenen Substrings ein Handle ist oder nicht. Liegt kein Handle vor, so wird das nächste Symbol gelesen (sog. Shift-Aktion) usw. Wird aber ein Handle erkannt, so wird im bisher gelesenen String die rechte Seite des Handles durch die linke Seite einer entsprechenden Produktion ersetzt (sog. Reduce-Aktion). Diese Vorgehensweise wird als *Shift-Reduce-Parsing* bezeichnet.

Mit den Problemen der Lokalisierung des Handles und des Nonterminals, das zur Ersetzung herangezogen wird, werden wir uns später befassen. Hier sei aber noch erwähnt, daß Shift-Reduce-Parsing in der gerade umrissenen Form bei der Verwendung mehrdeutiger Grammatiken scheitert. Es treten zwei Arten von Konflikten auf: *shift/reduce* Konflikte ergeben sich dann, wenn nicht entschieden werden kann, ob die nächste Aktion ein Shift oder ein Reduce ist; *reduce/reduce* Konflikte werden dann offenbar, wenn mehrere verschiedene Reduktionen möglich sind. Mehrdeutige Grammatiken können aber sehr wohl mit Shift-Reduce-Parsern behandelt werden, wenn man von der Möglichkeit der schon erwähnten Disambiguating Rules Gebrauch macht. Genaueres dazu findet man etwa in [SchFr85].

6.2.2 LR-Parsing

LR-Parsing ist eine effiziente, deterministische Bottom-Up-Methode der Syntaxanalyse nach dem Shift-Reduce-Prinzip. Die Klasse der Grammatiken, die von LR-Parsern erkannt werden können, ist sehr umfassend und kann etwa mit der Klasse der eindeutigen kontextfreien Grammatiken gleichgesetzt werden. Diese Klasse wird als LR(k) bezeichnet, die zugehörigen Parser als LR(k)-Parser. Das L bedeutet, daß die Eingabe von links nach rechts gelesen wird, der Buchstabe R steht für die Konstruktion einer umgekehrten Rechtsableitung, während k angibt, daß k Lookahead-Symbole vom Parser für seine Entscheidungen herangezogen werden.

Nun wollen wir uns einige Eigenschaften von LR(k)-Parsern vor Augen führen. LR(k)-Parser

- erkennen tendenziell alle kontextfreien Grammatiken.
- sind eine effiziente Methode, die ohne Backtracking auskommt.
- erkennen alle Grammatiken, die von LL(1)-Parsern erkannt werden.
- können syntaktische Fehler so früh als möglich erkennen (valid-prefix-property) [TrSo85].

Allerdings ist die manuelle Konstruktion von LR(k)-Parsern recht aufwendig und auch für Programmiersprachen deren Syntax vom Umfang her klein ist, praktisch nicht mehr durchführbar. In der Praxis werden meist LR(k)-Parser mit $k \leq 1$ angewendet. Die Konstruktionskomplexität derartiger Parser ist aber immer noch hoch. Man überläßt die Konstruktion solcher Parser speziellen Werkzeugen, sogenannten LR-Parser-Generatoren. Derartige Generatoren konstruieren aus einer kontextfreien Grammatik einen LR-Parser. Ein populäres Beispiel dafür ist der Parser-Generator YACC [John86, SchFr85, KePi84].

6.2.3 Die LR-Maschine

Nun wollen wir kurz einen Automaten beschreiben, der die Grundlage aller LR-Parser darstellt. Wir beschränken uns auf LR(1)-Parser.

Die zentralen Komponenten der LR(1)-Maschine (Abbildung 10) stellen sich wie folgt dar:

- Ein Eingabeband, welches Symbol für Symbol gelesen werden kann.
- Ein Stack, in dem Strings $s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m$ gespeichert werden. Die X_i sind Grammatiksymbole, die s_i die Zustände des Automaten. s_m liegt oben am Stack. Jedes s_i ist die Zusammenfassung der Information unter s_i am Stack.
- Eine Aktionstabelle *action*, mit der aus oberstem Stackzustand s_m und aktuellem Eingabesymbol a_i die nächste Aktion des Parsers bestimmt werden kann. Die möglichen Aktionen werden anschließend beschrieben.

