Parser

- Der Teil eines Compilers, der prüft, ob die Folge der Token einen gültigen Satz der Sprache bildet
- Unterscheidung nach der Strategie in TopDown oder ButtomUp Parser
- Unterscheidung nach der Arbeitsweise
- LL- oder LR-Parser
- LL: von links lesend, das am weitesten links stehende Nichtterminal ersetzend
- LR: von links lesend, das am weitesten rechts stehende Nichtterminal ersetzend

Kellerautomat

- Parser basieren auf Kellerautomaten.
- Vorstellbar als eine Menge von Automaten, die sich gegenseitig aufrufen, wie Funktionen. Der Zustand des aufrufenden Automaten wird im einem Stack konserviert.
- Kellerautomaten schalten in Abhängigkeit von den nächsten k Eingabesymbolen und dem obersten Stackelement und ihrem aktuellen Zustand.
- Der Stack ergibt sich bei vielen Verfahren implizit aus dem call-stack, zB. Beim Verfahren des rekursiven Abstiegs

- Ausgangspunkt ist immer die Grammatik der zugrunde liegenden Sprache.
- Grammatik und Parser hängen eng zusammen, LL(1) Parser verarbeiten LL(1) Grammatiken.
- Zur Analyse wird k Zeichen vorausgeschaut, um eine Alternative sicher zu erkennen, Bei LL(1) Grammatiken ist k=1.
- LL Grammatiken müssen linksrekursionsfrei sein
- Durch geeignete Umformungen können Linksrekursionen beseitigt werden.
- Je nach Verfahren werden weitere Umformungen nötig (BNF->EBNF, EBNF->BNF, (E)BNF → Graphen).

LL(1) Grammatik

- Eine Klasse von Grammatiken
- Jeder Ableitungsschritt wird eindeutig durch das nächste Eingabesymbol bestimmt, Lookahead=1.
- Linksrekursionen sind nicht erlaubt.
- First: die terminalen Anfänge aller Zeichenketten, die aus einem Nichtterminal hergeleitet werden können, müssen verschieden sein.
- Kann auch ε hergeleitet werden, so müssen zusätzlich alle auf das Nichtterminal folgenden terminalen Anfänge betrachtet werden und müssen ebenfalls unterschiedlich sein (Follow).

Regeln der Grammatik von PL0 nach N. Wirth in EBNF

```
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 ')'
```

First / Follow

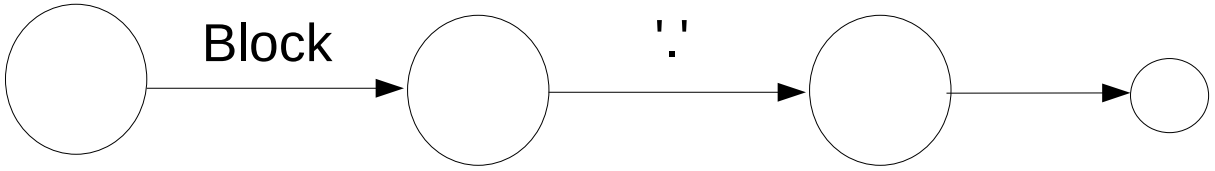
NichtTerminal X	First (X)	Follow (X)
block	CONST VAR PROCEDURE ident CALL BEGIN IF WHILE ? ! ε	. ;
statement	ident CALL BEGIN IF WHILE ? !	. ; end
condition	ODD + - ident Numeral (	THEN DO
expression	(ident numeral + -	) . ; = # < <= > >= THEN DO END
term	(ident numeral	) . ; + - = # < <= > >= THEN DO END
factor	(ident numeral	) . ; + - * / = # < <= > >= THEN DO END

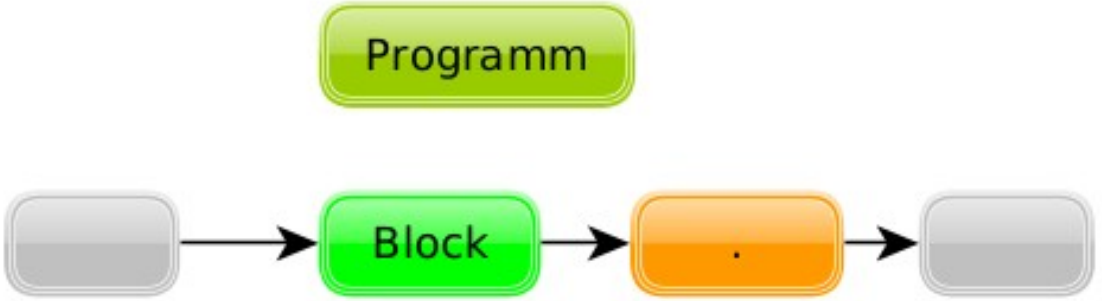
Syntaxgraphen

- Sytaxgraphen veranschaulichen die Syntaxregeln optisch sehr intuitiv.
- Der im weiteren betrachtete Parser basiert auf Syntaxgraphen.
- Es werden zwei Darstellungsformen der Syntaxgraphen vorgestellt.

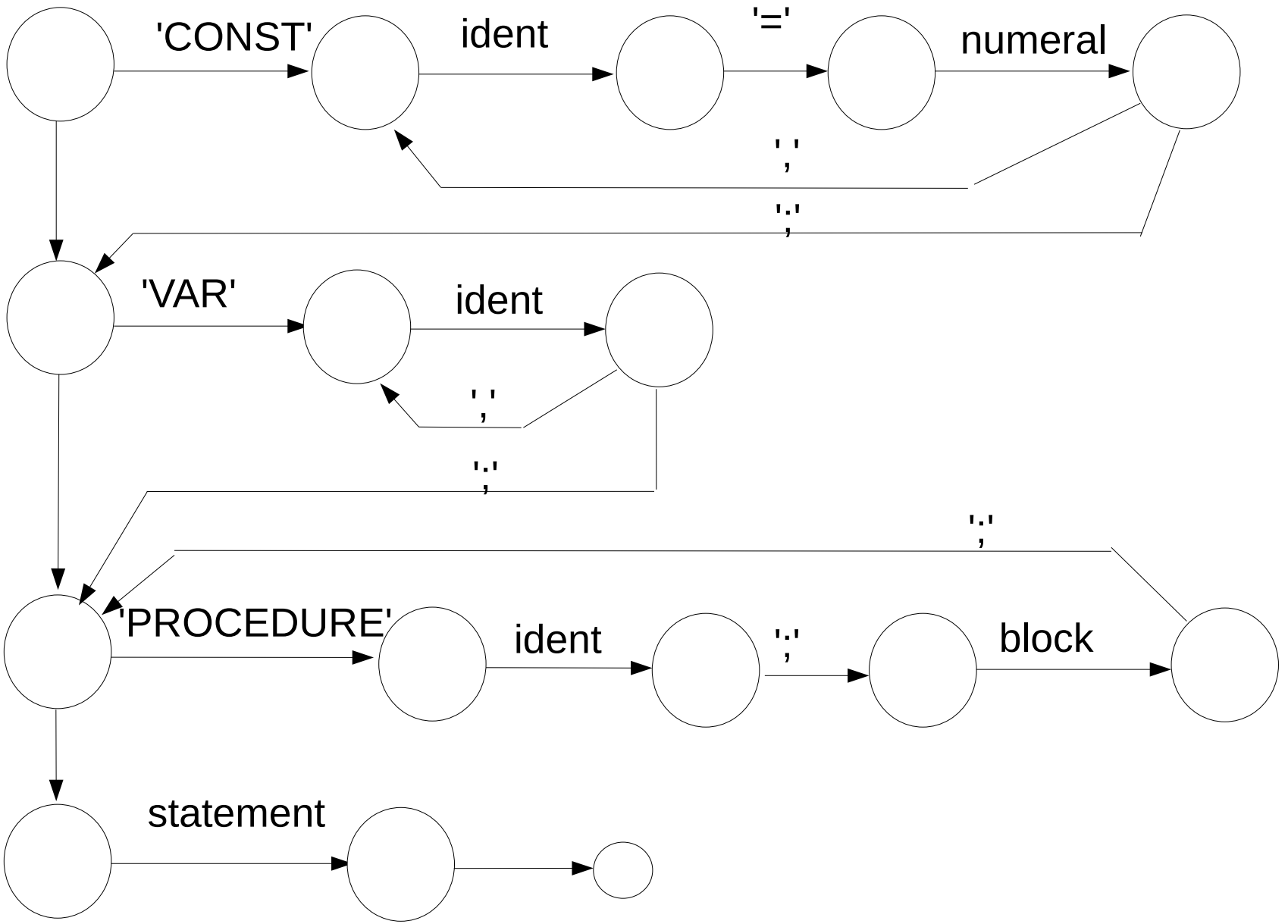
Zur Arbeit des Parsers stelle man sich vor, man geht einen Graphen vom Start bis zum Ende ab. Bei Verzweigungen orientiert man sich am aktuellen Eingabesymbol und wählt die dazu passende Alternative. Gibt es keine passende Alternative, so liegt ein Syntaxfehler in der Eingabe vor. Ist ein Teilstück des Weges (ein Bogen) mit einem Metasymbol (dem Namen eines anderen Graphen) markiert, so muss man erst einmal diesen abgehen, bevor man im aktuellen Graphen weiter gehen kann. Erreicht man das Ende des Graphen mit dem man angefangen hat, ist die Eingabe ein Satz der Sprache.

