

## 2 Sprachanalyse

### 2.1 Formale Sprachen und Grammatiken

#### Definition

- Ein *Symbol* ist eine unteilbare Einheit, dargestellt durch ein Zeichen, eine Zeichenfolge oder ein Schlüsselwort, z.B. + : ; != WHILE IF
- Ein *Alphabet*  $A$  ist eine Menge von Symbolen, z.B. enthält das C-Alphabet die Symbole - / \* ++ a b c A B C DO SWITCH FOR.
- Ein *String*  $\sigma$  über einem Alphabet  $A$  ist eine Folge  $\sigma = a_1, a_2, \dots, a_n$  von Symbolen  $a_i \in A$ . Die Länge von  $\sigma$  wird mit  $\#\sigma$  bezeichnet. Der *Leerstring* wird mit  $\epsilon$  (manchmal auch  $\lambda$ ) bezeichnet.
- Die *Menge aller Strings* über einem Alphabet  $A$  wird mit  $A^*$  bezeichnet.
- Die *Menge aller Strings der Länge  $\geq 1$*  über einem Alphabet  $A$  wird mit  $A^+$  bezeichnet.
- Eine *Sprache*  $L$  über dem Alphabet  $A$  ist eine Teilmenge von  $A^*$ .

Die meisten praktisch bedeutsamen Sprachen können nicht auf diese einfache Weise definiert werden. Man benötigt dazu das Konzept der Grammatiken. Hierbei werden zusätzliche Zwischensymbole (Begriffe wie *Name* oder *Block*) benötigt, die nur während der Erzeugung der Elemente der Sprache gebraucht werden.

Man nennt sie *nichtterminale* Symbole, während die bisherigen Symbole nun genauer *terminale* Symbole genannt werden.

- Eine *Grammatik*  $G$  besteht aus vier Komponenten  $G = (N, T, P, S)$ . Diese sind definiert als

N: Menge von nichtterminalen Symbolen  
 T: Menge von terminalen Symbolen  
 P: Menge von Produktionen  
 S: Startsymbol

Ferner muß gelten:  $N \cap T = \emptyset, S \in N$ .

$V := N \cup T$  ist das Vokabular (Gesamtalphabet) der Grammatik.

- Produktionen* sind Vorschriften zum Ersetzen von Strings durch Strings: Schreibweise  $\sigma \rightarrow \tau$ .

**Voraussetzung:**  $\sigma, \tau \in (N \cup T)^*$ .

- Ein String  $\tau$  heißt *direkt ableitbar* aus dem String  $\sigma$ :  $\sigma \Rightarrow \tau$ , falls  $\tau$  durch Anwendung einer einzigen Produktion aus  $P$  aus  $\sigma$  hervorgeht, d.h. ist  $\sigma = \alpha\delta\beta$  und  $\tau = \alpha\gamma\beta$  und ist  $\delta \rightarrow \gamma$  eine Produktion, so gilt  $\sigma \Rightarrow \tau$ .
- Ein String  $\tau$  heißt *ableitbar* aus dem String  $\sigma$ :  $\sigma \Rightarrow^* \tau$ , falls er durch  $N$  ( $N \geq 0$ ) Produktionen aus  $\sigma$  abgeleitet werden kann, d.h. es existiert eine Folge  $\alpha_0, \dots, \alpha_N$  mit  $\alpha_0 = \sigma, \alpha_{j-1} \Rightarrow \alpha_j, \alpha_N = \tau$ .
- Eine *Satzform* ist ein String, der aus dem Startsymbol der Grammatik ableitbar ist: ein String  $\tau$  mit  $S \Rightarrow^* \tau$ .

- l) Ein *Satz* ist ein String, der nur aus Terminalsymbolen besteht und aus dem Startsymbol abgeleitet werden kann: also ein String  $\tau \in T^*$  mit  $S \Rightarrow^* \tau$ .
- m) Eine *Sprache*  $L(G)$ , die von der Grammatik  $G$  produziert wird, ist definiert als  $L(G) = \{\tau \in T^* \mid S \Rightarrow^* \tau\}$ . Sie besteht aus den Sätzen der Grammatik.