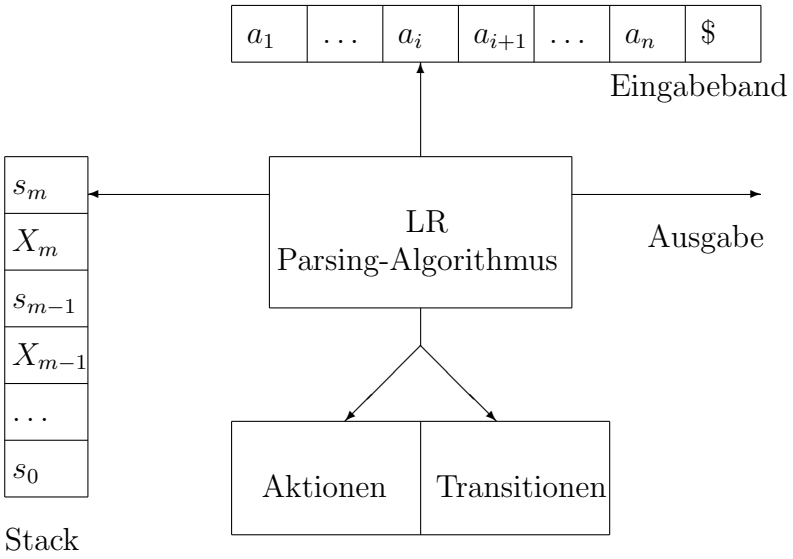

Abbildung 10: Der LR(1)-Automat

- Eine Transitionstabelle *goto*, mit der ebenfalls aus s_m und a_i der Folgezustand des Parsers ermittelt wird.

Unter einer *Konfiguration* einer LR-Maschine versteht man ein Paar, welches aus dem Stackinhalt und dem noch nicht konsumierten Teil des Eingabebandes besteht:

$$(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$$

Solch eine Konfiguration repräsentiert die Rechtssatzform

$$X_1 X_2 \dots X_m a_i a_{i+1} \dots a_n$$

Der Automat liest das momentane Eingabesymbol a_i. Zusammen mit dem obersten Stacksymbol s_m wird a_i benutzt, um die Aktionstabelle zu indizieren. $action[s_m, a_i]$ kann eine von vier Klassen von Einträgen aufnehmen:

Shift: Hier ist $action[s_m, a_i] = (\text{shift } s)$, wobei s ein Zustand ist. Der Parser gibt a_i gefolgt von s auf seinen Stack. a_{i+1} wird das aktuelle Eingabesymbol, der Parser besitzt nun folgende Konfiguration:

$$(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m a_i s, a_{i+1} \dots a_n \$)$$

Reduce: $action[s_m, a_i] = (\text{reduce } A \to \beta)$. Sei r die Länge von β, so nimmt der Parser r Paare von Zuständen und Grammatiksymbolen von seinem Stack. Nun ist s_{m-r} oberster Stackzustand. Dann wird A auf den Stack gegeben. Der neue Zustand s wird über $goto[s_{m-r}, A]$ ermittelt

und ebenfalls auf den Stack gegeben. Die Situation am Eingabeband ist unverändert, also lautet die neue Konfiguration:

$$(s_0 X_1 s_1 X_2 s_2 \dots X_{m-r} s_{m-r} A s, a_i a_{i+1} \dots a_n \$)$$

In Reduce-Aktionen erzeugt der Parser Ausgaben. Diese können wir uns hier einfach als Angabe der Produktion $A \to \beta$, nach der die Reduktion vorgenommen wurde, vorstellen.

Accept: $action[s_m, a_i]$ = accept. In diesem Fall wurde ein Wort erkannt, das Parsen ist beendet.

Error: $action[s_m, a_i]$ = error. Ein syntaktischer Fehler wurde entdeckt. Hier müssen nun geeignete Fehlerbehandlungsaktivitäten eingeleitet werden, auf die wir aber nicht eingehen werden.

Aus den soeben gemachten Bemerkungen kann für die Aktions- und für die Transitionstabelle festgelegt werden:

Aktions- und Transitionstabelle haben die gleiche Anzahl von Zeilen, die mit der Anzahl der Zustände des Automaten übereinstimmt.

Die Transitionstabelle hat $|N|$ Spalten, die einzelnen Einträge sind Folgezustände oder werden freigelassen.

Die Aktionstabelle besitzt $|T|$ Spalten. Die einzelnen Einträge sind jeweils entweder si,rj,ok oder frei. si bedeutet shift und Folgezustand i, rj bewirkt eine Reduktion durch die Produktion mit der Nummer j, bei ok wurde ein Wort erkannt und freie Einträge signalisieren das Auftreten von Fehlern.

In Abbildung 11 wird der LR(1)-Parsing-Algorithmus zusammengefaßt (nach [ASU86]).

Dieser Algorithmus soll anhand eines einfachen Beispiels verdeutlicht werden [ASU86].

Sei folgende Grammatik G gegeben:

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to$ F
(5) $F \to (E)$
(6) $F \to$ `id`

Die zugehörigen Aktions- und Transitionstabellen sind in Abbildung 12 zu finden.

Die Arbeitsweise des Automaten bei einem Eingabestring `id` $*$ `id` $+$ `id` \$ ist aus Abbildung 13 entnehmbar.

