Inhalt

Lehrziele

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie

- die Begriffe *kontextfreie Grammatik*, *Ableitung*, *Ableitungsbaum*, *Linksableitung* und *Rechtsableitung* erklären können,
- das allgemeine Prinzip der *Top-down-Analyse* mit *Backtracking* erklären und an einem Beispiel vorführen können,
- die Definition einer LL(*k*)-Grammatik angeben können,
- eine Grammatik durch Beseitigung von Linksrekursion und durch Linksfaktorisierung so modifizieren können, dass eine Top-down-Analyse ohne Backtracking möglich wird,
- für eine LL(1)-Grammatik FIRST-, FOLLOW- und Steuermengen definieren und berechnen können,
- eine Analysetabelle berechnen und erläutern können,
- die Implementierung eines vorgreifenden Analysators unter Verwendung einer Analysetabelle beschreiben können,
- die Implementierung eines vorgreifenden Analysators durch rekursiven Abstieg beschreiben können.

Kapitel 3

Syntaxanalyse

Die zweite Phase der Übersetzung ist die *Syntaxanalyse*. Die Aufgabe besteht darin, aus der Folge von Token, die aus der lexikalischen Analyse kommen, einen *Syntaxbaum* (auch *Ableitungsbaum* genannt) zu berechnen. In Kapitel 1 hatten wir dies mit Abb. 1.4 illustriert, die wir hier noch einmal wiedergeben.

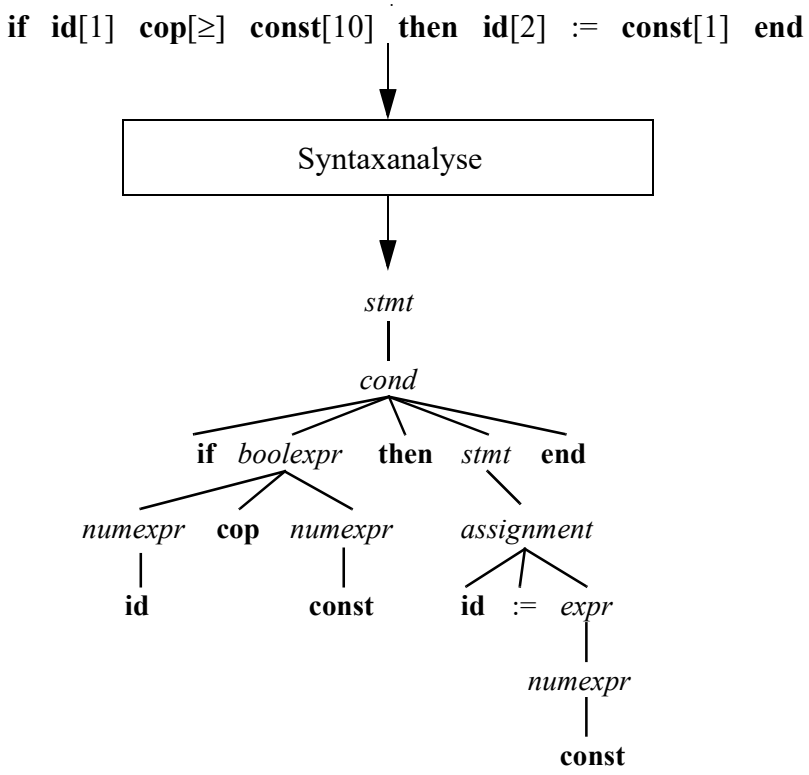


Abb. 3.1. Eingabe und Ausgabe der Syntaxanalyse

Basis für die Syntaxanalyse ist eine *Grammatik*, die die Syntax der Quellsprache beschreibt, also die Struktur von Programmen dieser Sprache. Einige Regeln der Grammatik für das Beispiel sind:

stmt ::= *assignment* | *cond*
cond ::= **if** *boolexpr* **then** *stmt* **end** |
if *boolexpr* **then** *stmt* **else** *stmt* **end**
numexpr ::= **id** | **const**

Es gibt nun zwei Strategien, um mit Hilfe der Grammatik den Baum aus der Tokenfolge zu berechnen. Bei der *Top-down-Analyse* baut man den Baum von der Wurzel aus zu den Blättern hin auf. Das heißt, man beginnt mit dem Startsymbol der Grammatik (hier *stmt*) als Wurzel des Baumes und „rät“ im ersten Schritt, dass die Regel

stmt ::= *cond*

anzuwenden ist. Das bedeutet, dass *cond* als Sohnknoten an *stmt* anzuhängen ist. Während der Analyse gibt es jeweils einen teilweise erzeugten Syntaxbaum und eine teilweise gelesene Eingabefolge; nach dem ersten Schritt hätten wir also folgende Situation:

if id cop const then id := const end
↑

Abb. 3.2. Beginn der Top-down-Analyse: Aufbau des Ableitungsbaums von der Wurzel aus

Oben ist der Baum gezeigt, unten die Eingabefolge, der Zeiger zeigt jeweils auf das nächste zu verarbeitende Token der Eingabefolge. Natürlich will man nicht wirklich raten, welche Regel anzuwenden ist. Stattdessen wird als Vorbereitung die Grammatik analysiert, so dass z. B. folgende Information verfügbar ist:

*Wenn das aktuelle Symbol im Syntaxbaum „stmt“ ist und das aktuelle Token in der Eingabefolge „***if***“ ist, dann ist die Regel „stmt ::= cond“ anzuwenden.*

Bei der zweiten Strategie, der *Bottom-up-Analyse*, wird der Baum von den Blättern her aufgebaut. Dabei ist die Grundidee, so lange Token der Eingabefolge zu lesen und sich zu merken, bis eine vollständige *rechte Seite* einer Grammatikregel gelesen worden ist. Die Symbole der rechten Seite werden dann als *Söhne* mit dem Symbol der linken Seite als *Vater* zu einem Teilbaum verbunden. Im obigen Beispiel wäre die erste vollständig gelesene rechte Seite das Token **id**, nämlich als rechte Seite der Regel

numexpr ::= **id**

Nach dem Lesen des Tokens **id** ergäbe sich also die Situation:

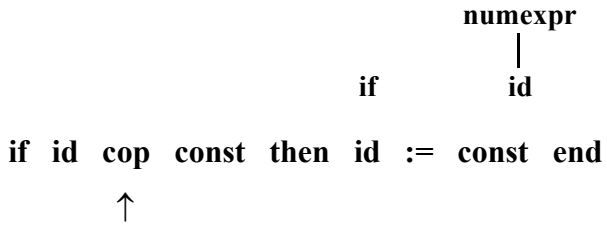


Abb. 3.3. Beginn der Bottom-up-Analyse: Aufbau des Ableitungsbaums von den Blättern aus

Bei der Bottom-up-Analyse hat man eine Liste von Teilbäumen zu verwalten, die nach und nach zu größeren Einheiten zusammengebaut werden. Der erste oben gezeigte Teilbaum, das Blatt **if**, kann offensichtlich erst dann mit einem Vater verbunden werden (nämlich *cond*), wenn die gesamte bedingte Anweisung gelesen wurde.

Im Folgenden wiederholen wir zunächst in Abschnitt 3.1 grundlegende Begriffe zu Grammatiken und betrachten dann in Abschnitt 3.2 die Top-down-Analyse. Die Bottom-up-Analyse wird in Abschnitt 3.3 behandelt.

3.1 Kontextfreie Grammatiken und Syntaxbäume

Die Syntax höherer Programmiersprachen, und allgemeiner der aus Sicht des Compilerbaus interessierenden Quellsprachen, wird durch *kontextfreie Grammatiken* beschrieben.

Definition 3.1: Eine *kontextfreie Grammatik* ist ein Quadrupel $G = (N, \Sigma, P, S)$, wobei gilt:

(i) N ist ein Alphabet von *Nichtterminalen*.

(ii) Σ ist ein Alphabet von *Terminalen*. Die Alphabete N und Σ sind disjunkt.

(iii) $P \subseteq N \times (N \cup \Sigma)^*$ ist eine Menge von *Produktionsregeln.*

(iv) $S \in N$ ist das *Startsymbol*. □

Wir erinnern uns, dass ein Alphabet eine endliche, nichtleere Menge ist. Im obigen Beispiel sind also *stmt*, *cond* usw. Nichtterminale, **if**, **id** usw. Terminale. Nichtterminale beschreiben größere strukturierte Einheiten der Quellsprache, Terminale die tatsächlich in der Eingabe vorkommenden Zeichen, in dieser Phase der Übersetzung allerdings die Zeichen, die die lexikalische Analyse liefert (Token). Produktionen

sind Paare, bestehend aus einem Nichtterminal und einer Folge von Nichtterminal- und/oder Terminalsymbolen. Formal entspricht die Notation

stmt ::= *assignment* | *cond*

also einer Menge von Regeln

{(*stmt*, *assignment*), (*stmt*, *cond*)}

Produktionsregeln werden in formalen Betrachtungen gewöhnlich mit Pfeilen notiert; anstelle einer Regel (A, α) schreibt man $A \to \alpha$. Eine Menge von Produktionen für dasselbe Nichtterminal A, also $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_n$, schreibt man auch als

$$A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

Der senkrechte Strich trennt die Alternativen für A. Allgemein sind folgende Konventionen üblich, die wir auch benutzen werden:

Lateinische Großbuchstaben A, B, C, X, Y, Z, … bezeichnen Nichtterminale, also Elemente aus N. Kleinbuchstaben am Anfang des Alphabets a, b, c, … stehen für Terminale (Elemente aus Σ), Kleinbuchstaben am Ende des Alphabets u, v, w, … bezeichnen Folgen von Terminalen (also Elemente aus Σ^*). Griechische Kleinbuchstaben α, β, γ, φ, ψ, … stehen für Worte aus $(N \cup \Sigma)^*$. – Wir werden diese Notationen abkürzend auch in Definitionen oder Sätzen benutzen, also z. B. sagen „es existiert ein Wort α“ anstatt „es existiert ein Wort α aus $(N \cup \Sigma)^*$“.

Produktionen sind Ersetzungsregeln; die Produktion $A \to \alpha$ besagt, dass man ein Auftreten des Nichtterminals A innerhalb eines Wortes φ durch die Folge von Symbolen α ersetzen darf. Damit verwandelt sich das gegebene Wort φ in ein Wort ψ. Das bringt uns zum Begriff der *Ableitung*:

Definition 3.2: Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik. ψ ist aus φ *direkt ableitbar* (oder φ *produziert* ψ *direkt*), notiert als $\varphi \Rightarrow \psi$, wenn es Worte σ, τ gibt und eine Produktion $A \to \alpha$, so dass gilt: $\varphi = \sigma A \tau$ und $\psi = \sigma\alpha\tau$. Wir sagen ψ ist aus φ *ableitbar* (φ *produziert* ψ), notiert als $\varphi \Rightarrow^* \psi$, falls es eine Folge von Worten $\varphi_1, \ldots, \varphi_n$ gibt ($n \geq 1$), so dass gilt: $\varphi = \varphi_1$, $\psi = \varphi_n$, und $\varphi_i \Rightarrow \varphi_{i+1}$ für $1 \leq i < n$. Die Folge von Worten $\varphi_1, \ldots, \varphi_n$, für die ja gilt

$$\varphi_1 \Rightarrow \varphi_2 \Rightarrow \varphi_3 \Rightarrow \ldots \Rightarrow \varphi_n$$

heißt eine *Ableitung* von ψ aus φ in G. □

Wir betrachten als Beispiel eine Grammatik, die einfache arithmetische Ausdrücke beschreibt.

$$G = (\{E, T, F\}, \{\mathbf{id}, +, *, (,)\},$$
$$\{E \to E + T \mid T, T \to T * F \mid F, F \to \mathbf{id} \mid (E)\}, E)$$

Hier steht E für *expression*, T für *term* und F für *factor*. In dieser Grammatik lässt sich aus dem Startsymbol E z. B. ableiten:

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow \mathbf{id} + T \Rightarrow \mathbf{id} + T * F \Rightarrow \mathbf{id} + F * F$$

$$\Rightarrow \mathbf{id} + \mathbf{id} * F \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \qquad (1)$$

Die Menge aller aus dem Startsymbol ableitbaren Terminalworte ist gerade die von einer Grammatik erzeugte Sprache.

Definition 3.3: Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik. Die *von G erzeugte Sprache* ist

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

Ein Wort $w \in L(G)$ heißt ein *Satz von G*. Ein Wort $\alpha \in (N \cup \Sigma)^*$ mit $S \Rightarrow^* \alpha$ heißt eine *Satzform von G*. □

Eine kontextfreie Grammatik kann prinzipiell Nichtterminalsymbole enthalten, die zur erzeugten Sprache nichts beitragen, weil sie entweder vom Startsymbol aus nicht erreichbar sind oder weil sich aus ihnen kein Terminalwort ableiten lässt. Solche Symbole stören evtl. in Definitionen oder Sätzen.

Definition 3.4: Ein Nichtterminal A heißt *unerreichbar*, falls es keine Worte α, β gibt, so dass $S \Rightarrow^* \alpha A \beta$. A heißt *unproduktiv*, falls es kein Wort $w \in \Sigma^*$ gibt, so dass $A \Rightarrow^* w$. Eine kontextfreie Grammatik heißt *reduziert*, wenn sie keine unerreichbaren oder unproduktiven Nichtterminale enthält. □

Offensichtlich kann man diese Symbole und alle Produktionen, in denen sie vorkommen, aus einer Grammatik entfernen, ohne dass sich die erzeugte Sprache ändert. Wir nehmen im Folgenden stets an, dass die betrachteten Grammatiken reduziert sind.