**Bsp.:**

$T = \{a, b, c, \dots, z, 0, 1, \dots, 9\}$

$N = \{\text{NAME}, \text{NREST}, \text{BU}, \text{BUZI}\}$

$P = \{\text{NAME} \rightarrow \text{BU NREST} \mid \text{BU};$   
 $\text{NREST} \rightarrow \text{BUZI NREST} \mid \text{BUZI};$   
 $\text{BUZI} \rightarrow a|b|\dots|z|0|1|\dots|9;$   
 $\text{BU} \rightarrow a|b|\dots|z\}$

Startsymbol ist NAME

Diese Grammatik dient zur Erzeugung von Namen in einer Sprache wie Pascal. Wie muß sie modifiziert werden, um C-Namen zu erzeugen?

## 2.2 Die Chomsky-Hierarchie

Zur Klassifizierung werden Grammatiken nach der Form ihrer Produktionen eingeteilt. Dabei ergibt sich eine echt aufsteigende Hierarchie von Grammatikklassen.

### Typ-0-Grammatiken

Die Produktionen  $\sigma \rightarrow \tau$  mit  $\sigma, \tau \in (N \cup T)^*$  unterliegen keinen Restriktionen, insbesondere ist auch  $\sigma \rightarrow \varepsilon$  erlaubt. Für praktische Anwendungen im Compilerbau ist diese Klasse zu allgemein.

### Typ-1-Grammatiken (kontext-sensitive Grammatiken)

Die Produktionen sind von der Form

$$(i) \quad \alpha \rightarrow \beta \quad \text{mit } \#\alpha \leq \#\beta, \alpha, \beta \in (N \cup T)^*$$

bzw. in *Normalform* von der Gestalt

$$(ii) \quad \sigma A \tau \rightarrow \sigma \gamma \tau \quad \text{mit } \sigma, \tau \in (N \cup T)^*, A \in N^+, \gamma \in (N \cup T)^+.$$

In der Literatur sind auch leicht modifizierte Definitionen zu finden.

### Typ-2-Grammatiken (kontext-freie Grammatiken)

Die Produktionen haben die Form

$$A \rightarrow \sigma$$

mit  $A \in N$  und  $\sigma \in (N \cup T)^*$ .

Die Grammatiken, die man im Zusammenhang mit Programmiersprachen benutzt, sind in der Regel kontextfrei mit gewissen Ausnahmen, die man häufig durch einige zusätzliche Regeln erreicht.

### Typ-3-Grammatiken (reguläre Grammatiken)

Die Produktionen haben die Gestalt

$$\begin{array}{ll} A \rightarrow Ba & \\ A \rightarrow a & \text{linkslinier} \end{array}$$

oder

$$A \rightarrow aB$$

rechtslinear

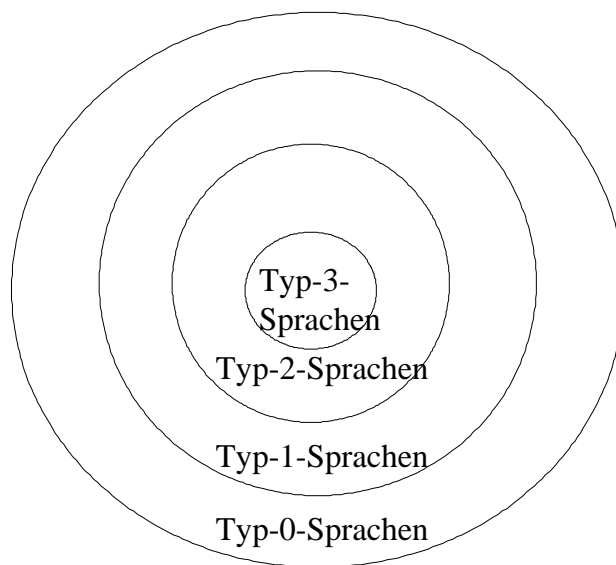
$$A \rightarrow a$$

mit  $A, B \in N$ ,  $a \in T$ , möglicherweise auch  $a = \epsilon$ .

Reguläre Grammatiken beschreiben ziemlich restriktive, lokale Eigenschaften von Programmiersprachen.

### Hierarchie der Sprachklassen

Eine Sprache  $L$  heißt vom *Typ*  $i$ , falls es eine Grammatik  $G$  vom Typ  $i$  gibt, die sie erzeugt:  $L = L(G)$ .