LR-Parser sind deshalb mächtiger als LL-Parser, weil die LL(k)-Parser darauf angewiesen sind, allein aus den dem Parser vorliegenden k Lookahead-Symbolen eine Produktion anzuwenden, während LR(k)-Parser zur Erkennung einer rechten Seite einer Produktion neben den k Lookahead-Symbolen auch die schon vorliegende bisher ermittelte Ableitung dieser Produktion zur Verfügung steht.

Das Eingabeband enthalte den Eingabestring w.
Setze den Lesekopf an den Beginn des Eingabebandes.
Stelle den Startzustand s_0 auf den Stack.
loop
    Sei s der oberste Stackzustand und a das aktuelle Eingabesymbol.
    if $action[s, a] = \text{s}i$ **then**
        gib a gefolgt von i auf den Stack.
        Bewege den Lesekopf um ein Symbol nach rechts.
    else if $action[s, a] = \text{r}j$ **then**
        Sei $A \to \beta$ die Produktion mit der Nummer j.
        Entferne $2 * l(\beta)$ Symbole vom Stack.
        Der oberste Stackzustand sei dann s'.
        Gib A, gefolgt von $goto[s', A]$ auf den Stack.
        Drucke die Produktion $A \to \beta$.
    else if $action[s, a] = \text{ok}$ **then**
        return
    else
        $error()$.
forever

Abbildung 11: LR-(1) Parsing Algorithmus

State	*action*						*goto*		
	`id`	+	∗	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				ok			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Abbildung 12: Parsing-Tabellen für die Grammatik G

	Stack	Eingabe	Aktion
1	0	`id` $*$ `id` $+$ `id` \$	shift
2	0`id` 5	$*$`id` $+$ `id` \$	reduce durch $F \to$ `id`
3	$0F3$	$*$`id` $+$ `id` \$	reduce durch $T \to F$
4	$0T2$	$*$`id` $+$ `id` \$	shift
5	$0T2*7$	`id` $+$ `id` \$	shift
6	$0T2*7$`id` 5	$+$`id` \$	reduce durch $F \to$ `id`
7	$0T2*7F10$	$+$`id` \$	reduce durch $T \to T*F$
8	$0T2$	$+$`id` \$	reduce durch $E \to T$
9	$0E1$	$+$`id` \$	shift
10	$0E1+6$	`id` \$	shift
11	$0E1+6$`id` 5	\$	reduce durch $F \to$ `id`
12	$0E1+6F3$	\$	reduce durch $T \to F$
13	$0E1+6T9$	\$	reduce durch $E \to E+T$
14	$0E1$	\$	accept

Abbildung 13: Verhalten der LR-Maschine für `id` $*$ `id` $+$ `id` \$

6.3 Konstruktion der Parsing-Tabellen

In diesem Abschnitt wird gezeigt, wie für eine Teilmenge der LR(k)-Grammatiken, die sogenannten SLR(1)-Grammatiken (Simple-LR(1)), die Aktions- und die Transitionstabelle generiert werden kann.

6.3.1 Viable Prefixes und Items

Ein *viable Prefix* (viable, engl.: lebensfähig) einer Rechtssatzform $\alpha\beta w$ mit $w \in T^*$ und dem Handle β ist jeder Präfix von $\alpha\beta$.

Ein viable Prefix ist also ein Präfix einer Rechtsatzform, der nicht weiter als bis zum rechten Ende des am weitesten rechts stehenden Handles reicht. Diese Definition bewirkt, daß es immer möglich ist, eine Rechtssatzform durch Anhängen von Terminals an das Ende eines solchen viable Prefix zu erzeugen. Solange also die Eingabe auf einen viable Prefix reduzierbar ist, ist kein syntaktischer Fehler aufgetreten.

Man kann nun einen endlichen Automaten konstruieren, der genau alle viable Prefixes aller Rechtsableitungen einer Grammatik erkennt. Dazu benötigen wir unter anderem noch einen weiteren Begriff.

Unter einem LR(0)-*Item* (kurz Item) versteht man eine Produktion der Grammatik, deren rechte Seite an einer beliebigen Stelle mit einer Marke "$\circ$" versehen wurde. Zur Produktion $E \to E+T$ gehören die Items $E \to \circ E+T$, $E \to E \circ +T$ sowie $E \to E+\circ T$ und $E \to E+T\circ$.

Die Position der Marke gibt an, welcher Teil der Produktion schon konsumiert wurde. Die Marke kann man sich also auch als den Lesekopf des LR-Automaten vorstellen. Liegt ein Item $A \to \alpha \circ \beta$ vor, so kann man einerseits sagen, daß gerade ein String, der aus α ableitbar ist, gelesen wurde, andererseits aber im Eingabestrom ein String erwartet wird, der aus β ableitbar ist.

Zu jedem viable Prefix kann man eine endliche Menge von *gültigen Items* angeben.