Innerhalb einer Ableitung wird in jedem Schritt ein Nichtterminal durch die rechte Seite einer zugehörigen Produktion ersetzt. Dazu gibt es eine Baumdarstellung, die wir schon mehrfach gezeigt haben, ohne sie formal zu definieren. Zu der obigen Ableitung (1) sieht der zugehörige Baum so aus:

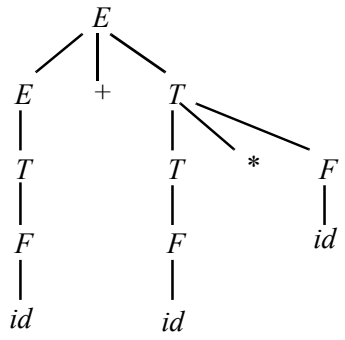


Abb. 3.4. Ableitungsbaum zur Ableitung (1)

Definition 3.5: Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik. Sei T ein Baum, dessen innere Knoten mit Nichtterminalen und dessen Blätter mit Terminalen von G oder mit dem leeren Wort ε markiert sind. T heißt *Syntaxbaum* (oder *Ableitungsbaum*) für das Wort $w \in \Sigma^*$ und für $X \in N$, falls gilt:

(i) Für jeden inneren Knoten p, der mit $Y \in N$ markiert ist und dessen Söhne (von links nach rechts) $q_1, \ldots, q_n$ mit $Q_1, \ldots, Q_n \in (N \cup \Sigma)$ markiert sind, gibt es eine Produktion $Y \to Q_1 \ldots Q_n$ in P. Falls p einen einzigen Sohn hat, der mit ε markiert ist, so existiert eine Produktion $Y \to \varepsilon$.

(ii) Die Wurzel des Baumes ist mit X markiert, und die Konkatenation der Markierungen der Blätter ergibt w. □

Ein Syntaxbaum „vergisst" gewisse Informationen aus einer Ableitung, nämlich in welcher Reihenfolge Nichtterminalsymbole ersetzt wurden. In der obigen Ableitung (1) haben wir jeweils das am weitesten links stehende Nichtterminal innerhalb einer Satzform ausgewählt. Wir machen das durch Unterstreichung in der Ableitung deutlich:

$$\underline{E} \Rightarrow \underline{E} + T \Rightarrow \underline{T} + T \Rightarrow \underline{F} + T \Rightarrow \mathbf{id} + \underline{T} \Rightarrow \mathbf{id} + \underline{T} * F \Rightarrow \mathbf{id} + \underline{F} * F$$

$$\Rightarrow \mathbf{id} + \mathbf{id} * \underline{F} \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \qquad (1)$$

Eine andere Ableitung ersetzt z. B. jeweils das am weitesten rechts stehende Nichtterminal und führt zum selben Syntaxbaum:

$$\underline{E} \Rightarrow E + \underline{T} \Rightarrow E + T * \underline{F} \Rightarrow E + \underline{T} * \mathbf{id} \Rightarrow E + \underline{F} * \mathbf{id} \Rightarrow \underline{E} + \mathbf{id} * \mathbf{id}$$

$$\Rightarrow \underline{T} + \mathbf{id} * \mathbf{id} \Rightarrow \underline{F} + \mathbf{id} * \mathbf{id} \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \qquad (2)$$

Natürlich gibt es noch viele andere Ableitungen, die zu demselben Syntaxbaum gehören, indem man z. B. mal das am weitesten rechts, mal das am weitesten links stehende oder auch einmal ein Nichtterminal in der Mitte der Satzform ersetzt. Die beiden gezeigten Ableitungen sind allerdings von besonderem Interesse; sie heißen *Linksableitung* bzw. *Rechtsableitung*. Wie wir später sehen werden, werden bei der Top-down-Analyse gerade Linksableitungen, bei der Bottom-up-Analyse Rechtsableitungen erkannt.

Definition 3.6: Sei $\varphi_1, \ldots, \varphi_n$ eine Ableitung mit $S = \varphi_1$, $\varphi = \varphi_n$. $\varphi_1, \ldots, \varphi_n$ heißt *Linksableitung von* φ, falls in jedem Schritt von φ_i nach φ_{i+1} in φ_i jeweils das am weitesten links stehende Nichtterminal ersetzt wird, also gilt $\varphi_i = wA\sigma$ und $\varphi_{i+1} = w\alpha\sigma$. Analog heißt $\varphi_1, \ldots, \varphi_n$ *Rechtsableitung von* φ, falls in jedem Schritt das am weitesten rechts stehende Nichtterminal ersetzt wird, das heißt, $\varphi_i = \sigma Aw$ und $\varphi_{i+1} = \sigma\alpha w$. Wir notieren das als

$S \underset{l}{\Rightarrow} {}^* \varphi$ $\quad$ S erzeugt φ mittels Linksableitung

$S \underset{r}{\Rightarrow} {}^* \varphi$ $\quad$ S erzeugt φ mittels Rechtsableitung

Eine Satzform innerhalb einer Linksableitung (Rechtsableitung) heißt *Linkssatzform* (*Rechtssatzform*). □

Die verschiedenen Ableitungen zu einem gegebenen Syntaxbaum für ein Wort w unterscheiden sich im Grunde nur unwesentlich, da es für die Struktur des Wortes w keine Rolle spielt, in welcher Reihenfolge Produktionen angewandt oder bei der Analyse erkannt werden. Es ist allerdings auch möglich, dass zu einem gegebenen Terminalwort verschiedene Ableitungsbäume existieren. In diesem Fall nennt man die zugrundeliegende Grammatik *mehrdeutig*. Mehrdeutigkeit ist ein ernstes Problem, da die Struktur des Syntaxbaums im Allgemeinen die Bedeutung des entsprechenden Wortes (bzw. Programmtextes) festlegt. Bei der Definition von Grammatiken für Programmiersprachen vermeidet man deshalb Mehrdeutigkeit unter allen Umständen.

3.2 Top-down-Analyse

3.2.1 Das Prinzip der Top-down-Analyse

Das Ziel der Syntaxanalyse besteht darin, zu einer gegebenen Grammatik und einer gegebenen Eingabesymbolfolge, die als Ergebnis der lexikalischen Analyse entstanden ist, einen Ableitungsbaum zu konstruieren. Wie bereits erwähnt, kann man den Baum entweder von der Wurzel aus „top-down“ oder von den Blättern aus „bottom-up“ aufbauen; wir betrachten zunächst die Top-down-Analyse.

Der Aufbau des Baumes muss natürlich irgendwie durch Betrachtung der Eingabefolge kontrolliert werden. Die Strategie dazu kann man so skizzieren: Die Blattfolge des bisher erzeugten Ableitungsbaumes wird mit der Eingabesymbolfolge verglichen, d. h., beide Symbolfolgen werden von links nach rechts gelesen. Solange beide Folgen gleiche Terminalsymbole enthalten, kann man weiterlesen. Enthält die Eingabefolge ein Terminalsymbol und das entsprechende Blatt des Baumes ein Nichtterminal, so wird eine Produktion der Grammatik ausgewählt, die auf dieses Nichtterminal anwendbar ist; dadurch wird die Blattfolge des Baumes lokal verändert. Falls zwei nicht übereinstimmende Terminalsymbole angetroffen werden, so ist entweder eine vorher getroffene Auswahl einer Produktion falsch gewesen und rückgängig zu machen, oder die Eingabefolge ist syntaktisch nicht korrekt.

Wir wollen das im Folgenden an einem Beispiel betrachten und definieren dazu eine Beispielgrammatik, die einen kleinen Ausschnitt einer imperativen Programmiersprache beschreibt. Teile dieser Grammatik haben wir schon in den einführenden Beispielen benutzt; die hier folgende vollständigere Version ist gegenüber diesen Beispielen leicht modifiziert (bedingte Anweisungen werden mit **fi** anstatt mit **end** abgeschlossen).

Wir gehen zunächst naiv vor und kümmern uns nicht um irgendwelche Anforderungen, die das spezielle Verfahren der Top-down-Analyse an eine solche Grammatik stellt. Wir werden dann sehen, welche Schwierigkeiten auftreten und wie man sie beheben kann.

stmt	→	*assignment* | *cond* | *loop*	(1)
assignment	→	**id** := *expr*	(2)
cond	→	**if** *boolexpr* **then** *stmt* **fi** |	(3)
		if *boolexpr* **then** *stmt* **else** *stmt* **fi**	
loop	→	**while** *boolexpr* **do** *stmt* **od**	(4)
expr	→	*boolexpr* | *numexpr*	(5)
boolexp	→	*numexpr* **cop** *numexpr*	(6)
numexpr	→	*numexpr* + *term* | *term*	(7)
term	→	*term* ∗ *factor* | *factor*	(8)
factor	→	**id** | **const** | (*numexpr*)	(9)

In solchen „konkreten“ Beispielgrammatiken sind Nichtterminale durch kursiv geschriebene Wortsymbole gekennzeichnet. Fettdruck stellt terminale Wortsymbole dar, die bereits durch die lexikalische Analyse als solche erkannt sind. Sonderzeichen oder Gruppen von Sonderzeichen ohne Zwischenräume (z. B. „:=“) sind ebenfalls Terminalsymbole.

Wir erläutern die Grammatik kurz für den Fall, dass Sie mit den englischen Begriffen nicht so vertraut sind: Eine Anweisung („statement“) ist eine Zuweisung, eine bedingte Anweisung oder eine Schleife (1). Eine Zuweisung weist einer Variablen, die aus der lexikalischen Analyse als **id**-Token hervorgegangen ist, den Wert eines Ausdrucks zu (2). Es gibt zwei Formen von bedingten Anweisungen (3) und eine **while**-Schleife (4). Ein Ausdruck kann ein Boolescher Ausdruck oder ein numerischer Ausdruck sein (5). Ein Boolescher Ausdruck hat hier die einfache Form „numerischer Ausdruck – Vergleichsoperator – numerischer Ausdruck“ (6). **cop** („comparison operator“) steht dabei für Operatoren der Art $\{=, \neq, <, >, \leq, \geq\}$, die alle von der lexikalischen Analyse in ein **cop**-Token verwandelt werden. Ein numerischer Ausdruck ist eine Folge von durch ein „+“-Zeichen verknüpften Termen (7). Ein Term ist analog eine Folge durch „*“ verknüpfter Faktoren (8). Ein Faktor ist ein Variablenname, eine Konstante oder ein numerischer Ausdruck in Klammern (9).

In dieser Sprache könnte man z. B. die Anweisung

```
if b > 0 then a := 1 fi
```

formulieren. Zunächst entsteht daraus als Ergebnis der lexikalischen Analyse eine Symbolfolge:

if id cop const then id := **const fi**

Wir werden nun betrachten, wie diese Anweisung mit dem Top-down-Verfahren analysiert würde. Leider wird sich bald herausstellen, dass die Top-down-Analyse für die gegebene Grammatik nicht durchführbar ist!

Das ist durchaus ein typisches Erlebnis, das wir Ihnen nicht vorenthalten wollten: Wenn man eine Grammatik auf „natürliche“ Weise definiert, um möglichst klar die Struktur der Sprache zu beschreiben, wird diese Grammatik im Allgemeinen nicht top-down analysierbar sein. Man kann aber anschließend Modifikationen vornehmen, die die Grammatik, nicht aber die Struktur der Sprache verändern, um so Top-down-Analysierbarkeit und sogar mehr als das, nämlich effiziente Analysierbarkeit, zu erreichen. Das wird im Folgenden gezeigt.

Die Produktionen, die zunächst Top-down-Analysierbarkeit überhaupt verhindern, sind (7) und (8). Das liegt daran, dass diese Produktionen *linksrekursiv* sind, d. h., das zu ersetzende Nichtterminal tritt auf der rechten Seite der Produktion wieder ganz links auf. Wir werden gleich sehen, warum das Schwierigkeiten macht. Um zunächst das Verfahren der Top-down-Analyse unbehindert erklären zu können, ersetzen wir vorübergehend diese Produktionen durch die Produktionen

numexpr → **id** | **const** (7')

Für die Analyse der obigen Beispielanweisung genügt das, da alle dort vorkommenden numerischen Ausdrücke Variablen oder Konstanten sind.

Es sei bekannt, dass eine *Anweisung* analysiert werden soll; das Startsymbol unserer Grammatik, das zur Wurzel des Ableitungsbaumes wird, ist also *stmt*. Zu Anfang erzeugt man diese Wurzel und setzt in der Eingabefolge einen Zeiger auf das erste Element.

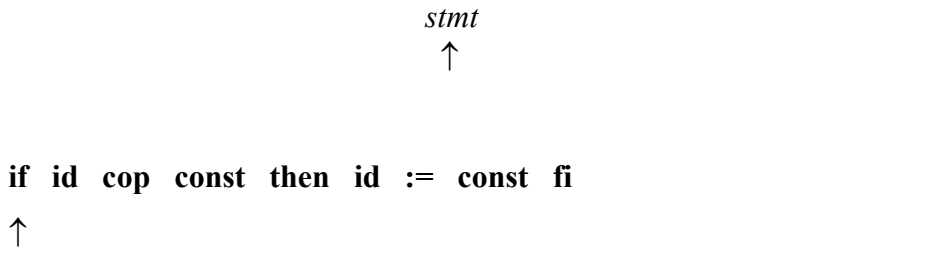


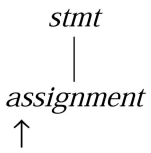
Abb. 3.5. Beispiel für Top-down-Analyse mit Backtracking (1)

Es gibt einen weiteren Zeiger in die Blattfolge des bisher erzeugten Ableitungsbaumes; dieser zeigt zu Anfang auf das einzige Blatt, das gleichzeitig die Wurzel ist.

Das Symbol **if** und das Blatt *stmt* stimmen nicht überein; das Nichtterminal *stmt* ist zu expandieren, d. h., eine Produktion für *stmt* ist auszuwählen und die rechte Seite in den Baum einzuhängen. Die erste vorkommende Produktion ist

stmt → *assignment*

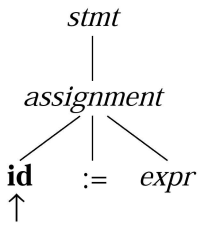
und sie wird ausgewählt:



if id cop const then id := const fi
↑

Abb. 3.6. Beispiel für Top-down-Analyse mit Backtracking (2)

Wieder steht der Zeiger im Baum auf einem Nichtterminal, also wird *assignment* expandiert. In diesem Fall gibt es nur eine Möglichkeit.