**Bsp.:** Reguläre Grammatik für Pascal-Namen

$$N = \{NAME\}$$

$$T = \{a, b, c, \dots, z, 0, 1, \dots, 9\}$$

$$P = \{NAME \rightarrow NAME a \mid NAME b \mid \dots \mid NAME z ;$$

$$NAME \rightarrow NAME 0 \mid NAME 1 \mid \dots \mid NAME 9 ;$$

$$NAME \rightarrow a \mid b \mid \dots \mid z \}$$

Startsymbol ist NAME

**Bsp.:** Kontextsensitive Grammatik  $G = (N, T, P, S)$  mit

$$N = \{A, B, C\}$$

$$T = \{a, b, c\}$$

$$P = \{A \rightarrow aABC \mid abC ;$$

$$CB \rightarrow BC ;$$

$$bB \rightarrow bb ;$$

$$bC \rightarrow bc ;$$

$$cC \rightarrow cc \}$$

Startsymbol ist A

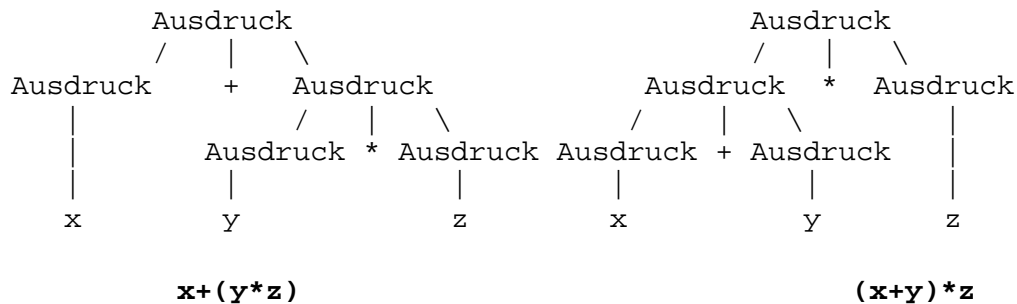
*Frage:* Ist der String aabbcc ein Satz dieser Grammatik?

## Zusammenfassung der Chomsky-Hierarchie

Typ	Name	Form der Produktionen	Äquivalenter Automat	Beispiele
0	<i>Nicht eingeschränkte Grammatik, Semi-Thue-System</i>	$\alpha \rightarrow \beta$ mit beliebigen $\alpha, \beta \in (N \cup T)^*$ , $\alpha \neq \epsilon$	Turing-Maschine	$S \rightarrow ACaB, Ca \rightarrow aaC,$ $CB \rightarrow DB, CB \rightarrow E,$ $aD \rightarrow Da, AD \rightarrow AC,$ $aE \rightarrow Ea, AE \rightarrow \epsilon$
1	<i>Kontext-sensitive Grammatik</i>	$\alpha \rightarrow \beta$ mit $\alpha, \beta \in (N \cup T)^*$ $\# \alpha \leq \# \beta$  Normalform: $\sigma A \tau \rightarrow \sigma \gamma \tau$ $\sigma, \tau, \gamma \in (N \cup T)^*$	Linear beschränkter Automat  (Turing-Maschine mit durch Markierungen eingeschränktem Band)	$S \rightarrow Abc, Ab \rightarrow aABb,$ $Bb \rightarrow bB, Bc \rightarrow bcc,$ $A \rightarrow a$
2	<i>Kontext-freie Grammatik</i>	$A \rightarrow \sigma, \sigma \in (N \cup T)^*$  Normalform: $A \rightarrow BC$ $A \rightarrow a$	Stack-Automat	$S \rightarrow AB, A \rightarrow aAb,$ $B \rightarrow Bc, B \rightarrow c,$ $A \rightarrow ab$
3	<i>Reguläre Grammatik</i>  <i>Rechtslineare Grammatik</i>  <i>Linkslineare Grammatik</i>	$A \rightarrow \alpha B, A \rightarrow \alpha$ $\alpha \in T^*$  $A \rightarrow B\alpha, A \rightarrow \alpha\alpha \in T^*$  Normalformen: $A \rightarrow aB, A \rightarrow a$ $A \rightarrow Ba, A \rightarrow a$	Endlicher Automat	$S \rightarrow 0A,$ $A \rightarrow 10A \mid \epsilon,$ $S \rightarrow S10 \mid 0$



Der Satz  $x + y * z$  hat zwei verschiedene Ableitungsbäume:



Wegen der verschiedenen Bedeutung der beiden Bäume ist diese Grammatik ungeeignet. Mehrdeutige Grammatiken müssen in eindeutige Grammatiken überführt werden bzw. durch zusätzliche semantische Regeln ergänzt werden.

