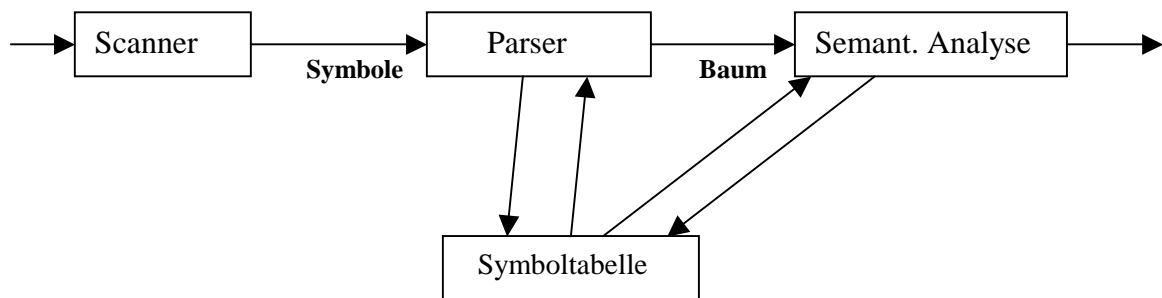


4 Syntaktische Analyse



4.1 Zum Prinzip der Top-Down-Analyse

Der Parser arbeitet die Eingabe i.a. von links nach rechts symbolweise ab. Beim Top-Down-Verfahren beginnt die Konstruktion des Parse-Baums mit der Wurzel und schreitet in Richtung der Blätter fort.

Die Top-Down-Konstruktion des Parse-Baums beginnt an der Wurzel. Die folgenden Schritte werden i.a. als Teil einer einfachen, *sequentiellen* Abarbeitung des Eingabestroms implementiert. Das gerade zu untersuchende Symbol heißt

Look-ahead-Symbol.

0. Die Wurzel wird mit dem *Startsymbol* markiert.

Dann führt man wiederholt die beiden folgenden Schritte aus:

1. Ist der Knoten n mit dem *Nichtterminal* A markiert, so wähle in Abhängigkeit vom Look-ahead-Symbol eine der Produktionen für A und erzeuge für jedes Grammatiksymbol auf der rechten Seite der Produktion je einen Nachfolger des Knotens n .

Ist der gerade behandelte Knoten n mit einem *Terminalsymbole* markiert und stimmt dieses mit dem Look-ahead-Symbol überein, so gehe sowohl im Parse-Baum, als auch in der Eingabe einen Schritt weiter. Wenn die beiden *Terminalsymbole* nicht übereinstimmen, so ist ein Fehler gefunden.

2. Suche den nächsten zu behandelnden Knoten n .

Sind die Nachfolger eines Knotens erzeugt, so behandeln wir als nächstes diese Nachfolger von links nach rechts.

Bsp.: Die folgende kleine Grammatik erzeugt eine Untermenge der Pascal-Datentypen

```

type  → simple | ^id | array [ simple ] of type
simple → integer | char | num dotdot num
  
```

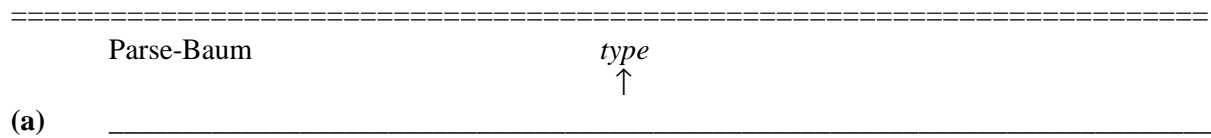
Der Satz:

array [num dotdot num] of integer

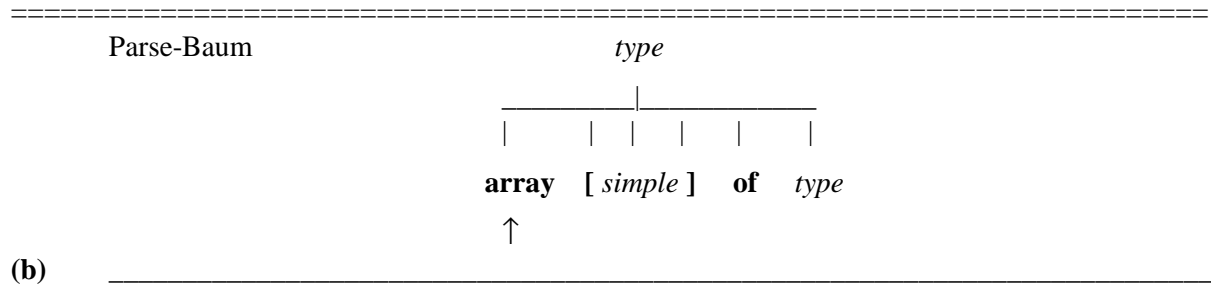
soll analysiert werden.