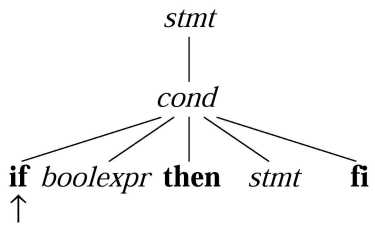
if id cop const then id := const fi
↑

Abb. 3.7. Beispiel für Top-down-Analyse mit Backtracking (3)

An dieser Stelle stehen beide Zeiger auf Terminalsymbolen, die aber nicht übereinstimmen. Wir betrachten zur Zeit noch den allgemeinsten Fall der Top-down-Analyse, nämlich die *Top-down-Analyse mit Backtracking*. Bei der Analyse ohne Backtracking, die wir später studieren, läge jetzt ein Syntaxfehler vor; so aber sind hier vorher getroffene Auswahlen von rechten Seiten zu revidieren. Für die Ersetzung von *assignment* gab es keine weitere Möglichkeit, deshalb ist auch die Auswahl

stmt → *assignment*

zu revidieren und die zweite Alternative für *stmt* auszuwählen. Weiter wählen wir die erste Alternative für *cond* und erhalten

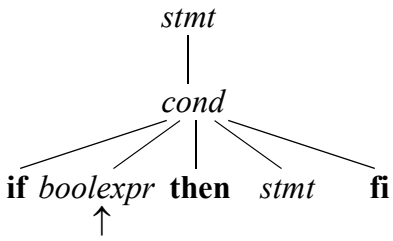


if id cop const then id := const fi

↑

Abb. 3.8. Beispiel für Top-down-Analyse mit Backtracking (4)

Nun stimmen die Terminalsymbole überein, und die beiden Zeiger rücken auf die nächste Position:

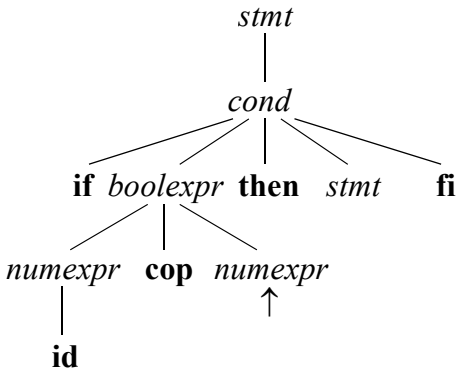


if id cop const then id := const fi

↑

Abb. 3.9. Beispiel für Top-down-Analyse mit Backtracking (5)

Für *boolexpr* gibt es nur eine Möglichkeit, danach wird für *numexpr* die erste Alternative aus (7') ausgewählt, wonach die Zeiger um zwei Symbole vorrücken können:

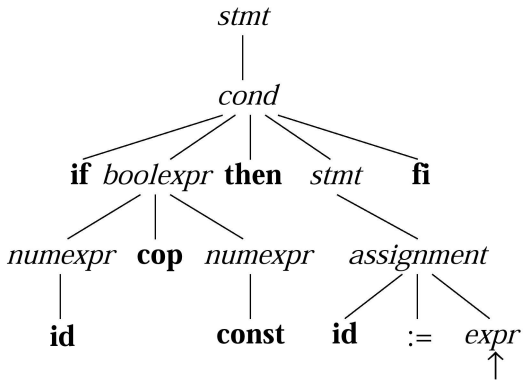


if id cop const then id := const fi

↑

Abb. 3.10. Beispiel für Top-down-Analyse mit Backtracking (6)

Anschließend wird *numexpr* zunächst fälschlich zu **id** expandiert; das wird dann zurückgenommen und die Alternative **const** gewählt, woraufhin Übereinstimmung festgestellt wird und die Zeiger über **then** bis auf *stmt* und **id** vorrücken. Für *stmt* wird die *assignment*-Alternative gewählt und *assignment* expandiert; daraufhin können beide Zeiger über **id** und „:=“ bis auf *expr* und **const** weiterrücken, so dass wir den folgenden Zustand erhalten:

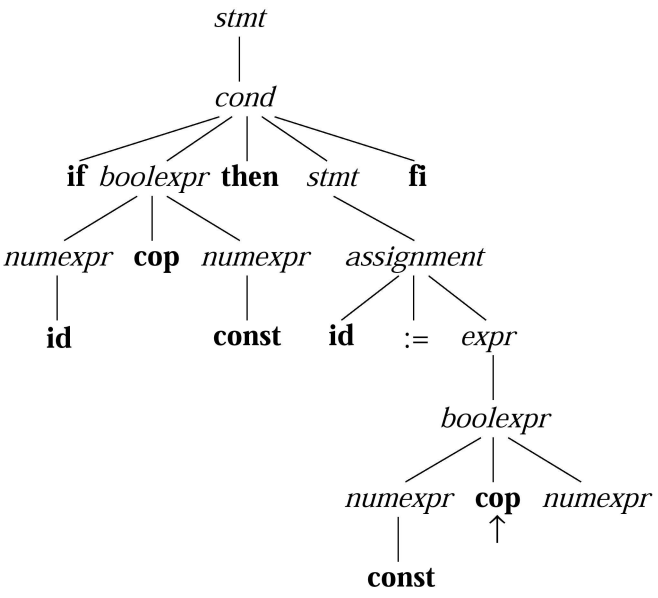


if id cop const then id := const fi

↑

Abb. 3.11. Beispiel für Top-down-Analyse mit Backtracking (7)

Das Verfahren läuft an dieser Stelle in eine relativ lange Sackgasse: *expr* wird mit *boolexpr* expandiert, dieses durch „*numexpr* **cop** *numexpr*“, dann *numexpr* zunächst durch **id**, anschließend durch **const,** woraufhin die Zeiger weiterrücken und wir die Situation erhalten:



if id cop const then id := const fi
↑

Abb. 3.12. Beispiel für Top-down-Analyse mit Backtracking (8)

Da nun **fi** und **cop** nicht übereinstimmen, müssen alle diese Entscheidungen zurückgenommen werden. Übrigens muss auch der Zeiger in der Eingabefolge zurückgesetzt werden. Anschließend wird *expr* mit *numexpr* expandiert, woraufhin schließlich die Analyse erfolgreich beendet werden kann mit dem in Abb. 3.13 gezeigten Ableitungsbaum.

Es ist nun klar, warum sich unsere ursprünglich gewählte Grammatik mit den linksrekursiven Produktionen (7) und (8) für die Top-down-Analyse nicht eignet: In einer Situation, in der *numexpr* aktuelles Blatt des Ableitungsbaumes ist, wird die rechte Seite „*numexpr* + *term*“ eingehängt, woraufhin *numexpr* wiederum aktuelles Blatt ist und sich der Zeiger in der Eingabefolge auch nicht bewegt hat. Das Verfahren würde also in eine Endlosschleife geraten.

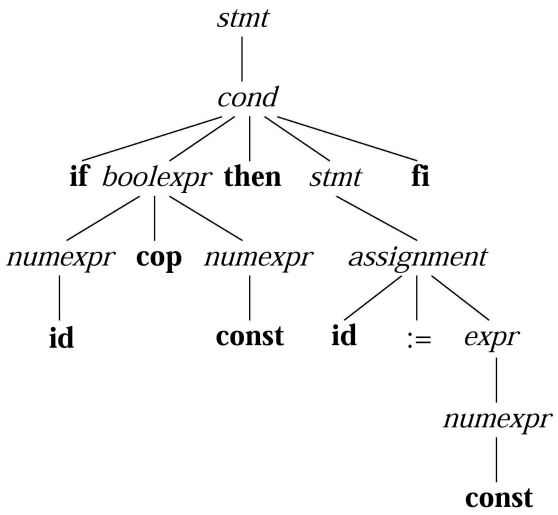


Abb. 3.13. Beispiel für Top-down-Analyse mit Backtracking (9)

Wir sehen, dass bei der Top-down-Analyse zwei Probleme auftreten, nämlich *linksrekursive Produktionen* und *Verlaufen in Sackgassen*. Das erste Problem macht diese Art der Analyse ganz unmöglich; das zweite macht sie wegen Ineffizienz für die Praxis uninteressant. Unser Ziel wird also darin bestehen, Grammatiken so zu konstruieren bzw. unsere Beispielgrammatik so zu modifizieren, dass beide Probleme nicht auftreten.

Beseitigung von Linksrekursion

Zunächst müssen wir Linksrekursion etwas genauer charakterisieren. Wir haben bisher *direkte* Linksrekursion beobachtet; diese liegt vor, wenn es Produktionen der Form

$$A \rightarrow A\alpha$$

gibt. *Indirekte* Linksrekursion liegt vor, wenn es zu einem Nichtterminal A eine Ableitung

$$A \Rightarrow^* A\alpha$$

gibt.

In vielen praktischen Fällen kommt nur direkte Linksrekursion vor; diese kann mit relativ einfachen Techniken beseitigt werden. Betrachten wir ein Paar linksrekursiver Produktionen

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

wobei α, β Folgen von terminalen und nichtterminalen Symbolen sind, die nicht mit *A* beginnen. Durch die beiden Produktionen werden Bäume der in Abb. 3.14 (a) dargestellten Form bzw. Folgen der Form βα* erzeugt.

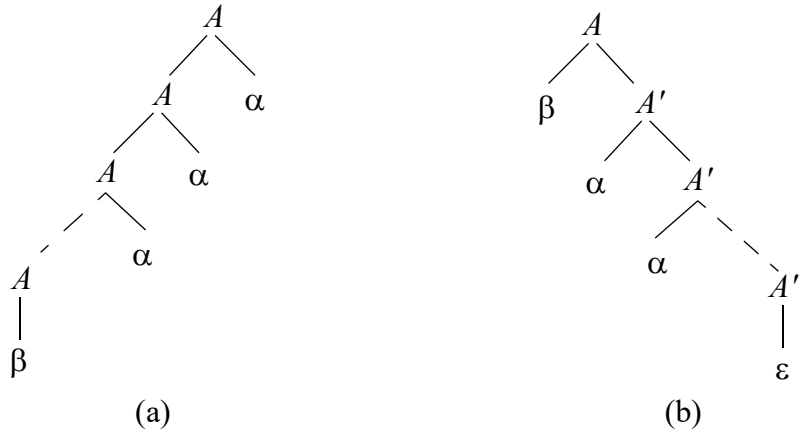


Abb. 3.14. Ableitungsbaum mit (a) linksrekursiver bzw. (b) rechtsrekursiver Produktion

Dieselben Folgen kann man erzeugen mit Produktionen

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

wobei *A*' ein neues Nichtterminal und die zweite Produktion *rechtsrekursiv* ist. Die entstehenden Bäume haben nun die in Abb. 3.14 (b) gezeigte Gestalt.

Wie man an den Ableitungsbäumen sieht, ändert sich aber bei solchen Grammatikänderungen auch die Assoziativität, d. h., bei Linksrekursion sind Operatoren linksassoziativ, bei Rechtsrekursion umgekehrt. Dies mußssman bei der Weiterverarbeitung der Ableitungsbäume beachten.

Wenn wir diese Technik auf die linksrekursiven Produktionen (7)

$$\textit{numexpr} \rightarrow \textit{numexpr} + \textit{term} \mid \textit{term}$$

unserer Beispielgrammatik anwenden, erhalten wir

$$\textit{numexpr} \rightarrow \textit{term}\ \ \textit{numexpr}' \tag{7a}$$

$$\textit{numexpr}' \rightarrow + \textit{term}\ \ \textit{numexpr}' \mid \varepsilon \tag{7b}$$

Ebenso kann man (8) ersetzen durch

$$\textit{term} \rightarrow \textit{factor}\ \ \textit{term}' \tag{8a}$$

$$\textit{term}' \rightarrow * \textit{factor}\ \ \textit{term}' \mid \varepsilon \tag{8b}$$

Wir werden im Folgenden mit der so modifizierten Version der Grammatik weiterarbeiten.

Die Technik lässt sich verallgemeinern für den Fall, dass mehr als zwei A-Produktionen vorhanden sind. Seien

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$$

alle A-Produktionen, wobei die β_i nicht mit A beginnen. Wir ersetzen die A-Produktionen durch

$$A \rightarrow \beta_1 A' \mid \ldots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

Auf diese Art lässt sich also jede Form direkter Linksrekursion beseitigen. Ein Algorithmus zur Beseitigung auch indirekter Linksrekursion findet sich in (Aho, Sethi und Ullman 1986, Abschnitt 4.3).

Unsere Beispielgrammatik eignet sich nun schon prinzipiell für die Top-down-Analyse. Wie kann man es nun noch vermeiden, bei der Analyse in Sackgassen zu laufen? Die Idee dazu ist eigentlich sehr einfach. Betrachten wir noch einmal die Ausgangssituation im Beispiel (Abb. 3.5).

Das Verfahren lief sofort in eine Sackgasse, weil sozusagen „blind“ *stmt* zu *assignment* expandiert wurde. Durch Ansehen des aktuellen Symbols **if** in der Eingabefolge hätte man aber ohne weiteres feststellen können, dass es sich um eine bedingte Anweisung handelt, dass also die Alternative *cond* auszuwählen ist.

Die Idee des „*predictive parsing*“, also der „vorausschauenden Syntaxanalyse“ (wir übersetzen „predictive parser“ mit „vorgreifendem Analysator“), besteht nun darin, die gegebene Grammatik vorher zu analysieren und sich irgendwo, etwa in einer Tabelle, zu merken, dass beim Auftreten von *stmt* im Ableitungsbaum und **if** in der Eingabefolge die Produktion

stmt → *cond*

auszuwählen ist. Beim Analysieren der Eingabefolge sieht man dann einfach nach und wählt direkt die richtige Produktion.

