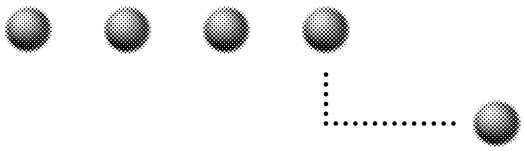


**Berner Fachhochschule**

Hochschule für
Technik und Architektur Bern

# PARSING METHODEN

**Dr. J. Boillat**

# 1 Formale Sprachen

## 1.1 Motivation

Die Verwendung von formalen Sprachen ermöglicht eine präzise und einfache Syntaxdefinition, ferner sind Syntaxanalysealgorithmen aus der Grammatik ableitbar und die Strukturierung eines Programms gemäss Ableitungsbaum vereinfacht die Codierung.

In der Praxis wird eine formale Grammatik ähnlich wie reguläre Definitionen aufgebaut. Dabei ist aber Rekursion erlaubt.

## 1.2 Theoretische Konzepte

**Definition 1.1 (Alphabet)** *Ein **Alphabet** $A$ ist eine endliche Menge von Zeichen. Das leere Zeichen $\epsilon$ gehört immer zum Alphabet.*

Für $\epsilon$ wird auch das Literal `""` verwendet.

**Definition 1.2 (Grammatik)** *Eine **Grammatik** $G$ ist ein 4-Tupel $G = (T, N, P, s)$ , wobei*

- $T$ *die Menge der **Terminalsymbole**,*
- $N$ *die Menge der **Nichtterminale**,*
- $P$ *die Menge von **Produktionen** der Form $v ::= w$ ; wobei $v$ und $w$ Strings über $V$ sind. Mit Hilfe der Produktionen wird ein **Syntaxbaum** entwickelt wird. Jede Produktion besteht aus einer linken und rechten Seite. Die rechte Seite einer Produktion wird aus dem String auf der linken Seite hergeleitet.*
- *$s$ ist ein Nichtterminalsymbol und bezeichnet das **Startsymbol** der Grammatik, aus dem jeder Satz in einer Sprache hergeleitet werden kann.*

**Notation 1.1 (Wörter)** *$V$ ist die Vereinigungsmenge der terminalen ( $T$ ) und nichtterminalen ( $N$ ) Symbole.*

$V^*$ bezeichnet die Menge der Strings über $V$ .

$V^0$ hat nur ein Element, den leeren String, der mit $\epsilon$ oder `""` bezeichnet wird.

Terminalsymbole werden entweder mit Grossbuchstaben oder mit Grossbuchstaben in spitzen Klammern (e.g. Token `<PLUS>`) odee direkt mit dem Entsprechenden Lexem (e.g. Literal `"+"`) dargestellt. Nichtterminalsymbole werden wie übliche Java Bezeichner dargestellt.

Die Terminalsymbole sind die Grundelemente der Sprache. Sie entsprechen den Wörtern [1] einer gewöhnlichen Sprache (Deutsch). Nichtterminale sind Abstraktionen. In einer gewöhnlichen Sprache entsprechen sie Begriffen wie Subjekt, Verb, etc. Aus Terminalsymbolen können entsprechend Produktionen gültige Sätze der Grammatik konstruiert werden. In der Sprache Deutsch ist ein Satz der Form *Subjekt Verb* syntaktisch korrekt gebildet.

[1] Terminale sind auch Wortklassen falls Tokens gebraucht werden.

**Beispiel 1.1** [Skript] Dieses Dokument wurde in XML redigiert, und hat somit eine eindeutige Struktur. Die Grammatik, die dieses Dokument beschreibt, besteht (sehr vereinfacht) aus folgenden Produktionen:

```
          document ::= frontmatter sections
                   ;
       frontmatter ::= title author date
                   ;
             title ::= <TEXT>
                   ;
            author ::= <TEXT>
                   ;
              date ::= <TEXT>
                   ;
          sections ::= section sections
                   ;
          sections ::= ""
                   ;
           section ::= stitle subsections
                   ;
       subsections ::= subsection subsections
                   ;
       subsections ::= ""
                   ;
        subsection ::= stitle subsubsections
                   ;
    subsubsections ::= subsubsection subsubsections
                   ;
    subsubsections ::= ""
                   ;
     subsubsection ::= stitle pars
                   ;
              pars ::= par pars
                   ;
              pars ::= ""
                   ;
               par ::= <TEXT>
                   ;
            stitle ::= <TEXT>
                   ;
```

**Definition 1.3 (Chomsky Sprachhierarchie)** *Die Idee, Sprachen und ihre Grammatiken mathematisch exakt zu definieren, geht auf den Linguist N. Chomsky zurück.*

**Reguläre Grammatik (Typ 3 Grammatik)** *alle Produktionen haben die Form* $\alpha ::= b A$ *; bzw.* $\alpha ::= A$ *; oder die Form* $\alpha ::= A b$ *; bzw.* $\alpha ::= A$ *; wobei* $\alpha$ *und* $b$ *Nichtterminalsymbole sind.*

**Kontextfreie Grammatik (Typ 2 Grammatik)** *alle Produktionen haben die Form* $a ::= w$ *; wobei a ein Nichterminalsymbol und* $w$ *ein String aus* $V$ *ist. Bei einer Herleitung werden einzelne Nichtterminale durch eine möglicherweise leere Folge von Terminalen und Nichtterminalen ersetzt.*

**Kontextsensitive Grammatik (Typ 1 Grammatik)** *alle Produktionen haben die Form* $v ::= w$ *; wobei* $v$ *und* $w$ *Strings aus* $V$ *sind und die Länge von* $v$ *ist kleiner gleich der Länge von* $w$ *. Die Sprache heisst* ***kontextsensitiv*** *weil die linke Seite einer Produktion auch Terminale enthalten darf.*

**Allgemeine Grammatik (Typ 0 Grammatik)** *alle Produktionen haben die Form* $v ::= w$ *; wobei* $v$ *und* $w$ *Strings aus* $V$ *sind.*

**Bemerkung 1.1** [Kontextfrei] Eine Produktion $a ::= w$ ; heisst **kontextfrei**, wenn ihre linke Seite aus einem einzigen Nichtterminal a besteht. D.h. dass $a$ durch $w$ ersetzt werden kann, unabhängig des Kontexts, in dem $a$ vorliegt.

**Bemerkung 1.2** [Praxis] In der Praxis genügen Typ 2 und Typ 3 Grammatiken für die Beschreibung einer Programmiersprache, obwohl bestimmte Konstrukte nicht kontextfrei beschreibbar sind, z.B. die Forderung, dass jeder Bezeichner vor seiner Benutzung deklariert wurde, oder die Forderung, dass die Anzahl Parameter in den Prozedurendeklarationen mit der Anzahl der Parameter beim Prozeduraufruf übereinstimmt. Diese Forderungen werden mit Zusatzregeln in Form von Prozeduren in der semantischen Analyse beschrieben.

**Definition 1.4 (Herleitung)** *Ein String* $v$ *kann aus dem String* $w$ ***direkt hergeleitet*** [2] *werden, genau dann wenn es eine Produktion v::=w; gibt.*

*Ein String* $s_n$ *kann aus einem String* $s_0$ *hergeleitet werden, genau dann wenn es Strings* $s_1$ *,..., * $s_{n-1}$ *so gibt, dass* $s_{i+1}$ *aus* $s_i$ *direkt hergeleitet werden kann (* $i = 1, n-1$ *).*

**Definition 1.5 (Sprache)** *Die Sprache* $L(G)$ *die von einer Grammatik* $G$ *erzeugt wird ist die Menge der Strings über dem Alphabet der terminalsymbole* $T$ *für die es eine Herleitung aus dem Startsymbol* $s$ *der Grammatik gibt.*

Von nun an ist jede Grammatik (somit auch jede Produktion) regulär.

**Beispiel 1.2** [Taschenrechner] Gegeben sei Folgende Grammatik `G=(T,N,P,s)` wobei

```
T = { "+", "-",  "*", "/", "(", " )", <ID<> }

N = { expr }

P:
```

[2] Genauer: $v$ kann aus dem String $w$ **direkt hergeleitet** werden, genau dann wenn es Strings $a$ , $b$ , $v'$ und $w'$ sowie eine Produktion $p : v' ::= w'$ ; so gibt, dass $v = av'b$ und $w = aw'b$ . Man sagt dann, dass die Produktion $p$ in **Kontext** von $a$ und $b$ Anwendung findet ( $a$ und $b$ dürfen leer sein).

```
      expr ::= expr "+" expr ;
      expr ::= expr "-" expr ;
      expr ::= expr "*" expr ;
      expr ::= expr "/" expr ;
      expr ::= "(" expr ")" ;
      expr ::= "-" expr ;
      expr ::= <ID> ;

s = expr
```

Die Eingabe `-(id+id)` lässt sich wie folgt als Rechtsableitung

```
expr ::= "-" expr
     ::= "-" "(" expr ")"
     ::= "-" "(" expr "+" expr ")"
     ::= "-" "(" expr "+" <ID> ")"
     ::= "-" "(" <ID> "+" <ID> ")"
```

oder als Linksableitung

```
expr ::= "-" expr
     ::= "-" "(" expr ")"
     ::= "-" "(" expr "+" expr ")"
     ::= "-" "(" <ID> "+" expr ")"
     ::= "-" "(" <ID> "+" <ID> ")"
```

herleiten.

### 1.2.1 Backus-Naur Form

**Notation 1.2 (BNF)** *Gib es mehrere Produktionen mit dem selben Symbol auf der linken Seite, so wird oft die sog. Backus-Naur Form (BNF) verwendet:* $v ::= w_1$ *;,* $v ::= w_2$ *;,...,* $v ::= w_n$ *;,* $v ::= w_1|w_2...|w_n$ *;, abgekürzt.*

**Beispiel 1.3** [Postfix] Grammatik $G = (T, N, P, s)$ für arithmetische Ausdrücke in Postfix Notation

```
T = { "0", "1", "2", "3", "4", "5", "6",
      "7", "8", "9", "+", "-", "*", "/" }

N = { num, digit, op, addop, subop,
      mulop, divop, expr }

P:
    num    ::=   num digit
```

```
            | digit
            ;
    digit  ::=   "0" | "1" | "2" | "3" | "4"
               | "5" | "6" | "7" | "8" | "9"
            ;
    expr   ::=   expr expr op
               | num
            ;
    op     ::=   addop
               | subop
               | mulop
               | divop
            ;
    addop  ::=   "+"
            ;
    subop  ::=   "-"
            ;
    mulop  ::=   "*"
            ;
    divop  ::=   "/"
            ;

s = expr
```

In der Praxis wird die Grammatik aus Beispiel 1.3, unter Verwendung von Tokens, wie folgt vereinfacht:

**Beispiel 1.4** [Postfix] Grammatik $G = (T, N, P, s)$ für arithmetische Ausdrücke in Postfix Notation

```
T = { <DIGIT>, <ADDOP>, <SUBOP>, <MULOP>, <DIVOP> }

N = { num, digit, op, expr }

P:
     num  ::=  num <DIGIT>
             | <DIGIT>
            ;
     expr ::=  expr expr op
             | num
            ;
     op   ::=  <ADDOP>
             | <SUBOP>
             | <MULOP>
             | <DIVOP>
            ;

s = expr
```

Die BNF-Notation lässt sich selbstverständlich in BNF darstellen.

