Begriffe (Wiederholung)

Menge aller Sätze
terminales / nicht terminales
Zeichen aus denen die Sätze der Sprache bestehen
Hilfszeichen zum Bilden von Regeln
Aneinanderreihung von Zeichen aus A
Länge der Zeichenkette
`s=>t`, wenn `s->s1->s2-> ... sn->t`
s= s1 s2 s3, t= s1 t2 s3, S2 -> t2
G = (T,N,s,R)

Analysestrategien

Top down

Ausgehend vom Startsymbol werden Metasymbole durch Anwendung von Regeln ersetzt. Dabei geht man von links nach rechts vor. Es wird immer das am weitesten links stehende Metasymbol ersetzt. (Linksableitung)

Bottom up

Ausgehend vom zu analysierenden Satz wird durch Reduktion versucht, das Startsymbol herzuleiten. Es wird immer versucht, ausgehend vom am weitesten rechts stehenden Metasymbol an Hand der Regeln soviel wie möglich zu reduzieren. (Rechtsreduktion)

Beispielbetrachtung

```
N={SATZ,SUBJEKT,PRÄDIKAT,OBJEKT,ARTIKEL,VERB
   ,SUBSTANTIV}
T= {der, den, hund, briefträger, beißt}
s= SATZ
R= {
   SATZ       -> SUBJEKT PRÄDIKAT OBJEKT (1)
   SUBJEKT    -> ARTIKEL SUBSTANTIV      (2)
   OBJEKT     -> ARTIKEL SUBSTANTIV      (3)
   PRÄDIKAT   -> VERB                    (4)
   SUBSTANTIV -> hund                    (5)
   SUBSTANTIV -> briefträger             (6)
   VERB       -> beißt                   (7)
   ARTIKEL    -> der                     (8)
   ARTIKEL    -> den                     (9)
   }
```

Top down

Satzform	Regel

Satz	
SUBJEKT PRÄDIKAT OBJEKT	1
ARTIKEL SUBSTANTIV PRÄDIKAT OBJEKT	2
der SUBSTANTIV PRÄDIKAT OBJEKT	8
der hund PRÄDIKAT OBJEKT	5
der hund VERB OBJEKT	4
der hund beißt OBJEKT	7
der hund beißt ARTIKEL SUBSTANTIV	3
der hund beißt den SUBSTANTIV	9
der hund beißt den briefträger	6

Bottom up

Satzform	Regel
der hund beißt den briefträger	
ARTIKEL hund beißt den briefträger	8
ARTIKEL SUBSTANTIV beißt den briefträger	5
SUBJEKT beißt den briefträger	2
SUBJEKT VERB den briefträger	7
SUBJEKT PRÄDIKAT den briefträger	4
SUBJEKT PRÄDIKAT ARTIKEL briefträger	9
SUBJEKT PRÄDIKAT ARTIKEL SUBSTANTIV	6
SUBJEKT PRÄDIKAT OBJEKT	2
SATZ	1

Grammatik und Parser

LL(k)-Grammatik bildet die Grundlage für einen LL(k)-Parser, der die Eingabezeichen von links liest und immer das am weitesten links stehende Metasymbol ersetzt (Linksableitung). LL(k)-Parser arbeiten nach der Strategie top down

LR(k)-Grammatik bildet die Grundlage für einen LR(k)-Parser, der die Eingabezeichen von links liest und immer die am weitesten rechts stehenden Metasymbole ersetzt (Rechtsreduktion). LR(k)-Parser arbeiten nach der Strategie bottom up.

(Subjekt Praedikat Artikel Substantiv → Subjekt Praedikat Objekt)

Es wird immer die Vorausschau um k Zeichen benötigt, um die richtige alternative Regel zu finden.

BNF

Backus Naur Form oder Backus Normalform

Entstand im Rahmen der Entwicklung von Algol 60

Einführung der Metasymbole (Nicht Terminale Symbole) als linke Seiten der Regeln

Metasymbole werden häufig in '<' '>' gesetzt.

Linke und rechte Seite der Regeln werden durch das Ableitungssymbol '::=' getrennt

Regeln mit gleichen linken Seiten werden zusammengefasst, die Alternativen werden durch das Symbol '|' getrennt

Es gibt eine Reihe von Modifikationen, wie das Weglassen der spitzen Klammern, Anfügen eines Punktes am Regelende oder andere Definitionssymbole als '::='.