Dieses „predictive parsing“ ist die eigentlich interessante Form der Top-down-Analyse; es läuft nicht in Sackgassen und kommt daher ohne Backtracking aus. Die Eingabefolge braucht nur einmal sequentiell gelesen zu werden.

Allerdings ist die Frage, ob man denn durch Ansehen des aktuellen Symbols der Eingabefolge immer entscheiden kann, welche Produktion auszuwählen ist. Das ist leider im Allgemeinen nicht der Fall. Betrachten wir z. B. die Produktionen

cond → **if** *boolexpr* **then** *stmt* **fi** |
if *boolexpr* **then** *stmt* **else** *stmt* **fi**

Durch Ansehen des ersten Symbols **if** kann man nicht entscheiden, ob die erste oder zweite Alternative auszuwählen ist. Tatsächlich kann die Entscheidung sogar erst sehr viel später, nämlich beim Lesen von **fi** oder **else**, getroffen werden.

Die Klasse von Grammatiken, bei denen immer eine eindeutige Entscheidung durch Ansehen der nächsten k Terminalsymbole getroffen werden kann, nennt man *LL(k)-Grammatiken*. Diese Klasse werden wir im folgenden Abschnitt formal charakterisieren. Danach werden wir in Abschnitt 3.2.3 unsere Beispielgrammatik so umformen, dass sie vom Typ LL(1) wird, dass also durch Ansehen des aktuellen Symbols eine sackgassenfreie Analyse möglich wird.

Selbsttestaufgabe 3.1: Beseitigen Sie die Linksrekursion in den Produktionen

$A \rightarrow A\mathbf{b} \mid AB\mathbf{cd} \mid \mathbf{e} \mid \mathbf{f}$ □

3.2.2 LL(*k*)-Grammatiken

Wie muss eine kontextfreie Grammatik beschaffen sein, damit eine sackgassenfreie Analyse unter Vorausschau auf die jeweils nächsten k Zeichen möglich ist? Die folgende Definition verrät uns noch nicht allzuviel darüber, sondern verlangt einfach, dass die im Analyseprozess zu treffende Entscheidung, durch welche rechte Seite ein Nichtterminal expandiert werden soll, eindeutig ist. Die Definition spricht von Linksableitungen; man kann sich leicht klar machen, dass in der Top-down-Analyse Linksableitungen berechnet werden, da ja in der Blattfolge des Syntaxbaumes (die gerade einer Linkssatzform entspricht) jeweils das am weitesten links stehende Nichtterminal expandiert wird.

Zunächst benötigen wir noch eine technische Definition, die „Vorausschau auf die nächsten k Zeichen“ formalisiert.

Definition 3.7: Sei $L \subseteq \Sigma^*$ eine beliebige Sprache und sei $k > 0$. Dann ist

$$start_k(L) := \{w \mid (w \in L \text{ und } |w| < k) \text{ oder (es existiert } wu \in L \text{ und } |w| = k)\}$$

Für ein Wort $v \in \Sigma^*$ sei

$$start_k(v) := \begin{cases} v & \text{falls } |v| < k \\ u & \text{falls } u, t \text{ existieren mit } |u| = k, ut = v \end{cases}$$

Dabei bezeichnet $|w|$ die Länge von w, d. h. die Anzahl der Zeichen in w. Die Funktion $start_k$ liefert also Worte oder Anfangsstücke von Worten bis zur Länge k einer Sprache oder eines Wortes. □

Definition 3.8: Eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$ heißt *LL(k)-Grammatik*, wenn gilt: Aus

$$S \underset{l}{\Rightarrow}^* wA\sigma \underset{l}{\Rightarrow} w\alpha\sigma \underset{l}{\Rightarrow}^* wx,$$

$$S \underset{l}{\Rightarrow}^{*} wA\sigma \underset{l}{\Rightarrow} w\beta\sigma \underset{l}{\Rightarrow}^{*} wy,$$

und $start_k(x) = start_k(y)$

folgt $\alpha = \beta$. □

Das bedeutet also, wenn das Terminalwort w gelesen ist und man aus $A\sigma$ Terminalworte x und y ableiten kann, die mit den gleichen k Zeichen beginnen, dann gibt es nur eine einzige Produktion $A \rightarrow \alpha$, über die x und y abgeleitet werden können. Die allgemeine Situation innerhalb der LL(k)-Analyse ist in Abb. 3.15 dargestellt. Dort wird oben der Ableitungsbaum mit seiner Blattfolge gezeigt.

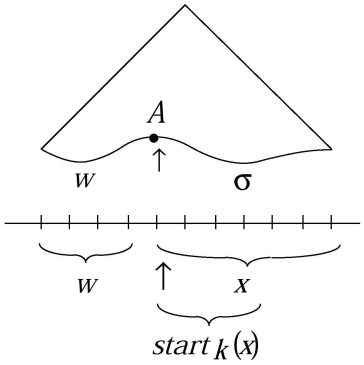


Abb. 3.15. Allgemeine Situation innerhalb der LL(k)-Analyse

Die Bezeichnung LL(k) steht übrigens für „Lesen von **l**inks nach rechts, Berechnen einer **L**inksableitung unter Vorausschau auf die nächsten ***k*** Zeichen“.

Für die Praxis interessanter ist eine leicht eingeschränkte Klasse von LL(k)-Grammatiken, die sogenannten *starken LL*(k)-Grammatiken.

Definition 3.9: Eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$ heißt *starke LL*(k)-*Grammatik*, wenn gilt: Aus

$$S \underset{l}{\Rightarrow}^{*} w_1 A\sigma_1 \underset{l}{\Rightarrow} w_1\alpha\sigma_1 \underset{l}{\Rightarrow}^{*} w_1 x,$$

$$S \underset{l}{\Rightarrow}^{*} w_2 A\sigma_2 \underset{l}{\Rightarrow} w_2\beta\sigma_2 \underset{l}{\Rightarrow}^{*} w_2 y,$$

und $start_k(x) = start_k(y)$

folgt $\alpha = \beta$. □

Der Unterschied liegt darin, dass es in Definition 3.8 auf den Kontext der Ableitung ankommt; die Entscheidung zwischen α und β muss nur eindeutig getroffen werden

können, wenn wir die Linkssatzform $wA\sigma$ kennen. In Definition 3.9, also bei starken LL(k)-Grammatiken, spielt die Umgebung w und σ keine Rolle, sie kann variieren.

Unser Ziel besteht nun darin, eine praktischere Charakterisierung von LL(k)-Grammatiken zu finden, die auch „implementierbar“ ist. Das konkrete Problem innerhalb der Top-down-Analyse besteht offensichtlich darin, die richtige rechte Seite auszuwählen, wenn ein Nichtterminal A expandiert werden soll. Seien

$$A \quad \rightarrow \quad \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

alle A-Produktionen. Wir können vorausschauen auf die nächsten k Terminalzeichen. Also ist es von Bedeutung, die Menge der Terminalworte zu kennen, die aus einer Zeichenfolge α ableitbar sind.

Definition 3.10: Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik, $\alpha \in (N \cup \Sigma)^*$ und $k > 0$. Dann ist

$$\text{FIRST}_k(\alpha) := start_k(\{w \mid \alpha \Rightarrow^* w\})$$

Die Menge $\text{FIRST}_k(\alpha)$ beschreibt also gerade die Anfangsstücke bis zur Länge k von aus α ableitbaren Terminalworten. □

Wenn ein Wort aus $\text{FIRST}_k(\alpha_i)$ kürzer als k ist, dann wird die Vorausschau auf die nächsten k Zeichen noch Zeichen enthalten, die nicht aus α_i abgeleitet sind, sondern aus der „Umgebung“, in der das Nichtterminal A stand. Diese Umgebung beschreibt man über eine Menge $\text{FOLLOW}_k(A)$.

Definition 3.11: Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik, $A \in N$, $k > 0$.

$$\text{FOLLOW}_k(A) := \{w \mid S \Rightarrow^* uAv \text{ und } w \in \text{FIRST}_k(v)\}$$

$\text{FOLLOW}_k(A)$ beschreibt also Terminalzeichenfolgen bis zur Länge k, die innerhalb von Ableitungen in G auf das Nichtterminal A folgen können. □

Nehmen wir nun an, dass innerhalb der Top-down-Analyse das Nichtterminal A expandiert werden soll. Falls die rechte Seite α_i in der Ableitung gewählt wurde (α_i also die richtige Entscheidung ist), dann muss die Folge der nächsten k Zeichen, auf die wir vorausschauen, in der Konkatenation der Mengen $\text{FIRST}_k(\alpha_i)$ und $\text{FOLLOW}_k(A)$ liegen. Wir definieren derartige Konkatenationen als *Steuermengen*.

Definition 3.12: Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik, $A \in N$, $k > 0$, und sei $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ die Menge der A-Produktionen. Dann ist für $1 \leq i \leq n$ die *Steuermenge* $D_k(A \rightarrow \alpha_i)$ definiert als

$$D_k(A \rightarrow \alpha_i) := start_k(\text{FIRST}_k(\alpha_i)\ \text{FOLLOW}_k(A)) \qquad \square$$

Anstelle von $D_k(A \rightarrow \alpha_i)$ schreiben wir gelegentlich auch kurz $D_k(\alpha_i)$, wenn klar ist, um welches A es geht. Der Buchstabe D steht für „director set“. In der Definition sind $\text{FIRST}_k(\alpha_i)$ und $\text{FOLLOW}_k(A)$ ja jeweils Wortmengen, also Sprachen; die Konkatenation von Sprachen hatten wir bereits in Kapitel 2 definiert. Wir wählen Anfangsstücke der Konkatenation ebenfalls bis zur Länge k.

Die Entscheidung unter den α_i kann nun eindeutig getroffen werden, wenn die Mengen $D_k(A \to \alpha_1)$, …, $D_k(A \to \alpha_n)$ alle paarweise disjunkt sind, es also kein Anfangsstück eines abgeleiteten Terminalwortes gibt, das in zwei oder mehr dieser Mengen vorkommt.

Satz 3.13: Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik. G ist eine starke LL(k)-Grammatik genau dann, wenn für jedes Nichtterminal A mit einer Menge von A-Produktionen $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ gilt

$$D_k(A \to \alpha_i) \cap D_k(A \to \alpha_j) = \varnothing \qquad \text{für alle } i, j \in \{1, \ldots, n\} \text{ mit } i \neq j$$

Beweis: „⇐“ Wir nehmen an, die Bedingung des Satzes gilt, aber G ist keine starke LL(k)-Grammatik. Dann gibt es ein Nichtterminal A und Ableitungen

$$S \underset{l}{\Rightarrow}^* w_1 A \sigma_1 \underset{l}{\Rightarrow} w_1 \alpha \sigma_1 \underset{l}{\Rightarrow}^* w_1 x,$$

$$S \underset{l}{\Rightarrow}^* w_2 A \sigma_2 \underset{l}{\Rightarrow} w_2 \beta \sigma_2 \underset{l}{\Rightarrow}^* w_2 y,$$

für die gilt $start_k(x) = start_k(y)$, aber $\alpha \neq \beta$. Für die beiden A-Produktionen $A \to \alpha \mid \beta$ gilt dann aber $start_k(x) \in D_k(A \to \alpha)$ und $start_k(x) \in D_k(A \to \beta)$, somit also

$$start_k(x) \in D_k(A \to \alpha) \cap D_k(A \to \beta)$$

Das ist aber ein Widerspruch zur Annahme.

„⇒“ Sei nun G starke LL(k)-Grammatik, und nehmen wir an, dass ein z existiert, das im Durchschnitt zweier Steuermengen liegt, also $z \in D_k(A \to \alpha) \cap D_k(A \to \beta)$ mit $\alpha \neq \beta$.

Fall 1: $|z| = k$. Betrachten wir eine Ableitung $S \underset{l}{\Rightarrow}^* wA\sigma$. Dann kann A durch α oder β ersetzt werden, was zu Ableitungen führt

$$S \underset{l}{\Rightarrow}^* wA\sigma \underset{l}{\Rightarrow} w\alpha\sigma \underset{l}{\Rightarrow}^* wzu,$$

$$S \underset{l}{\Rightarrow}^* wA\sigma \underset{l}{\Rightarrow} w\beta\sigma \underset{l}{\Rightarrow}^* wzv$$

Diese Ableitungen existieren, da z in beiden Steuermengen liegt. Nun gilt $start_k(zu) = z = start_k(zv)$, aber $\alpha \neq \beta$, ein Widerspruch zur Annahme, dass G (starke) LL(k)-Grammatik ist.

Fall 2: $|z| < k$. Dann muss es Linksableitungen

$$S \underset{l}{\Rightarrow}^* w_1 A \sigma_1 \underset{l}{\Rightarrow} w_1 \alpha \sigma_1 \underset{l}{\Rightarrow}^* w_1 x,$$

$$S \underset{l}{\Rightarrow}^* w_2 A \sigma_2 \underset{l}{\Rightarrow} w_2 \beta \sigma_2 \underset{l}{\Rightarrow}^* w_2 y$$

geben, für die $x = z = y$ gilt. Denn ein z, das weniger als k Zeichen hat, kann nur dann in der Steuermenge der Produktion $A \to \alpha$ liegen, wenn es eine Linksableitung in ein Terminalwort gibt, bei der auf z keine weiteren Zeichen folgen. Dies sei die erste Produktion, also gilt $x = z$. Das gleiche Argument gilt für die Steuermenge von $A \to \beta$, so dass gilt $y = z$, also auch $x = z = y$. Folglich gilt aber auch $start_k(x) = x = z = y = start_k(y)$, aber $\alpha \neq \beta$, ein Widerspruch zur Annahme, dass G starke LL(k)-Grammatik ist. □

Von besonderer Bedeutung für den Übersetzerbau sind (starke) LL(1)-Grammatiken, da das Vorausschauen um genau ein Zeichen am einfachsten zu handhaben ist. In diesem Fall lässt sich die Bedingung über die Disjunktheit der Steuermengen so formulieren:

Eine kontextfreie Grammatik ist genau dann eine LL(1)-Grammatik, wenn für jedes Nichtterminal A mit A-Produktionen $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ gilt:

(i) Die Mengen $\text{FIRST}_1(\alpha_1)$, …, $\text{FIRST}_1(\alpha_n)$ sind paarweise disjunkt.
(ii) Genau eine der Mengen $\text{FIRST}_1(\alpha_1)$, …, $\text{FIRST}_1(\alpha_n)$ darf das leere Wort ε enthalten. Wenn $\varepsilon \in \text{FIRST}_1(\alpha_i)$, dann gilt: $\text{FOLLOW}_1(A)$ ist disjunkt von allen anderen Mengen $\text{FIRST}_1(\alpha_j)$, $j \neq i$.