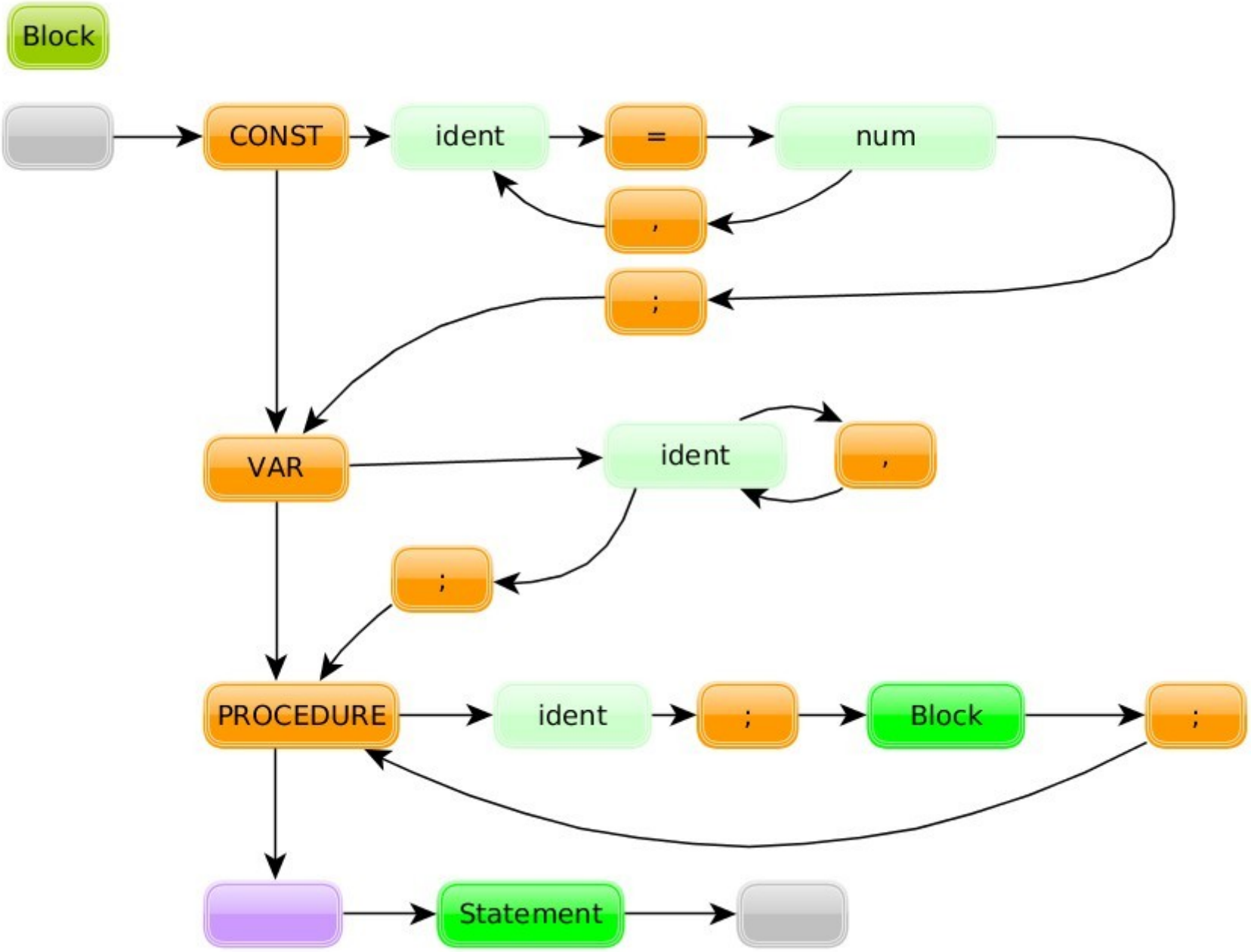
programm



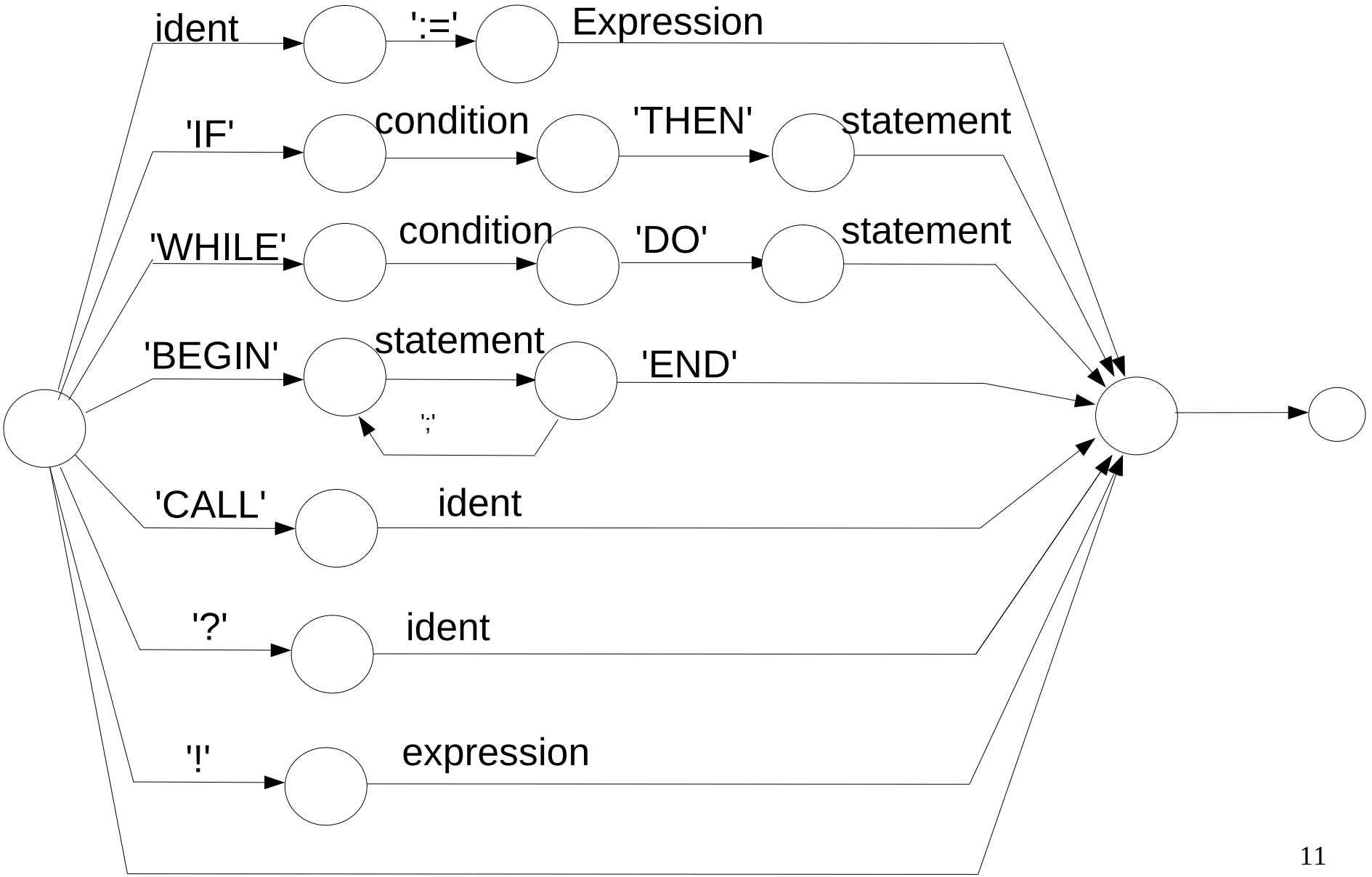


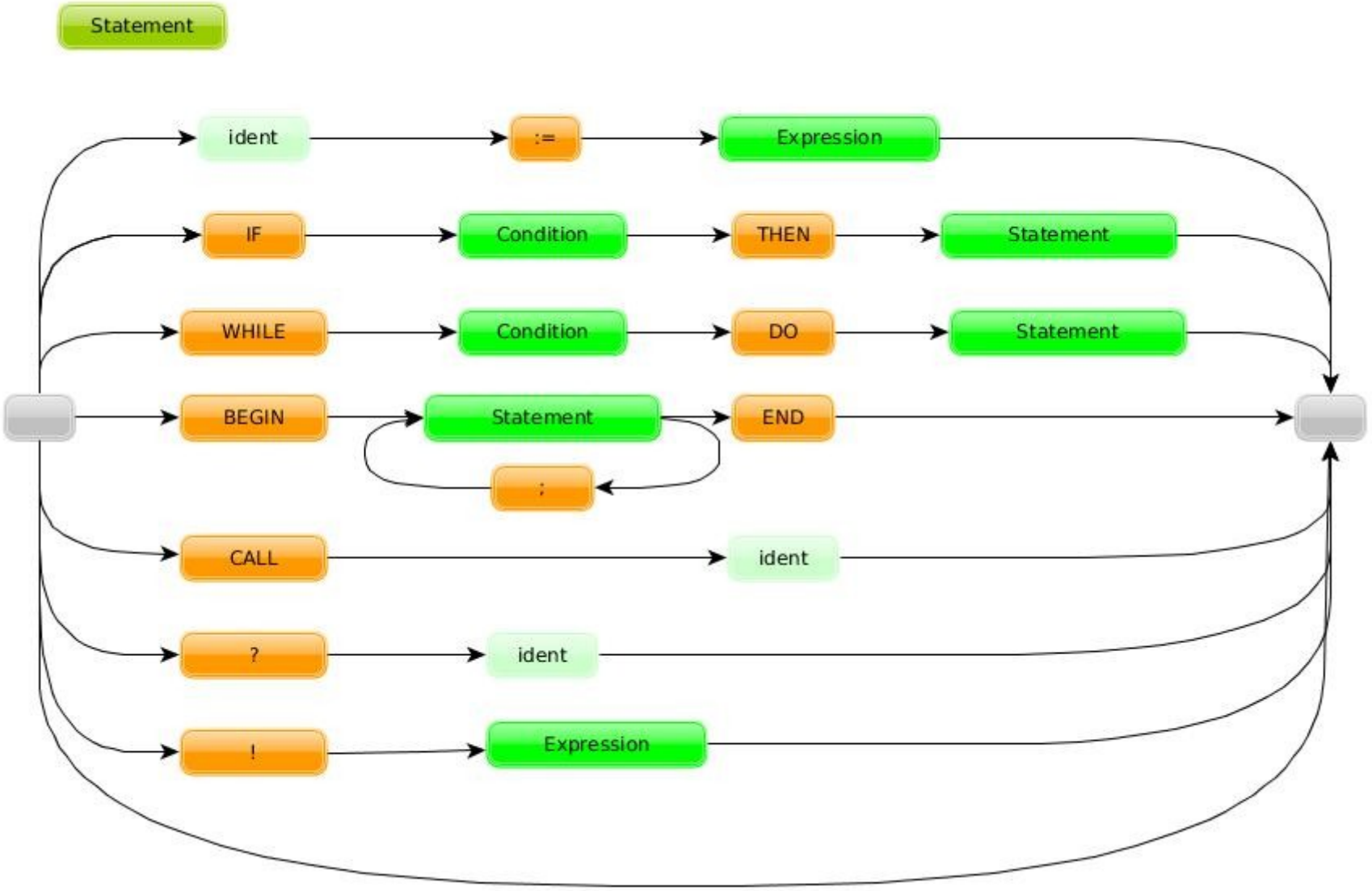
block


'CONST'
ident
'='
numeral
','
';'
'VAR'
ident
','
';'
';'
'PROCEDURE'
ident
':'
block
statement


Block
CONST
ident
=
num
,
;
VAR
ident
,
;
PROCEDURE
ident
;
Block
;
Statement

statement


ident
':='
Expression
'IF'
condition
'THEN'
statement
'WHILE'
condition
'DO'
statement
'BEGIN'
statement
'END'
';'
'CALL'
ident
'?'
ident
'!'
expression

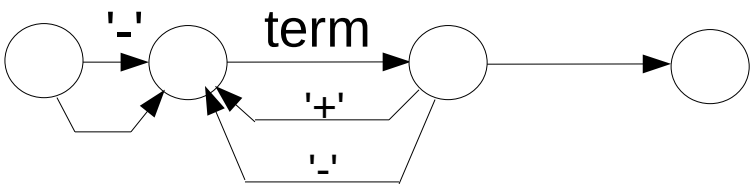

Statement
ident
:=
Expression
IF
Condition
THEN
Statement
WHILE
Condition
DO
Statement
BEGIN
Statement
END
;
CALL
ident
?
ident
!
Expression

Anmerkungen zu expr

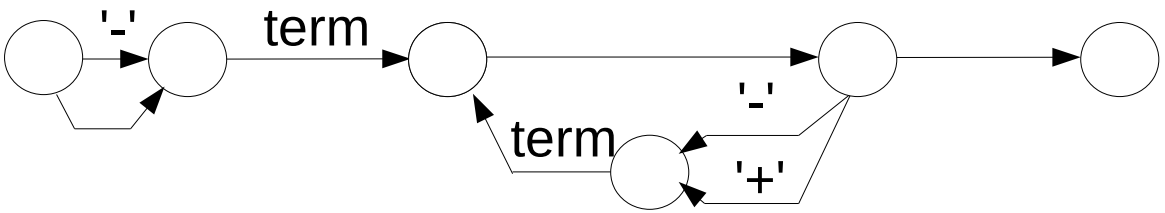
- Es gibt verschiedene Möglichkeiten die Syntax von expession abzubilden.
- Sehr einfach ist Variante 1. Sie ist vor allem sinnvoll, wenn es nur darum geht, die sytaktische Richtigkeit zu prüfen.
- Soll auch der Code generiert werden (typisch bei Einpasscompilern), so muss man sich den Operator (+/-) in einem Kellerspeicher merken, da die Operation erst nach dem Akzeptieren von term generiert wird.
- Hier bietet Variante 3 den Vorzug, dass man bei jedem Bogen term weiß, welche Operation vorangegengen ist.
- Es wird daher dringend empfohlen, Variante 3 zu implementieren.
- Selbiges gilt für den Graphen term.