**Beispiel 1.5** [BNF in BNF]

```
bnf        ::=   produktion bnf
                | ""
                ;
produktion ::=   <NONTERMINAL> "::=" expression ";"
                ;
expression ::=   term
                | term "|" expression
                ;
term       ::=   factor
                | factor term
                ;
factor     ::=   <NONTERMINAL>
                | <TOKEN>
                | <LITERAL>
                ;
```

## 1.3 Ableitungen

Im Folgenden ist die Grammatik kontextfrei.

**Definition 1.6 (Ableitung)** *Eine Herleitung heisst Linksableitung, bzw. Rechtsableitung, wenn jeweils das am weitesten links, bzw. rechts stehende nichtterminale Symbol ersetzt wird (Siehe Bsp. 1.2).*

**Definition 1.7 (Ableitungsbaum)** *Ein Ableitungsbaum ist eine Graphische Darstellung von Ersetzungsschritten. Für jede Produktion a::=bcd; existiert ein Teilableitungsbaum mit Wurzel a und Unterbäume b, c und d (Siehe Abb. 1-1 und 1-2).*

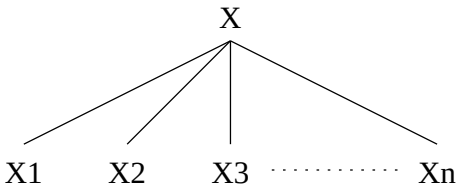



Abbildung 1-1: Ableitungsbaum für die Produktion a::=bcd;

**Definition 1.8 (Mehrdeutigkeit)** *Eine Zeichenkette* $x$ *heisst* ***mehrdeutig****, falls sie* $n > 1$ *verschiedene Ableitungsbäume bezüglich einer Grammatik* $G$ *besitzt (Siehe Abb. 1-3).*

*Eine Grammatik* $G$ *heisst* ***mehrdeutig*** *bzw.* ***eindeutig****, falls sie eine bzw. keine mehrdeutige Zeichenkette aus* $L(G)$ *besitzt.*

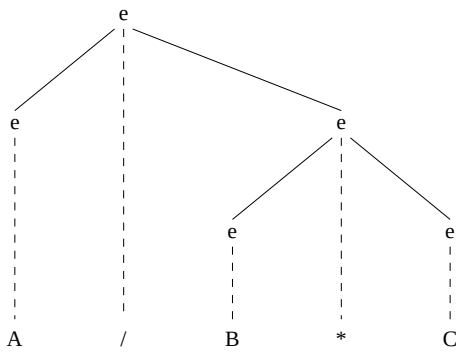



Abbildung 1-2: Ableitungsbaum für die Eingabe `A / B * C`

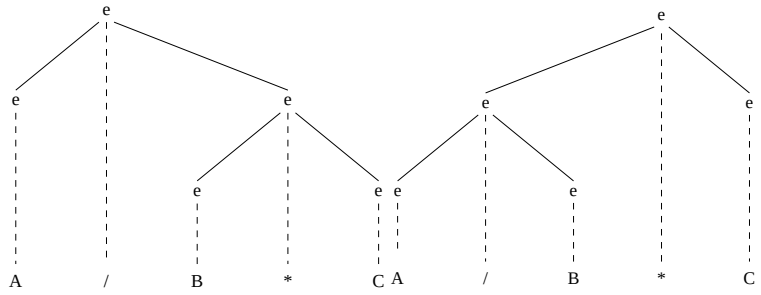



Abbildung 1-3: Mehrdeutige Grammatik

**Beispiel 1.6** [Mehrdeutig] Die Grammatik aus Bsp. 1.2 ist mehrdeutig (Siehe Abb. 1-3).

Bei der Syntaxdefinition von Programmiersprachen sind eindeutige Grammatiken erforderlich. I.a. ist eine Umformung nichteindeutiger Grammatiken möglich.

**Beispiel 1.7** [Mehrdeutig] Die Grammatik $G = (T, N, P, s)$ ist nicht eindeutig

```
T = { "+", "*", "(", ")", <ID> }

N = { expr }

P:
     expr ::=   expr "+" expr
              | expr "*" expr
              | "(" expr ")"
              | <ID>
            ;

s = expr
```

Sie lässt sich aber einfach in eine eindeutige Grammatik $G' = (T', N', P', s)$ umwandeln, nämmlich:

```
T' = { "+", "*", "(", ")", <ID> }

N' = { addExpr, mulExpr, PriExpr }

P':
     addExpr ::=   mulExpr "+" addExpr
                 | mulExpr
               ;
     mulExpr ::=   priExpr "*" mulExpr
                 | priExpr
               ;
     priExpr::=   "(" addExpr ")"
                 | <ID>
               ;

s = addExpr
```

## 1.4 Anwendung

Zur Syntaxdefinition von Programmiersprachen werden kontextfreie Grammatiken verwendet.

- Startsymbol ist `program`
- Terminals sind Tokens. Sie werden in der lexikalischen Analyse ermittelt.
- Nonterminals entsprechen sog. syntaktischen Kategorien, z.B. Ausdrücke, Anweisungen, etc.
- Produktionen: Aufbau syntaktischer Kategorien aus einfacheren Bestandteilen (i.a. rekursiv).
- Erzeugte Sprache: Menge aller syntaktisch korrekten Programme.

**Beispiel 1.8** [Eine Grammatik für ein C-Subset]

- Startsymbol- `program`
- Terminals:
  - in Grossbuchstaben Schlusselwörter: `<WHILE>`, `<IF>`, etc, sowie weitere Token, die in der lexikalischen Analyse erkannt werden: `"+"`, `"-"`, etc.
  - Sonderzeichen: `"("`, `")"`, `":"`, `"{"`, etc.
- Produktionen:

```
program  ::=   varDecls funProts funDecls
           ;
varDecls ::=   varDecl varDecls
             | ""
           ;
varDecl  ::=   <INT> <ID> ";"
           ;
funProts ::=   funProt funProts
             | ""
           ;
funProt  ::=   <INT> <ID> "(" form ")" ";"
           ;
form     ::=   formPars
             | ""
           ;
formPars ::=   <INT> <ID>
             | <INT> <ID> "," formPars
           ;
funDecls ::=   funDecl funDecls
             | ""
           ;
funDecl  ::=   <INT> <ID> "(" form ")" block
           ;
block    ::=   "{" varDecl stats "}"
           ;
stats    ::=   stat stats
             | ""
           ;
stat     ::=   <ID> "=" addExpr ";"
             | <PRINT> "(" addExpr ")" ";"
             | <WHILE> "(" logiExpr ")" stat
             | <WHILE> "(" logiExpr ")" block
             | <IF> "(" logiExpr ")" stat
             | <IF> "(" logiExpr ")" block
           ;
logiExpr ::=   addExpr<RELOP> addExpr
           ;
addExpr  ::=   mulExpr
             | mulExpr <ADDOP> addExpr
           ;
mulExpr  ::=   unaExpr
             | unaExpr <MULOP> mulExpr
           ;
unaExpr  ::=   "-" priExpr
             | priExpr
           ;
priExpr  ::=   "(" addExpr ")"
```

```
                | <ID>
                | <ID> "(" aparl ")"
                | <NUM>
             ;
aparl     ::=   actPars
                | ""
             ;
actPars   ::=   addExpr
                | addExpr "," actPars
             ;
```

## 1.5 Verwandte Methoden

### 1.5.1 Erweiterte Backus-Naur Form

**Notation 1.3 (EBNF)** $\nu ::= (w)*$; *bzw.* $\nu ::= \{w\}$; *Repetition von einem String* $w$ *(* $\nu ::= w\nu|\epsilon$; *)*

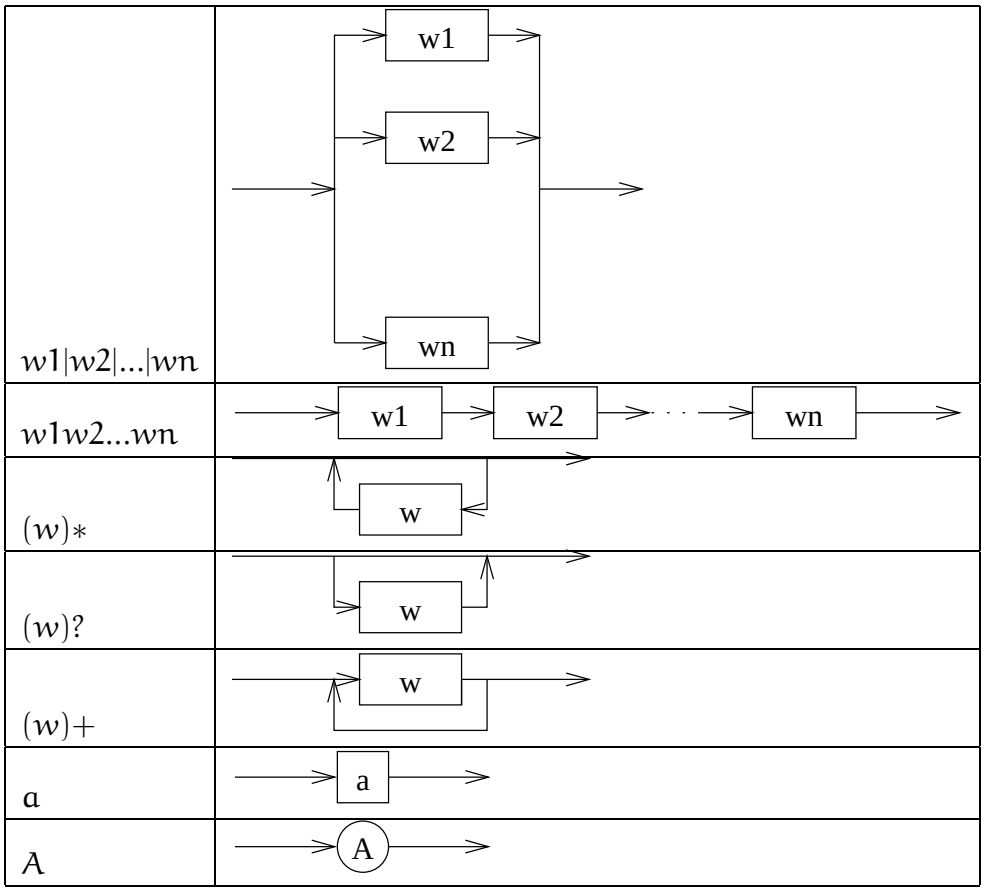
$\nu ::= (w)?$; *bzw.* $\nu ::= [w]$; *optionales String (* $\nu ::= w|\epsilon$; *)*

$\nu ::= (w)+$; *Mindestens eine Repetition von* $w$ *(* $\nu ::= w\nu|w$; *)*

### 1.5.2 Syntaxdiagramme, Syntaxgraphen

Die Darstellung einer Syntax in BNF ist nur eine von verschiedenen Möglichkeiten. Eine Andere, in vieler Hinsicht vorteilhafte Art der Darstellung beruht auf der Verwendung von Diagrammen oder Graphen. Der Hauptvorteil ist die bessere Überschaubarkeit. Der Nachteil ist in vielen Fällen die Grösse der Diagramme.