Zu Beginn ist das Symbol **array** das Look-ahead-Symbol. Vom Parse-Baum ist bisher nur die Wurzel bekannt, die in (a) mit dem Startsymbol *type* markiert ist. Ziel ist es, den Rest des Parse-Baums so zu konstruieren, daß der vom Parse-Baum erzeugte Satz mit dem Eingabestring übereinstimmt.

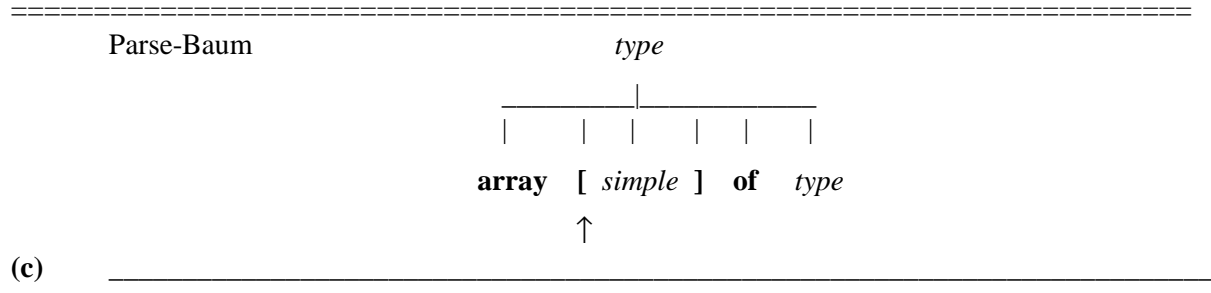
Damit beide Wörter überhaupt übereinstimmen können, muß das Nichtterminal *type* in (a) ein Wort herleiten, das mit dem Look-ahead-Symbol **array** beginnt. In der Grammatik gibt es für *type* nur eine Produktion, die ein solches Wort herleiten kann. Wir wählen diese Produktion aus und erzeugen für die Wurzel Nachfolgerknoten, die mit den Symbolen auf der rechten Seite der Produktion markiert werden.



Eingabe **array [num dotdot num] of integer**
↑



Eingabe **array [num dotdot num] of integer**
↑



Eingabe **array [num dotdot num] of integer**
↑

=====
 usw ...

In den drei Momentaufnahmen sind das Look-ahead-Symbol in der Eingabe und der aktuell behandelte Knoten des Parse-Baums jeweils durch Pfeile gekennzeichnet. Sind die Nachfolger eines Knotens erzeugt, dann behandeln wir als nächstes den am weitesten links stehenden Nachfolger. In

(b) wurden gerade die Nachfolger der Wurzel konstruiert und der am weitesten links stehende, mit **array** markierte Nachfolger wird nun behandelt.

Wenn der gerade behandelte Knoten des Parse-Baums für ein Terminal steht und dieses Terminal mit dem Look-ahead-Symbol übereinstimmt, gehen wir sowohl im Parse-Baum als auch in der Eingabe einen Schritt weiter. Das nächste Eingabesymbol wird zum neuen Look-ahead-Symbol, und im Parse-Baum wird der nächste Nachfolger betrachtet. In (c) ist der Pfeil im Parse-Baum zum nächsten Nachfolger der Wurzel weitergewandert und der Pfeil in der Eingabe steht nun auf dem nächsten Symbol "[". Nach einem weiteren Schritt wird der Pfeil im Parse-Baum auf den Nachfolgerknoten zeigen, der mit dem Nichtterminal `simple` markiert ist. Wenn ein mit einem Nichtterminal markierter Knoten bearbeitet wird, wählen wir erneut eine Produktion für das Nichtterminal aus.

Probleme bei der Top-Down-Analyse:

1. Es können *Sackgassen* entstehen, in die man bei der Auswahl der Produktionen läuft, und die wieder rückgängig gemacht werden müssen ("Backtracking").