expr

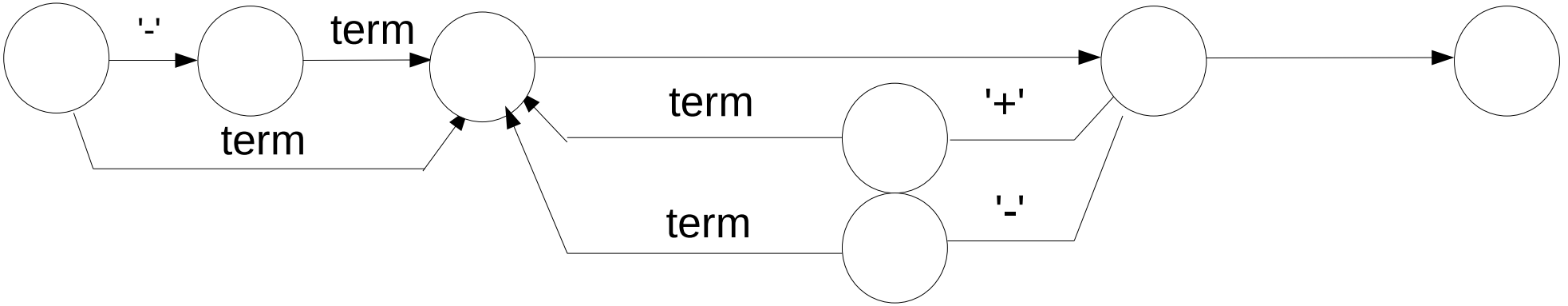
Variante 1



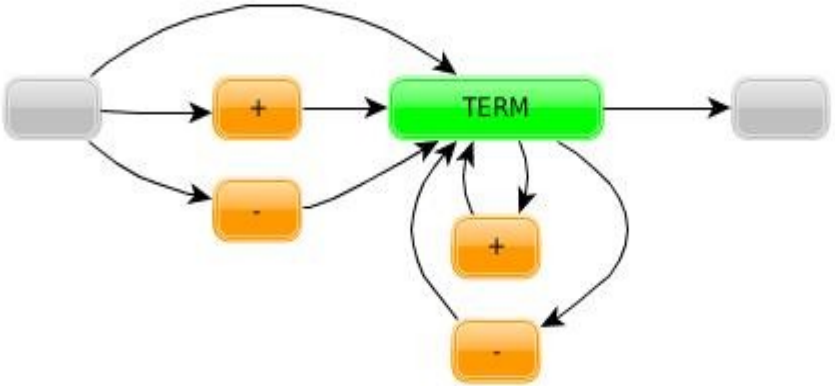
Variante 2



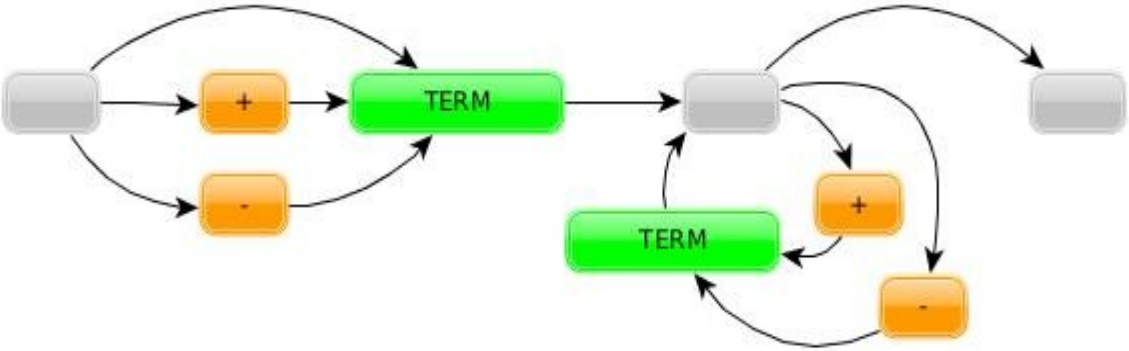
Variante 3



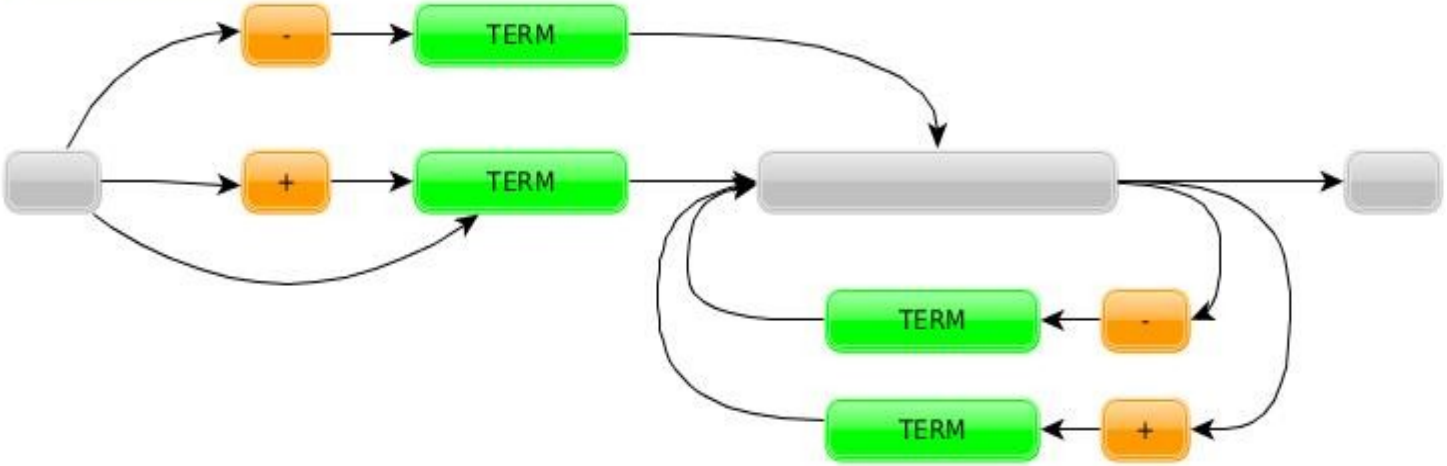
Expression I



Expression II

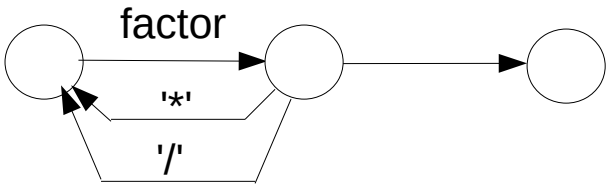


Expression III

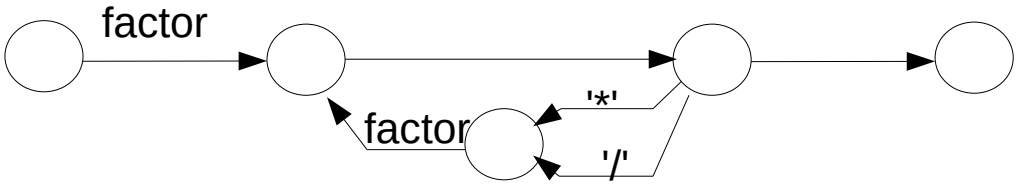


term

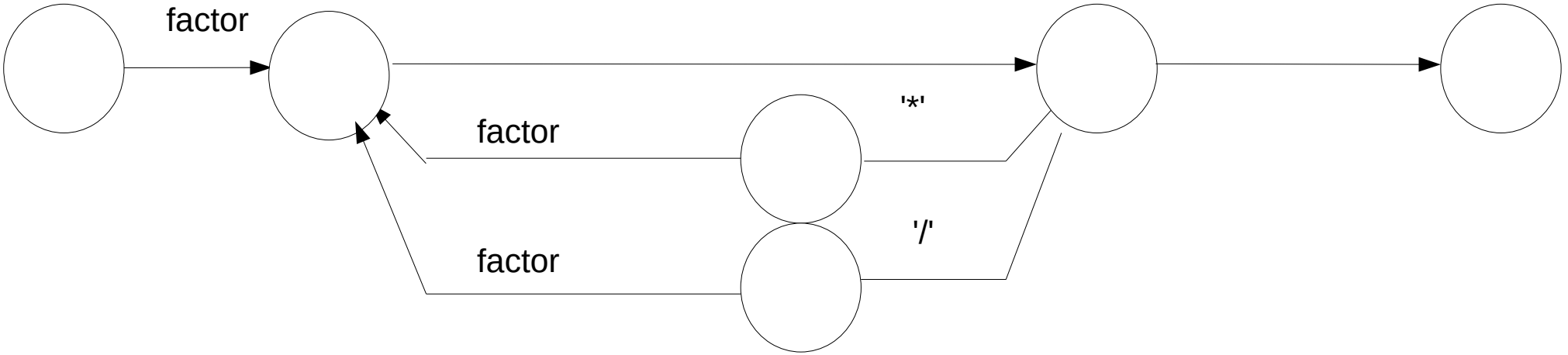
Variante 1



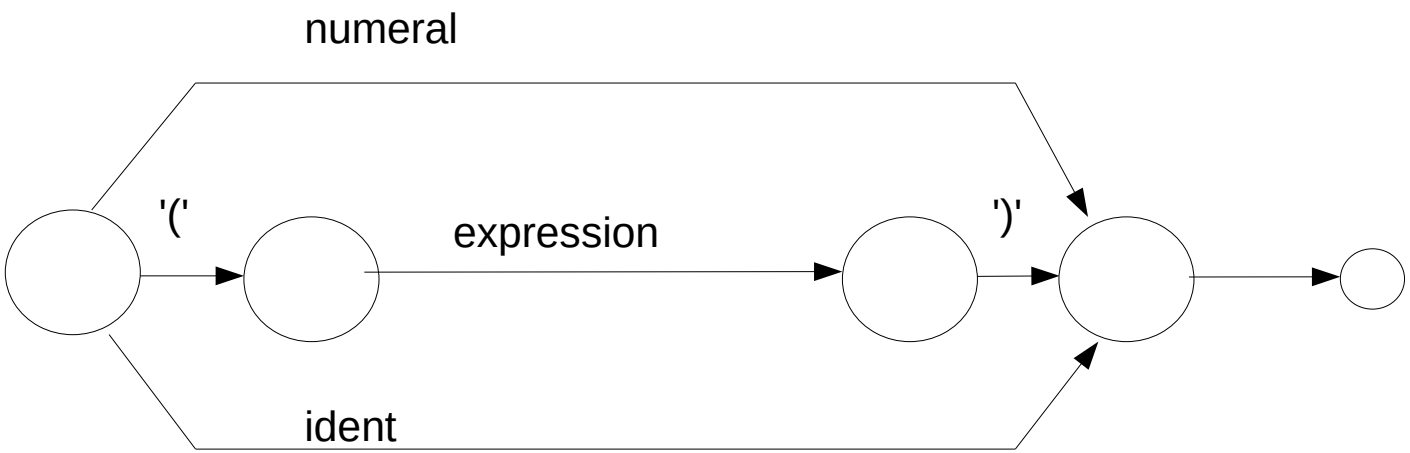
Variante 2

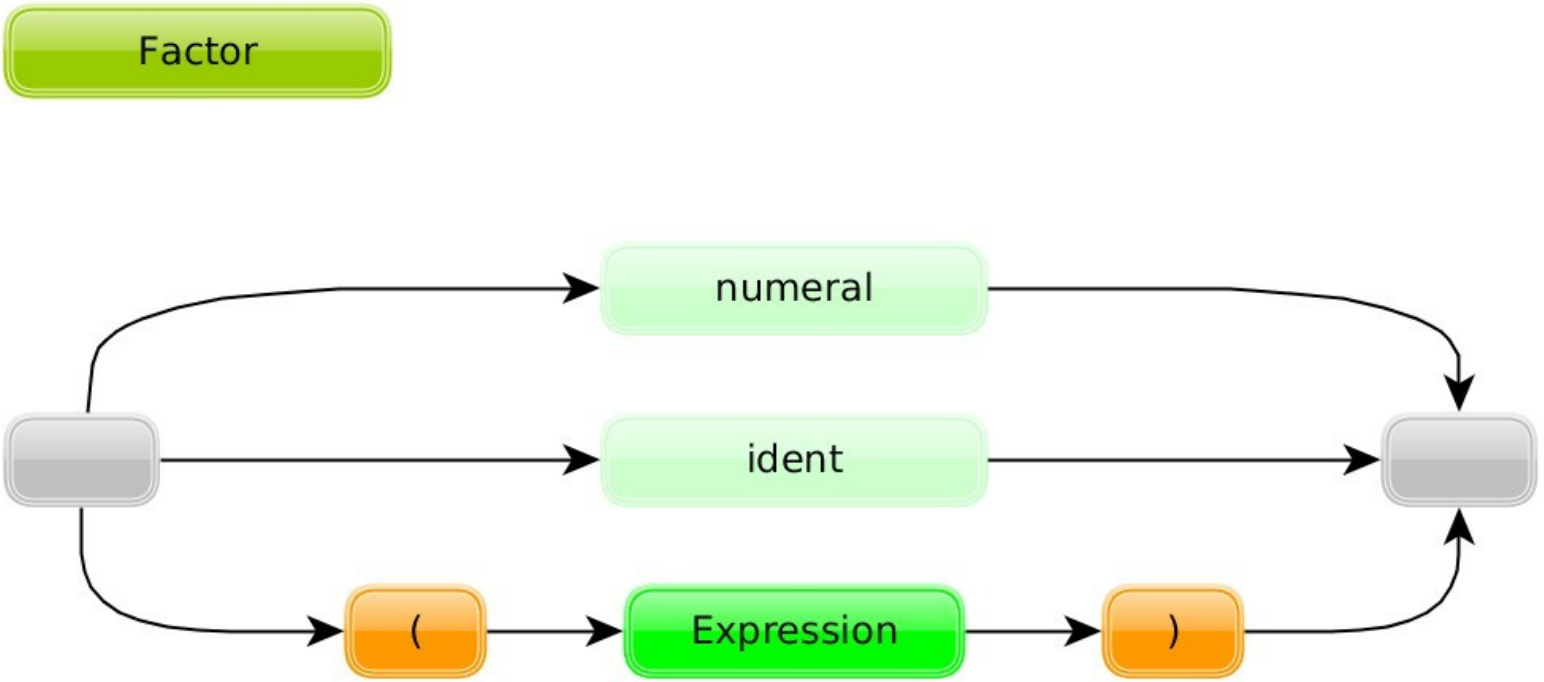


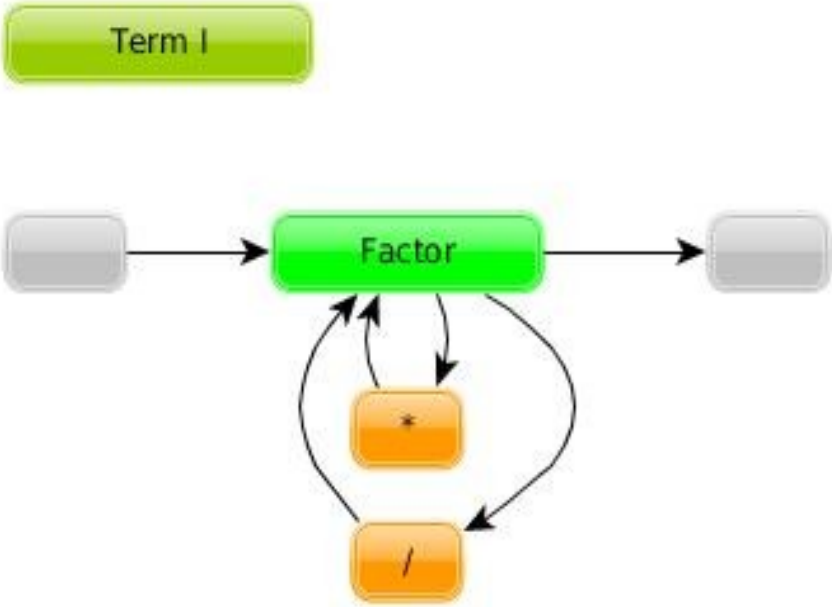
Variante 3

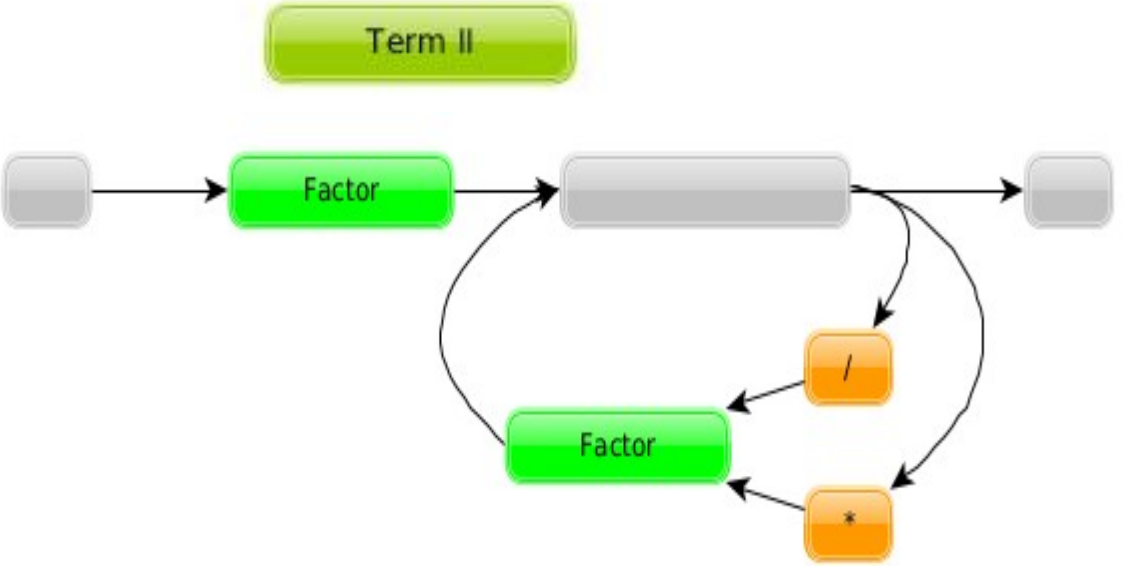


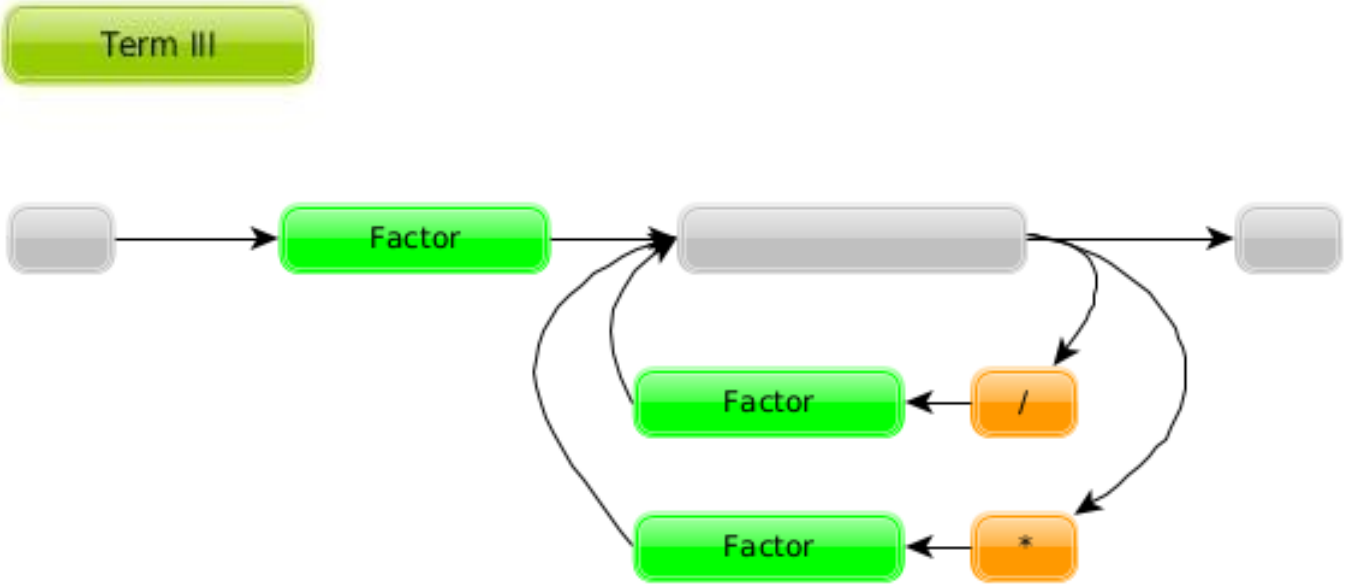
factor


numeral
'('
expression
')'
ident

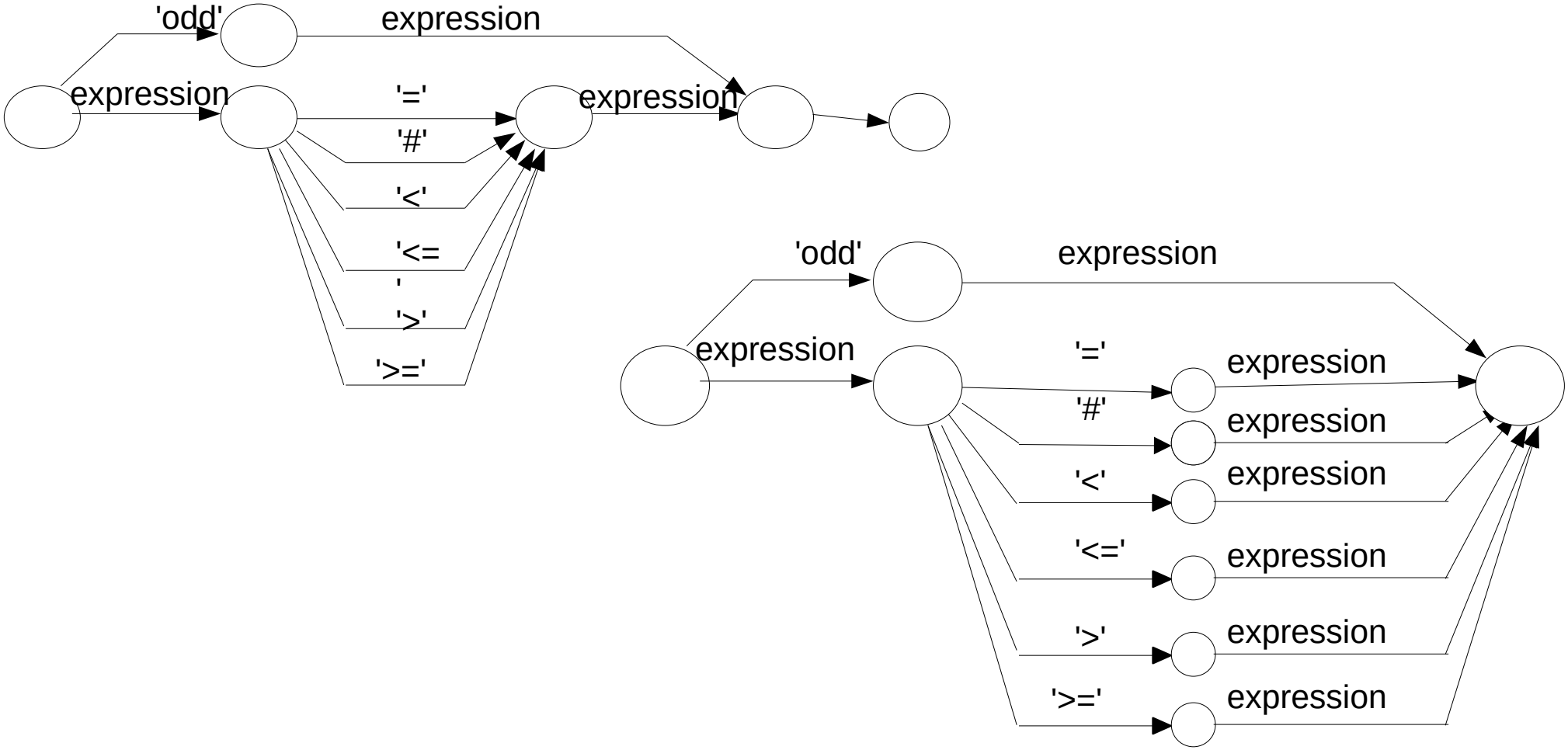

Factor
numeral
ident
(
Expression
)

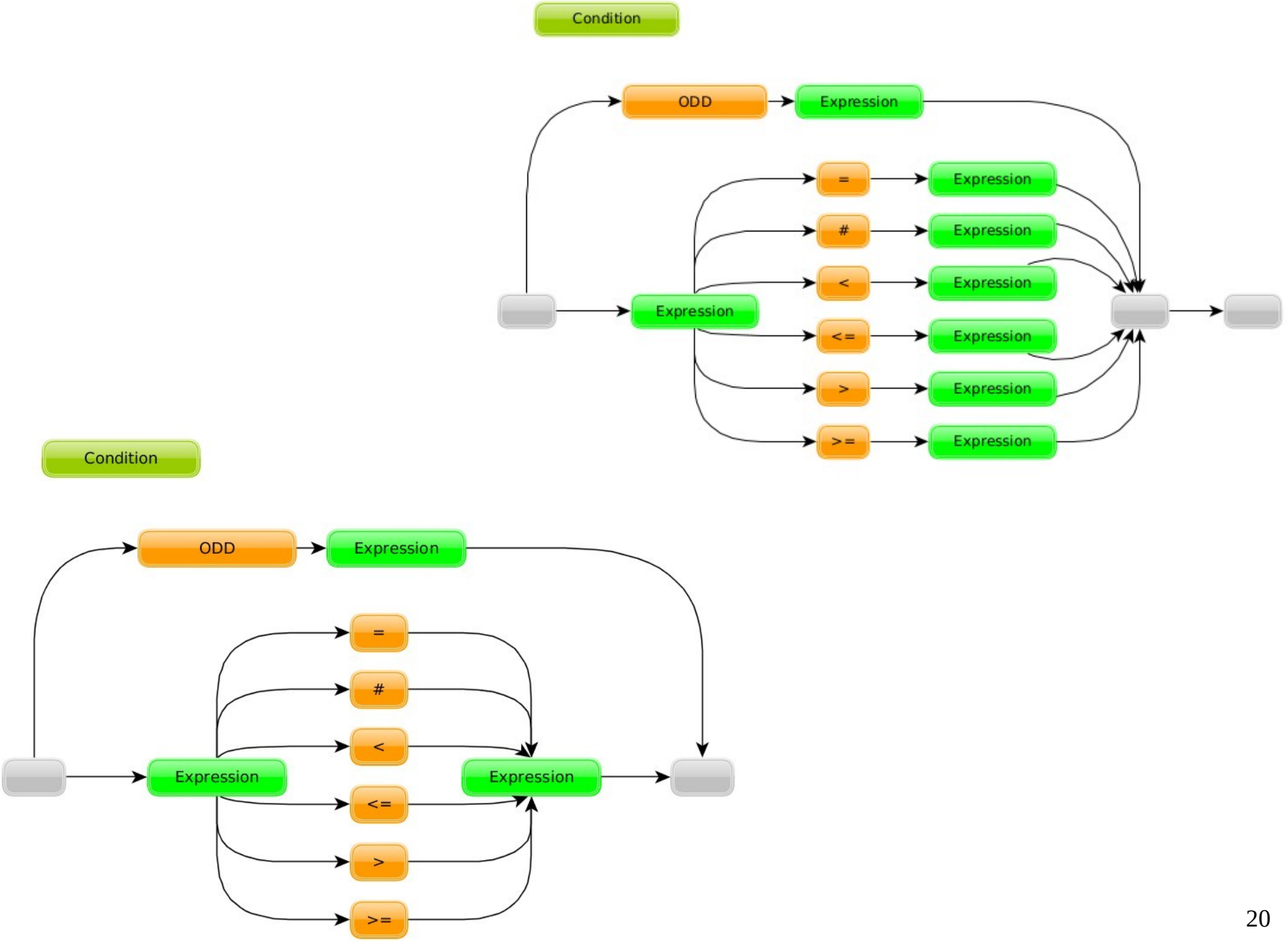

Term I
Factor
*
/


Term II
Factor
/
Factor
*


Term III
Factor
Factor
/
Factor
*

condition


'odd'
expression
expression
'='
expression
'#'
'<'
'<=
'
'>'
'>='
'odd'
expression
expression
'='
expression
'#'
expression
'<'
expression
'<='
expression
'>'
expression
'>='
expression


Condition
ODD
Expression
=
#
<
<=
>
>=
Expression
Expression
Expression
Expression
Expression
Expression
Expression
Condition
ODD
Expression
=
#
<
<=
>
>=
Expression
Expression

Implementation von Graphen

- Ein Graph kann als Array von Bogenbeschreibungen implementiert werden.