**Definition 1.9 (Syntaxdiagramme)** *Die EBNF-Konstrukte werden wie folgt in* ***Syntaxdiagramme*** *umgesetzt:*




**Beispiel 1.9** [Reelle Zahl]

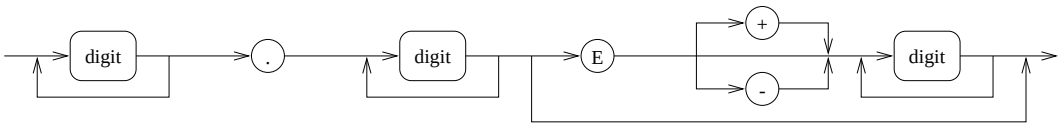



Abbildung 1-4: Syntaxdiagramm für eine Vorzeichenlose reele Zahl

## 1.6 Aufgaben

**Aufgabe 1.1** [Sprachhierarchie] Zeigen Sie, dass eine Sprache vom Typ $i$ auch eine Sprache von Typ $m-1$ ist, $i = 1...3$.

**Lösung**

Bei Typ 0 Grammatiken gibt es keine Restriktionen. Bei Typ 1 Grammatiken ist die linke Seite einer Produktion kürzer als die rechte Seite, somit ist eine Typ 1 Grammatik auch eine Typ 0 Grammatik.

Bei Typ 2 Grammatik besteht die linke Seite einer Produktion aus einem einzigen Nichtterminal. Die linke Seite kann somit nicht länger als die rechte Seite sein. Somit ist eine Typ 2 Grammatik auch eine Typ 1 Grammatik.

Bei Typ 3 Grammatik besteht die linke Seite einer Produktion aus einem einzigen Nichtterminal. Ferner gelten Restriktionen für die recht Seite, die bei Bei Typ 2 Grammatiken nicht vorhanden sind. Somit ist eine Typ 3 Grammatik auch eine Typ 2 Grammatik.

**Aufgabe 1.2** [Reguläre Ausdrücke und Grammatiken] Zeigen Sie, dass reguläre Ausdrücke und Typ 3 Grammatiken äquivalent sind.

**Lösung**

Zur Errinerung: Sei $A$ ein Alphabet. Die regulären Ausdrücke $\alpha$ über $A$ und die zugeordneten regulären Sprachen $L(\alpha) \subseteq A^*$ werden folgendermassen rekursiv definiert:

1. $\emptyset, \epsilon$ und $a \in A$ sind reguläre Ausdrücke und es gilt:

   $L(\emptyset) = \emptyset$
   $L(\epsilon) = \{\epsilon\}$
   $L(a) = \{a\}$

2. Sind $\alpha, \alpha_1, \alpha_2$ reguläre Ausdrücke, so auch $(\alpha), \alpha_1|\alpha_2, \alpha_1\alpha_2, \alpha^*$ und es gilt:

   $L((\alpha)) = L(\alpha)$
   $L(\alpha_1|\alpha_2) = L(\alpha_1) \cup L(\alpha_2)$
   $L(\alpha_1\alpha_2) = L(\alpha_1)L(\alpha_2)$
   $L(\alpha^*) = L(\alpha)^*)$

Wir zeigen zuerst, dass Reguläre Konstrukte mit einer Typ 3 Grammakik realisierbar sind. Anschliessend die Umkehrung.

1. Im wesentlichen müsen wir zeigen, dass mit einer Typ 3 Grammakik die Konstrukte Sequenz, Auswahl und Wiederholung konstruierbar sind.

   Die Sequenz `"a"`, `"b"`, `"c"` kann mit den Produktionen *seq* ::=`"a"` *seq1* ;, *seq1* ::=`"b"` *seq2* ;, *seq2* ::=`"c"`; realisiert werden.

   Die Auswahl `"a"|"b"` kann mit den Produktionen *choice* ::=`"a"`;, *choice* ::=`"b"`; realisiert werden.

   Die Wiederholung `"a"*` kann mit den produktionen *rep* ::=`"a"` *rep* ;, *rep* ::=`""`; realisiert werden.

2. Sofern Produktionen von Typ $a$ ::= $Ab$ ; keine (indirekte) Rekursionen enthalten, werden ausschliesslich Sequenzen gebildet. Dies ist mit Hilvfe von regulären Ausdrücken realisierbar.

   Rekursionen sind im wesentlichen Produktionen vom Typ $a$ ::= $Aa$ ;. Dies entspricht dem regulären Ausdruck `( A )*`.

**Aufgabe 1.3** [Klammerpaare] Zeigen Sie, dass kontextfreie Grammatiken zur Beschreibung von Klammerpaaren geeignet sind, z.B. `begin - end`.

**Lösung**

```
s::="(" s ")" | "";
```

**Aufgabe 1.4** [Listen] Konstruieren Sie eine Grammatik $G = (T, N, P, s)$ mit `T = { "x", "+", "(", ")" }` so dass `x`, `(x)`, `x + x`, `((x + (x + (x))))` und `x + x + x` Elemente von $L(G)$ sind. Die Grammatik soll sowohl in EBNF-Darstellung als auch in Form eines Syntaxdiagramms angegeben werden.

**Lösung**

```
e ::= e "+" e | "(" e ")" | "x" ;
```

**Aufgabe 1.5** [Eindeutigkeit] Gegeben sei eine Kontextfreie Grammatik G mit Produktionen `s::= s s "+" | s s "*" | "a";` und Startsymbol `s`

1. Zeigen Sie, dass `aa+a*` zur Sprache von G gehört.
2. Konstruieren Sie den entsprechenden Ableitungsbaum.
3. Ist diese Grammatik eindeutig?
4. Welche Sprache $L(G)$ erzeugt diese Grammatik?

**Lösung**

1. Siehe Ableitungsbaum.
2. 
```
s ::= s s "*"
  ::= s s "+" s "*"
  ::= "a" s "+" s "*"
  ::= "a" "a" "+" s "*"
  ::= "a" "a" "+" "a" "*"
```
3. Umgekehrte Polnische Notation (HP taschenrechner)
4. Eindeutig da für Taschenrechner verwendet.

**Aufgabe 1.6** [Vereinfachung] Vereinfachen Sie die Grammatik aus Beispiel 1.8 unter Verwendung der EBNF-Notation. Eliminieren Sie insbesondere alle `""` Literals.

**Lösung**

```
program  ::=  (varDecl)* (funProt)* (funDecl)*
          ;
varDecl  ::=  <INT> <ID> ";"
          ;
funProt  ::=  <INT> <ID> "(" (formPars)? ")" ";"
          ;
formPars ::=  <INT> <ID> ( "," formPars )*
          ;
funDecl  ::=  <INT> <ID> "(" (formPars)? ")" block
          ;
block    ::=  "{" (varDecl)* (stat)* "}"
```

```
            ;
stat      ::=   <ID> "=" addExpr ";"
              | <PRINT> "(" addExpr ")" ";"
              | <WHILE> "(" logiExpr ")" stat
              | <WHILE> "(" logiExpr ")" block
              | <IF> "(" logiExpr ")" stat
              | <IF> "(" logiExpr ")" block
            ;
logiExpr  ::=   addExpr<RELOP> addExpr
            ;
addExpr   ::=   mulExpr (<ADDOP> addExpr)?
            ;
mulExpr   ::=   unaExpr (<MULOP> mulExpr)?
            ;
unaExpr   ::=   ("-")? priExpr
            ;
priExpr   ::=   "(" addExpr ")"
              | <ID>
              | <ID> "(" (actPars)? ")"
              | <NUM>
            ;
actPars   ::=   addExpr ("," actPars)*
            ;
```

**Aufgabe 1.7** [EBNF in EBNF] Entwerfen Sie eine EBNF-Grammatik für EBNF-Grammatiken.

**Lösung**

```
ebnf       ::=   ( produktion )*
              ;
produktion ::=   <NONTERMINAL> "::=" choice ";"
              ;
choice     ::=   sequence ( "|" sequence )*
              ;
sequence   ::=   ( primitive )+
              ;
primitive  ::= "(" choice ")" rep
       | <NONTERMINAL>
                | <TOKEN>
                | <LITERAL>
              ;
rep        ::= ( "*" | "+" | "?" )?
    ;
```

**Aufgabe 1.8** [Römischen Zahlen] Konstruieren Sie eine Grammatik für die römischen Zahlen im Bereich 1 bis 2000 .

**Lösung**

```
u ::= "I" | "II"  | "III" | "IV" | "V" |
      "VI" | "VII" | "VIII" | "IX"
   ;
d ::= "X" | "XX"  | "XXX" | "XL" | "L" |
      "LX" | "LXX" | "LXXX" | "XC"
   ;
c ::= "C" | "CC"  | "CCC" | "CD" | "D" |
      "DC" | "DCC" | "DCCC" | "CM"
   ;
m ::= "M" | "MM"
   ;
r ::= m (c)? (d)? (u)? | c (d)? (u)? | d (u)? | u
   ;
```

# 2 Top-down Parsing

## 2.1 Aufgabe der Syntaxanalyse

- Prüfung, ob das Programm syntaktisch korrekt ist.
- Rekonstruktion des Ableitungsbaumes (Rechts-, Linksableitung); damit wird die Bedeutung der Token innerhalb des gesamten Programs festgelegt.

### 2.1.1 Parsing

Ein Programm zur Analyse der Syntax eines Programms wird Parser genannt. Die Eingabe eines Parsers besteht aus einer Folge von Tokens, die von der lexikalischen Analyse ermittelt worden sind. Die Ausgabe ist ein Ableitungsbaum oder eine äquivalente Darstellung.

Es gibt grundsächlich zwei Methoden der Syntaxanalyse:

**top-down** Der Ableitungsbaum wird von der Wurzel zu den Blättern hin konstruiert.

**bottom-up** Der Ableitungsbaum wird von den Bättern zur Wurzel hin konstruiert.

In beiden Fällen wird die Eingabekette (Tokenfolge) von links nach rechts abgearbeitet.

## 2.2 Top-down Parsing

**Beispiel 2.1** [Wirth] Aus [Wirt86a] und [Wirt96a]. Anhand folgender Grammatik wollen wir nun zeigen, wie ein einfacher top-down Parser vorgeht:

```
s ::= a b ;
a ::= "x" | "y";
b ::= "z" | "w";
```

Wir nehmen an, dass er den Satz `"x""w"` zu erkennen habe. `"x""w"` gehört nur dann zur Sprache, wenn es aus dem Startsymbol `s` abgeleitet werden kann. Aus `s` ist jedoch nur die Folge `ab` direkt herleitbar. Wir ersetzen daher `s` auf jeden Fall durch `ab`. Nun muss untersucht werden, ob sich der Anfang des Satzes `"x""w"` aus `ab` herleiten lässt. In der tat bestätigt die Produktion a::=`"x"`;, dass dies möglich ist; `"x"` lässt sich also, zusammen mit `a`, als erledigt abstreichen. Es bleibt noch zu zeigen, dass `"w"` sich aus `b` ableiten lässt. Beim Durchsuchen der Syntax erkennen wir dir Produktion b::=`"z"`; als nicht anwendbar; hingegen bringt uns die Produktion b::=`"w"`; ans Ziel. Wir können uns die einzelnen Schritte wie folgt aufzeichnen, wobei wir links die Symbolfolge auftragen, aus welcher der noch verbleibende Rest des zu erkennenden Satzes herleitbar sein muss. Rechts steht der Rest des Satzes selbst:

```
s           "x" "w"
a b         "x" "w"
"x" b       "x" "w"
b           "w"
"w"         "w"
""          ""
```

Man beachte, dass in diesem Beispiel jeder Schritt sich eindeutig aus dem zunächst zu verfolgenden Ziel (z.B. a) und den nächsten vorliegenden Terminalsymbol (z.B. "x") bestimmen lässt. Dies ist leider nicht in allen Fällen zutreffend.