Forderung: Grammatik muß eine sackgassenfreie Top-Down-Analyse ermöglichen,

2. In manchen Fällen kann nicht entschieden werden, welche von - im einfachsten Fall - zwei alternativen Produktionen auszuwählen ist, z.B. bei: $A \rightarrow \alpha\beta \mid \alpha\gamma$. Hier haben beide Produktionen einen gemeinsamen "Faktor" α . Deshalb verlagert man die Entscheidung auf einen späteren Zeitpunkt durch die sog. Links-Faktorisierung, die Aufspaltung in zwei Produktionen: $A \rightarrow \alpha B$, $B \rightarrow \beta \mid \gamma$.
3. Möglichkeit von Endlos-Schleifen bei linksrekursiven Produktionen.

eine Grammatik-Transformation ermöglicht die Entfernung von Links-Rekursionen in einfachen Fällen: aus $A \rightarrow A\alpha \mid \beta$ wird $A \rightarrow \beta B$ und $B \rightarrow \alpha B \mid \epsilon$.

Bsp.: $\text{EXPR} \rightarrow \text{EXPR} + \text{TERM}$

Beispiel für Linksrekursionen

Wir betrachten die erste Produktion der Standardgrammatik für arithmetische Ausdrücke

$$\text{EXPR} \rightarrow \text{TERM} \mid \text{EXPR} + \text{TERM} \mid \text{EXPR} - \text{TERM}$$

Wir können sie transformieren

1. in eine *rechtsrekursive* Produktion:

$$\text{EXPR} \rightarrow \text{TERM} \mid \text{TERM} + \text{EXPR} \mid \text{TERM} - \text{EXPR}$$

Dies kann jedoch zu Problemen führen, wie Eintausch der Link-Assoziativität eines Operators gegen die Rechts-Assoziativität!

2. durch die obige Transformation in zwei Produktionen der Form:

$$\text{EXPR} \rightarrow \text{TERM} E$$

$$E \rightarrow + \text{TERM} E \mid - \text{TERM} E \mid \epsilon$$

3. in eine *Iteration*

$$\text{EXPR} \rightarrow \text{TERM} \{ + \text{TERM} \mid - \text{TERM} \}$$

Dieses ist eine äquivalente kürzere Darstellung von 2.

Weitere Algorithmen zur Entfernung von Linksrekursionen findet man in Aho, Sethi, Ullman, S. 215.

Implementationstechniken für Top-Down-Parser

1. *Recursive Descent*-Parser:

Jedem Nonterminal entspricht eine Prozedur, das Anhängen von Knoten an den Parse-Baum geschieht durch einen Prozedur-Aufruf.

2. *Tabellengesteuerter Top-Down*-Parser:

- Universelles Analyseprogramm
- Grammatik-spezifische Parse-Tabelle
- Verwaltung durch Stack
- Parse-Tabelle muß/kann durch Programm erzeugt werden

4.2 LL(1)-Grammatiken

Wir wollen eine Klasse von kontextfreien Grammatiken charakterisieren, für die eine sackgassenfreie Top-Down-Analyse möglich ist. Wenn eine Produktion der Form

$$X \rightarrow \sigma_1 | \sigma_2 | \dots | \sigma_k$$

vorliegt, so muß allein durch Betrachten des Look-ahead-Symbols klar sein, *welche* der Alternativen $\sigma_1, \sigma_2, \dots, \sigma_k$ in Frage kommt.

Man nennt dies *prädiktive Syntaxanalyse* (der zugehörige Parser heißt *prädiktiver* Parser).

FIRST- und FOLLOW-Mengen

Def.:

a) Sei $G = (N, T, P, S)$, $\sigma \in V^*$. Dann ist

$$\text{FIRST}(\sigma) := \{t \in T \mid \sigma \Rightarrow^* t\dots\} \cup \{\varepsilon\}, \text{ falls } \sigma \Rightarrow^* \varepsilon, \text{ bzw.}$$

$$\text{FIRST}(\sigma) := \{t \in T \mid \sigma \Rightarrow^* t\dots\}, \text{ andernfalls.}$$

Dies ist also die Menge der *terminalen Anfangszeichen* aller Strings, die aus σ abgeleitet werden können.

b) Sei $A \in N$. Dann ist

$$\text{FOLLOW}(A) := \{t \in T \mid S \Rightarrow^* \alpha A t \beta, \alpha, \beta \text{ beliebig}\}$$

Dies ist die Menge aller Terminalsymbole, die in einer Satzform *direkt rechts neben* A stehen können.