- Jede Bogenbeschreibung hat dabei ihren Index im Array und beinhaltet den Index des Folgebogens, sowie den Index eines Alternativbogens, falls vorhanden, wenn der Bogen nicht akzeptiert wird.
- Die Akzeptanz eines Bogens ist abhängig von der Bogenbewertung und dem aktuellen Morphem.
- Ein rekursiver Algorithmus pars durchläuft jeweils einen Bogen an Hand der Eingabe. Stößt er auf einen Graph-Bogen, so ruft er sich rekursiv mit dem neuen Bogen auf.

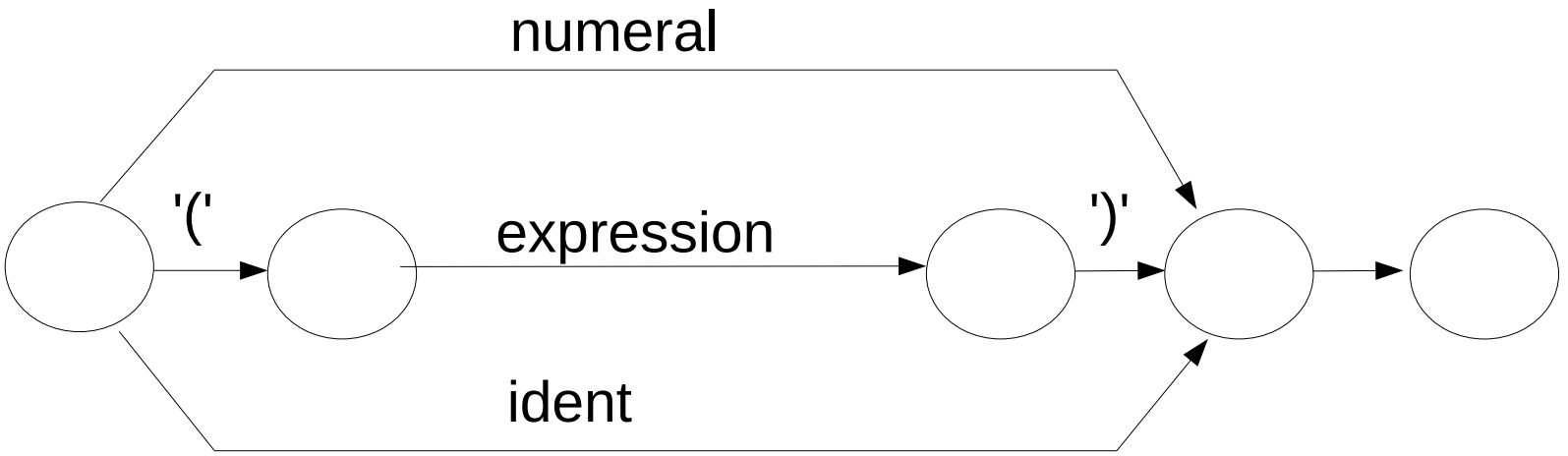
Struktur eines Bogens

```
typedef struct BOGEN
{
  tBg BgD;              // Bogentyp (Nil, Symbol, Name/Zahl, Graph)
  union BGX             // Bogenbeschreibung
  {
    unsigned long X;    // für Initialisierung notwendig
    int           S;    // Symbol (Ascii oder Wortsymbolcode aus enum)
    tMC           M;    // Morphemtyp (Zahl oder Bezeichner)
    struct BOGEN* G;    // Verweis auf Graph
  }   BgX;
  int (*fx)(void);      // Funktionspointer (Funktion, wenn Bogen akzeptiert)
  int iNext;            // Folgebogen, wenn Bogen akzeptiert
  int iAlt;             // Alternativbogen, wenn Bogen nicht akzeptiert
                        // oder 0 oder -1, wenn es keinen
                        // Alternativbogen gibt.
}tBog;
```

```
typedef enum BOGEN_DESC
          {BgNl= 0,  // NIL
           BgSy= 1,  // Symbol
           BgMo= 2,  // Morphem
           BgGr= 4,  // Graph
           //BgEn= 8,// Graphende (altenativ 0 oder -1 als Folgebogen)
           }tBg;
```