**Bsp.:** „Standardgrammatik“ für arithmetische Ausdrücke  $G = (N, T, P, S)$  mit

- $N = \{ \text{EXPR, TERM, FACT} \} ;$
- $T = \{ v, +, -, *, /, (, ) \} ;$
- $P = \{ \text{EXPR} \rightarrow \text{TERM} \mid \text{EXPR} + \text{TERM} \mid \text{EXPR} - \text{TERM} ;$
- $\text{TERM} \rightarrow \text{FACT} \mid \text{TERM} * \text{FACT} \mid \text{TERM} / \text{FACT} ;$
- $\text{FACT} \rightarrow v \mid (\text{EXPR}) \}$
- $S = \text{EXPR}$

Sie ist eindeutig und berücksichtigt die üblichen Vorrangregeln zwischen arithmetischen Operatoren.

## Analysetechniken

Die Syntaxanalyse eines Compilers erzeugt zu einem gegebenen Programm einen Parsebaum bzgl. der Grammatik der Programmiersprache. Dazu gibt es zwei im Ansatz unterschiedliche Methoden.

### 1. Top-Down-Analyse

Sie geht vom Startsymbol  $S$  der Grammatik aus und versucht, von oben nach unten eine Baumstruktur aufzubauen, die in ihrer Basis dem zu analysierenden Satz entspricht.

### 2. Bottom-Up-Analyse

Der eingelesene Text wird reduziert, bis zum Schluß der gesamte Text in das Startsymbol zurückgeführt ist. Der Baum wächst von unten nach oben!

## 2.4 Darstellung von Produktionen

Für die Darstellung von Produktionen einer Grammatik im Zusammenhang mit dem Compilerbau werden in der Regel folgende Notationen benutzt:

BNF (Backus-Naur-Form)  
EBNF (Extended-Backus-Naur-Form)  
Syntaxdiagramme

### 1. BNF

Produktionen haben die Gestalt:

Leftside ::= Definition

wobei Leftside  $\in (N^+ \cup T^*)$ , aber normalerweise  $\in N$ ,  
Definition  $\in (N \cup T)^*$ . Nichtterminale werden durch  $\langle \rangle$  gekennzeichnet.  
Das erste Nichtterminal ist das Startsymbol.

**Bsp.:**  $\langle \text{integer} \rangle ::= \langle \text{digit} \rangle \langle \text{rest of integer} \rangle$   
 $\langle \text{rest of integer} \rangle ::= \langle \text{digit} \rangle \langle \text{rest of integer} \rangle \mid \epsilon$   
 $\langle \text{digit} \rangle ::= 0 \mid 1 \mid 2 \mid \dots \mid 9$

### 2. EBNF

Zusätzliche Symbole zur Erleichterung der Schreibweise.

- a) { } : geschweifte Klammern bedeuten die Wiederholung eines Strings (null mal oder beliebig oft).
- b) [ ] : eckige Klammern bezeichnen ein optionales Symbol (es steht null mal oder einmal).

**Bsp.:** So läßt sich folgender (*rekursiver*) Grammatikausschnitt in BNF-Notation

$\langle \text{integer} \rangle ::= \langle \text{unsigned integer} \rangle \mid \langle \text{sign} \rangle \langle \text{unsigned integer} \rangle$   
 $\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{unsigned integer} \rangle$   
 $\langle \text{sign} \rangle ::= + \mid -$

einfacher in EBNF-Notation mit *Iteration* schreiben:

$\langle \text{integer} \rangle ::= [ \langle \text{sign} \rangle ] \langle \text{unsigned integer} \rangle$   
 $\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle \{ \langle \text{digit} \rangle \}$   
 $\langle \text{sign} \rangle ::= + \mid -$

Die runden Klammern ( und ) werden zur Zusammenfassung benutzt. Bsp.:  $\langle \text{Faktor} \rangle (* \mid /) \langle \text{Faktor} \rangle$   
Angabe einer Anzahl von Wiederholungen bei { }:

$\langle \text{FORTRAN identifier} \rangle ::= \langle \text{letter} \rangle \{ \langle \text{letter} \rangle \mid \langle \text{digit} \rangle \}^5_0$