Bsp.: $G = (N, T, P, S)$ mit $N = \{S, A, B\}$, $T = \{a, b, c\}$

$$P = \{S \rightarrow A \mid B, A \rightarrow cA \mid a, B \rightarrow cB \mid b\}$$

$$L(G) = \{c^n a \mid n \geq 0\} \cup \{c^n b \mid n \geq 0\}$$

FIRST-Mengen

σ	FIRST(σ)
a	{a}
b	{b}
cA	{c}
cB	{c}
A	{c, a}
B	{c, b}
S	{a, b, c}

FOLLOW-Mengen

T	FOLLOW(T)
A	\emptyset
B	\emptyset
S	\emptyset

Bsp.: $G = (N, T, P, S)$ mit
 $N = \{S\}$, $T = \{t, +\}$, $P = \{S \rightarrow S+t \mid \epsilon\}$
 $L(G) = \{(+t)^n \mid n \geq 0\} = \{\epsilon, +t, +t+t, \dots\}$

σ	FIRST(σ)	FOLLOW(σ)
S	{+, ϵ }	{+}
+t	{+}	undef.
ϵ	{ ϵ }	undef.

Bestimmung von FIRST- und FOLLOW-Mengen

FIRST-Mengen

Für alle Grammatiksymbole X wird FIRST(X) berechnet, indem die folgenden Regeln solange angewendet werden, bis zu keiner FIRST-Menge mehr ein neues Terminalsymbol oder ϵ hinzukommt:

1. Wenn X Terminalsymbol ist, dann ist $FIRST(X) = \{X\}$.
2. Wenn $X \rightarrow \epsilon$ eine Produktion ist, so füge ϵ zu FIRST(X) hinzu.
3. Wenn X Nichtterminal und $X \rightarrow Y_1 Y_2 \dots Y_k$ eine Produktion ist, dann nimm a zu FIRST(X) hinzu, falls a für irgendein i in FIRST(Y_i) und ϵ in allen FIRST(Y_1), ... FIRST(Y_{i-1}) enthalten ist. d.h. $Y_1 \dots Y_{i-1} \Rightarrow^* \epsilon$. Wenn ϵ für alle $j=1, 2, \dots, k$ in FIRST(Y_j) enthalten ist, nimm ϵ hinzu. Zum Beispiel gehört jedes Element aus FIRST(Y_1) auch zu FIRST(X). Ist ϵ nicht aus Y_1 herleitbar, braucht nichts mehr zu FIRST(X) hinzugefügt zu werden. Gilt aber $Y_1 \Rightarrow^* \epsilon$, müssen wir FIRST(Y_2) hinzunehmen.

FOLLOW-Mengen

FOLLOW(A) wird für alle Nichtterminale A berechnet, indem die folgenden Regeln solange angewendet werden, bis keine FOLLOW-Menge mehr vergrößert werden kann:

1. Wenn es eine Produktion $A \rightarrow \alpha B \beta$ gibt, wird jedes Element von FIRST(β) mit Ausnahme von ϵ auch in FOLLOW(B) aufgenommen.
2. Wenn es Produktionen $A \rightarrow \alpha B$ oder $A \rightarrow \alpha B \beta$ gibt und FIRST(β) ϵ enthält (d.h. $\beta \Rightarrow^* \epsilon$), dann gehört jedes Element von FOLLOW(A) auch zu FOLLOW(B).

LL(1)-Grammatiken

(Erklärung von LL(1): Analyse verarbeitet den Satz von links nach rechts, erzeugt Linksableitung, 1 Zeichen voraus!)

Definition: Eine kontextfreie Grammatik $G = (N, T, P, S)$, die die Bedingungen E_1 und E_2 erfüllt, heißt LL(1)-Grammatik.

(E_1) Falls es zum Nichtterminal N zwei alternative Produktionen $N \rightarrow \sigma_1$ und $N \rightarrow \sigma_2$ gibt, so muß gelten

$$\text{FIRST}(\sigma_1) \cap \text{FIRST}(\sigma_2) = \emptyset$$

(E_2) Falls aus einem Nichtterminal N der Leerstring ϵ abgeleitet werden kann, so muß gelten

$$\text{FIRST}(N) \cap \text{FOLLOW}(N) = \emptyset$$

Satz: Eine LL(1)-Grammatik erlaubt eine sackgassenfreie Top-Down-Analyse und besitzt keine Linksrekursivitäten.