Beispiel Factor

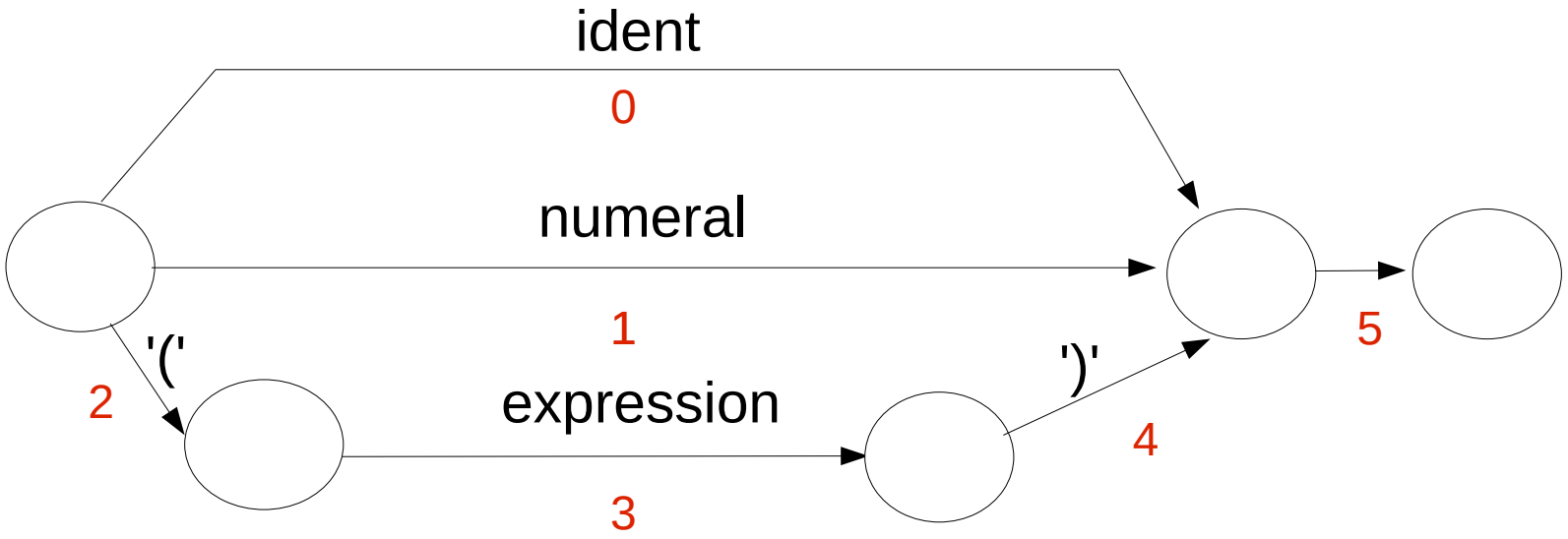
```
typedef unsigned long ul;

tBog gFact[]=
{
/* 0*/ {BgMo,{(ul)mcIdent}, NULL, 5, 1},/*(0)---ident--->(E)*/
/* 1*/ {BgMo,{(ul)mcNumb }, NULL, 5, 2},/* +---number--->(E)*/
/* 2*/ {BgSy,{(ul)'('    }, NULL, 3, 0},/*(+)----'('---->(3)*/
/* 3*/ {BgGr,{(ul)gExpr  }, NULL, 4, 0},/*(1)---express->(4)*/
/* 4*/ {BgSy,{(ul)')'    }, NULL, 5, 0},/*(0)----')'---->(E)*/
/* 5*/ {BgEn,{(ul)0      }, NULL, 0, 0} /*(E)--------(ENDE) */
};
```



Umsetzung in OOP

- In der objektorientierten Programmierung gibt es keinen union Typ.
- Man wird eine Klasse Bogen definieren und davon Klassen BogenNil, BogenSymbol usw. ableiten.
- Für die Funktion kann ein Objekt einer Klasse, die ein Interface implementiert, angelegt werden. Dabei enthält das Interface die Deklaration der Funktion. In Java bieten sich dann anonymous classes zur Implementierung der auszuführenden Funktion an (so ähnlich, wie ActionListener bei swing/AWT).
- Der Bogentyp kann entfallen, er ist durch die abgeleitete Klasse bereits festgelegt.

Hinweise zum Erstellen der Graphen



Graphen ausdrucken

Bögen mit 0 beginnend nummerieren

Bogenbeschreibungen zusammenstellen

Bogen 1 ist Altenativbogen zu Bogen 0 und Bogen 2 ist Altenativebogen zu Bogen 1

Aternativbogen 0 heißt, dass es keine Alternative gibt.

```
tBog gFact[]=
{
/* 0*/ {BgMo,{(ul)mcIdent}, NULL, 5, 1},
/* 1*/ {BgMo,{(ul)mcNumb }, NULL, 5, 2},
/* 2*/ {BgSy,{(ul)'('    }, NULL, 3, 0},
/* 3*/ {BgGr,{(ul)gExpr  }, NULL, 4, 0},
/* 4*/ {BgSy,{(ul)')'    }, NULL, 5, 0},
/* 5*/ {BgEn,{(ul)0      }, NULL, 0, 0}
};
```

Hinweise zur Umsetzung von Graphen

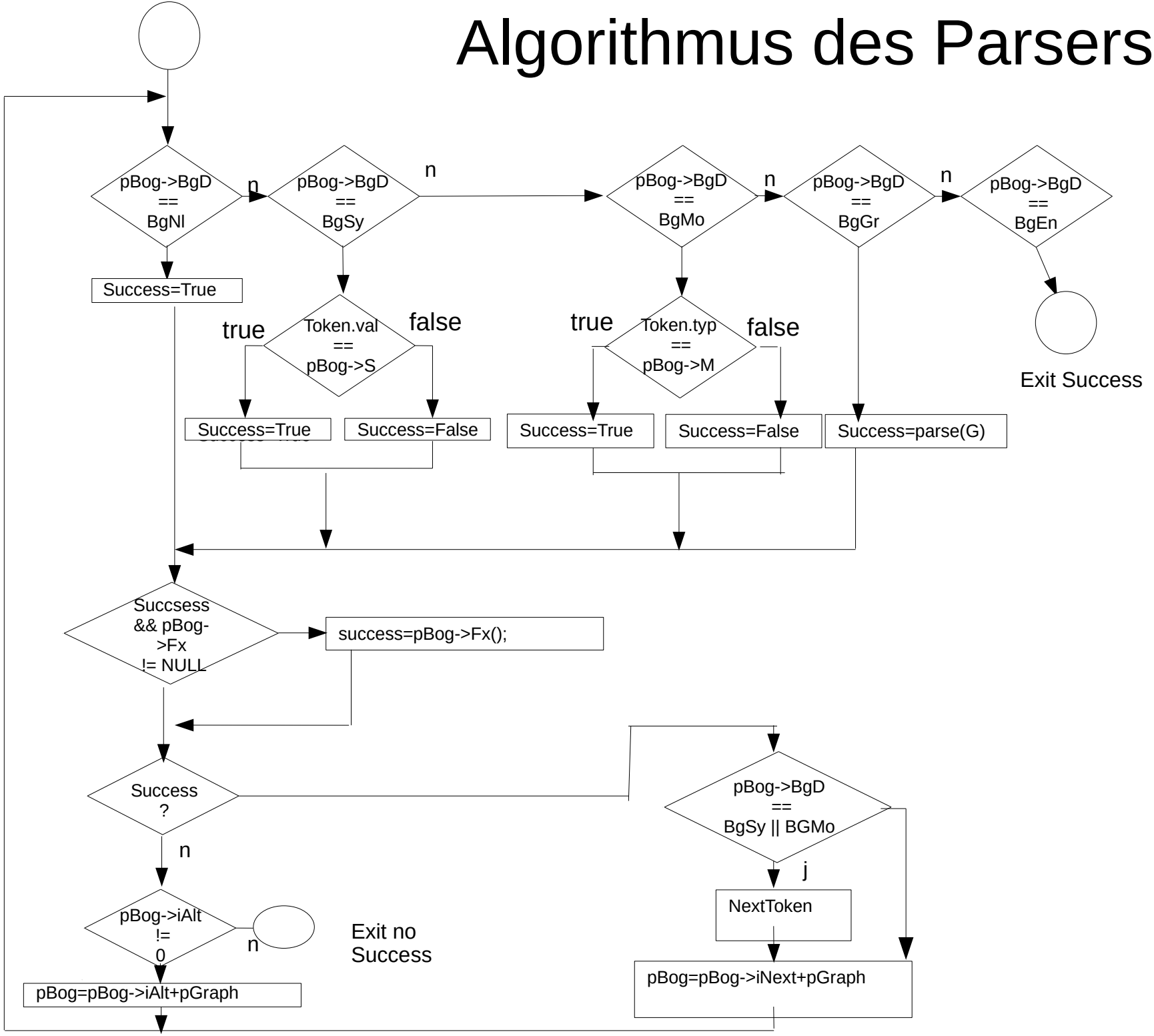
Bei alternativen Bögen sind zunächst die Symbol-und Morpohermbögen, die mit einem auszurufenden Graphen beweteten und **zuletzt die nil-Bögen** als **Alternative** angegeben werden.

Index als Kommentar mitführen

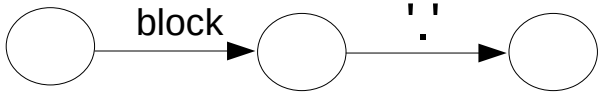
Wenn der Bogen mit dem Index 0 nirgends als Alternativbogen auftaucht, kann die Angabe von 0 als Index auf Alternativbogen als „keine Alternative vorhanden“ genutzt werden.

Andere Möglichkeiten das Vorhandensein von Alternativen zu steuern bestehen in dem Wert -1 für kein Alternativbogen, oder im Definieren eines Bit im Beschreibungsbyte des Bogens (Bogentyp)