Bei kontextfreien Grammatiken ohne Einschränkungen ist ein nichtdeterministischer Analysealgorithmus erforderlich, da in bestimmten Situationen u.U. eine falsche Alternative gewählt wird und deshalb Backtracking erfordert (siehe Bsp. 2.2).

**Beispiel 2.2** [Wirth] Aus [Wirt86a], und [Wirt96a] Gegeben sei folgende Grammatik:

```
s ::= a | b ;
a ::= "x" a | "y";
b ::= "x" b | "z";
```

Wir versuchen, den Satz "x""x""z" (der tatsächlich zur definierten Sprache gehört) zu erkennen:

```
s           "x" "x" "z"
a           "x" "x" "z"
"x" a       "x" "x" "z"
a           "x" "z"
"x" a       "x" "z"
a           "z"
```

An dieser Stelle zeigt es sich, dass es unmöglich ist, "z" aus a abzuleiten. Die richtige Lösung ist:

```
s           "x" "x" "z"
b           "x" "x" "z"
"x" b       "x" "x" "z"
b           "x" "z"
"x" b       "x" "z"
b           "z"
"z"         "z"
""          ""
```

Leider aber lässt sich in diesem fall der Entscheid für a oder b nicht aus dem einen vorliegenden Eingabesymbol "x" ableiten. Nur ein vorausblicken und Erkennen des Symbols "z" hätte den richtigen Entscheid ermöglicht. Die einzige Bearbeitungsmöglichkeit besteht in Backtracking, da keine feste Grenze für die Anzahl der zu betrachteten Symbole gegeben werden kann, denn jeder Algorithmus, der fähig wäre, über $n$ Symbole vorauszublicken, könnte mit einer Folge von $n$ Symbolen "x" gefolgt von einem "z" zu Fall gebracht werden.

Selbstverständlich ist Backtracking zu diesem Zweck geeignet. Diese Methode ist aber nicht effizient und oft gefährlich (z.B. Risiko von Endlosschlaufen).

Die Zeitkomplexität bei der top-down Syntaxanalyse mit Backtracking beträgt $O(c^n)$ , die Speicherkomplexität beträgt $O(n)$ , wobei $n$ die Länge des Eingabewortes ist.

In der Praxis wird die Syntax so eingeschränkt, dass das Vorausblicken von mehr als ein Symbol überflüssig wird.

### 2.2.1 first-Mengen

Im Beispiel 2.2 kann "x" sowohl der Anfang der rechten Seite der Produktion a::="x"; als auch der Anfang der rechten Seite der Produktion b::="x"; sein.

Wir bezeichnen die Menge aller Symbole, die am Anfang eines Terms $w$ stehen können, die aus $w$ herleitbar ist mit $\mathtt{first}(w)$

**Definition 2.1 (First)** *Es sei $w$ ein String aus $V$ . Dann ist $\mathtt{first}(w)$ die Menge aller Terminale, mit denen ein aus $w$ hergeleiteter String beginnen kann. Ist $\epsilon$ aus $w$ herleitbar, so gehört auch $\epsilon$ zu $\mathtt{first}(w)$ . D.h. falls $w \Rightarrow \epsilon$ : $\mathtt{first}(w) = \{A \in T; w \Rightarrow Av\} \cup \{\epsilon\}$ sonst: $\mathtt{first}(w) = \{A \in T; w \Rightarrow Av\}$*

Die Grammatik wird wie folgt eingeschränkt:

**Regel 2.1 (first)** *Für jede Produktion $a ::= w_1|w_2|...|w_n$ ; wird verlangt, dass die Initialsymbolmengen aller Terme $w_i$ disjunkt sind, d.h. der Durchschnitt $\mathtt{first}(w_i) \cap \mathtt{first}(w_j)$ ist leer für verschiedene $i$ und $j$ .*

Die first Mengen können mit folgendem Algorithmus berechnet werden.

**Algorithmus 2.1 (First)** *Bestimmung von $\mathtt{first}(x)$*

1. *Falls $x \in T$ , dann $\mathtt{first}(x) = \{x\}$*
2. *Falls $x ::= \epsilon$ ; eine Produktion ist, dann füge $\epsilon$ zu $\mathtt{first}(x)$ hinzu.*
3. *Falls $x$ nicht Terminalsymbol ist und $x ::= y_1y_2...y_n$ ; eine Produktion ist, dann nimm $A$ zu $\mathtt{first}(x)$ hinzu, falls $A$ für irgendein $i$ in $\mathtt{first}(y_i)$ und $\epsilon$ in allen $\mathtt{first}(y_1)$ , $\mathtt{first}(y_2)$ ... $\mathtt{first}(y_{i-1})$ enthalten ist. Wenn $\epsilon$ in allen $\mathtt{first}(y_i)$ enthalten ist, nimm $\epsilon$ zu $\mathtt{first}(x)$ hinzu.*

In der Syntax vom Beispiel 2.2 ist "x" sowohl in $\mathtt{first}(a)$ als auch in $\mathtt{first}(b)$ enthalten. Die Regel 2.1 ist also durch die Produktion a ::= a|b ; verletzt.

**Beispiel 2.3** [Wirth] Aus [Wirt86a]und [Wirt96a]. Wir gehen nun die Probleme aus Beispiel 2.2 wie folgt um, indem eine aequivalente Syntax definiert wird, welche die Regel 2.1 beachtet:

```
s ::= c | "x" s;
c ::= "y" | "z";
```

Der Satz "x""x""z" lässt sich nun eindeutig erkennen:

```
s          "x" "x" "z"
"x" s      "x" "x" "z"
s          "x" "z"
"x" s      "x" "z"
s          "z"
c          "z"
"z"        "z"
""         ""
```

### 2.2.2 follow-Mengen

Leider genügt Regel 2.1 noch nicht, um alle Schwierigkeiten auszuschliessen:

**Beispiel 2.4** [Wirth] Aus [Wirt86a] und [Wirt96a]. Gegeben sei folgende Grammatik:

```
s ::= a "x";
a ::= "x" | "";
```

Falls wir nun versuchen, den Satz `"x"` zu erkennen, geraten wir erneut in eine Sackgasse:

```
s          "x"
a "x"      "x"
"x" "x"    "x"
"x"        ""
```

Die Richtige Lösung wäre die Anwendung der Produktion s::=""; für die letzte Herleitung.

```
s          "x"
a "x"      "x"
"" "x"     "x"
"x"        "x"
""         ""
```

Eine Situation wie im Beispiel 2.4 kann nur entstehen, wenn ein Symbol die leere Folge erzeugen kann. Das Symbol `"x"` gehört zwar zu $\mathtt{first}(\alpha)$, `"x"` kann aber auch `a` unmittelbar folgen!

Die Menge aller Symbole, die einer aus $\alpha$ hergeleiteten Folge unmittelbar nachfolgen können, wird mit $\mathtt{follow}(\alpha)$ bezeichnet:

**Definition 2.2 (follow)** *Es sei* $\alpha \in N$ . $\mathtt{follow}(\alpha)$ *ist die Menge aller Terminale* $A$ *, die in einer Satzform direkt rechts neben* $\alpha$ *stehen können.*

$\mathtt{follow}(\alpha) = \{A \in T; s \Rightarrow w_1 \alpha A w_2; w_1, w_2 \in V\}$

**Regel 2.2 (follow)** *Für jedes Nichtterminalsymbol* $\alpha$ *, aus welchen die leere Folge hergeleitet werden kann, muss die Menge* $\mathtt{first}(\alpha)$ *seiner Initialsymbole disjunkt von der menge* $\mathtt{follow}(\alpha)$ *der Folgesymbole sein:*

*Aus* $\alpha \Rightarrow \epsilon$ *folgt dass* $\mathtt{first}(\alpha) \cap \mathtt{follow}(\alpha)$ *leer ist.*

Für die Berechnung der follow-Mengen wird die Grammatik mit einer neuen Produktion $s'$ ::= s <EOF>; sowie einem neuen Terminal <EOF> erweitert. $s'$ ist das neue Startsymbol der erweiterten Grammatik. Die follow-Mengen können mit folgendem Algorithmus berechnet werden.

**Algorithmus 2.2 (follow)** *Bestimmung von* $\mathtt{follow}(a)$ *:*

1. *Nimm* <EOF> *in* $\mathtt{follow}(s)$ *hinzu.*
2. *Falls* $a ::= w_1 b w_2 ; \in P$ *, wird jedes Element von* $\mathtt{first}(w_2)$ *mit Ausnahme von* $\epsilon$ *auch in* $\mathtt{follow}(b)$ *aufgenommen.*
3. *Falls es Produktionen* $a ::= w_1 b$ *; oder* $a ::= w_1 b w_2$ *; gibt und falls* $\epsilon$ *in* $\mathtt{first}(w_2)$ *enthalten ist, dann gehört jedes Element von* $\mathtt{follow}(a)$ *auch zu* $\mathtt{follow}(b)$ .

**Beispiel 2.5** [first und follow] Gegeben sei folgende Grammatik für arithmetische Ausdrücke

```
e  ::= t e' ;
e' ::= "+" t e' | "" ;
t  ::= f t' ;
t' ::= "*" f t' | "" ;
f  ::= "(" e ")" | <ID> ;
```

Es gilt dann:

```
first(e)  = first(t) = first(f) = { "(", <ID> }
first(e') = { "+", "" }
first(t') = { "*", "" }
follow(e) = follow(e') = { ")", <EOF> }
follow(t) = follow(t') = { "+", ")", <EOF> }
follow(f) = { "*", "+", ")", <EOF> }
```

### 2.2.3 LL(1) Grammatiken

**Definition 2.3 (LL(1) Grammatik)** *Eine* ***LL(1) Grammatik*** *ist eine Grammatik, die die Regeln 2.1 und 2.2, erfüllt*

Dabei bedeutet das erste L in LL(1), dass die Eingabe von Links nach rechts gelesen wird; das zweite L dass eine Linksherleitung erzeugt wird; und die 1 dass in jedem Schritt des Parseprozesses ein Lookahead Symbol benötigt wird, um zu entscheiden, welche Aktionen durchzuführen ist. Präziser wird eine LL(1) Grammatik wie folgt definiert:

**Definition 2.4 (LL(1) Grammatik)** *Eine LL(1)-Grammatik ist eine Grammatik, deren Parse-Tabelle keine Mehrfacheinträge besitzt (siehe Abschnitt A.2).*

**LL(k)-Grammatiken** sind Verallgemeinerungen. Dabei werden $k$ -Lookaheadsymbole benötigt, um zu entscheiden, welche Aktionen durchzuführen sind.

### 2.2.4 Semantik

**Bemerkung 2.1** [Semantik] Die Syntax ist nur Mittel zu einem höheren Zweck, nämlich zur Sichtbarmachung der Bedeutung eines Satzes. Wird eine Syntax umgeformt, so muss stets beachtet werden, dass damit die Bedeutungsstruktur der Sprache nicht in Mitleidenschaft gezogen wird.