Ein Item $A \to \alpha \circ \beta$ ist genau dann gültig für einen viable Prefix $\pi\alpha$, wenn es eine Rechtsableintung $S \overset{R*}{\Rightarrow} \pi A w \overset{R}{\Rightarrow} \pi\alpha\beta w$ gibt, mit $w \in T^*$.

Wenn $A \to \alpha \circ \beta$ ein gültiges Item für $\pi\alpha$ ist, und $\pi\alpha$ am Stack der LR-Maschine ist, so kann man erkennen, ob man eine Shift- oder eine Reduce-Aktion vornehmen soll. Ist $\beta = \epsilon$, steht also die Marke am Ende des Items, so kann man annehmen, daß $A \to \alpha$ ein Handle ist und man reduziert dieses Handle. Ist aber $\beta \neq \epsilon$, so ist dies ein Indikator dafür, daß sich noch kein vollständiges Handle am Stack befindet; deshalb wird man die Shift-Aktion wählen. Allerdings können mehrere gültige Items uns für ein und denselben viable Prefix zu verschiedenen Aktionen veranlassen. Viele, aber nicht alle dieser Konflikte können durch entsprechenden Lookahead gelöst werden.

Bei der Konstruktion des endlichen Automaten macht man sich nun zunutze, daß es zwar in einer Grammatik unendlich viele viable Prefixes geben kann, daß aber jeder viable Prefix nur eine endliche Anzahl von gültigen Items besitzt. Jeder Zustand des endlichen Automaten besitzt daher zwar eine potentiell unendliche Menge von viable Prefixes, die in diesem Zustand erkannt werden, aber nur eine endliche Menge von gültigen Items. Mit Hilfe dieser Items kann dann in jedem Zustand in der schon beschriebenen Weise die korrekte Aktion ermittelt werden.

Die weiteren Unterabschnitte beschäftigen sich nun Schritt für Schritt mit der Konstruktion des Automaten bzw. mit der Erstellung der Tabellen für die LR-Maschine. Hauptsächlich stützen wir uns hierbei wie auch schon bisher in diesem Abschnitt auf [ASU86] und [TrSo85].

6.3.2 Erweiterte Grammatiken

Unter der Erweiterung einer Grammatik $G = (N, T, S, P)$ versteht man die Grammatik $G' = (N \cup \{S'\}, T, S', P \cup \{S' \to S\})$. Man ersetzt also das Startsymbol S durch ein neues Symbol S' und ergänzt die Produktionen um die Regel $S' \to S$.

Die Begründung für diese Transformation ist einfach: die neue Startproduktion wird verwendet, um dem Parser die Erkennung eines Wortes zu signalisieren; ein Wort soll genau dann vom Parser akzeptiert werden, wenn dieser gerade dabei ist durch die neue Produktion $S' \to S$ reduzieren.

Für die weiteren Beispiele sei nachstehende erweiterte Grammatik G_e definiert:

(0) $E' \to E$
(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to \mathrm{F}$
(5) $F \to (E)$
(6) $F \to$ `id`

6.3.3 Die Hülle einer Item-Menge

Sei $A \rightarrow \alpha \circ B\beta \in I$, so wird dadurch ja ausgedrückt, daß man als nächsten Input einen String erwarten kann, der aus $B\beta$ abgeleitet werden kann. Ist aber $B \in N$, so gibt es in einer (reduzierten) Grammatik mindestens eine Produktion der Form $B \rightarrow \gamma$. Also kann man auch einen String erwarten, der aus γ ableitbar ist, als Item notiert $B \rightarrow \circ\gamma$. Die Hülle einer Item-Menge I ist nun die Zusammenfassung aller Items, die auf Grund der gegebenen Menge I für die Ableitung eines Strings relevant sind.

Die Hülle $closure(I)$ einer Menge I von Items wird wie folgt berechnet:

1) $closure(I) = I$
2) Wiederhole
   Sei $A \rightarrow \alpha \circ B\beta \in closure(I)$
   und $B \rightarrow \gamma$ eine Produktion, dann
   $closure(I) = closure(I) \cup \{B \rightarrow \circ\gamma\}$
   bis $closure(I)$ unverändert bleibt

Dazu ein Beispiel unter G_e:
Sei $I = \{F \rightarrow (\circ E)\}$, dann sind die Elemente von $closure(I) =$

$$
\begin{array}{lcl}
E & \rightarrow & \circ E + T \\
E & \rightarrow & \circ T \\
T & \rightarrow & \circ T * F \\
T & \rightarrow & \circ F \\
F & \rightarrow & \circ(E) \\
F & \rightarrow & \circ \texttt{id}
\end{array}
$$

6.3.4 Die Goto-Operation

Diese Operation beschreibt, welche neuen Items sich durch die Konsumation eines Grammatiksymboles X aus einer Ausgangsmenge I von Items erzeugen lassen.