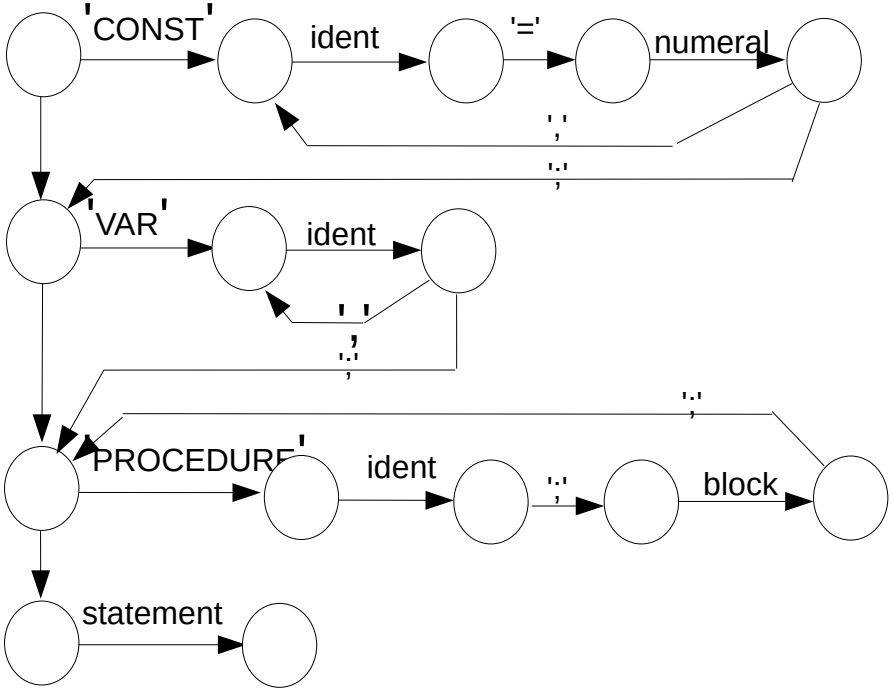
Algorithmus des Parsers



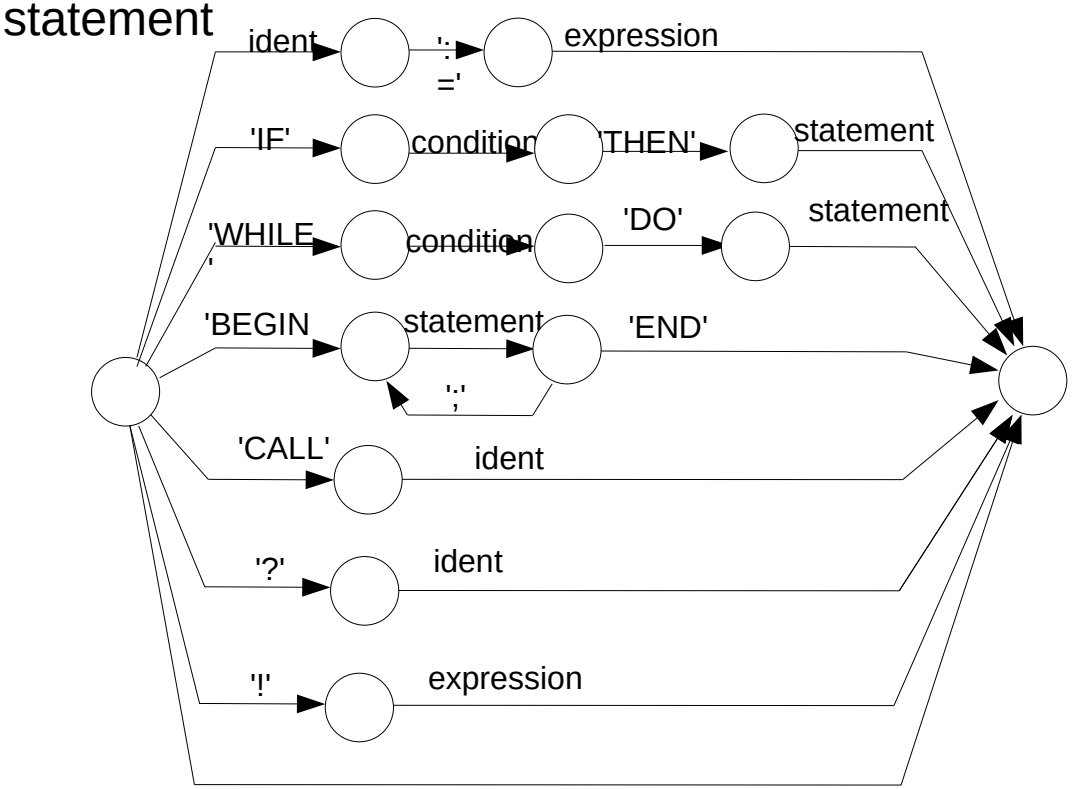
programm



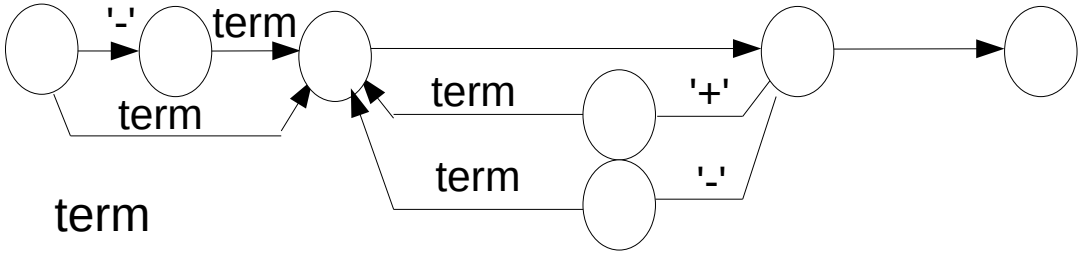
block



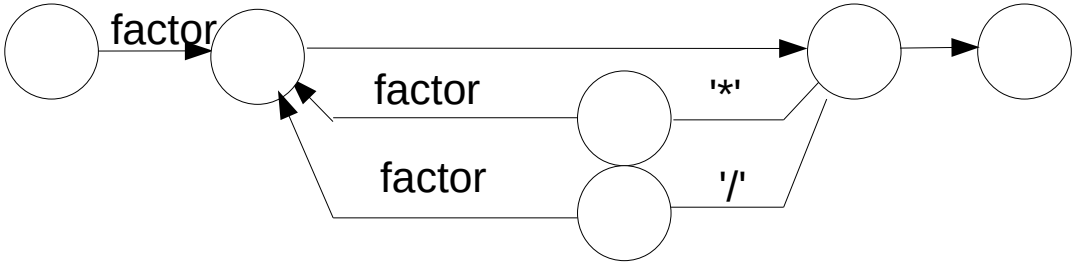
statement



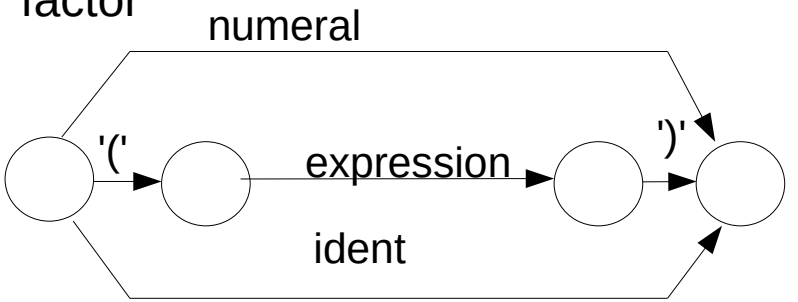
expression



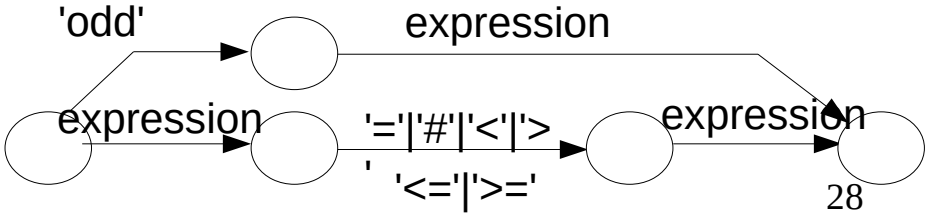
term



factor



condition



Die rekursive Funktion pars

```
int pars(tBog* pGraph)
{
  tBog* pBog=pGraph;
  int succ=0;
  if (Morph.MC==mcEmpty)Lex();
  while(1)
  {
    switch(pBog->BgD)
    {
      case BgNl:succ=1;                                  break;
      case BgSy:succ=(Morph.Val.Symb==pBog->BgX.S);break;
      case BgMo:succ=(Morph.MC==(tMC)pBog->BgX.M); break;
      case BgGr:succ=pars(pBog->BgX.G);                  break;
      case BgEn:return 1;   /* Ende erreichet - Erfolg */
    }
    if (succ && pBog->fx!=NULL) succ=pBog->fx();

    if (!succ)/* Alternativbogen probieren */
      if (pBog->iAlt != 0)
        pBog=pGraph+pBog->iAlt;
      else return FAIL;
    else /* Morphem formal akzeptiert (eaten) */
    {
      if (pBog->BgD & BgSy || pBog->BgD & BgMo) Lex();
      pBog=pGraph+pBog->iNext;
    }
  }/* while */
}
```

Die nachfolgenden Inhalte sind für die Implementation mit dem gewählten Verfahren nicht nötig. Die Erzeugung eines Parsbaumes fördert das Verständnis und wird deshalb hier gezeigt.

Generierung des Parsebaums

- Der Baum wird bei dem gewählten Verfahren nicht benötigt.
- Er gibt aber Aufschluss über die syntaktische Struktur von Programmen und fördert das Verständnis.

- Der Baum ergibt sich als Logbuch des Parsens.
- Der Baum besteht aus Listen, die jeweils ein Element für jeden akzeptierten Bogen enthalten.
- Jedes Listenelement beschreibt einen akzeptierten Bogen.
- Enthielt der Bogen ein Metasymbol (Graph), wird eine neue Liste für den aufgerufenen Graphen angelegt.
- Bei erfolgreicher Kompilierung erfolgt die Ausgabe des Baumes in XML.

Baumblatt/Erzeugung eines Baumblattes

```
typedef struct
{
  int    line;
  char   Type[10];
  tList* pList;
  char * Descr;
}TreeItem;


TreeItem* crItem(tList* pList,  // neue Liste od. NULL
                 char* Descr,   // Description
                 char* Type,    // Bogentyp
                 int line)      // Quellzeile
{
  TreeItem*ptmp=malloc(sizeof(TreeItem));
  strncpy(ptmp->Type,Type,9);
  ptmp->pList=pList;
  ptmp->Descr=Descr;
  ptmp->line=line;
  return ptmp;
}
```

Aufruf:

```
InsertHead(pTree,crItem((pl=CreateList()),"Programm","MetaSymb",0));
    if (pars(gProg,pl)==1)
```

Neue Liste für den Graphen

```
int pars(tBog* pGraph,tList* pList)
{
  int verarbMorph=0;
  tBog* pBog=pGraph;
  int succ;

   tList* pl=NULL;
   TreeItem *pItem=NULL;

  if (Morph.MC==mcEmpty)Lex();
  while(1)
  {

. . .
  }
}
```

Arrays der Graphen mit Pointern und Strings für den Baum. Diese Arrays können auch für das Debugging wertvoll sein, weil man ausgeben kann, welcher Graph aktuell bearbeitet/aufgerufen wird. Die Bogenbeschreibung enthält dann nur einen Index.

```
tBog* vGr[]={gFact,
             gTerm,
             gExpr,
             gCond,
             gStmnt,
             gBlock,
             gProg,
             gAssign,
             gCall,
             gBegin,
             gIf,
             gWhile,
             gInput,
             gOutput,
             gConstList,
             gConstDecl,
             gVarList,
             gVarDecl,
             gProcDecl};
```

```
char* StrGr[]={"Factor",
               "Term",
               "Expression",
               "Condition",
               "Statement",
               "Block",
               "Program",
               "AssignmentStatement",
               "PrCall",
               "CompoundStatement",
               "ConditionalStatement",
               "LoopStatement",
               "InputStatement",
               "OutputStatement",
               "ConstList",
               "ConstDecl",
               "VarList",
               "VarDecl",
               "ProcDecl"};
```

Hinweise zum Aufbau des Parsebaumes

- Die nachfolgenden Seiten enthalten Quelltextausschnitte, die aufzeigen sollen, wie der Parsebaum aufgebaut wird.
- Er wird zunächst aus Listen gebaut, wobei jedes Listenelement einen akzeptierten Bogen repräsentiert.
- Handelt es sich dabei um einen Bogen, der mit einem aufzurufenden Graphen bewertet ist, so enthält das Listenelement einen Listenkopf für eine neue Liste.
- Am Prgrammende wird der so entstandene Baum als XML-File exportiert.

```
switch(pBog->BgD & (BgNl+BgSy+BgMo+BgGr+BgEn))
{
  case BgNl:succ=1;
                                    break;
  case BgSy:succ=(Morph.Val.Symb==pBog->BgX.S);
          if (succ)
            InsertTail(pList,crItem(NULL,(crStr(Morph.vBuf)),"Symbol",Morph.PosLine));
                                    break;
  case BgMo:succ=(Morph.MC==(tMC)pBog->BgX.M);
          if (succ)
            InsertTail(pList,crItem(NULL,(crStr(Morph.vBuf)),"Morph",Morph.PosLine));
                                    break;
  case BgGr:
          pl=CreateList();
          pItem=crItem((pl),crStr(StrGr[pBog->BgX.G]),"MetaSymb",Morph.PosLine);
          InsertTail(pList,pItem);
          succ=pars(vGr[pBog->BgX.G],pl);
          if (succ!=OK)
          {DeleteList(pl); free(pItem->Descr);free(pItem); GetLast(pList); RemoveItem(pList);}
                                    break;
  case BgEn:return 1;   /* Ende erreichet - Erfolg */
                                    break;
}
```

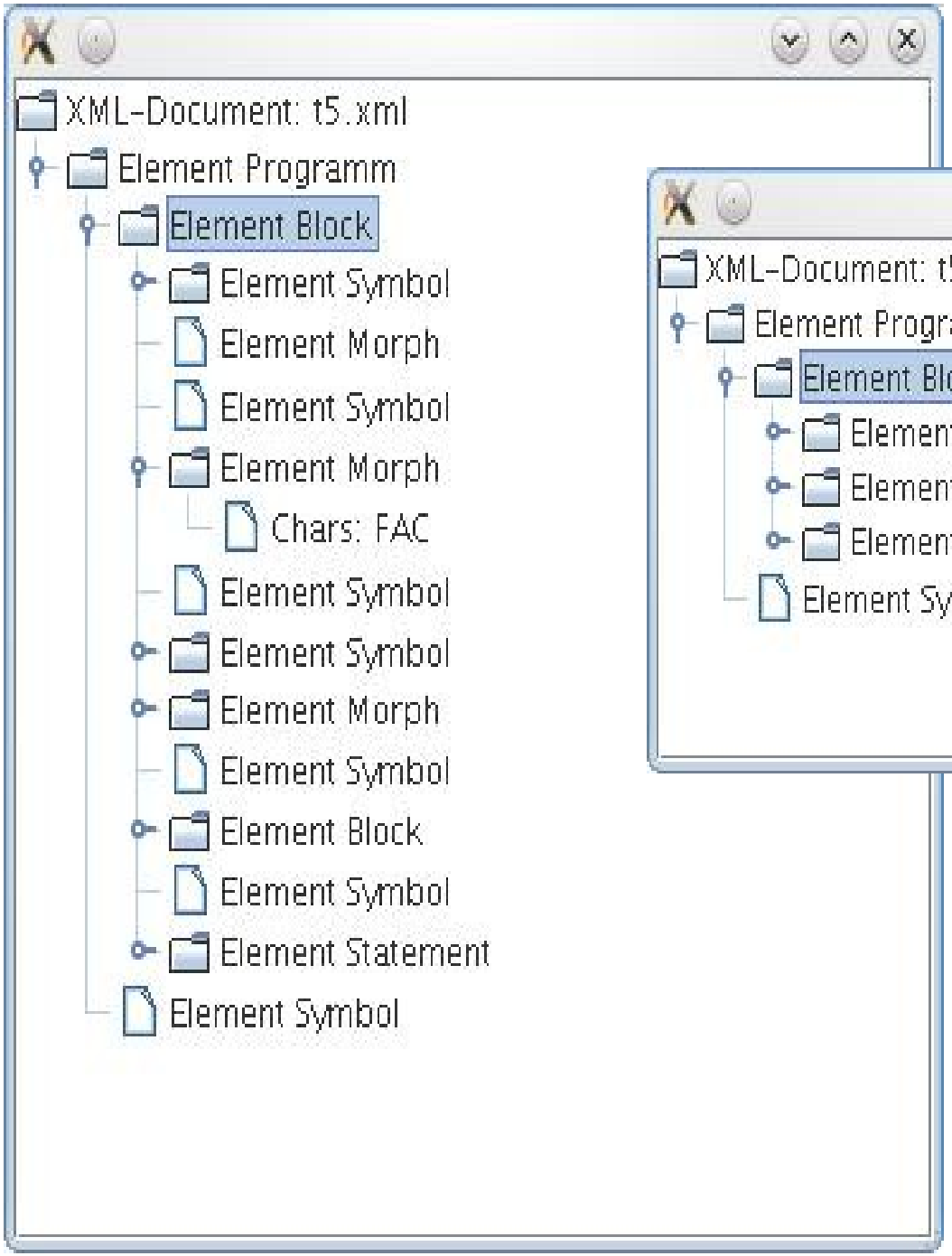
```
char* crStr(char* pstr)
{
  char* ptmp=(char*)malloc(strlen(pstr)+1);
  strcpy(ptmp,pstr);
  return ptmp;
}
```

```
if (succ && pBog->fx!=NULL) succ=pBog->fx();

if (!succ)
/* Alternativbogen probieren */
{
  if (pBog->iAlt != 0)
  {
    pBog=pGraph+pBog->iAlt;
  }
  /* Graph nicht erfolgreich verlassen                    */
  else if (verarbMorph) return ERROR; /* Syntaxfehler                  */
       else
       {
           //printf("go back\n");
           while(pItem=GetFirst(pList)){free(pItem->Descr);free(pItem); RemoveItem(pList);}
           return FAIL;  /* vielleicht gibt es noch Alternat. */
       }
}
else /* Morphem formal akzeptiert */
{
  if (pBog->BgD & BgSy || pBog->BgD & BgMo)
  {
    verarbMorph=1;
   Lex();
  }
  pBog=pGraph+pBog->iNext;
}
```