Bew.: für Nichtexistenz von Linksrekursivitäten ($A \rightarrow A\sigma$)

a) Falls außer einer linksrekursiven Produktion $A \rightarrow A\sigma$ auch noch $A \Rightarrow^* \epsilon$ gilt, dann ist:

$$\text{FIRST}(\sigma) \subset \text{FIRST}(A)$$

und

$$\text{FIRST}(\sigma) \subset \text{FOLLOW}(A)$$

also

$$\text{FIRST}(\sigma) \in \text{FIRST}(A) \cap \text{FOLLOW}(A) \neq \emptyset$$

Damit folgt ein Widerspruch zu (E_2)!

b) Falls es eine linksrekursive Produktion $A \rightarrow A\sigma$ gibt, jedoch *nicht* $A \Rightarrow^* \epsilon$ gilt, dann muß es eine alternative Produktion $A \rightarrow \tau$ und ein Zeichen $a \in T$ geben mit $\tau \Rightarrow^* a..$

Dann ist $a \in \text{FIRST}(A\sigma) \cap \text{FIRST}(\tau) \neq \emptyset$.

Dies ist ein Widerspruch zu (E_1)!

4.3 Recursive-Descent-Parser

Wir wollen zeigen, wie man systematisch zu einer gegebenen LL(1)-Grammatik $G = (N,T,P,S)$ einen Recursive-Descent-Parser entwickeln kann.

G besitze die Nichtterminalsymbole $S, S_1, S_2, S_3, \dots, S_n$. S sei das Startsymbol.

Dann hat der Parser die folgende Grobstruktur:

```

/* Parser */
int symbol;

void error() {...}
void s1() {...}
...
void sn() {...}
void s() {...}

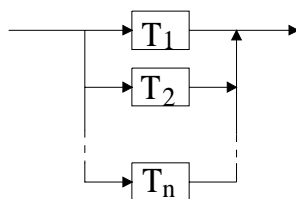
main() {
    getsym(); /* fordert erstes Symbol von Scanner */
    s();
}

```

Regeln für die Ableitung des Codes der Prozeduren S, S_1, S_2, \dots, S_n aus den *Syntaxdiagrammen* der Grammatik.

- Es bedeuten: T, T_i : Teilgraphen der Syntax
- $P(T)$: Code, der einem Teilgraphen zugeordnet wird
- X : Nichtterminalsymbole
- a : Terminalsymbole

a) Auswahl



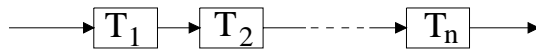
Diesem Diagramm wird zugeordnet:

```

if (symbol ∈ FIRST(T1)) P(T1);
else if (symbol ∈ FIRST(T2)) P(T2);
else if . . .
else if (symbol ∈ FIRST(Tn)) P(Tn);
else error();

```

b) Sequenz



Diesem Diagramm wird zugeordnet:

```
{ P(T1); P(T2); ... P(Tn) }
```

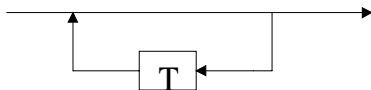
c) Optionale Produktion



Diesem Diagramm wird zugeordnet:

```
if (symbol ∈ FIRST(T)) P(T);
```

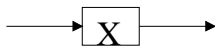
d) Iteration



Diesem Diagramm wird zugeordnet:

```
while (symbol ∈ FIRST(T)) P(T);
```

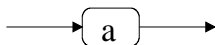
e) Nichtterminal



Diesem Diagramm wird zugeordnet:

```
X (Aufruf von X)
```

f) Terminal



Diesem Diagramm wird zugeordnet:

```
if (symbol == asym) getsym(); else error();
```

Bsp.: Ein Parser für arithmetische Ausdrücke (Vorstufe für PL/0-Parser)

$G = (N, T, P, S)$ mit $S = \text{EXPR}$
 $N = \{ \text{EXPR}, \text{TERM}, \text{FACT} \}$
 $T = \{ \text{ident}, \text{number}, +, -, *, /, (,) \}$
 $P = \{ \text{EXPR} \rightarrow [+|-] \text{TERM} \{ + \text{TERM} \mid - \text{TERM} \};$
 $\quad \text{TERM} \rightarrow \text{FACT} \{ * \text{FACT} \mid / \text{FACT} \};$
 $\quad \text{FACT} \rightarrow \text{ident} \mid \text{number} \mid (\text{EXPR}) \quad \}$

Die LL(1)-Eigenschaften sind erfüllt, wie man leicht nachprüft, vgl. Kopp.