Die Transitionsfunktion $goto(I, X)$ einer Menge I von Items und eines Grammatiksymboles X kann wie folgt berechnet werden:

1) $goto(I, X) = \emptyset$
2) Für alle Items der Form $A \rightarrow \alpha \circ X\beta \in I$ wiederhole
   $goto(I, X) = goto(I, X) \cup \{A \rightarrow \alpha X \circ \beta\}$
3) Bilde die Hülle:
   $goto(I, X) = closure(goto(I, X))$

In Schritt 2) werden also neue Items durch Simulation des Lesens von X gebildet. Danach berechnet man die Hülle dieser neu gewonnenen Items.

Folgende Aussage sollte einsichtig sein: Ist I die Menge der gültigen Items für einen viable Prefix π, so ist $goto(I, X)$ die Menge der gültigen Items für den viable Prefix πX.

Auch hier ein Beispiel unter G_e zur Verdeutlichung:

Sei $I = \{E \to T\circ, T \to T \circ *F\}$. $goto(I, *)$ enthält dann die Elemente:

$$\begin{aligned} T &\to T * \circ F \\ F &\to \circ(E) \\ F &\to \circ \texttt{id} \end{aligned}$$

Das erste angeführte Item entsteht aus Schritt 2) der Berechnungsvorschrift, die restlichen Items ergeben sich aus der Bildung der Hülle.

6.3.5 Die Konstruktion der Item-Mengen

Aus dem Startitem $S' \to \circ S$ kann man durch Bildung der Hülle eine Menge I_0 von gültigen Items für den viable Prefix ϵ bestimmen. I_0 ordnet man dem Zustand 0 zu. Aus I_0 können dann mit der *goto*-Operation weitere Mengen von gültigen Prefixes gewonnen werden, auf die dann wieder die *goto*-Operation angewendet werden kann usw. Identische Mengen von gültigen Items werden in einem Zustand zusammengefaßt. Die Bildung der Item-Mengen kann so beschrieben werden:

1) Bilde die Hülle der Startproduktion:
   $C = \{closure(\{S' \to \circ S\})\}$
2) Wiederhole
   Für jede Menge von Items $I \in C$ und
   für jedes Grammatiksymbol X
   berechne $goto(I, X)$
   falls $goto(I, X) \neq \emptyset$ und $goto(I, X) \notin C$, dann
   $C = C \cup \{goto(I, X)\}$
   bis keine Veränderungen in C auftreten.

Die Item-Mengen entsprechen den Zuständen des Automaten, für jede *goto*-Menge der Form $I_j = goto(I_i, X)$ zeichnet man eine Transition von Zustand i nach Zustand j ein, und versieht diese mit dem Symbol X.

Dazu ein Beispiel:

Wir wollen die Item-Mengen unter G_e berechnen:

$I_0 = closure(\{E' \to \circ E\})$

$$\begin{aligned} E' &\to \circ E \\ E &\to \circ E + T \\ E &\to \circ T \\ T &\to \circ T * F \end{aligned}$$

$$
\begin{aligned}
T &\rightarrow \circ F \\
F &\rightarrow \circ (E) \\
F &\rightarrow \circ \texttt{id}
\end{aligned}
$$

$I_1 = goto(I_0, E)$:

$$
\begin{aligned}
E' &\rightarrow E \circ \\
E' &\rightarrow E \circ + T
\end{aligned}
$$

$I_2 = goto(I_0, T)$:

$$
\begin{aligned}
E &\rightarrow T \circ \\
T &\rightarrow T \circ * F
\end{aligned}
$$

$I_3 = goto(I_0, F)$:

$$
T \rightarrow F \circ
$$

$I_4 = goto(I_0, ()$:

$$
\begin{aligned}
F &\rightarrow (\circ E) \\
E &\rightarrow \circ E + T \\
E &\rightarrow \circ T \\
T &\rightarrow \circ T * F \\
T &\rightarrow \circ F \\
F &\rightarrow \circ (E) \\
F &\rightarrow \circ \texttt{id}
\end{aligned}
$$

$I_5 = goto(I_0, \texttt{id}\ )$:

$$
F \rightarrow \texttt{id}\ \circ
$$

$I_6 = goto(I_1, +)$:

$$
\begin{aligned}
E &\rightarrow E + \circ T \\
T &\rightarrow \circ T * F \\
T &\rightarrow \circ F \\
F &\rightarrow \circ (E) \\
F &\rightarrow \circ \texttt{id}
\end{aligned}
$$

$I_7 = goto(I_2, *)$:

$$\begin{aligned} T &\rightarrow T * \circ F \\ F &\rightarrow \circ(E) \\ F &\rightarrow \circ\texttt{id} \end{aligned}$$

$I_8 = goto(I_4, E)$:

$$\begin{aligned} F &\rightarrow (E\circ) \\ E &\rightarrow E \circ + T \end{aligned}$$