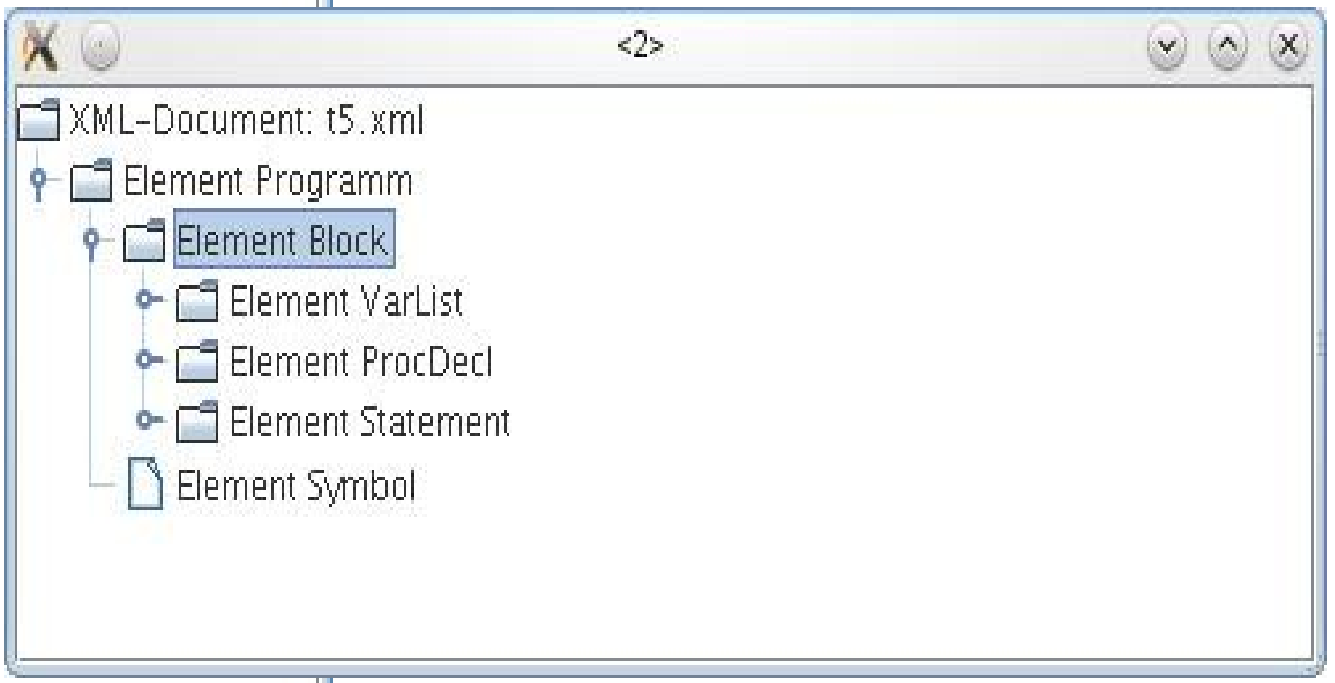
Vorzüge / Nachteile

- Die Graphen sind ein direktes Abbild der Regeln der EBNF nach N. With.
- Alternativen werden sofort am Beginn des Grafen sackgassenfrei erkannt (schnelles parsen).
- Protokolliert man die akzeptierten Bögen, so ergibt sich der Parsebaum.
- Der Parsebaum dient hier nur zur Visualisierung und zum besseren Verständnis. Er wird für das vorgestellte Verfahren nicht benötigt.
- Der entstehende Parsebaum hat keine schöne Struktur, wird aber für das angewandte Verfahren nicht benötigt. Dennoch sollen Betrachtungen angestellt werden, einen schöneren Baum zu erhalten.

Variante 1

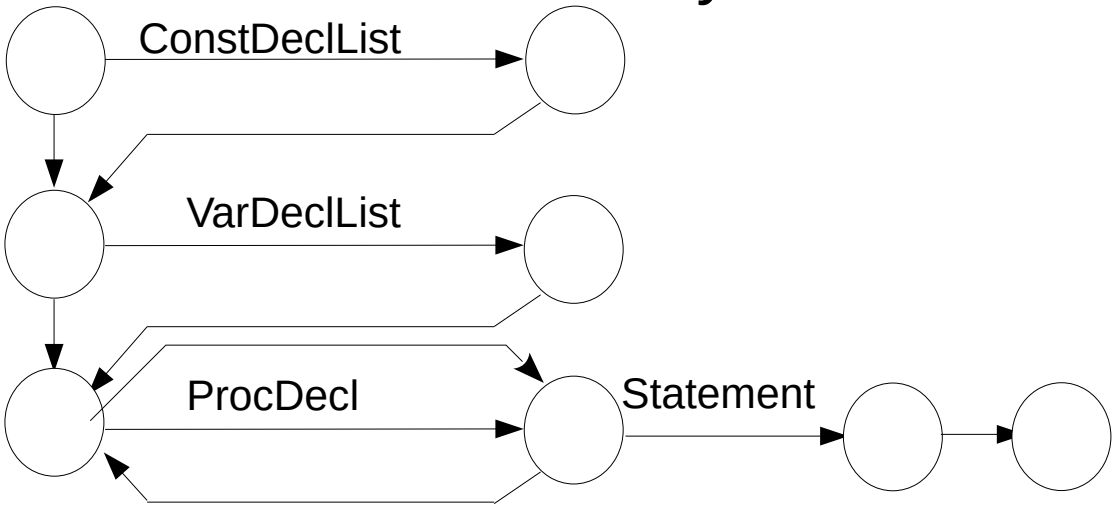


Variante 2

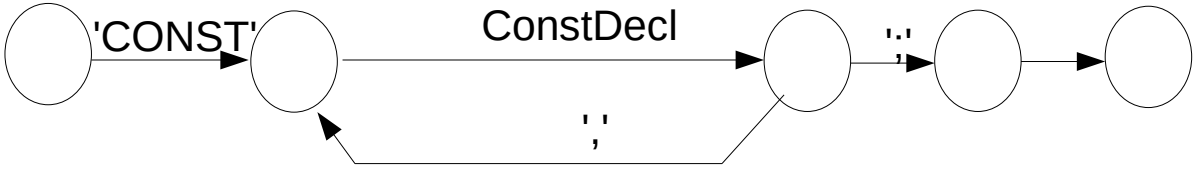


Block

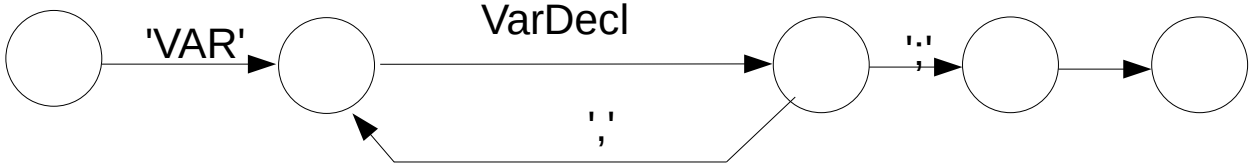
alternative Syntax für bessere Baumstruktur



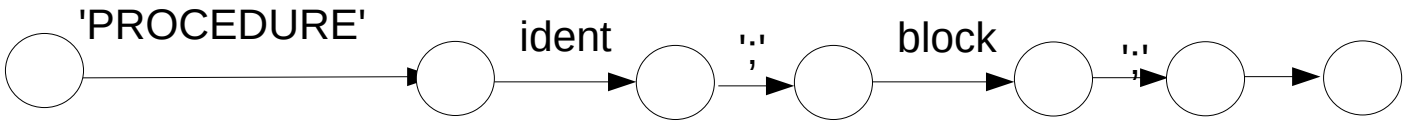
ConstDeclList::



VarDeclList::



ProcDecl::

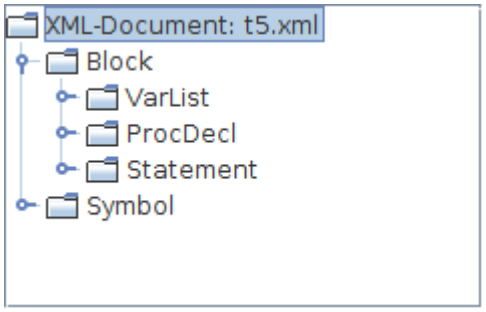


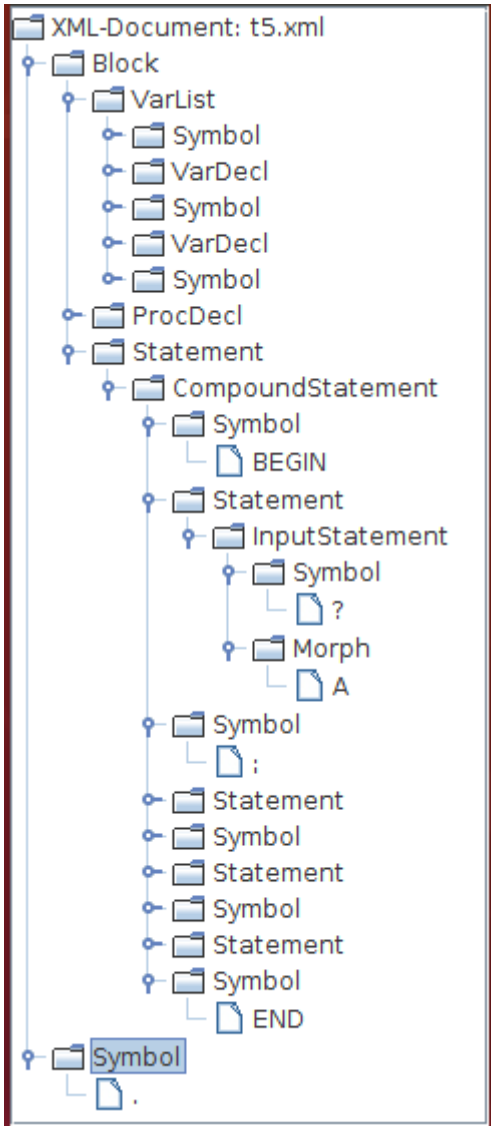
In dieser Variante gibt es mehr Metasymbole. Dadurch wird die syntaktische Struktur eines Quelltextes sehr viel besser abgebildet.

Vergleicht man auch diese Variante wieder mit dem Durchlaufen der Bögen, so kommt man hier an Alternativen, die mit Metasymbolen bewertet sind. Man geht also zunächst in den angegebenen Graphen und stellt erst dort fest, ob es der passende Weg ist. Wenn das nicht der Fall ist und im aktuellen Bogen noch kein Eingabsymbol verarbeitet worden ist, geht man zur Aufrufstelle zurück und sucht dort nach einer Alternative (Backtracking). Das ist aufwändiger und langsamer, aber der erzeugte Baum ist schöner strukturiert.

Vorzüge / Nachteile

- Es entsteht ein sehr gut lesbarer Syntaxbaum
- Alternativen werden nicht sofort erkannt
- Es entstehen Sackgassen
- Wird eine Sackgasse erkannt, bevor ein Token verarbeitet wurde, ist ein Backtracking bis zur nächsten Alternative möglich. Gelangt die Analyse in eine Sackgasse nachdem ein Token in dem aktuellen Graphen verarbeitet worden ist, so liegt ein Syntaxfehler vor.

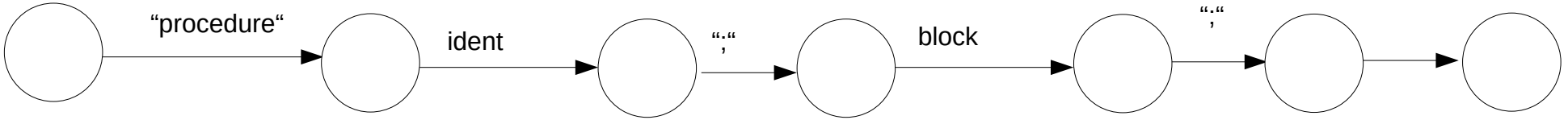




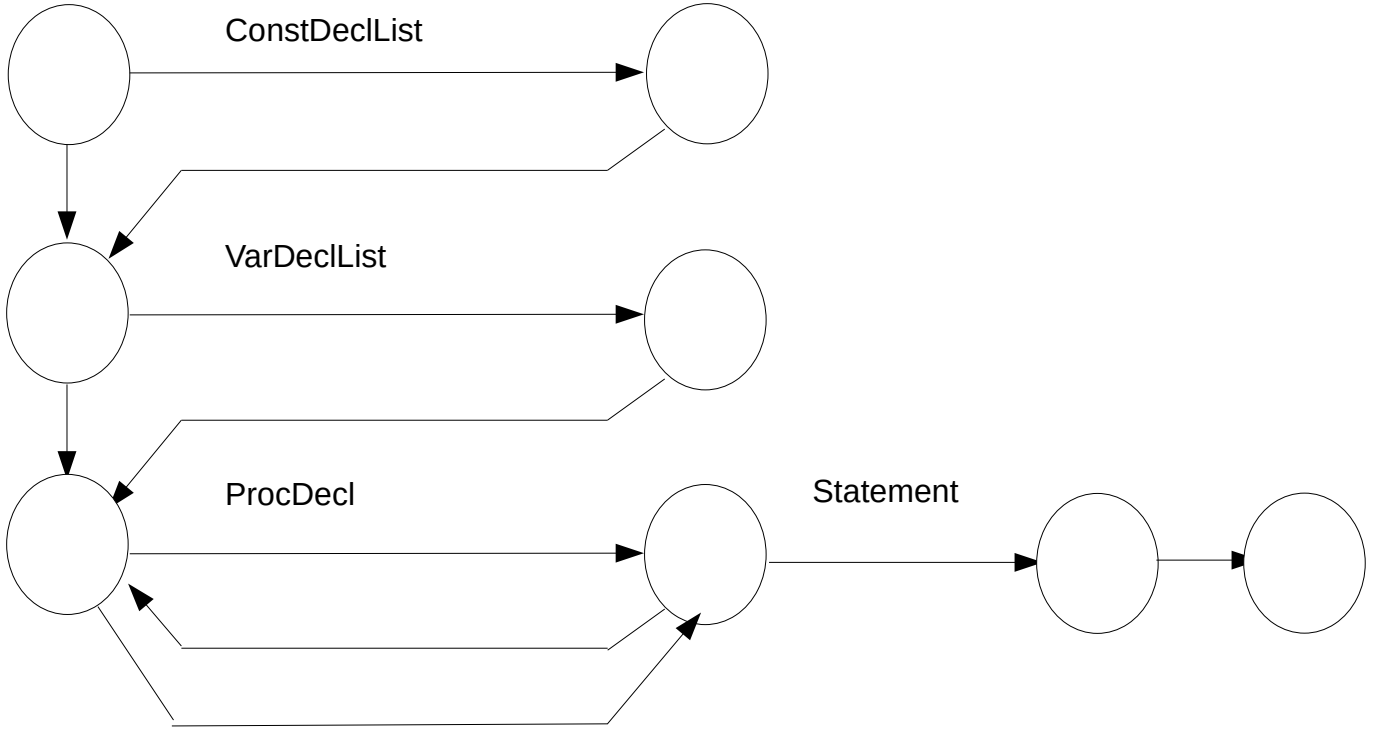
Der hier teilweise aufgeklappte Baum enthält viel mehr Metasymbole als Zwischensymbole, wie CompoundStatement, InputStatement usw.. Dadurch ergibt sich eine sehr viel systematischere Stuktur.