Der folgende Pseudocode zeigt die Struktur des Parsers für arithmetische Ausdrücke. Getsym ist der lexikalische Analysator, der das nächste Symbol in der Variablen *symbol* speichert.

```

/* parser */
int symbol;

void expression() {
    if (symbol == plus || symbol == minus)
    {
        getsym(); term();
    }
    else
        term();
    while (symbol == plus || symbol == minus)
    {
        getsym(); term();
    }
} /* expression */

void term() {
    factor();
    while (symbol == times || symbol == slash)
    {
        getsym(); factor();
    }
} /* term */

void factor() {
    switch (symbol) {
        case ident : getsym(); break;
        case number : getsym(); break;
        case lparen : getsym();
                    expression();
                    if (symbol == rparen) getsym();
                    else error();
                    break;
    }
} /* factor */

main()
{
    getsym();
    expression();
} /* parser */

```

4.4 Der PL/0-Parser als Beispiel

Die meisten FIRST- und FOLLOW-Mengen zur PL/0-Grammatik sind in folgender Tabelle enthalten:

X	FIRST(X)	FOLLOW(X)
BLOCK	const, var, procedure, ident, call, begin, if, while, ε	., ;
STATEMENT	ident, call, begin, if, while, write, writeln, read, ε	., ;, end, else
CONDITION	odd, +, -, ident, number, (then, do
EXPRESSION	+, -, ident, number, (), ., ;, =, #, <, <=, >, >=, then, do, end
TERM	(, ident, number), ., ;, =, #, <, <=, >, >=, then, do, end, +, -
FACTOR	(, ident, number), ., ;, =, #, <, <=, >, >=, then, do, end, +, -, *, /

Verifikation der LL(1)-Eigenschaften: Übungsaufgabe

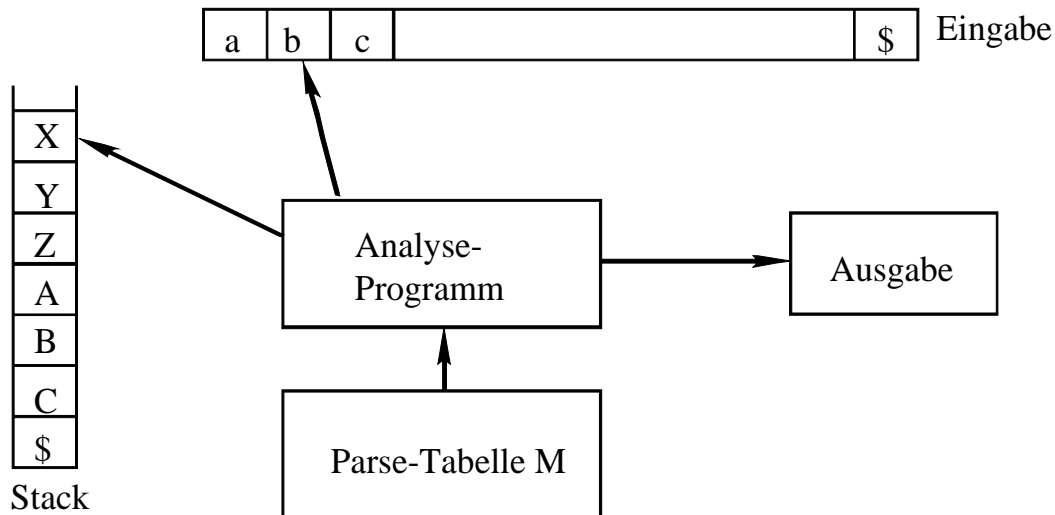
Aufgaben des PL/0-Parsers

- Syntaxprüfung
- Fehlererkennung: Aufruf einer Fehlerprozedur, die die Analyse abbricht
- Aufbau einer Symboltabelle (es gelten ähnliche Regeln für die Gültigkeit bzw. Lebensdauer der Symbole wie in Pascal).

4.5 Tabellengesteuerte Top-Down-Parser

Recursive Descent Parser eignen sich gut für die Implementierung von Hand. Eine tabellengesteuerte Analysetechnik hingegen ist besser automatisierbar und i.a. auch effizienter im Ablauf.

Wir betrachten ein Modell eines tabellengesteuerten Parsers:



Vorgehensweise

Der Parser liest die Eingabe von links nach rechts und verarbeitet sie nach den Vorschriften der Parse-Tabelle, die aus der Grammatik abgeleitet ist. Das Programm selbst ist von der speziellen Grammatik unabhängig. Der Stack führt Buch über noch nicht erledigte Teilaufträge.

Die prädiktive *Parse-Tabelle* M besitzt für jedes $x \in N$ eine Zeile und für jedes $a \in T$ und das $\$$ -Zeichen eine Spalte. $\$$ gehört nicht zum Alphabet und markiert aus rein technischen Gründen das *Ende* der *Eingabe* bzw. des *Stacks*. $\$$ wird in $FOLLOW(A)$ aufgenommen für alle $a \in N$, die am rechten Ende einer Satzform stehen können.