## 2.5 Die Modellsprache PL/0

Ursprünglich von N. Wirth stammt die Definition der vom Umfang her kleinen Programmiersprache PL/0, die als eine Art Untermenge von Pascal betrachtet werden kann. Diese Sprache wurde und wird häufig in Compilerbau-Lehrveranstaltungen als Modellsprache eingesetzt, in verschiedenen Variationen und unter verschiedenen Namen. Sie zeigt wesentliche Eigenschaften einer großen Programmiersprache:

- Prozeduren
- Rekursion
- Schleifenkonstrukte

Sie verzichtet aber auf viele wesentliche Sprachelemente wie vor allem Datentypen, was zu einem übersichtlichen und nicht zu umfangreichen Compiler führt. In vielen Büchern sind Varianten dieser Sprache aufgetaucht. Auch hier wird eine etwas erweiterte Form dargestellt und ein Compiler dafür beschrieben.

Wir beschreiben die Syntax unserer Version von PL/0 hier durch die EBNF und durch Syntaxdiagramme.

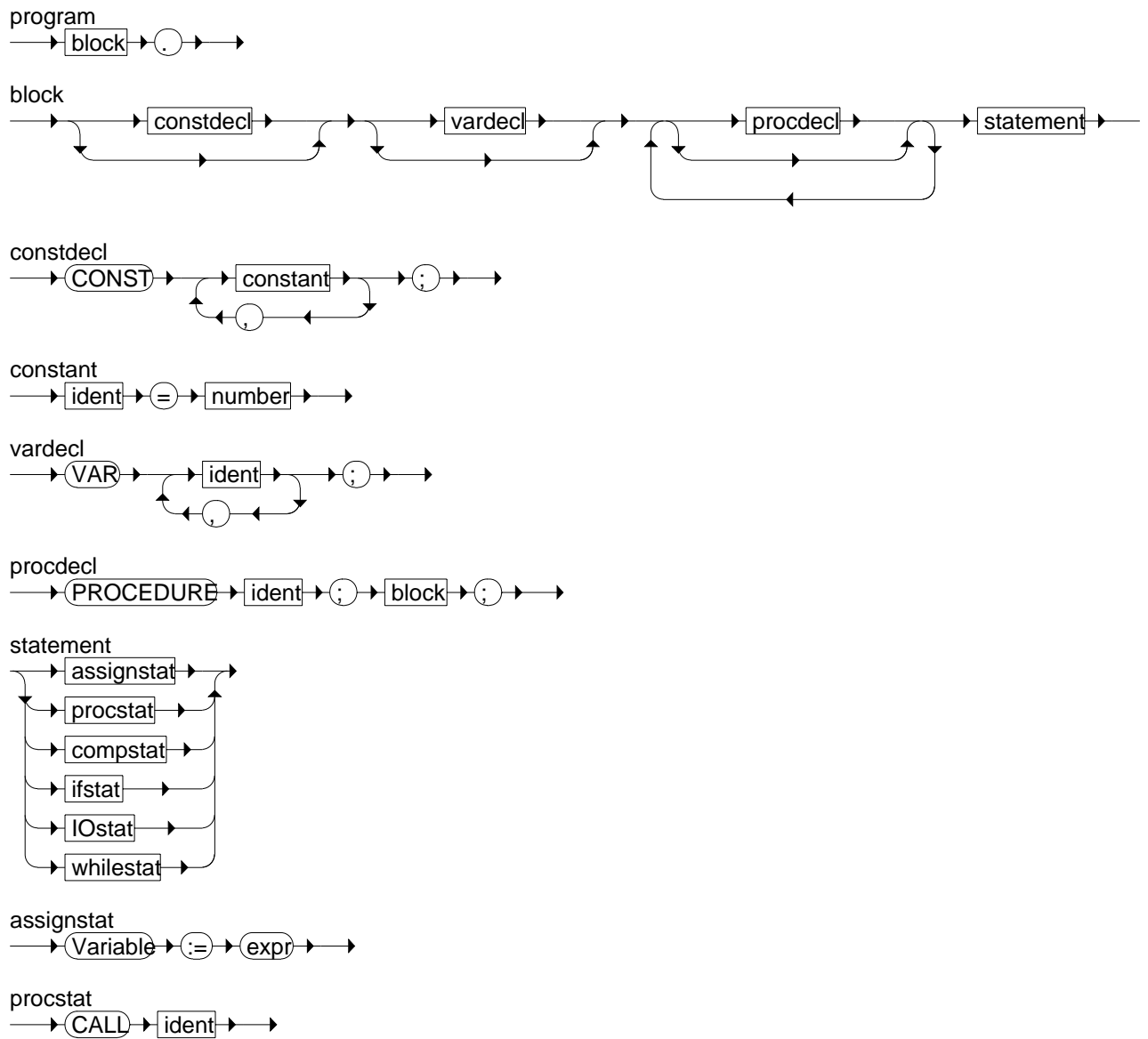
### PL/0-Syntax in EBNF

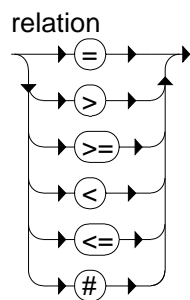
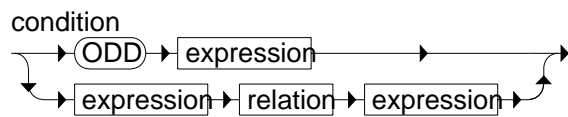
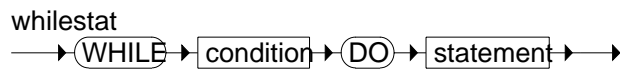
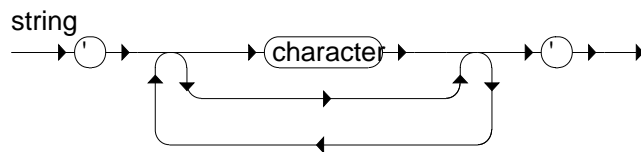
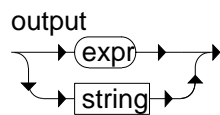
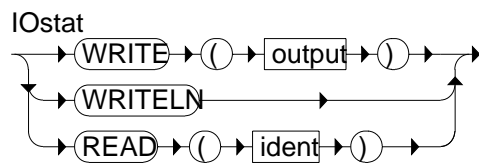