$I_a = goto(I_4, T) = I_2$:

$$\begin{aligned} E &\rightarrow T \circ \\ T &\rightarrow T \circ * F \end{aligned}$$

$I_b = goto(I_4, F) = I_3$:

$$T \rightarrow F \circ$$

$I_c = goto(I_4, () = I_4$:

$$\begin{aligned} F &\rightarrow (\circ E) \\ E &\rightarrow \circ E + T \\ E &\rightarrow \circ T \\ T &\rightarrow \circ T * F \\ T &\rightarrow \circ F \\ F &\rightarrow \circ(E) \\ F &\rightarrow \circ\texttt{id} \end{aligned}$$

$I_d = goto(I_4, \texttt{id}\ ) = I_5$:

$$F \rightarrow \texttt{id}\ \circ$$

$I_9 = goto(I_6, T)$:

$$\begin{aligned} E &\rightarrow E + T \circ \\ T &\rightarrow T \circ * F \end{aligned}$$

$I_e = goto(I_6, F) = I_3$:

$$T \rightarrow F \circ$$

$I_f = goto(I_6, () = I_4$:

$$\begin{aligned} F &\rightarrow (\circ E) \\ E &\rightarrow \circ E + T \\ E &\rightarrow \circ T \\ T &\rightarrow \circ T * F \\ T &\rightarrow \circ F \\ F &\rightarrow \circ(E) \\ F &\rightarrow \circ \texttt{id} \end{aligned}$$

$I_g = goto(I_6, \texttt{id}\ ) = I_5$:

$$F \rightarrow \texttt{id}\ \circ$$

$I_{10} = goto(I_7, F)$:

$$T \rightarrow T * F \circ$$

$I_h = goto(I_7, () = I_4$:

$$\begin{aligned} F &\rightarrow (\circ E) \\ E &\rightarrow \circ E + T \\ E &\rightarrow \circ T \\ T &\rightarrow \circ T * F \\ T &\rightarrow \circ F \\ F &\rightarrow \circ(E) \\ F &\rightarrow \circ \texttt{id} \end{aligned}$$

$I_i = goto(I_7, \texttt{id}\ ) = I_5$:

$$F \rightarrow \texttt{id}\ \circ$$

$I_{11} = goto(I_8,))$:

$$F \rightarrow (E) \circ$$

$I_j = goto(I_8, +) = I_6$:

$$\begin{aligned} E &\rightarrow E + \circ T \\ T &\rightarrow \circ T * F \\ T &\rightarrow \circ F \\ F &\rightarrow \circ(E) \\ F &\rightarrow \circ \texttt{id} \end{aligned}$$

$I_k = goto(I_9, *) = I_7$:

$$\begin{aligned} T &\rightarrow T * \circ F \\ F &\rightarrow \circ(E) \\ F &\rightarrow \circ \texttt{id} \end{aligned}$$

6.3.6 Ausfüllen der Tabellen

Aufbauend auf den soeben definierten Operationen ist es numehr möglich, die Einträge der Aktions- und die Transitionstabelle zu füllen. Dazu verwendet man den folgenden Algorithmus:

Konstruktion der SLR-Tabellen

1) Man konstruiere $C = \{I_0, I_1, \ldots, I_n\}$, die Menge der Item-Mengen für G'
2) Zu jeder Menge $I_i \in C$ erzeuge man einen Zustand i
3) Man bestimme für jeden Zustand i die Einträge der Aktionstabelle:
   a) Falls $A \rightarrow \alpha \circ x\beta \in I_i$, und $goto(I_i, x) = I_j$, dann
      trage man "shift j" in $action[i, x]$ ein.
      x muß ein Terminalsymbol sein.
   b) Falls $A \rightarrow \alpha\circ \in I_i$, dann
      bestimme man $FOLLOW(A)$ und
      trage für alle $x \in FOLLOW(A)$
      reduce $A \rightarrow x$ in $action[i, x]$ ein.
      A muß von S' verschieden sein.
   c) Falls $S' \rightarrow S\circ \in I_i$, dann
      trage man "accept" in $action[i, \$]$ ein.
4) Man bestimme für jeden Zustand i die Einträge der Transitionstabelle:
   Falls $goto(I_i, A) = I_j$, dann
   trage man "j" in $goto[i, A]$ ein.
5) Alle noch freien Einträge in den Tabellen setze man auf "error"
6) Startzustand wird derjenige Zustand i, dessen zugeordnete Item-Menge I_i das Item $S' \rightarrow \circ S$ enthält.

Zu diesem Algorithmus nun noch einige Bemerkungen:

In den Punkten 1) und 2) wird der die viable Prefixes erkennende Automat konstruiert. Die Punkte 3), 4) und 5) erzeugen die Tabellen. Punkt 6) schließlich ermittelt den Startzustand der LR-Maschine.