Gegeben sei folgende Grammatik:

```
s ::= a | s "-" a;
a ::= "a" | "b" | "c";
```

Da die linksrekursive Produktion für `s` Regel 2.1 verletzt, muss eine aequivalente Syntax gesucht werden:

```
s ::= a  b;
b ::= "-" s | "";
a ::= "a" | "b" | "c";
```

In der ersten Version der Grammatik wird dem Satz `"a""-""b""-""c"` die Struktur zugewiesen, die durch Klammerung `((a-b)-c)` hervorgehoben werden kann. In der zweiten Version wird demselben Satz die Struktur zugewiesen, die durch Klammerung `(a-(b-c))` hervorgehoben werden kann. Beide Versionen sind zwar syntaktisch, aber nicht semantisch aequivalent.

## 2.3 Probleme in der Praxis

Regeln 2.1 und 2.2 werden insbesondere durch jede linksrekursive Produktion und durch jede nicht faktorisierte Produktion verletzt In der Praxis [3] muss man darauf achten, dass die Grammatik faktorisiert ist und keine Links-Rekursionen enthält.

### 2.3.1 Links-Faktorisierung

Besitzen Produktionen einer Grammatik gemeinsame Präfixe, so wird die Syntaxanalyse erschwert, da je nach dem nicht einfach entscheidbar ist, welche Produktion zur Anwendung kommen soll. Erst nach Verarbeitung des Präfixes, kann entschieden werden, welche Produktion gewählt wird. Gemeinsame Präfixe könne einfach eliminiert werden:

**Beispiel 2.6** [Taschenrechner] Die Produktionen $e$ ::= $e$ `"+"` t|$e$ `"-"` t|t ; besitzen das Nichtterminal $e$ als gemeinsames Präfix. Durch Einführung eines neuen Nichtterminals $e'$ erhalten wir äquivalente Produktionen $e$ ::= $ee'$|t ; und $e'$ ::=`"+"` t| `"-"` t ;

**Algorithmus 2.3 (Links-Faktorisieren)** *Solange es Produktionen* $\alpha ::= \nu w_1$ *; und* $\alpha ::= \nu w_2$ *; gibt, mit* $\nu \neq \epsilon$

[3] `choice conflict` und `left recursion` sind die zwei typische Fehlermeldungen von JAVACC.

1. *Streiche* $\alpha ::= \nu w_1$ *; und* $\alpha ::= \nu w_2$ *; aus* $P$

2. *Füge folgende Produktionen* $\alpha ::= \nu \alpha'$ *; und* $\alpha' ::= w_1 | w_2$ *; hinzu, wobei* $\alpha'$ *ein neues Nichtterminal ist.*

### 2.3.2 Links-Rekursion

**Definition 2.5 (Links-Rekursion)** *Es sei* $G = (N, T, P, s)$ *eine kontextfreie Grammatik.* $G$ *heisst links-rekursiv, falls mit den Produktionen der Grammatik eine Herleitung der Form* $\alpha \Rightarrow \alpha w$ *gibt, mit* $\alpha \in N$ *und* $w \in V$

Insbesondere sind Produktionen der Form a ::= a <A> | ...; oder c ::= d <C> | ...; und d ::= c <D> | ...; links-rekursiv.

Eine Eliminierung der Linksrekursion ist ohne änderung der erzeugten Sprache möglich.

**Beispiel 2.7** [Elimination der Linksrekursion] Es sei $G = (T, N, P, s)$ eine kontextfreie Grammatik. Die Linksrekursion $\alpha ::= \alpha\nu | w_1 | w_2 | ... | w_n$ ; lässt sich in $\alpha ::= w_1\alpha' | w_2\alpha' | ... | w_n\alpha'$ ; und $\alpha' ::= \nu\alpha' | \epsilon$ ; umwandeln.

## 2.4 Rekursiver Abstieg

Wir wollen nun annehmen, dass die Gramatik die Bedingungen aus Regel 2.1 und Regel 2.2 erfüllt, d.h, dass die Grammatik die $LL(1)$ -Eigenschaft (siehe Abschnitt 2.2.3). Insbesondere sollen in der Grammatik keine Produktionen mit gemeinsamen Präfixen (gemeinsame Anfangstokens), sowie keine Linksrekursionen auftreten. In vielen Fällen ist eine Umformung der Grammatik möglich [4]. Die Idee vom rekursivem Abstieg ist die folgende:

- Eine Prozedur für jedes Nichtterminal der Grammatik.
- Die Prozeduren ergeben sich direkt aus der Grammatik.
- Der Ableitungsbaum wird erzeugt durch die Ausgabe der angewandten Produktionen.

In der Regel gibt es zu jedem Nichtterminal $\alpha$ mehrere Produktionen mit $\alpha$ auf der linken Seite. Um eine Auswahl zu treffen, muss das nächste Token mit dem übereinstimmen, was der Anfang dessen ist, was aus $\alpha$ herleitbar ist. Es sei $\alpha ::= w$ ; eine solche Produktion. Der Anfang von $\alpha ::= w$ ; wird mit Hilfe von $\mathtt{predict}(\ \alpha ::= w\ ;\ )$ bestimmt. $\mathtt{predict}(\ \alpha ::= w\ ;\ )$ ist die Menge der terminalsymbole, die eine Satzform beginnen können, die bei Verwenden der Produktion $\alpha ::= w$ ; durch G erzeugt werden kann:

**Definition 2.6 (predict)** *Falls* $\epsilon \in \mathtt{first}(w)$ *dann ist* $\mathtt{predict}(\ \alpha ::= w\ ;\ ) = (\mathtt{first}(w) - \epsilon) \cup \mathtt{follow}(w)$ *sonst ist* $\mathtt{predict}(\ \alpha ::= w\ ;\ ) = \mathtt{first}(w)$ .

[4] Es können aber dabei semantische Probleme auftreten

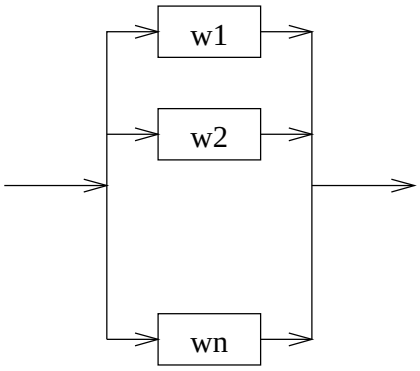
**Definition 2.7 (Rekursivem Abstieg)** *Ein Parser mit rekursivem Abstieg (recursive descent parser) enthält für jedes Nichtterminal* a *eine Methode* `a()` *und arbeitet wie folgt:*

1. *Er entscheidet, welche Produktion er anwendet, indem er das Lookahead-Symbol aus der Eingabe mit den* `predict` *-Mengen des aktuellen Nichtterminals vergleicht.*
2. *Der Parser bildet die Produktionen nach, indem er die rechte Seite der Produktionen wie folgt herleitet: Ein Nichtterminal* b *auf der rechten Seite führt zu einem Aufruf der korrespondierenden Methode* `b()` *(absteigen). Bei einem Terminalsymbol, das mit dem Lookahead-Symbol übereinstimmt, wird das nächste Token gelesen (konsummieren); stimmt es nicht überein, wird ein Fehler angezeigt.*

### 2.4.1 Rekursiver Abstieg: Kodierung

Aus [Wirt86a] und [Wirt96a]. Wir bezeichnen das programm, das aus der Übersetzung eines graphens $S$ hervorgeht, mit $p(S)$ .

- Jede Struktur mit der Form

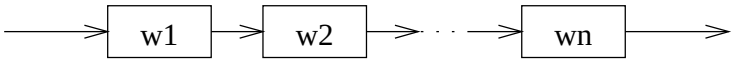



wird in eine bedingte oder selektive Anweisung übersetzt:

```
if (lookahead in L1) {
  p(w1);
}
else if (lookahead in L2) {
  p(w2);
}
.................................
else if (lookahead in Ln) {
  p(wn);
}
else {
  error();
}
```

Dabei bezeichnet $L_i$ die Menge $first(w_i)$ .

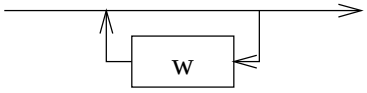
- Jede Struktur mit der Form




  wird in eine Sequenz übersetzt:

```
p(w1);
p(w2);
.....
p(wn);
```

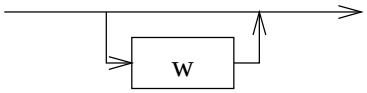
- Jede Struktur mit der Form




  wird in eine Wiederholung übersetzt:

```
while (lookahead in L) {
  p(w);
}
```

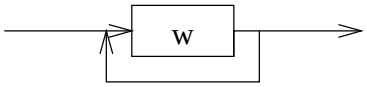
- Jede Struktur mit der Form




  wird in eine bedingte Anweisung übersetzt:

```
if (lookahead in L) {
  p(w);
}
```

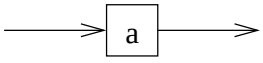
- Jede Struktur mit der Form




  wird in eine Wiederholung übersetzt:

```
do {
  p(w);
} while (lookahead in L);
```

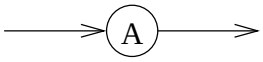
- Jede Struktur mit der Form




  wird in einem Aufruf der dem Graphen entsprechenden Prozedur übersetzt:

```
p(a);
```

- Jede Struktur mit der Form




Jede Referenz zu einem Terminalsymbol *A* wird wie folgt übersetzt (Prozedur `match()`):

```
if (lookahead == A) {
  eat(A);
}
else {
  error();
}
```

**Beispiel 2.8** [Rekursiver Abstieg] Gegeben seien die Produktionen

```
s  ::= "x" b "z";
b  ::= "y" | s | "";
```

Die zugeordnete Methoden `s()` und `b()` haben folgende Gestalt:

```
void s() {
  match("x");
  b();
  match("z");
}

void b() {
  switch (lookahead) {
    case "y": {
      match("y");
      break;
    }
    case "x": {
      s();
      break;
    }
    case "z": {
      break;
    }
    default: {
      error();
    }
  }
}
```

Die `match()` dient zur überprüfung, ob das nächste anstehende Token mit dem übereinstimmt, was erwartet wird. Ist dies der Fall, so ruft `match()` die `getNextToken()` auf,

die das nächste Token als neue `lookahead`-Symbol liefert. Andernfalls erfolgt eine Fehlermeldung.

Der Anfang von `s` ist das Terminal `"x"` es muss also konsumiert werden. Als nächstes wird `b()` aufgerufen. Anschliessend wird `"z"` konsumiert.

Der Anfang von `b` ist entweder `"y"`, der Anfang von `s` oder `""` Die Auswahl in `b()` entscheidet anhand von `lookahead` ob `"y"` konsumiert wird, `s()` aufgerufen wird, oder nichts gemacht wird (`""`). Bei der letzten Variante (`""`) soll das nächste Symbol in $follow(b) = \{$`"z"`$\}$ enthalten sein.