- Eignet sich zur Darstellung von Symbolfolgen und Alternativen
- Listen werden durch Rekursion formuliert
- BNF bildet die Grundlage für viele Parsergeneratoren

Beispiel

```
<const_def_list>   ::=  'const' <const_list> ';'
<const_list>       ::=  <const_def>
                   |    <const_list>','<const_def>
<const_def>        ::=  <const_name>=<const_symbol>
<const_name>       ::=  name
<const_symbol>     ::=  name | zahl
```

Quelltextausschnitt aus pl0.y (yacc)

```
ConstDecl: T_CONST constList ';'
   |        ;
ConstList: constList ',' T_Ident '=' T_Num {AcreateConst($5,$3);};
   |       T_Ident '=' T_Num               {AcreateConst($3,$1);};
```

Const X=0, eins=1;

Erweiterte BNF (EBNF)

- Eingeführt durch Niklaus Wirth im Rahmen der Definition von Pascal
- Darstellung optionaler Elemente durch Einführung eckiger Klammern '[' ']'
- Darstellung von Wiederholungen durch Einführung geschweifter Klammern '{' '}', die geklammerte Zeichenkette kann 0 mal, 1 mal oder beliebig oft auftreten
- Deutlich besser lesbar, durch top-down-Parser auch sehr elegant umsetzbar.

EBNF:

compound_stmt ::= "BEGIN" statement {";" statement} "END".

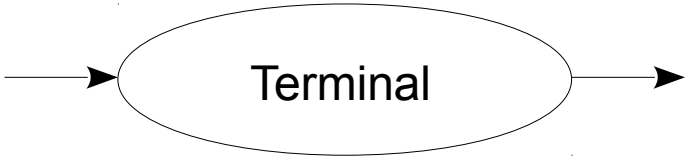
BNF (yacc)

```
statementList:    statementList ';' statement ;
                |   statement ;
                |   ;
compound_stmnt:   T_BEGIN statementList T_END;
```

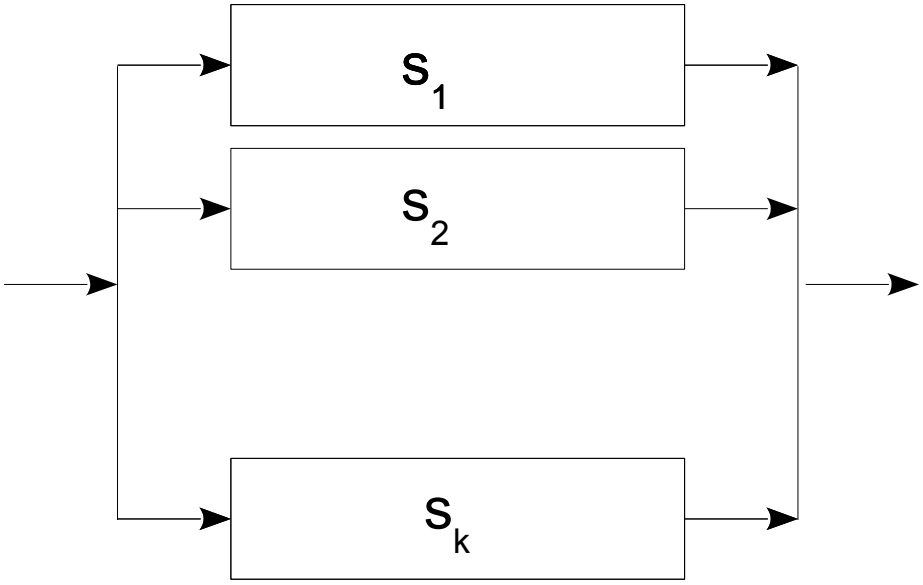
Syntaxgraphen

Darstellung eines Nicht Terminals:

Darstellung eines Terminals:



Darstellung von Alternativen:

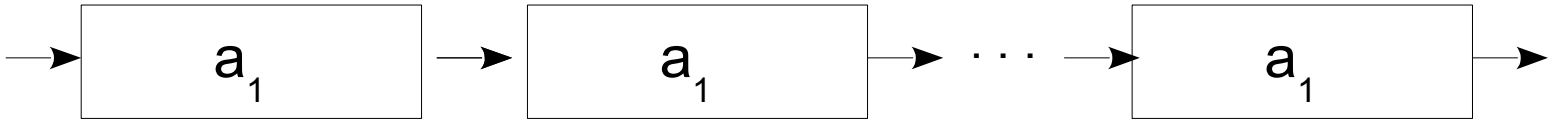


$A \rightarrow s_1 \mid s_2 \mid .... \; s_k$

Dabei sind s_i Teilgraphen, bestehen aus Termen, wie nachfolgend

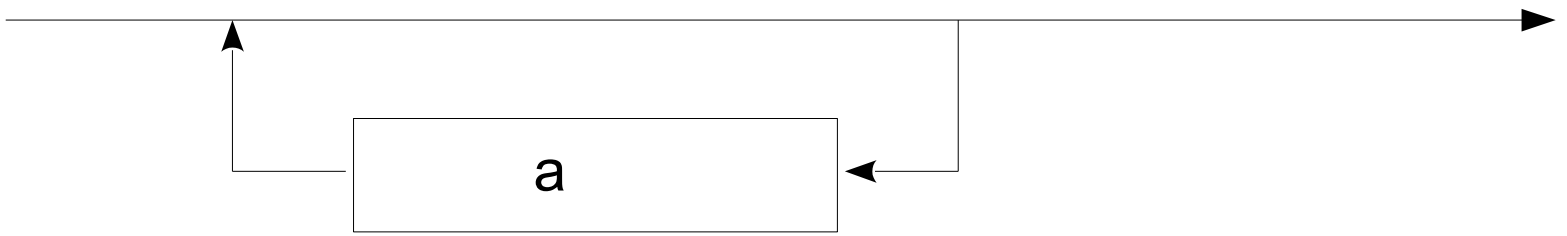
Sequenz

$S = a_1\ a_2\ ...a_k$



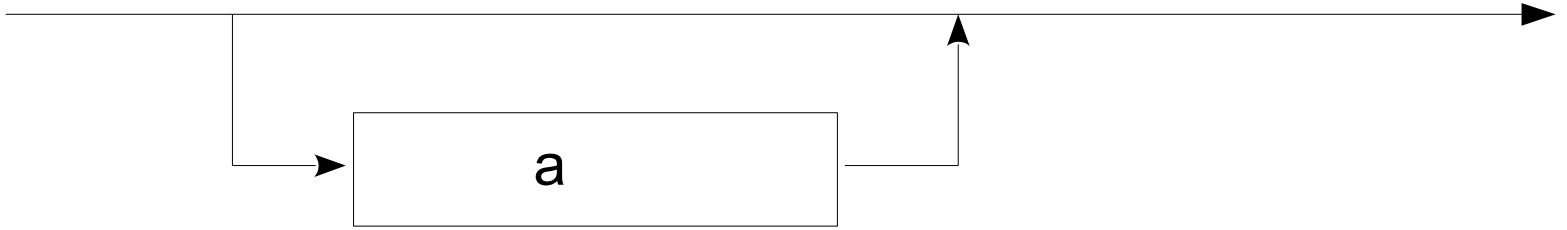
Iteration

$S = \{a\}$

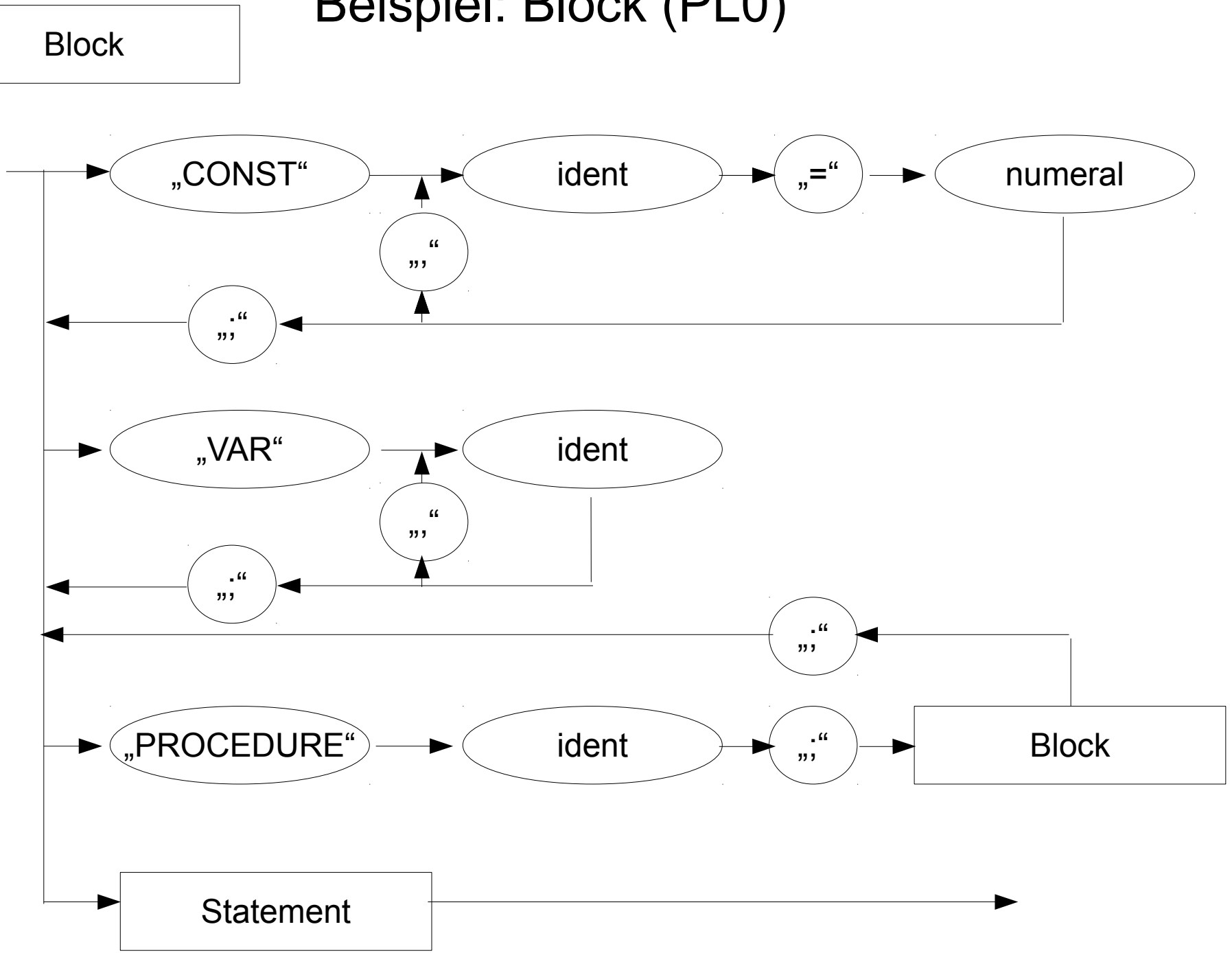


Option

$S = [a]$

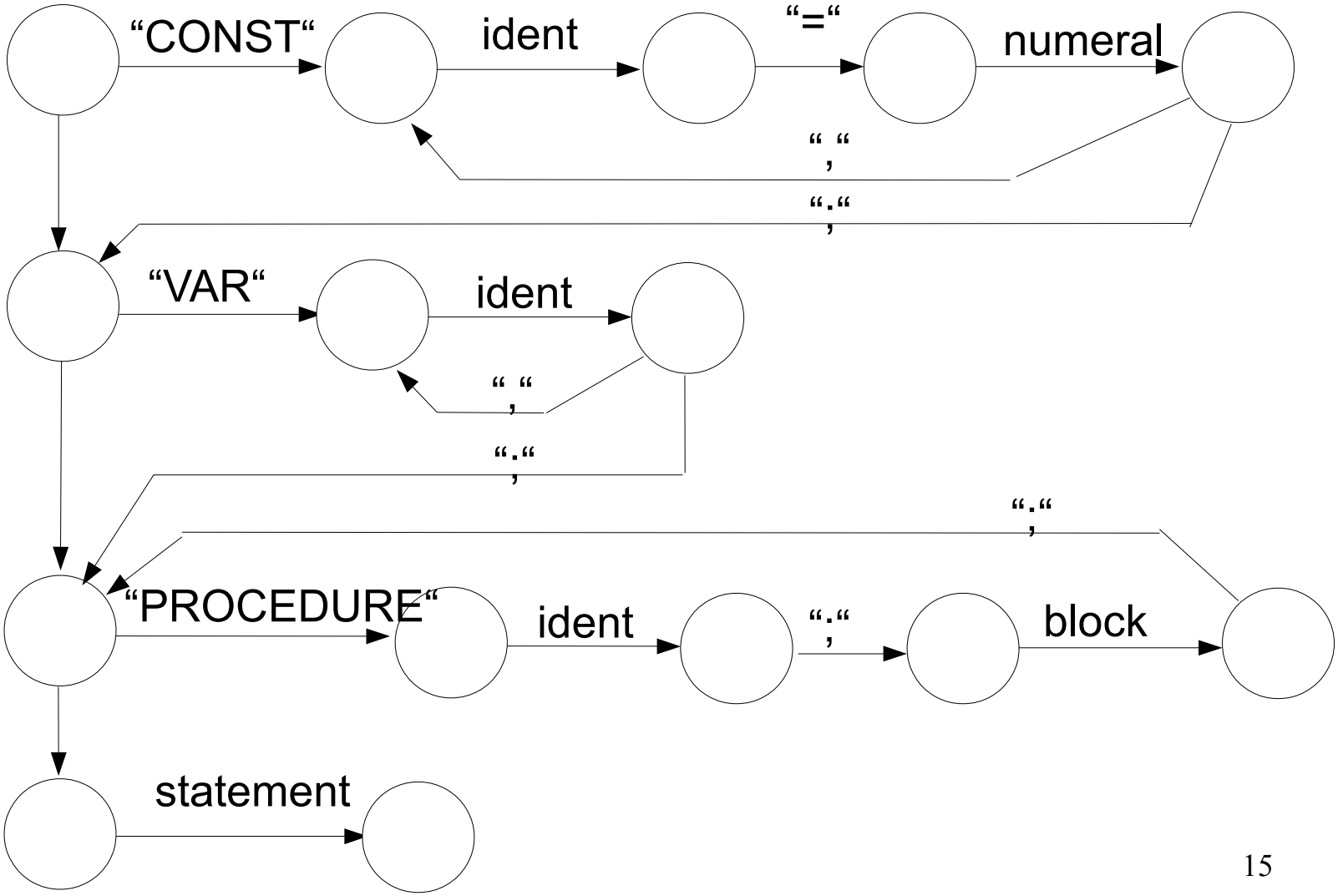


Beispiel: Block (PL0)



Andere Darstellung von Syntaxgraphen

block:



Grammatik von PL/0 nach N. Wirth

T	`+ | - | * | / | := | , | . | ; | ( | ) | ? | ! | # | = | < | > | >= | <= |`
	
	`BEGIN | CALL | CONST | DO | If | ODD | PROCEDURE | VAR | WHILE |`
	
	`numeral| ident`