Falls $k = 1$, reduziert sich die Definition der Steuermengen zu:

$$D(\alpha_i) := \begin{cases} \text{FIRST}_1(\alpha_i) & \text{falls } \varepsilon \notin \text{FIRST}_1(\alpha_i) \\ \text{FIRST}_1(\alpha_i) - \{\varepsilon\} \cup \text{FOLLOW}_1(A) & \text{sonst} \end{cases}$$

Im folgenden Abschnitt werden wir sehen, wie man FIRST- und FOLLOW-Mengen bzw. Steuermengen berechnen kann.

3.2.3 Berechnung von FIRST- und FOLLOW-Mengen, Modifikation von Grammatiken

Wir haben im letzten Abschnitt gesehen, dass die Kenntnis von FIRST- und FOLLOW-Mengen[1] entscheidende Voraussetzung für die Top-down-Analyse ohne Backtracking ist. Gleichzeitig kann man anhand dieser Mengen entscheiden, ob eine Grammatik vom Typ LL(1) ist, und störende Produktionen erkennen. In diesem Abschnitt wollen wir einerseits sehen, wie diese Mengen berechnet werden können, und andererseits unsere Beispielgrammatik daran überprüfen und ggf. verändern.

Wir hatten in Abschnitt 3.2.1 schon ein Problem „mit bloßem Auge“ erkannt: Die Produktionen (3)

cond → **if** *boolexpr* **then** *stmt* **fi** |
if *boolexpr* **then** *stmt* **else** *stmt* **fi**

[1] Wenn im folgenden von FIRST- bzw. FOLLOW-Mengen die Rede ist, sind immer die FIRST_1- bzw. FOLLOW_1-Mengen gemeint, sofern nicht explizit anders gekennzeichnet.

erlauben es nicht, durch Ansehen des ersten Symbols **if** die richtige Alternative eindeutig zu bestimmen. Wir betrachten jetzt eine Technik, solche Produktionen durch „LL(1)-fähige“ zu ersetzen.

Links-Faktorisierung

Das Problem tritt offensichtlich dann auf, wenn für ein Nichtterminal A die rechten Seiten verschiedener A-Produktionen ein gemeinsames Präfix haben, z. B.

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

Die Idee besteht nun darin, die Produktionen so umzuschreiben, dass bei der Analyse zunächst eindeutig eine Produktion ausgewählt werden kann, die das gemeinsame Präfix erzeugt, und die Entscheidung über die Fortsetzung verschoben werden kann, bis das Präfix abgearbeitet ist. Das heißt, wir wählen hier Produktionen

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \mid \beta_2 \end{aligned}$$

wobei A' ein neues Nichtterminal ist.

Mit dieser Technik können wir unsere Beispielgrammatik so modifizieren:

cond → **if** *boolexpr* **then** *stmt cond-rest*
cond-rest → **fi** | **else** *stmt* **fi**

Die Technik lässt sich verallgemeinern für den Fall, dass mehr als zwei Alternativen ein gemeinsames Präfix besitzen: Seien die A-Produktionen

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_m \mid \gamma_1 \mid \ldots \mid \gamma_n$$

mit $\alpha \neq \varepsilon$, wobei die γ_i nicht mit α beginnen. Dann kann man diese Produktionen ersetzen durch

$$\begin{aligned} A &\rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_n \\ A' &\rightarrow \beta_1 \mid \ldots \mid \beta_m \end{aligned}$$

mit einem neuen Nichtterminal A'.

Selbsttestaufgabe 3.2: Gegeben sei die Grammatik $G = (N, \Sigma, P, \mathit{modhead})$ mit

N = {*modhead*, *lists*, *implist*, *explist*, *A*, *B*, *idlist*}
Σ = {**MODULE**, **id**, **IMPORT**, **FROM**, **EXPORT**, **QUALIFIED**, **;**}
P = { *modhead* → **MODULE id ;** *lists*
lists → *implist explist* | *implist* | *explist* | ε
implist → *A* **IMPORT** *idlist* | *A* **IMPORT** *idlist implist*
A → **FROM id** | ε

explist → **EXPORT** *B idlist*
B → **QUALIFIED** | ε
idlist → **id ;** | *idlist* **id ;**
}.

G beschreibt den (etwas vereinfachten) Kopf eines Moduls der Sprache Modula-2. Geben Sie eine äquivalente Grammatik *G′* mit „LL(1)-fähigen" Produktionen an, indem Sie Linksrekursion beseitigen und Links-Faktorisierung durchführen. □

Wir wenden uns nun der Berechnung von FIRST- und FOLLOW-Mengen zu, um anschließend überprüfen zu können, ob die Beispielgrammatik inzwischen vom Typ LL(1) ist. Unser Ziel besteht zunächst darin, FIRST(X) für ein beliebiges Symbol $X \in (N \cup \Sigma)$ zu bestimmen. Der Fall $X \in \Sigma$ ist trivial: FIRST(X) = $\{X\}$. Für ein Nichtterminal $X = A \in N$ ist es sinnvoll, die Menge aller A-Produktionen

$$A \quad \rightarrow \quad \alpha_1 \mid \ldots \mid \alpha_n$$

gemeinsam zu betrachten und dabei neben FIRST(A) auch die Mengen FIRST(α_1), …, FIRST(α_n) zu berechnen. Letztere können wir auch als „initiale" Steuermengen auffassen (die endgültigen Steuermengen werden durch Einbezug der FOLLOW-Mengen gebildet).

Algorithmus 3.14: Berechnung von FIRST-Mengen und initialen Steuermengen

Eingabe Grammatik G und Menge von A-Produktionen $A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$

Ausgabe FIRST(A) und initiale Steuermengen $D_1, \ldots, D_n$, wobei D_i = FIRST(α_i) mit $i = 1 \ldots n$

Methode

```
FIRST(A) := ∅;
for i := 1 to n do
   if α_i = ε then (die Produktion hat die Form A → ε)
      D_i := {ε}
   else (die Produktion hat die Form A → X_1 X_2 … X_m)
      D_i := FIRST(X_1) \ {ε}; j := 1
      while ε ∈ FIRST(X_j) and j < m do
         j := j + 1;
         D_i := D_i ∪ FIRST(X_j) \ {ε}
      od;
      if j = m and ε ∈ FIRST(X_m) then
         D_i := D_i ∪ {ε}
      fi
   fi;
   FIRST(A) := FIRST(A) ∪ D_i
end for
```

FIRST(A) ist also die Vereinigung aller FIRST(α_i), und für eine Regel $A \to X_1 \ldots X_m$ ist das im Wesentlichen FIRST(X_1). Falls X_1 ein Terminalsymbol ist oder ein Nichtterminal, aus dem nicht ε abzuleiten ist, dann ist das alles. Nur für den Fall, dass FIRST(X_1) ε enthält, muss man FIRST(X_2) hinzunehmen, falls das ε enthält, auch FIRST(X_3) usw.

In einer Grammatik mit den Regeln

$$S \to ABcd \quad (1)$$
$$A \to a \mid B \quad (2)$$
$$B \to b \mid \varepsilon \quad (3)$$

wenden wir den Algorithmus nacheinander auf die Gruppen von Produktionen (3), (2), (1) an und erhalten FIRST-Mengen und Steuermengen:

$\{b, \varepsilon\}$	$B \to$	$b \mid$	$\{b\}$
		ε	$\{\varepsilon\}$
$\{a, b, \varepsilon\}$	$A \to$	$a \mid$	$\{a\}$
		B	$\{b, \varepsilon\}$
$\{a, b, c\}$	$S \to$	$ABcd$	$\{a, b, c\}$

Dabei haben wir FIRST-Mengen links und Steuermengen rechts neben Produktionen notiert. Man kann den obigen Algorithmus nun rekursiv benutzen, d. h., falls bei der Berechnung von FIRST(A) anhand der Produktion

$$A \to X_1 \ldots X_m$$

eine Menge FIRST(X_j) für ein Nichtterminal X_j benötigt wird, wird der Algorithmus für X_j aufgerufen. Für die praktische Berechnung „von Hand“ ist es interessant, eine Reihenfolge festzulegen, in der Nichtterminale betrachtet werden, so dass man die benötigten FIRST-Mengen jeweils schon zur Verfügung hat. Dazu kann man so vorgehen:

1. Bestimme eine Menge N_ε aller Nichtterminale, aus denen das leere Wort abgeleitet werden kann, also $N_\varepsilon := \{X \in N \mid X \Rightarrow^* \varepsilon\}$. (Überlegen Sie selbst, wie das geschehen könnte.)
2. Man zeichne einen Graphen, dessen Knoten die Nichtterminale sind. Für jede Produktion

   $$A \to X_1 \ldots X_m$$

   mit dem Nichtterminal X_1 füge man eine gerichtete Kante

   $$A \to X_1$$

   ein. Falls $X_1 \in N_\varepsilon$ und X_2 ein Nichtterminal ist, füge man auch Kante

   $$A \to X_2$$

   hinzu usw.

Eine Kante $A \rightarrow B$ drückt aus: FIRST(B) sollte vor FIRST(A) berechnet werden. Man kann nun den Graphen von den Blättern her abarbeiten, d. h. FIRST(X) erst dann berechnen, wenn die FIRST-Mengen aller Nachfolger von X bereits bekannt sind.

Selbsttestaufgabe 3.3: Dieses Verfahren funktioniert natürlich nicht, falls es Zyklen im Graphen gibt. Kann es Zyklen geben? Eine äquivalente Frage ist die, ob der obige FIRST-Algorithmus mit rekursiven Aufrufen mit Sicherheit terminiert. □

Wir wenden die Technik nun zur Berechnung von FIRST-Mengen für unsere Beispielgrammatik an. Hier noch einmal die aktuelle Version:

stmt	→	*assignment* |
		cond |
		loop
assignment	→	**id** := *expr*
cond	→	**if** *boolexpr* **then** *stmt* *cond-rest*
cond-rest	→	**fi** |
		else *stmt* **fi**
loop	→	**while** *boolexpr* **do** *stmt* **od**
expr	→	*boolexpr* |
		numexpr
boolexpr	→	*numexpr* **cop** *numexpr*
numexpr	→	*term* *numexpr'*
numexpr'	→	+ *term* *numexpr'* |
		ε
term	→	*factor* *term'*
term'	→	* *factor* *term'* |
		ε
factor	→	**id** |
		const |
		(*expr*)

Im ersten Schritt bestimmen wir Nichtterminale, aus denen ε abgeleitet werden kann:

$N_\varepsilon = \{term', numexpr'\}$

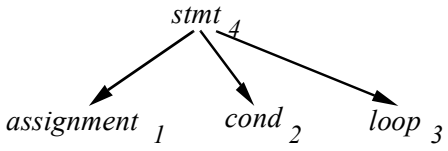
Dann zeichnen wir den Graphen, der die Berechnungsreihenfolge festlegt (Abb. 3.16). Die Zahlen an den Knoten geben die Bearbeitungsreihenfolge für den nächsten Schritt an.[2] Wir benutzen nun den Algorithmus FIRST:

{**id**}	*assignment*	→	**id** := *expr*	{**id**}
{**if**}	*cond*	→	**if** *boolexpr* **then** *stmt* *cond-rest*	{**if**}
{**while**}	*loop*	→	**while** *boolexpr* **do** *stmt* **od**	{**while**}
{**id**, **if**, **while**}	*stmt*	→	*assignment* |	{**id**}

[2] Wir haben Nichtterminale ohne Kanten weggelassen. Diese können in beliebiger Reihenfolge bearbeitet werden.

			cond |	{**if**}
			loop	{**while**}
{**id**, **const**, (}	*factor*	→	**id** |	{**id**}
			const |	{**const**}
			(*expr*)	{(}
{**id**, **const**, (}	*term*	→	*factor term*'	{**id**, **const**, (}
{**id**, **const**, (}	*numexpr*	→	*term numexpr*'	{**id, const**, (}
{**id**, **const**, (}	*boolexpr*	→	*numexpr* **cop** *numexpr*	{**id, const**, (}
{**id, const**, (}	*expr*	→	*boolexpr* |	{**id**, **const**, (}
			numexpr	{**id**, **const**, (}

Achtung: Die Alternativen für *boolexpr* und *numexpr* sind nicht disjunkt. Die Grammatik ist damit nicht vom Typ LL(1).



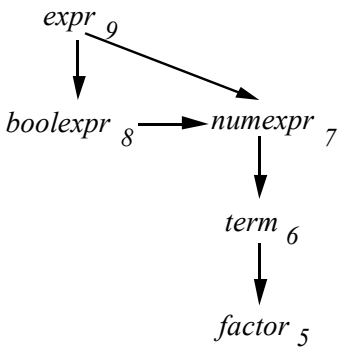


Abb. 3.16. Bestimmung der Berechnungsreihenfolge von FIRST-Mengen

Das Problem liegt offensichtlich darin, dass ein Ausdruck stets mit einem numerischen Ausdruck beginnt; ob er ein Boolescher Ausdruck ist, kann erst beim Antreffen eines **cop**-Symbols entschieden werden. Das Problem kann wieder mit Links-Faktorisierung behoben werden. Wir nehmen die Produktionen

expr → *boolexpr* | *numexpr*

heraus und ersetzen sie durch

expr → *numexpr bool-rest*
bool-rest → **cop** *numexpr* | ε

Im Graphen verschwindet dadurch die Kante *expr* → *boolexpr*, und das Verfahren lässt sich problemlos zu Ende führen. Die endgültige Version wird am Ende dieses Abschnitts vorgestellt.