Die Aktionstabelle wird von Punkt 3) erzeugt. Punkt 3a) bezieht sich auf das Vorliegen unvollständiger Handles und sorgt dafür, daß in solchen Situationen geshiftet wird. Punkt 3b) bezieht sich auf Konstellationen, in denen vollständige Handles vorliegen. Liegen solche vor, dann wird ein Reduce durchgeführt. Diese Reduce-Aktionen sind allerdings nur dann erlaubt, wenn das aktuelle Eingabesymbol in der $FOLLOW$-Menge der linken Seite der reduzierenden Produktion aufscheint. Also wird nur in den entsprechenden Spalten der Aktionstabelle eine Reduce-Aktion vermerkt.

Die Transitionstabelle wird in Punkt 4) auf die schon erwähnte Weise generiert.

Wird dieser Algorithmus auf G_e angewendet, so entstehen genau die SLR(1)-Parsingtabellen in Abbildung 12.

6.4 Werkzeuge zur Parser-Konstruktion

Wie man aus dem vorangegangenen Abschnitt ersehen kann, ist die Konstruktion eines Bottom-Up Shift-Reduce Parsers auch schon bei einer an Simplizität wohl kaum zu überbietenden Grammatik recht aufwendig. Will man aber nun Parser für Sprachen von realistischem Umfang konstruieren, so stellt diese hohe Konstruktionskomplexität einer manuellen Konstruktion bald Grenzen. In HIBOL-2 etwa, einer formularorientierten Programmiersprache zur Implementierung von Anwendungen im kommerziellen Bereich [MiWe87], deren kontextfreie Grammatik in einer EBNF-Darstellung nur wenige Seiten lang ist, umfaßt der Zustandsraum der Parsing-Maschine bereits über 400 Zustände.

Für die Praxis werden deshalb spezielle Software-Werkzeuge, sogenannte *Parser-Generatoren* eingesetzt. Solche Werkzeuge erzeugen aus einer ihnen vorgelegten kontextfreien Grammatik die Transitions- und die Aktionstabelle und stellen ebenfalls das die Tabellen verwendende Parsing-Programm zur Verfügung. Man benötigt nur mehr einen lexikalischen Analysator, der das Quellprogramm in eine Folge von Symbolen aus dem Vokabular der Quellsprache zerlegt und diese Symbolkette dem Parser sukzessive zur Verfügung stellt (Abbildung 14). Weiters muß hier aber auch vermerkt werden, daß Parsing ja nicht Selbstzweck ist, sondern lediglich einen Ausschnitt des mit der Analyse der Worte der Quellsprache befaßten Teils eines Compilers abdeckt. Um einen realen Compiler zu erstellen, muß neben der lexikalischen und syntaktischen Analyse auch die semantische Analyse und der zweite große Teil des Übersetzers, der für die Synthese des Zielprogrammes verantwortlich ist, implementiert werden. Obwohl Parser-Generatoren einen wesentlichen Beitrag zur einfachen Konstruktion von Compilern liefern, soll hier nicht der Eindruck entstehen, daß mit dem richtigen Werkzeug allein schon die Garantie für eine erfolgreiche Implementierung gegeben ist (diese Aussage besitzt übrigens in der Informatik generelle Gültigkeit).

Ein in dieser Arbeit schon erwähntes, auch sonst oft angeführtes Beispiel für solch einen Parser-Generator findet man als Bestandteil der *Programmers Workbench* im

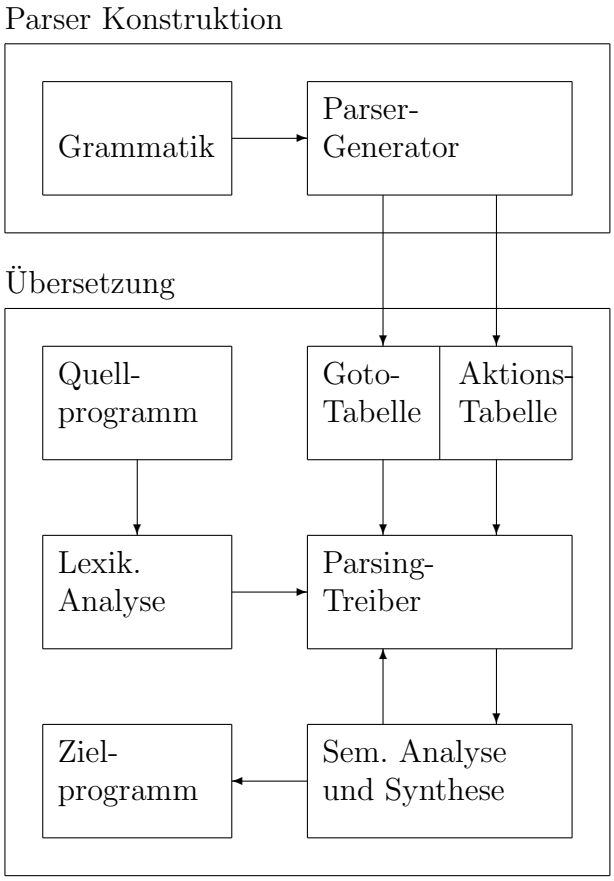

Abbildung 14: Parser Generator und generierter Parser (nach [Finn85])