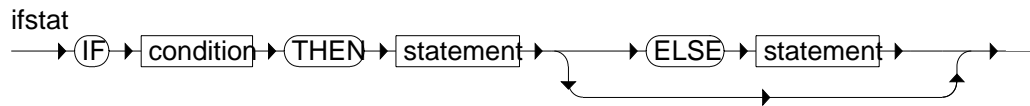
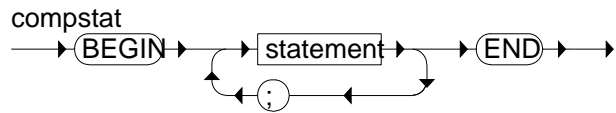
```

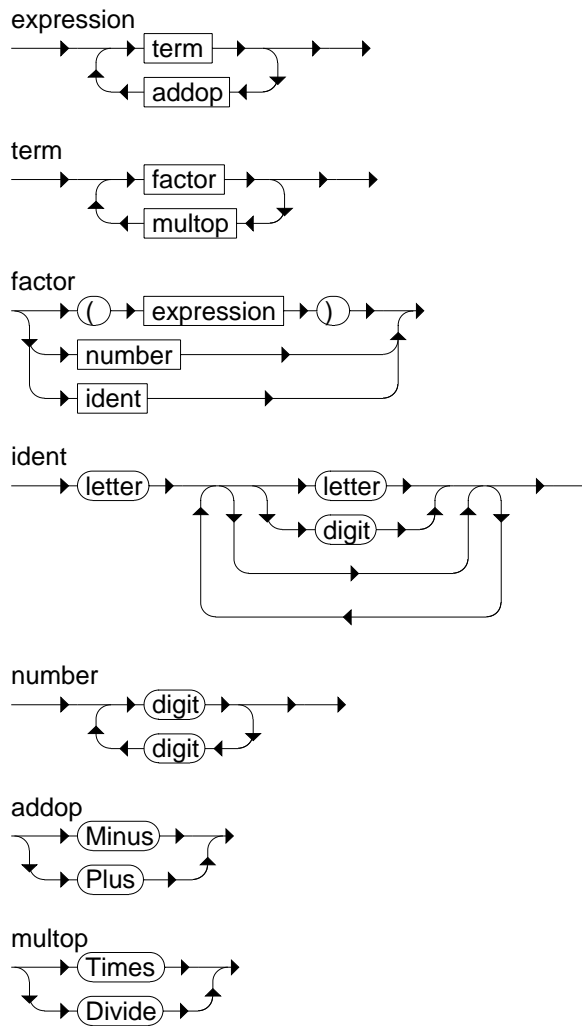
<Program> ::= <Block>.
<Block> ::= [<Constdecl>][<Vardecl>][<Procdecl>] <Statement>
<Constdecl> ::= CONST <Ident> = <Number> {, <Ident> = <Number> };
<Vardecl> ::= VAR <Ident> {, <Ident>};
<Procdecl> ::= {PROCEDURE <Ident>; <Block>;}
<Statement> ::= <AssignStat> | <ProcStat> | <CompStat> | <IfStat> | <IOStat> |
               <WhileStat> | ε
<AssignStat> ::= <Ident> := <Expression>
<ProcStat> ::= CALL <Ident>
<CompStat> ::= BEGIN <Statement> {; <Statement>} END
<IfStat> ::= IF <Condition> THEN <Statement> [ELSE <Statement>]
<IOStat> ::= WRITE( <Output> ) | WRITELN | READ(<Ident>)
<Output> ::= <Expression> | '{<Character>}'
<WhileStat> ::= WHILE <Condition> DO <Statement>
<Condition> ::= ODD <Expression> | <Expression> (=|#|<|<=>|=) <Expression>
<Expression> ::= [+|-] <Term> {(+|-) <Term>}
<Term> ::= <Factor> {(*|/) <Factor>}
<Factor> ::= <Ident> | <Number> | "(" <Expression> ")"
<Ident> ::= <Letter> { (<Letter> | <Digit> ) }
<Number> ::= <Digit> {<Digit>}
<Letter> ::= a | b | ... | z | A | B | ... | Z
<Digit> ::= 0 | 1 | ... | 9