Selbsttestaufgabe 3.4: Bestimmen Sie für die Grammatik *G′* aus Selbsttestaufgabe 3.2 die FIRST-Mengen und die initialen Steuermengen der Produktionen. □

Für die Produktionen

numexpr'	→	ε	{ε}
term'	→	ε	{ε}
bool-rest	→	ε	{ε}

enthält die Steuermenge nun ε (bzw. ist gleich {ε}). Für genau diese Nichtterminale werden die FOLLOW-Mengen benötigt.

Im Prinzip verläuft die Berechnung von FOLLOW-Mengen für alle Nichtterminale einer Grammatik nach folgenden Regeln:

1. Initialisiere die FOLLOW-Menge des Startsymbols mit {$} (wobei $ ein spezielles Symbol sein soll, das die Eingabefolge abschließt), die aller anderen Nichtterminale mit ∅.

Führe 2. und 3. durch, solange sich noch FOLLOW-Mengen ändern:

2. Für jede Produktion $A \to \alpha B \beta$ (B ein Nichtterminal, α, β beliebige Folgen) mit $\beta \neq \varepsilon$ füge alle Symbole in FIRST(β) außer ε in FOLLOW(B) ein.
3. Für jede Produktion $A \to \alpha B$ und jede Produktion $A \to \alpha B \beta$, bei der gilt $\varepsilon \in$ FIRST(β), füge alle Symbole aus FOLLOW(A) in FOLLOW(B) ein.

Diese Regeln sind einfach und einsehbar; es ist nur nicht ganz klar, wie man sie systematisch anwendet. Dies kann man so organisieren:

Algorithmus 3.15: Berechnung von FOLLOW-Mengen

Eingabe Grammatik $G = (N, \Sigma, P, S)$

Ausgabe FOLLOW(A) für alle $A \in N$

Methode

1. Trage alle Nichtterminale als Knoten in einen Graphen ein. Der Graph hat zu Anfang keine Kante. Markiere den Knoten für das Startsymbol mit dem Symbol $ (Ende der Eingabe).
2. Betrachte der Reihe nach alle Produktionen in P und für jede Produktion jedes Nichtterminal B auf der rechten Seite.
   (i) Die Regel hat die Form $A \to \alpha B \beta$ mit $\beta \neq \varepsilon$: Markiere den Knoten B mit allen Symbolen, die in FIRST(β) \ {ε} liegen. Falls $\varepsilon \in$ FIRST(β), dann füge eine Kante $A \to B$ hinzu (falls noch nicht vorhanden).
   (ii) Die Regel hat die Form $A \to \alpha B$: Füge die Kante $A \to B$ hinzu.

(Der Graph kann Zyklen haben; die FOLLOW-Mengen aller Knoten in einem Zyklus sind gleich. Deshalb:)

3. Berechne alle starken Komponenten[3] des Graphen und behandle fortan jede Komponente wie einen einzigen Knoten; seine Markierung ist die Vereinigung der Markierungen aller seiner Knoten.
4. Die FOLLOW-Menge eines Nichtterminals ist die Vereinigung seiner eigenen Markierung mit den Markierungen aller seiner Vorgänger im Graphen. Das heißt, ausgehend von den „Wurzeln“[4] des Graphen, propagiere Knotenmarkierungen entlang den Kanten bis hin zu den Blättern, um alle FOLLOW-Mengen zu erhalten.

Wenn wir diesen Algorithmus auf unsere Beispielgrammatik anwenden, erhalten wir nach Schritt 2 den in Abb. 3.17 gezeigten Graphen (ohne *kursive* Knotenmarkierungen).

Schritt 3 fällt aus, da keine Zyklen mit mehr als einem Knoten vorkommen. In Schritt 4 würde z. B. *assignmen*t die Markierungen {$, **od**, **fi**, **else**} von *stmt* übernehmen, *expr* hätte dann {$, **od**, **fi**, **else**,)} usw. Wir haben das Ergebnis von Schritt 4 im Graphen nur an den interessierenden Knoten *bool-rest*, *num-expr*' und *term*' fett und kursiv eingetragen, also an den Knoten, die Nichtterminale darstellen, die nach ε ableiten.

---

[3] Eine starke (Zusammenhangs-) Komponente in einem gerichteten Graphen ist eine maximale Menge von Knoten, in der für jedes Paar (v, v') von Knoten ein Pfad von v nach v' und von v' nach v existiert. Ein Zyklus ist ein Spezialfall davon.

[4] Hier sollen einmal „Wurzeln“ Knoten ohne Vorgänger und „Blätter“ Knoten ohne Nachfolger im gerichteten Graphen bezeichnen (dies ist keine Standardterminologie).

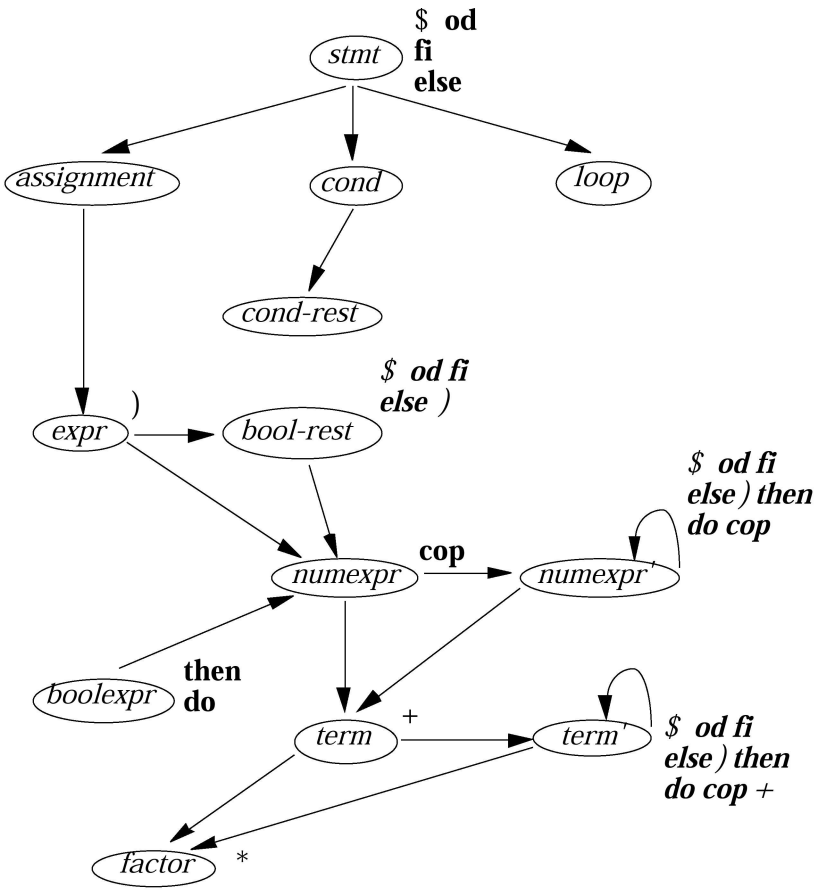


Abb. 3.17. Berechnung von FOLLOW-Mengen (nach Schritt 2 des Alg. 3.15)

Abschließend erhalten wir als komplette Version der Grammatik mit endgültigen Steuermengen:

(1)	*stmt*	→	*assignment* |	{**id**}
(2)			*cond* |	{**if**}
(3)			*loop*	{**while**}
(4)	*assignment*	→	**id** := *expr*	{**id**}
(5)	*cond*	→	**if** *boolexpr* **then** *stmt* *cond-rest*	{**if**}
(6)	*cond-rest*	→	**fi** |	{**fi**}
(7)			**else** *stmt* **fi**	{**else**}
(8)	*loop*	→	**while** *boolexpr* **do** *stmt* **od**	{**while**}
(9)	*expr*	→	*numexpr bool-rest*	{**id**, **const**, (}
(10)	*bool-rest*	→	**cop** *numexpr* |	{**cop**}
(11)			ε	{$, **od**, **fi**, **else**,)}
(12)	*boolexpr*	→	*numexpr* **cop** *numexpr*	{**id**, **const**,(}
(13)	*numexpr*	→	*term* *numexpr'*	{**id**, **const**,(}

(14)	*numexpr'*	→	+ *term numexpr'* |	{+}
(15)			ε	{$, **od**, **fi**, **else**,), **then**, **do**, **cop**}
(16)	*term*	→	*factor term'*	{**id**, **const,** (}
(17)	*term'*	→	* *factor term'* |	{*}
(18)			ε	{$, **od**, **fi**, **else**,), **then**, **do**, **cop**, +}
(19)	*factor*	→	**id** |	{**id**}
(20)			**cons**t |	{**const**}
(21)			(*expr*)	{(}

Wir sehen, dass für jedes Nichtterminal die Steuermengen seiner Alternativen disjunkt sind; die Grammatik ist also nun tatsächlich vom Typ LL(1). Da wir alle Steuermengen kennen, ist das Problem der Top-down-Analyse ohne Backtracking somit prinzipiell gelöst; es geht nun nur noch darum, eine geschickte Implementierung zu finden. Dafür gibt es zwei Möglichkeiten, die wir in den folgenden Abschnitten besprechen.

Selbsttestaufgabe 3.5: Bestimmen Sie die endgültigen Steuermengen für die Produktionen der Grammatik G' aus Selbsttestaufgabe 3.2. □

3.2.4 Implementierung eines vorgreifenden Analysators mit Analysetabelle

Wir hatten in Abschnitt 3.2.1 gesehen, dass die zentrale Idee des „predictive parsing“ darin besteht, beim Antreffen eines Nichtterminals A im Ableitungsbaum und eines Terminalsymbols a in der Eingabefolge einfach „irgendwo nachzusehen“, welche A-Produktion anzuwenden ist. In diesem Abschnitt betrachten wir die Implementierung eines „predictive parsers“ mit Hilfe einer Analysetabelle, wobei diese Idee sehr direkt umgesetzt wird; die Tabelle enthält an einer Position (A, a) einen Eintrag, der die anzuwendende Produktion identifiziert. Ein Parser mit Analysetabelle benutzt weiterhin einen Stack, um implizit den Durchlauf durch den Ableitungsbaum zu organisieren. Insgesamt kann man sich einen solchen Parser, in Analogie zu verschiedenen abstrakten Maschinen der theoretischen Informatik, etwa so vorstellen, wie in Abb. 3.18 gezeigt.

Der Parser benutzt ein Eingabeband, ein Ausgabeband, einen Stack und eine Analysetabelle. Auf dem Eingabeband steht zu Anfang die zu analysierende Folge von Terminalsymbolen; sie wird durch ein spezielles Symbol, hier „$“, abgeschlossen. Im Allgemeinen steht auf dem Eingabeband noch ein Reststück der Eingabefolge, und der Parser betrachtet jeweils das erste Symbol dieses Reststücks; dies hatten wir in Abschnitt 3.2.1 das „aktuelle Symbol“ genannt. Auf dem Stack steht zu Anfang ein $-Zeichen (das das untere Ende des Stacks markiert) und das Startsymbol der Grammatik. Im Allgemeinen, also während der Analyse, steht auf dem Stack zuoberst das gerade betrachtete Symbol in der Blattfolge des Ableitungsbaumes und darunter im Stack der Rest dieser Blattfolge nach rechts.

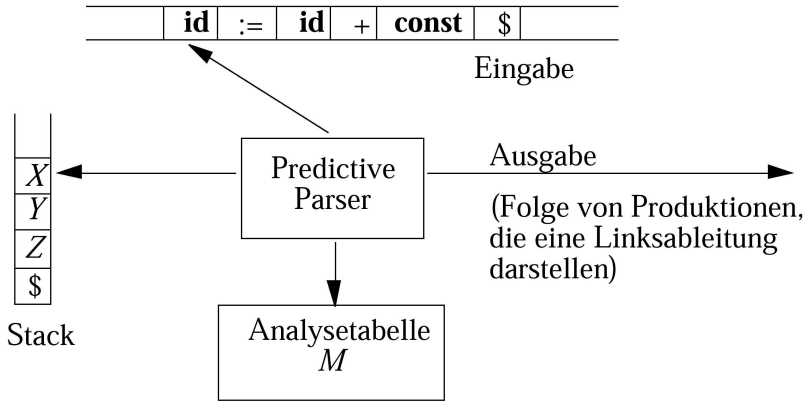


Abb. 3.18. Vorgreifender Analysator als abstrakte Maschine

Wir betrachten zum Vergleich noch einmal die allgemeine Situation innerhalb der LL(k)-Analyse:

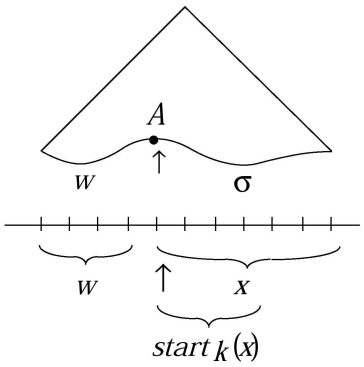


Abb. 3.19. Allgemeine Situation innerhalb der LL(k)-Analyse

In der dort dargestellten Situation steht also in der Eingabefolge gerade x\$ und auf dem Stack $A\sigma$\$. Die Ausgabe ist eine Folge von Produktionen oder Produktionsnummern, die den bisher erkannten Teil einer Linksableitung darstellen; offensichtlich ist diese Folge zu Anfang leer. Die *Analysetabelle M* ist eine Matrix mit einer Zeile für jedes Nichtterminal und einer Spalte für jedes Terminalsymbol der Grammatik; an jeder Position (A, a) enthält sie einen Eintrag, der entweder eine Produktionsnummer ist oder ein besonderes Symbol **error**.