Für das vorgestellte Verfahren spielt das aber keine Rolle. Der Baum wird hier nur zur Visualisierung zum besseren Verständnis erzeugt.

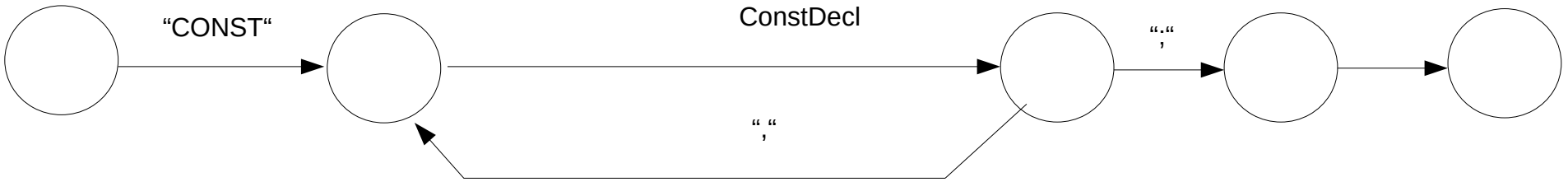
ProcDecl::



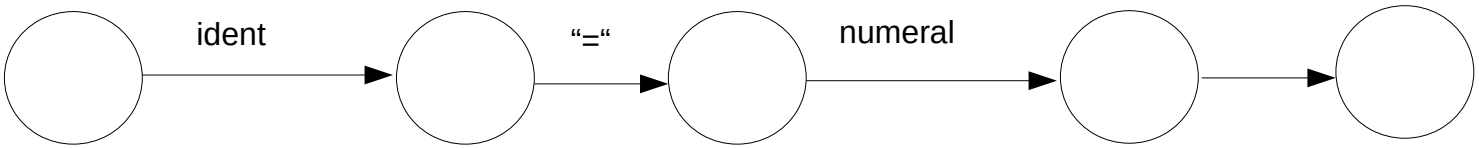
Block::



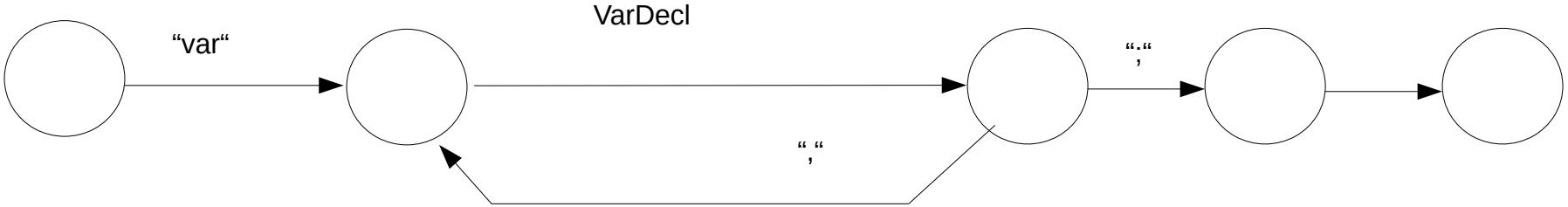
ConstDeclList::



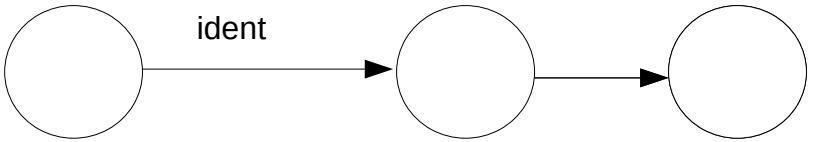
ConstDecl::



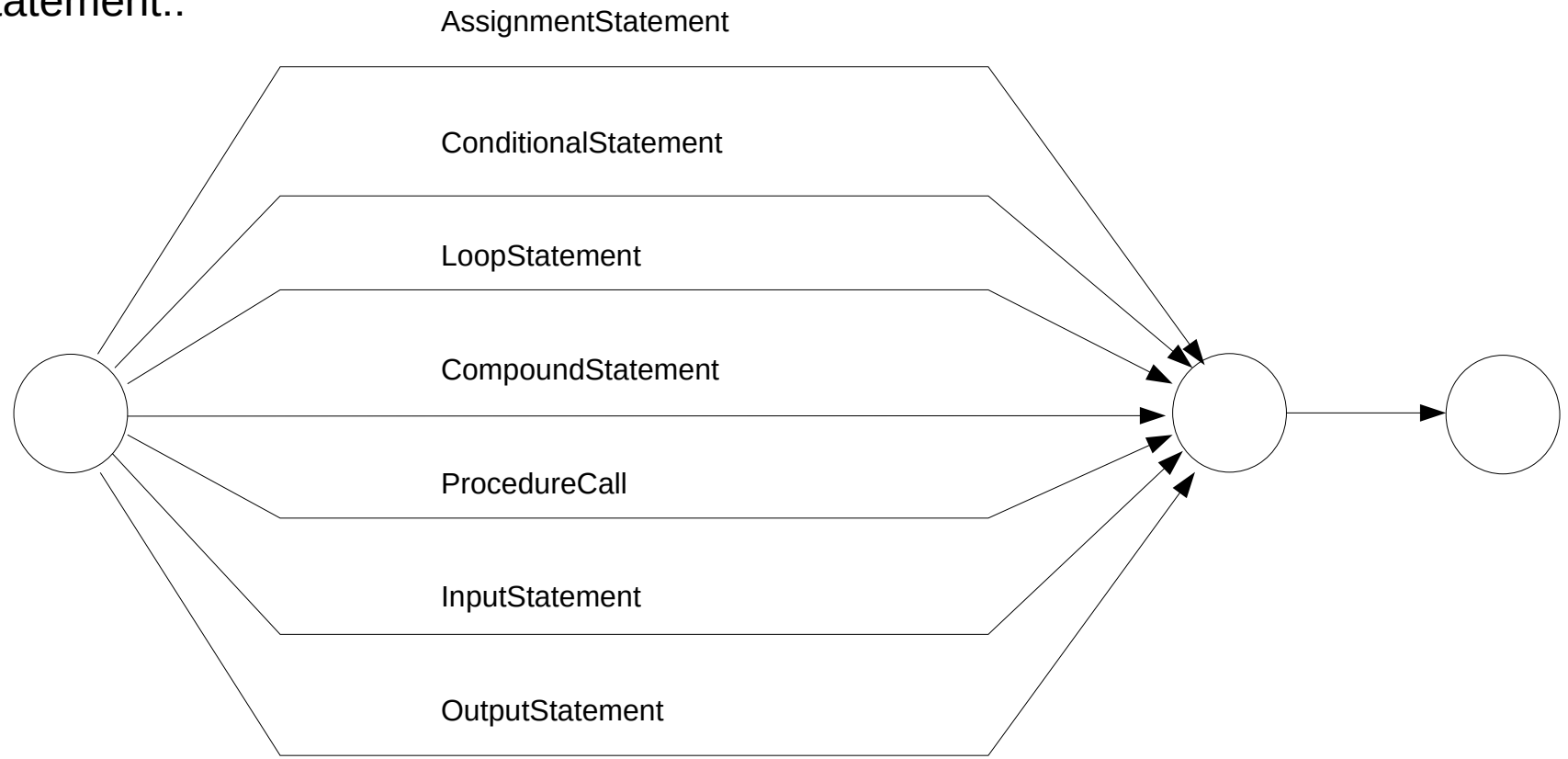
VarDeclList::



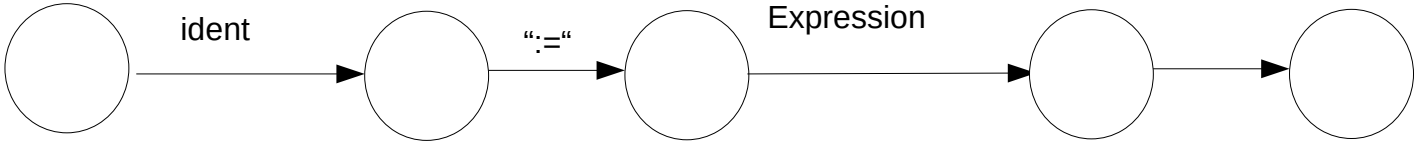
VarDecl::



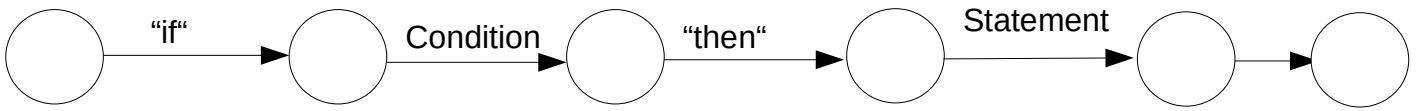
Statement::



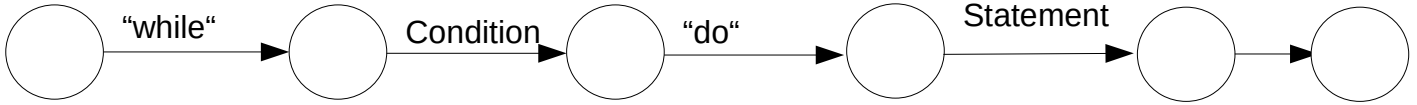
AssignmentStmt::



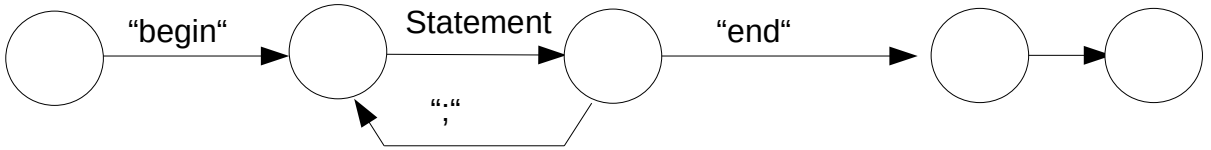
ConditionalStmnt::



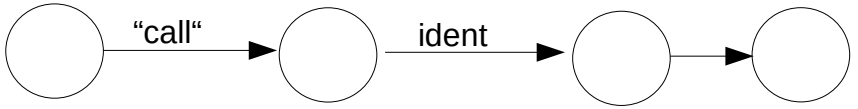
LoopStmnt::



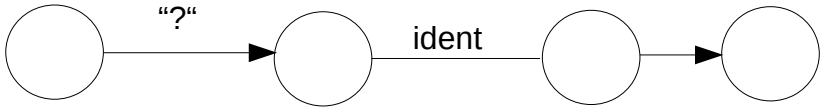
CompoundStmnt::



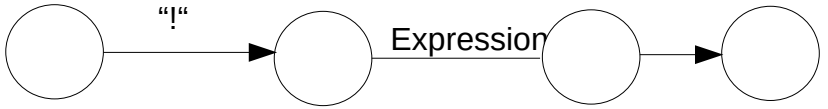
ProcedureCall::



InputStmnt::



OutputStmnt::



Auszug aus den implementierten geänderten Graphen

```
tBog gStmnt[]=
{
/* 0*/ {BgGr,{(unsigned long)iAssign}, NULL, 7, 1},
/* 1*/ {BgGr,{(unsigned long)iCall  }, NULL, 7, 2},
/* 2*/ {BgGr,{(unsigned long)iBegin }, NULL, 7, 3},
/* 3*/ {BgGr,{(unsigned long)iIf    }, NULL, 7, 4},
/* 4*/ {BgGr,{(unsigned long)iWhile }, St4 , 7, 5},
/* 5*/ {BgGr,{(unsigned long)iInput }, NULL, 7, 6},
/* 6*/ {BgGr,{(unsigned long)iOutput}, NULL, 7, 0},
/* 7*/ {BgEn,{(unsigned long)0      }, NULL, 0, 0}
};

tBog gAssign[]=
{
/* 0*/ {BgMo,{(unsigned long)mcIdent}, St0 , 1, 0},
/* 1*/ {BgSy,{(unsigned long)zErg   }, NULL, 2, 0},
/* 2*/ {BgGr,{(unsigned long)iExpr  }, St8 , 3, 0},
/* 3*/ {BgEn,{(unsigned long)0      }, NULL, 0, 0}
};
```