Betriebssystem UNIX; es ist der Parser Generator *YACC* [John86,KePi84,SchFr85], der aus einer um die bei Reduktionen zu exekutierenden semantischen Aktionen erweiterten, in BNF-ähnlicher Notation spezifizierten Grammatik einen Shift-Reduce Parser erzeugt.

7 Zusammenfassende Schlußbemerkungen

Aufbauend auf einigen theoretischen Grundlagen wurde in dieser Arbeit versucht, einen Überblick über das weite Gebiet des Parsings zu geben. Sowohl Top-Down als auch Bottom-Up Methoden wurden ansatzweise dargestellt.

Hier bleibt noch einiges zur Auswahl der geeigneten Parsing-Technik für eine konkrete Programmiersprache zu sagen. Welche Art von Parser im jeweiligen Fall zum Einsatz kommen wird, hängt von einer Reihe von Faktoren ab.

Neben der zentralen Frage der für die zu erkennende Sprache erforderlichen Mächtigkeit der Parsingmethode ist weiters das Kriterium des Implementierungsaufwandes für den Parser von großer Bedeutung für die Entscheidung zugunsten einer bestimmten Technik. Allerdings bedingt hohe Mächtigkeit aber auch hohen Aufwand und so muß wohl immer im jeweiligen Einzelfall zwischen den beiden Faktoren abgewogen werden.

Außer den beiden eben genannten Hauptkriterien sind wohl auch noch Platzbedarf und Geschwindigkeit (Raum- und Zeitlomplexität) des Parsers und die Möglichkeiten die die jeweilige Methode für die Fehlererkennung im Quellprogramm bietet, sowie die Einfachheit der Einbringung der semantischen Aktionen in den Parser zu berücksichtigen.

In einer konkreten Situation kann es aber müßig sein, die Auswahl einer Parsingmethode mit großer Akribie zu betreiben. Ist ein Parser-Generator verfügbar so werden wohl die hohen Personalkosten die Verwendung dieses Werkzeuges (freilich nur bei prinzipelller Eignung im vorliegenden Fall) indizieren.

Es sollte nicht unter den Tisch fallen, daß diese Arbeit keineswegs als auch nur einigermaßen vollständig zu verstehen ist. Neben der Einschränkung auf den Kernbereich der syntaktische Analyse mußte auch die Darstellung der Methoden unter Verzicht auf wesentliche Details vorgenommen werden. Trotzdem bleibt zu hoffen, daß die Arbeit dem Anspruch, einen ersten orientierenden Einstieg in dieses breite Feld zu bieten, gerecht werden konnte.

8 Literatur

ASU86: Aho,A.V., Sethi,R., Ullman,J.D.: Compilers, Principles, Techniques and Tools, Addison-Wesley, Reading, 1986

Back79: Backhouse,R.C.: Syntax of Programming Languages, Prentice-Hall, Englewood-Cliffs, 1979

BuMa84: Buchner,W., Maurer,H.: Theoretische Grundlagen der Programmiersprachen, Bibliographisches Institut, Mannheim, 1984

Finn85: Finn,G.D.: Extended Use of Null Productions in LR(1) Parser Applications, in: Communications of the ACM, Vol. 28 No. 9, September 1985

HoUl88: Hopcroft,J.E., Ullman,J.D.: Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie, Addison-Wesley, Reading, 1988

John86: Johnson,S.C.: Yacc: Yet Another Compiler Compiler, in: HP-UX Concepts and Tutorials, Vol.3: Programming Environment, HP, Fort Collins, 1986

KePi84: Kernighan,B.W., Pike,R.: The UNIX Programming Environment, Prentice-Hall, Englewood-Cliffs, 1984

MiWe87: Mittermeir,R., Wernhart, H.: Benutzerhandbuch zur Programmiersprache HIBOL-2, Interner Bericht, Institut für Informatik, Universität Klagenfurt, 1987

SchFr85: Schneider,A.T., Friedman,H.G.Jr.: Introduction to Compiler Construction with UNIX, Prentice-Hall, Englewood-Cliffs, 1985

Sedg83: Sedgewick,R : Algorithms, Addison-Wesley, Reading, 1983

TrSo85: Tremblay,J.P., Sorenson, P.G.: Theory and Practice of Compiler Writing, McGraw-Hill, New York, 1985

Wirth84: Wirth,N.: Compilerbau, Teubner, Stuttgart, 1984