Etwas formaler lässt sich ein solcher Parser so beschreiben: Sei $G = (N, \Sigma, P, S)$ die zugrundeliegende LL(1)-Grammatik. Die Produktionen in P seien nummeriert; bezeichne $\overline{P}$ die zugehörige Menge von Produktionsnummern und sei p_i die entsprechende Produktion zu $i \in \overline{P}$. Dann steht auf dem Eingabeband ein Wort aus

$\Sigma^*\{\$\}$ (also der Konkatenation der Sprachen Σ^* und $\{\$\}$), auf dem Stack ein Wort aus $(N \cup \Sigma)^*\{\$\}$, auf dem Ausgabeband ein Wort aus $\overline{P}^*$, die Analysetabelle ist mit N und $\Sigma \cup \{\$\}$ indiziert und enthält Einträge aus $\overline{P} \cup \{\textbf{error}\}$.

Ein aktueller Zustand während der Analyse lässt sich als Paar $(X\alpha, xw)$ beschreiben, wobei $X\alpha$ den Stackinhalt und xw den Rest der Eingabe beschreibt; X sei dabei das oberste Stacksymbol und x das aktuelle Symbol der Eingabefolge. Ein solches Paar bezeichnen wir als *Konfiguration* des Parsers. Zu Anfang der Analyse liegt die Konfiguration $(S\$, u\$)$ für eine zu analysierende Eingabefolge u vor. Der Parser führt dann eine Reihe von Schritten durch, die jeweils einen Übergang von einer Konfiguration K zu einer Konfiguration K' bewirken, notiert als

$$K \mapsto K'$$

bis entweder die Konfiguration ($, $) erreicht oder ein Fehler aufgetreten ist. Bei manchen Schritten werden die Nummern erkannter Produktionen ausgegeben; dies notieren wir formal als

$$K \overset{i}{\mapsto} K'$$

Für die Schritte gibt es folgende Möglichkeiten:

1. $X \in \Sigma, X = x$

   Das oberste Stacksymbol ist terminal und passt zum aktuellen Eingabesymbol; beide werden entfernt:

   $$(X\alpha, xw) \mapsto (\alpha, w)$$

2. $X \in \Sigma, X \neq x$

   Die Analyse wird abgebrochen und eine Fehlermeldung ausgegeben.[5]

3. $X \in N, M(X, x) = i$

   Mit Hilfe der Analysetabelle wird nun die richtige Produktion $p_i = (X \to X_1 \ldots X_m)$ ausgewählt und auf dem Stack X durch $X_1 \ldots X_m$ ersetzt:

   $$(X\alpha, xw) \overset{i}{\mapsto} (X_1 \ldots X_m\alpha, xw),$$

4. $X \in N, M(X, x) = \textbf{error}$

   Die Analyse wird abgebrochen und eine Fehlermeldung ausgegeben.

Die Arbeitsweise des Parsers ist damit klar; es ist lediglich noch der Inhalt der Analysetabelle festzulegen. Für eine LL(1)-Grammatik ergibt sich dies unmittelbar aus den Steuermengen: Sei für jedes Nichtterminal $A \in N$ die Menge der A-Produktionen

---

[5] In der Praxis wird eher eine Fehlerbehandlung eingeleitet und versucht, die Analyse fortzusetzen; darauf gehen wir hier aber nicht ein.

$$
\begin{array}{lll}
i_1 & A \rightarrow \alpha_1 \mid & D_1 \\
i_2 & \quad\ \alpha_2 \mid & D_2 \\
 & \ldots & \\
i_{n-1} & \quad\ \alpha_{n-1} \mid & D_{n-1} \\
i_n & \quad\ \alpha_n & D_n
\end{array}
$$

mit Steuermengen D_j und Produktionsnummern i_j gegeben. Dann ist die Analysetabelle M definiert durch

$$M(A,b) = \begin{cases} i_j & \text{falls } \exists j \in \{1 \ldots n\} : b \in D_j \\ \text{error} & \text{sonst} \end{cases}$$

für alle $A \in N$, $b \in (\Sigma \cup \$)$.

Für unsere Beispielgrammatik lässt sich anhand der im vorigen Abschnitt berechneten Steuermengen die in Abb. 3.20 gezeigte Analysetabelle berechnen. Alle in dieser Darstellung leeren Felder enthalten den Eintrag **error**; diese Einträge wurden der Übersichtlichkeit halber weggelassen.

Mit Hilfe der Tabelle läßt sich nun z. B. eine Anweisung

```
a := 3 * 5 + 7
```

analysieren. Durch die lexikalische Analyse ist daraus eine Symbolfolge entstanden:

id := **const** * **const** + **const**

Der Ablauf der Analyse ist in der Tabelle in Abb. 3.21 dargestellt. Jede Zeile enthält eine Konfiguration des Parsers. Der Stackinhalt ist so dargestellt, dass das oberste Symbol rechts steht, so dass sich X und x direkt gegenüberstehen. Die in einer Zeile gezeigte Ausgabe erfolgt beim Schritt von dieser Zeile zur nächsten. In der Tabelle kürzen wir **const** mit **c** ab.

Selbsttestaufgabe 3.6: Erstellen Sie zu der Grammatik G' eine Analysetabelle und analysieren Sie das Wort **MODULE id ; IMPORT id ; EXPORT id ;** . □

3.2.5 Implementierung eines vorgreifenden Analysators durch rekursiven Abstieg

Die zweite Möglichkeit, einen „predictive parser“ zu implementieren, besteht darin, ein System rekursiver Prozeduren anzulegen; dabei gibt es genau eine Prozedur für jedes Nichtterminal in der LL(1)-Grammatik. Die Idee dabei ist, zu Anfang die Prozedur für das Startsymbol aufzurufen; anschließend soll der Baum, der sich aus den wechselseitigen Prozeduraufrufen ergibt, direkt die Struktur des Ableitungsbaumes widerspiegeln. Jede aufgerufene Prozedur für ein Nichtterminal A entscheidet anhand des aktuellen Symbols in der Eingabefolge, welche A-Produktion auszuwählen ist; sie bearbeitet dann durch weitere Aufrufe die rechte Seite dieser Produktion.

	id	:=	**if**	**then**	**else**	**fi**	**while**	**do**	**od**	**cop**	+	*	**const**	(	)	$
statement	1		2				3									
assignment	4															
cond			5													
cond-rest					7	6										
loop							8									
expr	9												9	9		
bool-rest					11	11			11	10					11	11
boolexpr	12												12	12		
numexpr	13												13	13		
numexpr'				15	15	15		15	15	15	14				15	15
term	16												16	16		
term'				18	18	18		18	18	18	18	17			18	18
factor	19												20	21		

Abb. 3.20. Analysetabelle für die Beispielgrammatik

Stack	*Eingabe*	*Ausgabe*
$ *stmt*	**id** := **c** * **c** + **c** $	1
$ *assignment*	**id** := **c** * **c** + **c** $	4
$ *expr* := **id**	**id** := **c** * **c** + **c** $	
$ *expr* :=	:= **c** * **c** + **c** $	
$ *expr*	**c** * **c** + **c** $	9
$ *bool-rest numexpr*	**c** * **c** + **c** $	13
$ *bool-rest numexpr' term*	**c** * **c** + **c** $	16
$ *bool-rest numexpr' term' factor*	**c** * **c** + **c** $	20
$ *bool-rest numexpr' term'* **c**	**c** * **c** + **c** $	
$ *bool-rest numexpr' term'*	* **c** + **c** $	17
$ *bool-rest numexpr' term' factor* *	* **c** + **c** $	
$ *bool-rest numexpr' term' factor*	**c** + **c** $	20
$ *bool-rest numexpr' term'* **c**	**c** + **c** $	
$ *bool-rest numexpr' term'*	+ **c** $	18
$ *bool-rest numexpr'*	+ **c** $	14
$ *bool-rest numexpr' term* +	+ **c** $	
$ *bool-rest numexpr' term*	**c** $	16
$ *bool-rest numexpr' term' factor*	**c** $	20
$ *bool-rest numexpr' term'* **c**	**c** $	
$ *bool-rest numexpr' term'*	$	18
$ *bool-rest numexpr'*	$	15

$ *bool-rest*	$	11
$	$	**accept**

Abb. 3.21. Ablauf der Top-down-Analyse für tabellengesteuerten Parser

Die Struktur der Prozedur für Nichtterminal *A* ergibt sich unmittelbar aus der mit Steuermengen und Produktionsnummern versehenen Menge von *A*-Produktionen. Betrachten wir zunächst als Beispiel die Prozeduren für Nichtterminale *stmt* und *assignment* unserer Grammatik. Anhand der gegebenen Produktionen

(1)	*stmt*	→	*assignment*	{**id**}
(2)			*cond*	{**if**}
(3)			*loop*	{**while**}
(4)	*assignment* →		**id** := *expr*	{**id**}

werden Prozeduren *stmt* und *assignment* konstruiert, die folgende Gestalt haben:

```
procedure stmt;
begin
   if symbol = id then
        output(1); assignment
   elsif symbol  = if then
        output(2); cond
   elsif symbol  = while then
        output(3); loop
   else error
   fi
end;

procedure assignment;
begin
   if symbol = id then
      output(4); match(id); match(:=); expr
   else error
   fi
end;
```

In diesen Prozeduren bezeichne *symbol* eine globale Variable vom Typ *tsymbol* (Terminalsymbol), die jeweils das aktuelle Symbol der Eingabefolge enthält; bei Aufruf der Prozedur für das Startsymbol enthält sie das erste Symbol dieser Folge. Weiterlesen in der Eingabefolge erfolgt durch die Prozedur *match*:

```
procedure match(t: tsymbol)
begin
   if symbol = t then nextsymbol else error fi
end;
```

Parameter dieser Prozedur ist ein Terminalsymbol. Der Typ *tsymbol* könnte etwa numerische Codierungen aller Terminalsymbole enthalten, z. B. **if** = 17, := = 43 usw. In den gezeigten Prozeduren für *assignment* müsste also die entsprechende Codierung für die vorkommenden Terminalsymbole eingesetzt werden; der Klarheit halber haben wir das nicht getan und stattdessen Sonderzeichenfolgen, die Terminalsymbole bilden, fett und unterstrichen dargestellt. Die Prozedur *nextsymbol* liest das nächste Symbol in die Variable *symbol*. Die Prozedur *output* gibt lediglich die Nummer der ausgewählten Produktion aus. Die Prozedur *error* gibt eine Fehlermeldung aus.

Wir geben nun ein allgemeines Schema an, um die Prozedur für ein Nichtterminal zu beschreiben. Gegeben sei eine Menge von A-Produktionen mit Steuermengen D_j und Produktionsnummern i_j:

$$
\begin{array}{llllll}
i_1 & A & \rightarrow & \alpha_1 & | & D_1 \\
i_2 & & & \alpha_2 & | & D_2 \\
 & \ldots & & & & \\
i_{n-1} & & & \alpha_{n-1} & | & D_{n-1} \\
i_n & & & \alpha_n & & D_n
\end{array}
$$

Dann hat die Prozedur A folgende Gestalt:

```
procedure A;
begin
   if symbol ∈ D1 then
      output(i1); bearbeite α1
   elsif symbol ∈ D2 then
      output(i2); bearbeite α2
   elsif
      …
   elsif symbol ∈ Dn then
      output(in); bearbeite αn
   else error
   fi
end;
```

Für $D_j = \{t_1, \ldots, t_r\}$ lässt sich „$symbol \in D_j$“ beispielsweise weiter verfeinern zu

$$symbol = t_1 \textbf{ or } \cdots \textbf{ or } symbol = t_r$$

Eine andere Möglichkeit wäre, statt der **if** … **elsif** … **fi**-Kette eine **case**-Anweisung zu verwenden:

```
case symbol of
   t1,1, …, t1,r1 : begin output (i1); bearbeite α1 end;
   …
```

$t_{n,1}, \ldots, t_{n,r_n}$: **begin** *output* (i_n); bearbeite α_n **end**;
otherwise *error*
esac

Für $\alpha_j = \varepsilon$ ist „bearbeite α_j“ einfach die leere Anweisung. Andernfalls sei $\alpha_j = X_1 \ldots X_m$ mit $X_k \in N \cup \Sigma$. Dann verfeinern wir „bearbeite α_j“ zu

code(X_1); …; *code*(X_m),

dabei ist

$$code(X_k) = \begin{cases} match(a) & \text{falls } X_k = a \in \Sigma \\ B & \text{falls } X_k = B \in N \end{cases}$$

Als weitere Beispiele zeigen wir noch die Prozeduren für *term*, *term*' und *factor*:

```
procedure term;
begin
   if symbol = id  or symbol = const or symbol = ( then
      output (16); factor; term'
   else error
   fi
end

procedure term';
begin
   if symbol = * then
      output(17); match(*); factor; term'
   elsif symbol = $ or symbol = od or symbol = fi or
      symbol = else or symbol = ) or symbol = then or
      symbol = do or symbol = cop or symbol = +
   then output(18)
   else error
   fi
end;

procedure factor;
begin
   if symbol = id then
      output(19);  match(id)
   elsif symbol = const then
      output(20);  match(const)
   elsif symbol = ( then
      output(21);  match(();  expr;  match())
   else error
   fi
end;
```

Es ist Ihnen wahrscheinlich bei den vorgestellten Beispielen aufgefallen, dass darin viele redundante Tests auftreten. So müsste z. B. in der Prozedur *assignment* nicht mehr getestet werden, ob das aktuelle Symbol gleich **id** ist. Wäre es das nämlich nicht, wäre *assignment* erst gar nicht von *stmt* aufgerufen worden, d. h., der **else**-Zweig von *assignment* kann nie erreicht werden. Weiterhin könnte der Aufruf *match*(**id**) innerhalb von *assignment* durch einen von *nextsymbol* ersetzt werden, da *match* ebenfalls immer nur in den **then**-Zweig laufen würde.