**Beispiel 2.9** [Taschenrechner]

```
program ::=   statl <EOF>
          ;
statl   ::=   statl stat
            | ""
          ;
stat    ::=   <IDENTIFIER> "=" addExpr ";"
            | <PRINT> "(" addExpr ")" ";"
          ;
addExpr ::=   mulExpr
            | mulExpr "+" addExpr
            | mulExpr "-" addExpr
          ;
mulExpr ::=   unaExpr
            | unaExpr "*" mulExpr
            | unaExpr "/" mulExpr
          ;
unaExpr ::=   "-" priExpr
            | priExpr
          ;
priExpr ::=   "(" addExpr ")"
            | <IDENTIFIER>
            | <NUMBER>
          ;
```

Diese Grammatik ist für den rekursiven Abstieg noch nicht geeignet:

die Produktion statl::=statl stat; ist links rekursiv. Ferner müssen `addExpr` und `mulExpr` linksfaktorisiert werden. Nach geeigneten Transformationen erhalten wir folgende Grammatik in EBNF Schreibweise:

```
program ::=   statl <EOF>
          ;
statl   ::=   ( stat )*
          ;
stat    ::=   <IDENTIFIER> "=" addExpr ";"
            | <PRINT> "(" addExpr ")" ";"
```

```
          ;
addExpr ::=   mulExpr ( "+" addExpr | "-" addExpr )?
          ;
mulExpr ::=   unaExpr ( "*" mulExpr | "/" mulExpr ) ?
          ;
unaExpr ::=   ( "-" )? priExpr
          ;
priExpr ::=   "(" addExpr ")"
            | <IDENTIFIER>
            | <NUMBER>
          ;
```

Die first Mengen von G sind:

```
first(program)  = { <IDENTIFIER>, <PRINT>, "" }
first(statl)    = { <IDENTIFIER>, <PRINT>, "" }
first(stat)     = { <IDENTIFIER>, <PRINT>, "" }
first(addExpr)  = { "-", "(", <IDENTIFIER>, <NUMBER> }
first(mulExpr)  = { "-", "(", <IDENTIFIER>, <NUMBER> }
first(unaExpr)  = { "-", "(", <IDENTIFIER>, <NUMBER> }
first(priExpr)  = { "(", <IDENTIFIER>, <NUMBER> }
follow(program) = { <EOF> }
follow(statl)   = { <EOF> }
follow(stat)    = { <IDENTIFIER>, <PRINT>, <EOF> }
follow(addExpr) = { ")", ";" }
follow(mulExpr) = { "+", "-", ")", ";" }
follow(unaExpr) = {  "*", "/", "+", "-", ")", ";" }
follow(priExpr) = {  "*", "/", "+", "-", ")", ";" }
```

Für obige Grammatik hat ein Programmgerüst für die top-down Syntaxanalyse mit rekursivem Abstieg folgende Gestalt:

JAVACC-Spezifikation `Scanner`:

```
PARSER_BEGIN(Scanner)

public class Scanner { }

PARSER_END(Scanner)

TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < SEMICOLON: ";" >
| < ASSIGN: "=" >
```

```
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < PRINT: "print" >
| < IDENTIFIER:  ["a"-"z","A"-"Z"]
                 ( ["a"-"z","A"-"Z","0"-"9"] )* >
| < NUMBER: ( ["0"-"9"] )+ >
}

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}
```

JAVA-Klasse `Parser`:

```
class Parser implements ScannerConstants {

  static Token lookahead;
  static Scanner scanner;

  public static void main (String args[]) {
    scanner = new Scanner(System.in);
    lookahead = scanner.getNextToken();
    program();
    System.out.println("> parsing succesfull");
  }

  static void program() {
    statl();
    match(EOF);
  }

  static void statl() {
    while (lookahead.kind != EOF)
      stat();
  }

  static void stat() {
    switch(lookahead.kind) {
```

```
  case IDENTIFIER:
    match(IDENTIFIER);
    match(ASSIGN);
    addExpr();
    match(SEMICOLON);
    break;
  case PRINT :
    match(PRINT);
    match(LPAREN);
    addExpr();
    match(RPAREN);
    match(SEMICOLON);
    break;
  default:
    unexpectedToken(lookahead);
  }
}

static void addExpr() {
  mulExpr();
  switch(lookahead.kind) {
  case PLUS:
    match(PLUS);
    addExpr();
    break;
  case MINUS:
    match(MINUS);
    addExpr();
    break;
  }
}

static void mulExpr() {
  unaExpr();
  switch(lookahead.kind) {
  case STAR:
    match(STAR);
    mulExpr();
    break;
  case SLASH:
    match(SLASH);
    mulExpr();
    break;
  }
}

static void unaExpr() {
```

```
    if (lookahead.kind == MINUS) {
      match(MINUS);
    }
    priExpr();
  }

  static void priExpr() {
    switch (lookahead.kind) {
    case LPAREN:
      match(LPAREN);
      addExpr();
      match(RPAREN);
      break;
    case IDENTIFIER:
      match(IDENTIFIER);
      break;
    case NUMBER:
      match(NUMBER);
      break;
    default:
      unexpectedToken(lookahead);
    }
  }

  static void unexpectedToken(Token lookahead) {
    System.out.println("> at line " + lookahead.beginLine +
               ", column " + lookahead.beginColumn);
    System.out.println("> unexpected token : " +
               tokenImage[lookahead.kind]);
    System.exit(1);
  }

  static void match(int tokenKind) {
    if (lookahead.kind == tokenKind)
      lookahead = scanner.getNextToken();
    else
      unexpectedToken(lookahead);
  }
}
```

## 2.5 JavaCC

Der Parsergenerator JAVACC [5] ist natürlich besonders geeignet für die generierung von Top-Down Parser.

Falls man in der Lage ist, Parser mit rekursiven Abstieg selber zu entwerfen ist die Arbeit mit JAVACC ist besonders einfach. Eine Produktion der Form:

```
p ::= a b <C> | <D> e ;
```

wird wie folgt in JAVACC umgesetzt:

```
void p() : {}
{
  a() b() <C> | <D> e()
}
```

Dabei müssen `<C>` und `<D>` als Tokens definiert werden. Das leere Klammerpaar `{}` ist für lokale Deklarationen reserviert (wird im nächsten Abschnitt besprochen).

### 2.5.1 Verhalten von JavaCC

JAVACC übernimmt die Berechnung der predict -Mengen. Im Falle eines Konflikts wird eine entsprechende Fehlermeldung erzeugt. Wir zeigen nun wie die Beispiele von Wirth in JAVACC umgesetzt werden, und welche Fehlermeldungen erzeugt werden.

**Beispiel 2.10** [Wirth] JAVACC Spezifikation von Beispiel 2.2:

```
PARSER_BEGIN(Wirth01)

public class Wirth01 {
  public static void main (String args []) {
    Wirth01 parser = new Wirth01(System.in);
    try { parser.s(); }
    catch (Exception e) { }
  }
}

PARSER_END(Wirth01)

void s() : {}
{
  a() | b()
```

[5] JAVACC wurde 1996 von Sriram Sankar (Sun Microsystems) und Sreenivasa Viswanadha (SUNY at Albany) entwickelt. JAVACC ist frei erhältlich unter `http://www.metamata.com`

```
}

void a() : {}
{
  "x" a() | "y"
}

void b() : {}
{
  "x" b() | "z"
}
```

JAVACC Meldungen:

```
Copyright (c) 1996-2000 Sun Microsystems, Inc.
Copyright (c) 1997-2000 Metamata, Inc.
(type "javacc" with no arguments for help)
Reading from file wirth01.jj . . .
Warning: Choice conflict involving two expansions at
         line 15, column 3 and
         line 15, column 9 respectively.
         A common prefix is: "x" "x"
         Consider using a lookahead of 3 or
         more for earlier expansion.
Parser generated with 0 errors and 1 warnings.
```

Man beachte, dass hier nur eine Warnung erfolgt. Im Konfliktfall wird der parser immer die erstmögliche Wahl treffen und die Alternative ignorieren.

**Beispiel 2.11** [Wirth] JAVACC Spezifikation (mit Linksrekursion):

```
PARSER_BEGIN(Wirth02)

public class Wirth02 {
  public static void main (String args []) {
    Wirth02 parser = new Wirth02(System.in);
    try { parser.s(); }
    catch (Exception e) { }
  }
}

PARSER_END(Wirth02)

void s() : {}
{
  a() "x"
```

```
}

void a() : {}
{
  s() "y" | "z"
}
```

JAVACC Meldungen:

```
Java Compiler Compiler Version 2.0 (Parser Generator)
Copyright (c) 1996-2000 Sun Microsystems, Inc.
Copyright (c) 1997-2000 Metamata, Inc.
(type "javacc" with no arguments for help)
Reading from file wirth02.jj . . .
Error: Line 13, Column 1:
       Left recursion detected: "s... --> a... --> s..."
Detected 1 errors and 0 warnings.
```

Hier wird richtigerweise einen Fehler gemeldet. Sonst würde eine endlos Rekursion entstehen.

**Beispiel 2.12** [Wirth] JAVACC Spezifikation von Beispiel 2.4:

```
PARSER_BEGIN(Wirth03)

public class Wirth03 {
  public static void main (String args []) {
    Wirth03 parser = new Wirth03(System.in);
    try { parser.s(); }
    catch (Exception e) { }
  }
}

PARSER_END(Wirth03)

void s() : {}
{
  a() "x"
}

void a() : {}
{
  ( "x" )?
}
```

JAVACC Meldungen:

```
Java Compiler Compiler Version 2.0 (Parser Generator)
Copyright (c) 1996-2000 Sun Microsystems, Inc.
Copyright (c) 1997-2000 Metamata, Inc.
(type "javacc" with no arguments for help)
Reading from file wirth03.jj . . .
Warning: Choice conflict in [...] construct at line 20,
         column 3.
         Expansion nested within construct and expansion
         following construct have common prefixes, one
         of which is: "x"
         Consider using a lookahead of 2 or more for
         nested expansion.
Parser generated with 0 errors and 1 warnings.
```

Es folgt noch ein lauffähiges JAVACC Beispiel:

**Beispiel 2.13** [JavaCC] Für die Grammatik aus Beispiel 2.6 hat eine JAVACC Spezifikation für die top-down Syntaxanalyse folgende Gestalt.

JAVACC-Spezifikation `Parser`:

```
PARSER_BEGIN(Parser)

public class Parser {
  public static void main (String args []) {
    Parser parser = new Parser(System.in);
    try {
      parser.program();
      System.out.println("programm is correct");
    }
    catch (Exception e) {
      System.out.println("parse error");
    }
  }
}

PARSER_END(Parser)

TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < SEMICOLON: ";" >
| < ASSIGN: "=" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
```

```
| < SLASH: "/" >
| < PRINT: "print" >
| < IDENTIFIER:  ["a"-"z","A"-"Z"]
                 ( ["a"-"z","A"-"Z","0"-"9"] )* >
| < NUMBER: ( ["0"-"9"] )+ >
}

SKIP :
{
  " " | "\t" | "\n" | "\r" | "\f"
}

void program() : {}
{
  statl()  <EOF>
}

void statl() : {}
{
  ( stat() )*
}

void stat() : {}
{
  <IDENTIFIER> <ASSIGN> addExpr() <SEMICOLON>
| <PRINT> <LPAREN> addExpr() <RPAREN> <SEMICOLON>
}

void addExpr() : {}
{
  mulExpr() (( <PLUS> | <MINUS> ) addExpr() )?
}

void mulExpr() : {}
{
  unaExpr() (( <STAR> | <SLASH> ) mulExpr() ) ?
}

void unaExpr() : {}
{
  ( <MINUS> )? priExpr()
}

void priExpr() : {}
{
  <LPAREN> addExpr() <RPAREN>
| <IDENTIFIER>
```

```
| <NUMBER>
}
```

## 2.6 Aufgaben

**Aufgabe 2.1** [Linksrekursion] Es sei G eine Grammatik mit Startsymbol c . Eliminieren Sie die indirekte Linksrekursion in den Produktionen
c::=d<D> | <E>;
und
d::=c<C> | <F>;.