Die Einträge der Parse-Tabelle: $M(X,a)$ sind entweder *leer* oder mit einer Produktion der *Grammatik* besetzt.

Das *Verhalten des Parsers* wird bestimmt durch das aktuelle Eingabezeichen a und durch das oberste Zeichen X des *Stacks*:

1. Falls $X = a = \$$ ist, so wird das analysierte Programm akzeptiert.
2. Falls $X = a \neq \$$ ist, so wird X vom Stack entfernt und das nächste Eingabezeichen gelesen
3. Falls $X \in T$ und $X \neq a$ ist, so liegt ein Syntaxfehler vor.
4. Falls $X \in N$ und $M(X,a)$ leer ist, so liegt ein Syntaxfehler vor.
5. Falls $X \in N$ und $M(X,a) = X \rightarrow Y_1 \dots Y_k$ mit $Y_1, \dots, Y_k \in V$, dann ersetzt das Programm den obersten Stackeintrag X durch Y_k, \dots, Y_1 , so daß Y_1 das oberste Stackelement ist. In diesem Fall liefert er als Ausgabe die Produktion $X \rightarrow Y_1 \dots Y_k$.

Darstellung als Pseudocode (vgl. Kopp; Aho, Sethi, Ullman):

```

lies erstes Eingabezeichen a;
PUSH(Startsymbol);
REPEAT
  IF X ∈ T ∪ {$} THEN
    IF X = a THEN
      BEGIN
        POP; {Entferne X vom Stack}
        Lies nächstes Eingabezeichen
      END
    ELSE Error
  ELSE {X ∈ N}
    IF (M(X,a) = X → Y1 ... Yk) THEN
      BEGIN
        POP; {Entferne X vom Stack}
        FOR i := k DOWNTO 1 DO PUSH(Yi);
        Gib X → Y1 ... Yk aus
      END
    ELSE Error
UNTIL X = $

```

Konstruktion von prädiktiven Parse-Tabellen

Der Algorithmus zur Konstruktion einer Parse-Tabelle aus einer Grammatik G geht von folgender einfachen Idee aus:

Sei $A \rightarrow \alpha$ eine Produktion der Grammatik G .

Ist $a \in \text{FIRST}(\alpha)$, dann expandiert der Parser A zu α , wenn a das aktuelle Eingabesymbol ist.

Zu Komplikationen kann es kommen, wenn $\alpha = \varepsilon$ ist oder $\alpha \Rightarrow^* \varepsilon$ gilt. In diesem Fall muß A erneut zu α expandiert werden, wenn das aktuelle Eingabesymbol in $\text{FOLLOW}(A)$ ist oder wenn in der Eingabe die Endmarkierung $\$$ erreicht wurde und $\$$ in $\text{FOLLOW}(A)$ enthalten ist.

Insgesamt lautet der *Algorithmus* dann:

1. Führe für jede der Produktionen $A \rightarrow \alpha$ der Grammatik die Schritte 2 und 3 durch.
2. Trage für jedes Terminal $a \in \text{FIRST}(\alpha)$ die Produktion $A \rightarrow \alpha$ in $M[A, a]$ ein.
3. Wenn $\varepsilon \in \text{FIRST}(\alpha)$, trage $A \rightarrow \alpha$ für jedes Terminal $b \in \text{FOLLOW}(A)$ an der Stelle $M[A, b]$ ein. Ist $\varepsilon \in \text{FIRST}(\alpha)$ und $\$ \in \text{FOLLOW}(A)$, so trage $A \rightarrow \alpha$ in $M[A, \$]$ ein.
4. Trage in jedem undefinierten Eintrag *error* ein.

4.6 Grundlagen der Bottom-Up-Syntaxanalyse

Bei der Technik der Bottom-Up-Syntaxanalyse geht man von einem gegebenen Satz der Grammatik aus und *reduziert* diesen Schrittweise, bis das Startsymbol erreicht wird.

Stichwort: LR(k)-Grammatiken: Eingabestring wird von links nach rechts gelesen und es werden Reduktionen angewandt, so daß eine rechtskanonische Ableitung entsteht. k ist die Anzahl der Look-ahead-Symbole.