N	program, block, statement, condition, expression, term, factor
s	Programm
P	→ next Page

```
programm    = block “.“.
block       = [“CONST“ ident=num{“,“ ident=num}“;“]
              ["VAR" ident { "," ident } ";" ]
              {"PROCEDURE" ident ";" block ";" } statement.

statement   = [ident ":=" expression |
              "CALL" ident |
              "?" ident |
              "!" expression |
              "BEGIN" statement { ";" statement } "END" |
              "IF" condition THEN statement |
              "WHILE" condition DO statement].

condition   = "ODD" expression |
              expression ("="|"#"|"<"|"<="|">"|">=") expression.

expression  =["+" | "-"] term {("+" | "-") term }.

term        =faktor { ("*"|"/") faktor }.

factor      =ident | number | "(" expression ")".
```

Beispiele in PL0

```
var U,P,I,R;

procedure ProcP;
P:=7*U*U/R/10;

procedure ProcI;
I:=U/R;

begin
  ?U;
  ?R;
  if R>0 then
  begin
  call ProcI;
  call ProcP;
  ! I;
  ! P
  end
end.
```

```
const c=0,d=1;
var a,
    x,
    y,
    b;

begin
  ?a;
  ?b;
  x:=0;
  while x<=a do
  begin
  y:=0;
    while y<=b do
    begin
      !x*y;
      y:=y+1
    end;
    x:=x+1
  end
end.
```

```
var f,x;

procedure fakult;
var xl;
begin
  xl:=x;
  x:=x-1;
  if x>0 then
  begin
    call fakult;
    f:=f*xl
  end
end;

begin
  ?x;
  f:=1;
  call fakult;
  !f
end.
```

Rekursion