**Lösung**

Einsetzen von d in c ergibt die direkte Rekursion:
c::=c<C><D> | <F><D> | <E>;
(direkte Rekursion). Das Resultat ist nun:
c::=<F><D> e | <E> e;
e::=<C><D> e | "";

**Aufgabe 2.2** [if] Analysiere die `if` Anweisung der Programmiersprache Java unter Berücksichtigung der der Mehrdeutigkeit.

**Lösung**

Eine mögliche Grammatik für `if` ist:

```
stmt ::=   "if" expr "then" stmt
         | "if" expr "then" stmt "else" stmt
         | other
       ;
```

Dabei steht `other` für irgendeine andere Anweisung. Diese Grammatik ist mehrdeutig, weil `if e1 then if e2 then s2 else s3` zwei Ableitungsbäume besitzt.

Alle Programmiersprachen mit bedingten Anweisungen dieser Art geben der Ableitung den Vorzug wo `else s3` zu `if e2` gehört.Die allgemeine Regel heisst: Ein `else` gehört zum letzten noch freien `then`. Diese Regel zur Auflösung der Mehrdeutigkeit kann man in eine neue Grammatik einbauen. Die Idee dabei ist, dass eine Anweisung zwischen einem `then` und einem `else` geschlossen sein muss:

```
stmt          ::=   matchedStmt
                  | unmatchedStmt
                ;
matchedStmt   ::=   "if" expr "then" matchedStmt "else" matchedStmt
                  | other
                ;
unmatchedStmt ::=   "if" expr "then" stmt
                  | "if" expr "then" matchedStmt "else" unmatchedStmt
                ;
```

**Aufgabe 2.3** [XML] Gegeben sei die Grammatik aus Beispiel 2.9.

Instrumentieren Sie den zugehörigen rekursiven Parser so, dass bei folgender Eingabe:

```
x=1;
y=2;
z=x*y-3;
```

folgende XML-Datei als Ausgabe produziert wird:

```
<?xml version="1.0" ?>
<program>
  <statementlist>
    <statement>
      <match kind="12" value="x"/>
      <match kind="6" value="="/>
      <addexpr>
        <mulexpr>
          <unaexpr>
            <priexpr>
              <match kind="13" value="1"/>
            </priexpr>
          </unaexpr>
        </mulexpr>
      </addexpr>
      <match kind="5" value=";"/>
    </statement>
    <statement>
      <match kind="12" value="y"/>
      <match kind="6" value="="/>
      <addexpr>
        <mulexpr>
          <unaexpr>
            <priexpr>
              <match kind="13" value="2"/>
            </priexpr>
          </unaexpr>
        </mulexpr>
      </addexpr>
      <match kind="5" value=";"/>
    </statement>
    <statement>
      <match kind="12" value="z"/>
      <match kind="6" value="="/>
      <addexpr>
        <mulexpr>
          <unaexpr>
```

```
            <priexpr>
              <match kind="12" value="x"/>
            </priexpr>
          </unaexpr>
          <match kind="9" value="*"/>
          <mulexpr>
            <unaexpr>
              <priexpr>
                <match kind="12" value="y"/>
              </priexpr>
            </unaexpr>
          </mulexpr>
        </mulexpr>
        <match kind="8" value="-"/>
        <addexpr>
          <mulexpr>
            <unaexpr>
              <priexpr>
                <match kind="13" value="3"/>
              </priexpr>
            </unaexpr>
          </mulexpr>
        </addexpr>
      </addexpr>
      <match kind="5" value=";"/>
    </statement>
  </statementlist>
  <match kind="0" value=""/>
</program>
```

**Lösung**

JAVA-Programm `parserxml`:

```
class Parser implements ScannerConstants {

  static Token lookahead;
  static Scanner scanner;

  public static void main (String args[]) {
    scanner = new Scanner(System.in);
    lookahead = scanner.getNextToken();
    program("");
    System.out.println("> parsing succesfull");
  }

  static void program(String indent) {
    System.out.println(indent + "<program>");
```

```
  statl(indent + "  ");
  match(indent + "  ", EOF);
  System.out.println(indent + "</program>");
}

static void statl(String indent) {
  System.out.println(indent + "<statl>");
  while (lookahead.kind != EOF)
    stat(indent + "  ");
  System.out.println(indent + "</statl>");
}

static void stat(String indent) {
  System.out.println(indent + "<stat>");
  switch(lookahead.kind) {
  case IDENTIFIER:
    match(indent + "  ", IDENTIFIER);
    match(indent + "  ", ASSIGN);
    addExpr(indent + "  ");
    match(indent + "  ", SEMICOLON);
    break;
  case PRINT :
    match(indent + "  ", PRINT);
    match(indent + "  ", LPAREN);
    addExpr(indent + "  ");
    match(indent + "  ", RPAREN);
    match(indent + "  ", SEMICOLON);
    break;
  default:
    unexpectedToken(lookahead);
  }
  System.out.println(indent + "</stat>");
}

static void addExpr(String indent) {
  System.out.println(indent + "<addExpr>");
  mulExpr(indent + "  ");
  switch(lookahead.kind) {
  case PLUS:
    match(indent + "  ", PLUS);
    addExpr(indent + "  ");
    break;
  case MINUS:
    match(indent + "  ", MINUS);
    addExpr(indent + "  ");
    break;
  }
```

```
  System.out.println(indent + "</addExpr>");
}

static void mulExpr(String indent) {
  System.out.println(indent + "<mulExpr>");
  unaExpr(indent + "  ");
  switch(lookahead.kind) {
  case STAR:
    match(indent + "  ", STAR);
    mulExpr(indent + "  ");
    break;
  case SLASH:
    match(indent + "  ", SLASH);
    mulExpr(indent + "  ");
    break;
  }
  System.out.println(indent + "</mulExpr>");
}

static void unaExpr(String indent) {
  System.out.println(indent + "<unaExpr>");
  if (lookahead.kind == MINUS) {
    match(indent + "  ", MINUS);
  }
  priExpr(indent + "  ");
  System.out.println(indent + "</unaExpr>");
}

static void priExpr(String indent) {
  System.out.println(indent + "<priExpr>");
  switch (lookahead.kind) {
  case LPAREN:
    match(indent + "  ", LPAREN);
    addExpr(indent + "  ");
    match(indent + "  ", RPAREN);
    break;
  case IDENTIFIER:
    match(indent + "  ", IDENTIFIER);
    break;
  case NUMBER:
    match(indent + "  ", NUMBER);
    break;
  default:
    unexpectedToken(lookahead);
  }
  System.out.println(indent + "</priExpr>");
}
```

```
  static void unexpectedToken(Token lookahead) {
    System.out.println("> at line " + lookahead.beginLine +
              ", column " + lookahead.beginColumn);
    System.out.println("> unexpected token : " +
              tokenImage[lookahead.kind]);
    System.exit(1);
  }

  static void match(String indent, int tokenKind) {
    if (lookahead.kind == tokenKind) {
      System.out.println(indent + "<match kind=\"" + lookahead.kind +
 "\" image=\"" + lookahead.image + "\" />");
      lookahead = scanner.getNextToken();
    }
    else
      unexpectedToken(lookahead);
  }
}
```

**Aufgabe 2.4** [Modulo] Erweitere die Grammatik aus Beispiel 2.9 mit dem Modulo Operator. Erweitere den zugehörigen Parser dementsprechend.

**Lösung**

```
program ::=   statl <EOF>
          ;
statl   ::=   ( stat )*
          ;
stat    ::=   <IDENTIFIER> "=" addExpr ";"
            | <PRINT> "(" addExpr ")" ";"
          ;
addExpr ::=   mulExpr (("+" | "-") addExpr )?
          ;
mulExpr ::=   unaExpr (("%" | "*" | "/") mulExpr ) ?
          ;
unaExpr ::=   ( "-" )? priExpr
          ;
priExpr ::=   "(" addExpr ")"
            | <IDENTIFIER>
            | <NUMBER>
          ;
```

**Aufgabe 2.5** [Taschenrechner] In einem echten Taschenrechner erfolgt die Ausgabe automatisch und nicht mit Hilfe einer `print` Funktion wie es im Beispiel 2.9. Wir wollen nun unsere Applikation so modifizieren, dass die Ausgabe des Resultats beim drücken der `<NEWLINE>` Taste erfolgt. Die Interaktive Arbeit mit dem Taschenrechner ist in dieser Form erwünscht:

```
> x = 5
5
> y = 2 + 3
5
> (x - 1) * (y + 1)
24
```

Zu diesem Zweck könnte die Grammatik folgendermassen modifiziert werden:

```
program ::=   statl <EOF>
          ;
statl   ::=   ( stat )*
          ;
stat    ::=   <IDENTIFIER> "=" addExpr <NEWLINE>
            | addExpr <NEWLINE>
          ;
addExpr ::=   mulExpr ( "+" addExpr | "-" addExpr )?
          ;
mulExpr ::=   unaExpr ( "*" mulExpr | "/" mulExpr ) ?
          ;
unaExpr ::=   ( "-" )? priExpr
          ;
priExpr ::=   "(" addExpr ")"
            | <IDENTIFIER>
            | <NUMBER>
         ;
```

Erklaeren Sie, welche Probleme mit dieser Grammatik auftreten

Wie können Sie diese Probleme lösen?

**Lösung**

1. Fall die erste Eingabe eine Variable ist, ist es nicht klar, ob diese dei linke Seite einer Zuweisung oder der Anfang eines additiven Ausdruck ist, d.h.`<ID>` gehört sowohl zu `first(addexpr<NEWLINE>)` als auch zu `first(<ID> "=" addExpr <NEWLINE>)`,

2. Das Problem lässt sich am elegantesten ohne Veränderung der Grammatik lösen. Tritt das Token `<ID>` als erstes Token bei der Verarbeitung von `stat` auf, so ist der Entscheid mit hilfe des übernachsten Tokens möglich. Wir lesen also hier 2 Tokens zum Voraus.