Dennoch haben wir die Beispiele in dieser Form vorgestellt, da sie genau nach dem beschriebenen allgemeinen Schema konstruiert worden sind. Diese Konstruktion könnte nämlich auch durchaus automatisch erfolgen, d. h. durch einen Parser-Generator, der nur die Grammatik als Eingabe erhält. Die oben erwähnten Optimierungen könnten dann möglicherweise ebenfalls automatisch durchgeführt werden. Eine Regel wäre z. B., alle Aufrufe von *match*, die von einem auf der rechten Seite einer Produktion ganz links stehenden Terminalsymbol stammen, durch einen Aufruf von *nextsymbol* zu ersetzen.

Literaturhinweise

Gute Darstellungen der in diesem Kapitel behandelten Themen bieten (Aho et al. 2006) und (Parsons 1992); formale Definitionen bzw. Darstellungen der LL(k) und LR(k)-Analyse aus formaler Sicht finden sich in (Sudkamp 2005). Eine umfassende Behandlung der Theorie der Syntaxanalyse bieten (Sippu und Soisalon-Soininen 1988, 1990).

Top-down-Analyse mit rekursivem Abstieg war sehr früh eine beliebte Syntaxanalyse-Technik; ein solcher vorausschauender Analysator wird schon in (Conway 1963) beschrieben. LL(k)-Grammatiken stammen von Lewis und Stearns (1968); die Theorie dazu wurde in (Rosenkrantz und Stearns 1970) weiterentwikkelt. Das Buch (Lewis, Rosenkrantz und Stearns 1976) beschreibt den Einsatz vorausschauender Analysatoren im Compilerbau. Auch Knuth (1971b) studierte die Top-down-Analyse. Algorithmen, um Grammatiken in die LL(1)-Form zu bringen, wurden u. a. von Stearns (1971) und Soisalon-Soininen und Ukkonen (1979) entwickelt.

Lösungen zu den Selbsttestaufgaben

Aufgabe 3.1

Wir ersetzen die Produktionen durch

$A \rightarrow \mathbf{e}A' \mid \mathbf{f}A'$
$A' \rightarrow \mathbf{b}A' \mid B\mathbf{cd}A' \mid \varepsilon$

Aufgabe 3.2

G enthält die linksrekursive Produktion *idlist* → *idlist* **id ;** . Außerdem ist die jeweils richtige Alternative bei den Produktionen

lists → *implist explist* | *implist*
implist → *A* **IMPORT** *idlist* | *A* **IMPORT** *idlist implist*

offensichtlich nicht durch Betrachten des ersten Symbols der rechten Seite bestimmbar. Wir ersetzen

idlist → **id ;** | *idlist* **id ;**

durch

idlist → **id ;** *hidlist*
hidlist → **id ;** *hidlist* | ε.

Weiter wenden wir Links-Faktorisierung an und ersetzen

lists → *implist explist* | *implist* | *explist* | ε

durch

lists → *implist hlists* | *explist* | ε
hlists → *explist* | ε

sowie

implist → *A* **IMPORT** *idlist* | *A* **IMPORT** *idlist implist*

durch

implist → *A* **IMPORT** *idlist himplist*
himplist → *implist* | ε.

Dann erhalten wir $G_1 = (N_1, \Sigma, P_1, modhead)$ mit

$N_1 = N \cup \{hidlist, hlists, himplist\}$
$P_1 = \{$

modhead	→	**MODULE id ;** *lists*
lists	→	*implist hlists* | *explist* | ε
hlists	→	*explist* | ε
implist	→	*A* **IMPORT** *idlist himplist*
himplist	→	*implist* | ε
A	→	**FROM id** | ε
explist	→	**EXPORT** *B idlist*
B	→	**QUALIFIED** | ε
idlist	→	**id ;** *hidlist*
hidlist	→	**id ;** *hidlist* | ε

$\}$

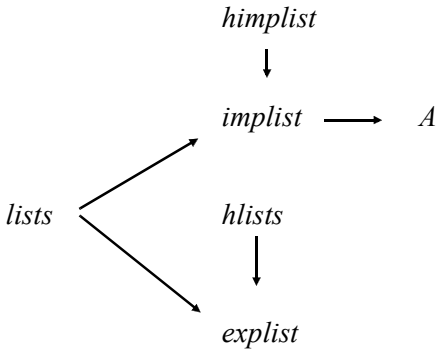
Aufgabe 3.3

Angenommen, es gibt einen Zyklus $A \rightarrow \ldots \rightarrow A$ im Graphen. Wie man sich leicht überlegt, bedeutet dies, dass dann eine Linksableitung $A \underset{l}{\Rightarrow} {}^{*} A\alpha$ existieren muss.

Die Grammatik ist also direkt oder indirekt linksrekursiv, solche Grammatiken hatten wir aber von der Betrachtung ausgeschlossen.

Aufgabe 3.4

Es ist $N_\varepsilon = \{lists, hlists, himplist, A, B, hidlist\}$. Wir zeichnen den Graphen, der die Berechnungsreihenfolge festlegt:



Die Benutzung des Algorithmus FIRST liefert dann

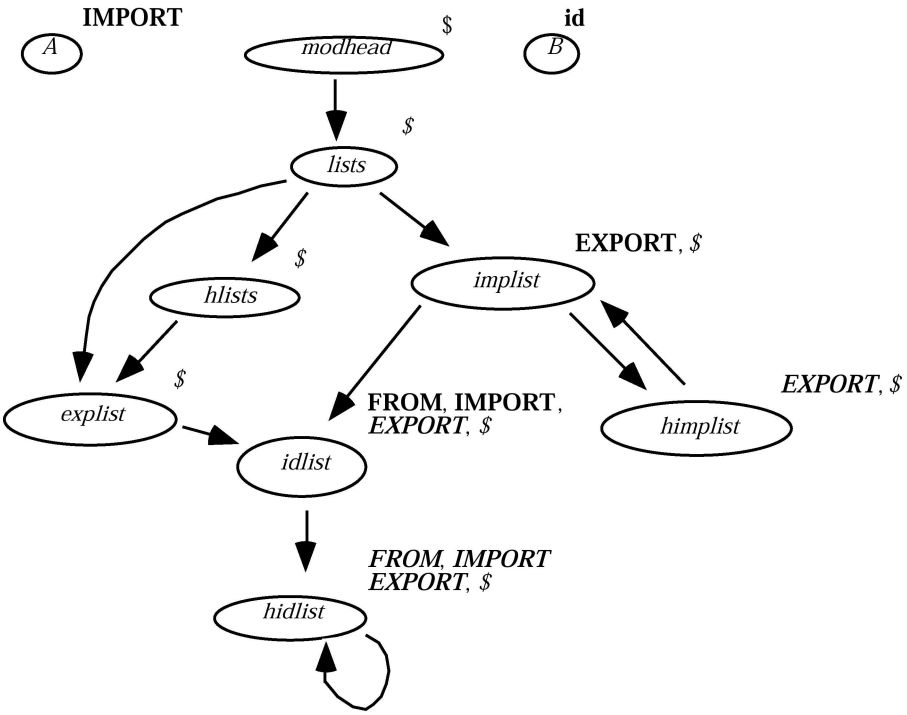
FIRST	*Produktion*		*Steuermenge*
{**MODULE**}	*modhead*	→ **MODULE id ;** *lists*	{**MODULE**}
{**FROM**, **IMPORT**, **EXPORT**, ε}	*lists*	→ *implist hlists* |	{**FROM**, **IMPORT**}
		explist |	{**EXPORT**}
		ε	{ε}
{**EXPORT**, ε}	*hlists*	→ *explist* |	{**EXPORT**}
		ε	{ε}
{**FROM**, **IMPORT**}	*implist*	→ *A* **IMPORT** *idlist* *himplist*	{**FROM**, **IMPORT**}
{**FROM**, **IMPORT**, ε}	*himplist*	→ *implist* |	{**FROM**, **IMPORT**}
		ε	{ε}
{**FROM**, ε}	*A*	→ **FROM id** |	{**FROM**}
		ε	{ε}
{**EXPORT**}	*explist*	→ **EXPORT** *B idlist*	{**EXPORT**}
{**QUALIFIED**, ε}	*B*	→ **QUALIFIED** |	{**QUALIFIED**}
		ε	{ε}
{**id**}	*idlist*	→ **id ;** *hidlist*	{**id**}
{**id**, ε}	*hidlist*	→ **id ;** *hidlist*	{**id**}
		ε	{ε}

Aufgabe 3.5

Für die Produktionen

lists → ε
hlists → ε
himplist → ε
A → ε
B → ε
hidlist → ε

ist die Steuermenge gleich ε. Für genau diese Nichtterminale benötigen wir die FOLLOW-Mengen. Die Benutzung des Algorithmus FOLLOW liefert den folgenden Graphen, wobei man die nicht kursiv gedruckten Knotenmarkierungen durch Schritt 2 und die kursiv dargestellten durch Schritt 4 des Algorithmus erhält. Beachten Sie bitte, dass in Schritt 3 eigentlich die beiden Knoten *implist* und *himplist* zusammengefasst werden.



Nun können wir die endgültigen Steuermengen ablesen und erhalten:

	Produktionen		*Steuermenge*
(1)	*modhead*	→ **MODULE id ;** *lists*	{**MODULE**}
(2)	*lists*	→ *implist hlists* |	{**FROM**, **IMPORT**}
(3)		*explist* |	{**EXPORT**}
(4)		ε	{$}
(5)	*hlists*	→ *explist* |	{**EXPORT**}
(6)		ε	{$}
(7)	*implist*	→ *A* **IMPORT** *idlist himplist*	{**FROM**, **IMPORT**}
(8)	*himplist*	→ *implist* |	{**FROM**, **IMPORT**}
(9)		ε	{**EXPORT**, $}
(10)	*A*	→ **FROM id** |	{**FROM**}
(11)		ε	{**IMPORT**}
(12)	*explist*	→ **EXPORT** *B idlist*	{**EXPORT**}
(13)	*B*	→ **QUALIFIED** |	{**QUALIFIED**}
(14)		ε	{**id**}

(15)	*idlist*	→ **id ;** *hidlist*	{**id**}
(16)	*hidlist*	→ **id ;** *hidlist* |	{**id**}
(17)		ε	{**FROM**, **IMPORT**, **EXPORT**, $}

Aufgabe 3.6

Wir erhalten die Analysetabelle:

	MODULE	**id**	**IMPORT**	**FROM**	**EXPORT**	**QUALIFIED**	**;**	**$**
modhead	1							
lists			2	2	3			4
hlists					5			6
implist			7	7				
himplist			8	8	9			9
A			11	10				
explist					12			
B		14				13		
idlist		15						
hidlist		16	17	17	17			17

Den Ablauf der Analyse kann man der folgenden Tabelle entnehmen.

Stack	*Eingabe*	*Ausg.*
$*modhead*	**MODULE id; IMPORT id ; EXPORT id ;**$	1
$*lists* **; id MODULE**	**MODULE id; IMPORT id ; EXPORT id ;**$	
$*lists* **; id**	**id; IMPORT id ; EXPORT id ;**$	
$*lists* **;**	**; IMPORT id ; EXPORT id ;**$	
$*lists*	**IMPORT id ; EXPORT id ;**$	2
$*hlists implist*	**IMPORT id ; EXPORT id ;**$	7
$*hlists himplist idlist* **IMPORT** *A*	**IMPORT id ; EXPORT id ;**$	11
$*hlists himplist idlist* **IMPORT**	**IMPORT id ; EXPORT id ;**$	
$*hlists himplist idlist*	**id ; EXPORT id ;**$	15
$*hlists himplist hidlist* **; id**	**id ; EXPORT id ;**$	
$*hlists himplist hidlist* **;**	**; EXPORT id ;**$	
$*hlists himplist hidlist*	**EXPORT id ;**$	17
$*hlists himplist*	**EXPORT id ;**$	9
$*hlists*	**EXPORT id ;**$	5
$*explists*	**EXPORT id ;**$	12
$*idlist B* **EXPORT**	**EXPORT id ;**$	
$*idlist B*	**id ;**$	14
$*idlist*	**id ;**$	15
$*hidlist* **; id**	**id ;**$	
$*hidlist* **;**	**;**$	
$*hidlist*	$	17
$	$	**accept**

Literatur

Aho, A.V., Lam, M.S., Sethi, R. und Ullman, J.D. (2006). *Compilers: Principles, Techniques, and Tools*. 2nd Edition, Addison-Wesley, Reading, MA.

Conway, M.E. (1963). Design of a Separable Transition-Diagram Compiler. *Communications of the ACM 6*, 396-408.

Knuth, D.E. (1971b). Top-Down Syntax Analysis. *Acta Informatica 1* (2), 79-110.

Lewis, P.M. II und Stearns, R.E. (1968). Syntax-directed Transduction. *Journal of the ACM 15*, 465-488.

Lewis, P.M. II, Rosenkrantz, D.J. und Stearns, R.E. (1976). *Compiler Design Theory*. Addison-Wesley, Reading, MA.

Parsons, T.W. (1992). *Introduction to Compiler Construction*. Computer Science Press, New York.

Rosenkrantz, D.J. und Stearns, R.E. (1970). Properties of Deterministic Top-down Grammars. *Information and Control 17*, 226-256.

Sippu, S. und Soisalon-Soininen, E. (1988). *Parsing Theory. Vol. I: Languages and Parsing*. EATCS Monographs on Theoretical Computer Science 15, Springer-Verlag, Berlin.

Sippu, S. und Soisalon-Soininen, E. (1990). *Parsing Theory. Vol. II: LR(k) and LL(k) Parsing*. EATCS Monographs on Theoretical Computer Science 20, Springer-Verlag, Berlin.

Soisalon-Soininen, E. und Ukkonen, E. (1979). A Method for Transforming Grammars into LL(k) Form. *Acta Informatica 12* (4), 339-369.

Stearns, R.E. (1971). Deterministic Top-down Parsing. *Proc. 5th Annual Princeton Conf. on Information Sciences and Systems*, S. 182-188.

Sudkamp, T. A. (2005). *Languages and Machines*: An Introduction to the Theory of Computer Science. 3rd Edition, Addison-Wesley, Reading, MA.

Index

Inhalt

XIV .