```

Es folgt eine Darstellung der PL/0-Syntax durch Syntaxdiagramme:







Die Semantik von PL/0 lehnt sich stark an die von Pascal an.

**Ein Beispielprogramm in PL/0:**

```

const c=10;
var n, f;

procedure fak;
var i;
begin
  f:= 1; i:= 2;
  while i <= n do
  begin
    f:= f*i; i:= i+1;
  end;
end;

begin
  n:= c; call fak; write(f);
end.

```

## 2.6 Darstellung der Semantik von Programmiersprachen

Bei der Spezifikation von PL/0 haben wir nichts über die Semantik (Bedeutung) der einzelnen syntaktischen Konstrukte festgelegt. In diesem Fall war das auch nicht nötig, da wir die Semantik der einzelnen Sprachelemente entweder von *Pascal* her kennen oder leicht aus der Namensgebung der Syntax ablesen können.

Man kann allgemein sagen, daß die Trennung von Syntax und Semantik in vielen Fällen nicht ganz klar vollzogen wird, da viele Produktionen durch ihre Namensgebung semantische Untertöne vernennen lassen. Manche Satzformen spiegeln schon die Vorrangregeln etwa von Operatoren wieder.

Man kann die Semantik einer Programmiersprache aufteilen in zwei Kategorien:

- **Statische Semantik:** hier geht es um Dinge, die man zur Compile-Zeit überprüfen kann:  
Typ-Prüfungen etwa bei einer Zuweisung wie `i:= i+1`,  
Parameterüberprüfung bei Prozeduraufruf.
- **dynamische Semantik:** Eigenschaften, die nur zur Laufzeit überprüft werden können:  
Arithmetische Fehler,  
Typ-Prüfungen etwa bei Unterbereichstypen.

Formale Darstellungen der Semantik von Programmiersprachen werden bisher wenig benutzt. Im allgemeinen werden semantische Aktionen durch *verbale* Erklärungen in englisch-sprachigen Dokumenten definiert, die schwer lesbar und häufig ungenau sind.

**Bsp.:** In einer Beschreibung der Sprache *Pascal* wird die Bedeutung des while-Statements erklärt durch:

```
<while-statement> ::= WHILE <Boolean-expression> DO <statement>
```

mit dem Kommentar:

```
"The <statement> is repeatedley executed while the <Boolean-expression> yields the value TRUE. If its value is FALSE at the beginning, the <statement> is not executed at all. The <while-statement>
```

```
    WHILE b DO body
```

is equivalent to

```
    IF b THEN REPEAT body UNTIL NOT b"
```