Ein Teilstring β einer Satzform $\alpha\beta$ heißt *Handle*, falls

- β die rechte Seite einer Produktion $N \rightarrow \beta$ ist,
- der Reduktionsschritt $\alpha\beta \leftarrow \alpha N$ sich zu einer linkskanonischen Produktion fortsetzen läßt.

Zur Speicherung des bereits bearbeiteten Teiles einer Satzform benutzen Bottom-Up-Parser in der Regel einen *Parse-Stack*. Das Handle ist das oberste Stackelement.

Bsp.: $G = (N,T,P,S)$ mit $N = \{E, T, S\}$, $T = \{a, +\}$ und

$$\begin{aligned}
 P = \{ & S \rightarrow E && (1) \\
 & E \rightarrow E + T \mid T && (2), (3) \\
 & T \rightarrow a && (4) \},
 \end{aligned}$$

Eingabestring: a+a+a

Schritt	Aktion	Produktion	Stack	Handle
1	Lies a		a	a
2	Reduziere	4	T	T
3	Reduziere	3	E	
4	Lies +		E+	
5	Lies a		E+a	a
6	Reduziere	4	E+T	E+T
7	Reduziere	2	E	
8	Lies +		E+	
9	Lies a		E+a	a
10	Reduziere	4	E+T	E+T
11	Reduziere	2	E	E
12	Reduziere	1	S	

Bottom-Up-Parser sind immer *tabellen-gesteuert*. Das Hauptproblem ist das Aufstellen der Parse-Tabellen. In der Regel werden hierfür spezielle Programme benötigt, da der Aufwand sehr groß ist. Eine Parse-Tabelle T ist eine rechteckige Matrix, die von zwei Variablen indiziert wird: dem Zustand des Parsers (erreichte Position innerhalb der Produktion) und dem *Eingabe-Symbol* (Terminal oder Nichtterminal).

Die Tabelleneinträge spezifizieren, ob der Parser

- die Eingabe korrekt akzeptiert (Accept)
- als inkorrekt zurückweist (Reject)
- in einen anderen Zustand übergeht (Shift)
- eine bestimmte Produktion reduziert (Reduce)

Im Gegensatz zu obigem Beispiel werden i.a. Zustände gestackt.

Der Algorithmus lautet als Pseudocode:

```

INPUTSYMBOL:= erstes Symbol des Satzes
STATUS:= 1
PUSH(STATUS)
PARSING:= true
repeat
  Bestimme ACTION aus TABLE[STATUS,INPUTSYMBOL]
  case ACTION of
    SHIFT: STATUS:= nächster Status aus TABLE
           PUSH(STATUS)
           if (INPUTSYMBOL ∈ T) THEN GETSYM
           INPUTSYMBOL:= SYM
    REDUCE: POP n Elemente vom Stack, wobei n die Länge der rechten
           Seite der Produktion ist (Spez. in Table)
           STATUS:= Status oben aus Stack
           INPUTSYMBOL:= linke Seite der angew. Produktion
    REJECT: Fehlermeldung
           PARSING:= false
    ACCEPT: Erfolgsmeldung
           PARSING:= false
until not PARSING
    
```

Parse-Tabelle zu unserem Beispiel (unbesetzte Einträge bedeuten Reject, \$: Eingabestring-Ende)

Symbol Zustand	S	E	T	a	+	\$
1	Accept	Shift 2	Shift 3	Shift 4		
2					Shift 5	Reduce 1
3					Reduce 3	Reduce 3
4					Reduce 4	Reduce 4
5			Shift 6	Shift 4		
6					Reduce 2	Reduce 3

Die folgende Tabelle zeigt den Ablauf des Parsens von a+a+a\$.

Zustand	Symbol	Stack	Aktion
1	a	1	Shift zu 4, lies a
4	+	14	Reduce (4), T → a
1	T	1	Shift zu 3
3	+	13	Reduce (3), E → T
1	E	1	Shift zu 2
2	+	12	Shift zu 5, lies +
5	a	125	Shift zu 4, lies a
4	+	1254	Reduce (4), T → a
5	T	125	Shift zu 6
6	+	1256	Reduce (2), E → E + T
2	E	1	Shift zu 2
2	+	12	Shift zu 5, lies +
5	a	125	Shift zu 4, lies a
4	\$	1254	Reduce (4), T → a
5	T	125	Shift zu 6

6	\$	1256	Reduce (2), $E \rightarrow E + T$
1	E	1	Shift zu 2
2	\$	12	Reduce (1), $S \rightarrow E$
1	S	1	Accept, fertig!

Bei Shift-Operationen werden Terminal-Symbole auf der rechten Seite gelöscht!