   Eine Modifikation der Grammatik ist auch möglich. Die neu entstehende Grammatik ist aber kaum lesbar:

   ```
   program     ::=   statl <EOF>
                      ;
   statl       ::=   ( stat <NEWLINE> )*
                      ;
   ```

```
stat        ::=   <IDENTIFIER> ( "=" addExpr | restAddexpr  )
        |
                  (
    "-" ( "(" addExpr ")"  | <IDENTIFIER> | <NUMBER> )
  |
                    "(" addExpr ")"
                  |
                    <NUMBER>
                  ) restAddExpr
                ;
addExpr     ::=   mulExpr ( "+" addExpr | "-" addExpr )?
                ;
mulExpr     ::=   unaExpr ( "*" mulExpr | "/" mulExpr ) ?
                ;
unaExpr     ::=   ( "-" )? priExpr
                ;
priExpr     ::=   "(" addExpr ")"
                | <IDENTIFIER>
                | <NUMBER>
;
restAddExpr ::=   restMulExpr ( "+" addExpr | "-" addExpr )?
                ;
restMulExpr ::=   ( "*" mulExpr | "/" mulExpr ) ?
                ;
```

Umsetzung mit JAVACC (LOOKAHEAD):

JAVACC-Spezifikation `treol`:

```
PARSER_BEGIN(Parser)

public class Parser {
  public static void main (String args []) {
    Parser parser = new Parser(System.in);
    try {
      parser.program();
      System.out.println("programm is correct");
    }
    catch (Exception e) {
      System.out.println("parse error");
    }
  }
}

PARSER_END(Parser)
```

```
TOKEN:
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < ASSIGN: "=" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < PRINT: "print" >
| < IDENTIFIER:  ["a"-"z","A"-"Z"]
                 ( ["a"-"z","A"-"Z","0"-"9"] )* >
| < NUMBER: ( ["0"-"9"] )+ >
}

TOKEN:
{
  <EOL: ( "\r" )? "\n" >
}

SKIP:
{
  " " | "\t" | "\f"
}

void program() : {}
{
  statl()  <EOF>
}

void statl() : {}
{
  ( stat() <EOL> )*
}

void stat() : {}
{
  LOOKAHEAD(2)
  <IDENTIFIER> <ASSIGN> addExpr()
| addExpr()
}

void addExpr() : {}
{
  mulExpr() (( <PLUS> | <MINUS> ) addExpr() )?
}
```

```
void mulExpr() : {}
{
  unaExpr() (( <STAR> | <SLASH> ) mulExpr() ) ?
}

void unaExpr() : {}
{
  ( <MINUS> )? priExpr()
}

void priExpr() : {}
{
  <LPAREN> addExpr() <RPAREN>
| <IDENTIFIER>
| <NUMBER>
}
```

Umsetzung mit JAVACC (faktorisierte Grammatik):

JAVACC-Spezifikation `trfakt`:

```
PARSER_BEGIN(Parser)

public class Parser {
  public static void main (String args []) {
    Parser parser = new Parser(System.in);
    try {
      parser.program();
      System.out.println("programm is correct");
    }
    catch (Exception e) {
      System.out.println("parse error");
    }
  }
}

PARSER_END(Parser)

TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < SEMICOLON: ";" >
| < ASSIGN: "=" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
```

```
| < NEWLINE: "\n" >
| < IDENTIFIER:  ["a"-"z","A"-"Z"]
                 ( ["a"-"z","A"-"Z","0"-"9"] )* >
| < NUMBER: ( ["0"-"9"] )+ > }

SKIP :
{
  " " | "\t" | "\r" | "\f"
}

void program() : {}
{
  statl() <EOF>
}

void statl() : {}
{
  ( stat() <NEWLINE> )*
}


void stat() : {}
{
  <IDENTIFIER> ( "=" addExpr() | restAddExpr() )
               |
               (
                 "-" ( "(" addExpr() ")" | <IDENTIFIER> | <NUMBER> )
               |
                 "(" addExpr() ")"
               |
                  <NUMBER>
               ) restAddExpr()
}

void addExpr() : {}
{
  mulExpr() ( "+" addExpr() | "-" addExpr() )?
}

void mulExpr() : {}
{
  unaExpr() ( "*" mulExpr() | "/" mulExpr() ) ?
}

void unaExpr() : {}
{
  ( "-" )? priExpr()
```

```
}

void priExpr() : {}
{
  "(" addExpr() ")"
|
  <IDENTIFIER>
|
  <NUMBER>
}

void restAddExpr() : {}
{
  restMulExpr() ( "+" addExpr() | "-" addExpr() )?
}

void restMulExpr() : {}
{
  ( "*" mulExpr() | "/" mulExpr() ) ?
}
```

**Aufgabe 2.6** [EBNF] Schreibe eine JAVACC-Spezifikation für die EBNF-Notation aus Abschnitt 1.5.1.

**Lösung**

Wir verwenden die Grammatik aus Aufgabe 1.7.

```
ebnf       ::=   ( produktion )*
               ;
produktion ::=   <NONTERMINAL> "::=" choice ";"
               ;
choice     ::=   sequence ( "|" sequence )*
               ;
sequence   ::=   ( primitive )+
               ;
primitive  ::= "(" choice ")" rep
       | <NONTERMINAL>
                | <TOKEN>
                | <LITERAL>
               ;
rep        ::= ( "*" | "+" | "?" )?
    ;
```

Daraus resultiert folgende JAVACC-Umsetzung

JAVACC-Spezifikation `EBNFParser`:

```
PARSER_BEGIN(EBNFParser)

public class EBNFParser {
  public static void main (String args []) {
    EBNFParser parser = new EBNFParser(System.in);
    try {
      parser.ebnf();
      System.out.println("correct EBNF notation");
    }
    catch (Exception e) {
      System.out.println("parse error");
    }
  }
}

PARSER_END(EBNFParser)

TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < SEMICOLON: ";" >
| < IS: "::=" >
| < OR: "|" >
| < STAR: "*" >
| < PLUS: "+" >
| < WHY: "?" >
| < NONTERMINAL: ["a"-"z"] ( ["a"-"z"] | ["A"-"Z"] | ["0"-"9"] )* >
| < TERMINALTOKEN: "<" ["A"-"Z"] ( ["A"-"Z"] | ["0"-"9"] )* ">" >
| < TERMINALSTRING: "\""
                    (   (~["\"","\\","\n","\r"])
                      | (   "\\"
                            (   ["n","t","b","r","f","\\","'","\""]
                              | ["0"-"7"] ( ["0"-"7"] )?
                              | ["0"-"3"] ["0"-"7"] ["0"-"7"]
                            )
                        )
                    )+
                    "\""
  >
}

SKIP :
{
  " "
| "\t"
| "\n"
```

```
| "\r"
| "\f"
}

void ebnf() :
{}
{
  ( production() )* <EOF>
}

void production():
{}
{
  <NONTERMINAL> <IS> choice() <SEMICOLON>
}

void choice() :
{}
{
  sequence() ( <OR> sequence() )*
}

void sequence() :
{}
{
  ( primitive() )+
}

void primitive() :
{}
{
  <NONTERMINAL>
| <TERMINALTOKEN>
| <TERMINALSTRING>
| <LPAREN> choice() <RPAREN> ( <WHY> | <PLUS> | <STAR> )?
}
```

# 3 Semantische Analyse

Einige notwendige Eigenschaften von Programmen sind nicht durch eine kontextfreie Grammatik beschereibbar. Diese Eigenschaften werden durch Prädikate auf Kontextinformation, sog. Kontextbedingungen, beschrieben. Dazu gehören insbesondere (siehe auch [WiMa92a]:

- Die **Gültigkeitsregeln** legen für ein im Programm deklarierte Identifier fest, in welchem Teil des Programms Deklarationen einen Effekt haben.
- Die **Sichtbarkeitsregeln** bestimmen, wo in seinem Gültigkeitsbereich ein Identifier sichtbar bzw. verdeckt ist.
- Die **Deklarierheitseigenschaften** bestimmen etwa, dass zu jedem angewandt auftretenden Bezeichner eine explizite Deklaration gegeben werden muss, und dass Doppeldeklarationen verboten sind.
- Die **Typkonsistenz** eines Programms garantiert, dass zur Ausführungszeit keine Operationen (ausser Eingabeoperationen) auf Operanden angewendet wird, aud die sie von ihren Argumenttypen her nicht passt.

Zur Beschreibung dieser Eigenschaften werden sog. **Attributgrammatiken** (attribute grammars) verwendet (Siehe [WiMa92a]) für eine exakte Definition). Informell werden dabei die Produktionen mit sog. semantischen Aktionen erweitert.

In einer solchen Grammatik wird allen Symbolen Attributen und allen Produktionen Operationen über dieser Attribute angeheftet. Die Attribute besitzen Hilfsinformation als Wert und Steuern die Erzeugung von Zwischencode. Werden bei der top-down Syntaxanalyse Symbole expandiert (bzw. bei der bottom-up Syntaxanalyse Symbole reduziert), so erfolgt gleichzeitig die Ausführung der in den Produktionen spezifizierten Operationen. Dies behinhaltet auch z.B. die Erzeugung von Zwischencode. Die Operationen über den Attributen nennt man auch semantische Aktionen und die Vorgehensweise allgemein syntaxgesteuerte übersetzung.

**Notation 3.1 (Attribute)** *Sei $x \in V$ ein Grammatiksymbol, so bezeichnen wir die zu $x$ Attribute $\alpha$ , $\beta$ , etc. mit $x.\alpha$ , $x.\beta$ , etc.*

**Definition 3.1 (Attributtypen)**

1. ***Synthetisierte Attribute*** *(synthetized attributes). Hier werden die Attributwerte der linken Seite einer Produktion aus den Attributwerten der rechten Seite berechnet, z.B.: Für die Produktion $a ::= xy$; und Attrribute $a.\alpha$ , $x.\alpha$ , $y.\alpha$ gilt $a.\alpha = f(x.\alpha, y.\alpha)$ , wobei $f$ eine hier nicht weiter interessierende Funktion bezeichne*
2. ***Ererbte Attribute*** *(inherited attributes). Die Attribute der rechten Seite berechnen sich aus der linken Seite, z.B.: Für die Produktion $a ::= xy$; und Attrribute $a.\alpha$ , $x.\alpha$ , $y.\alpha$ gilt $x.\alpha = g(a.\alpha)$ , $y.\alpha = g(a.\alpha)$*

**Notation 3.2 (Semantische Aktion)** *Wir schreiben die semantischen Aktionen in geschweiften Klammerpaaren, d.h.*

```
a:a1 ::= b:b1 a:a2 <D>:d
         { a1.attr = f( b1.attr, a2.attr, d.attr) }
       ;
```

*bedeutet dass die Attribute der linken Seite der produktion* `a::=b a "D";` *als Funktion der Attribute der linken Seite berechnet werden.*

**Beispiel 3.1** [Deklarierheitseigenschaft] Wir wollen nun die Grammatik aus Beispiel 2.9 mit Deklarierheitseigenschaft als Attributgrammatik erweitern:

```
program ::=   statl <EOF>
          ;
statl   ::=   ( stat )*
          ;
stat    ::=   <IDENTIFIER>:i
              { i.decl = true; } "=" addExpr ";"
            | <PRINT> "(" addExpr ")" ";"
          ;
addExpr ::=   mulExpr ( "+" addExpr | "-" addExpr )?
          ;
mulExpr ::=   unaExpr ( "*" mulExpr | "/" mulExpr )?
          ;
unaExpr ::=   ( "-" )? priExpr
          ;
priExpr ::=   "(" addExpr ")"
            | <IDENTIFIER>:i
              { if (! i.decl) error(); }
            | <NUMBER>
          ;
```

In der Praxis wird zu diesem Zweck eine Symboltabelle verwendet:

Die Symboltabelle wird in unserem Beispiel vom Parser gesteuert. Ein Symboltabelleneintrag besteht aus dem Namen des Lexems und der zugehörigen Registernummer. Die Datenstruktur für die Symboltabelleneinträge hat folgende Gestalt:

JAVA-Klasse `Entry`:

```
class Entry {
    String lexem;
    int  address;
    Entry( String lexem, int  address) {
      this.lexem = lexem;
      this.address = address;
   }
}
```

Die Symboltabelle wird als Hashtabelle von `Entry`-Objekten mit Schlüssel `lexem` realisiert werden.

JAVA-Klasse `Parser`:

```
import java.util.Hashtable;

class Parser implements ScannerConstants {

  static Token  lookahead;
  static Scanner scanner;
  static Hashtable symbolTable = new Hashtable();
  static int adress = 0;

  public static void main (String args[]) {
    scanner = new Scanner(System.in);
    lookahead = scanner.getNextToken();
    program();
    System.out.println("> parsing succesfull");
  }

  static void program() {
    statl();
    match(EOF);
  }

  static void statl() {
    while (lookahead.kind != EOF)
      stat();
  }

  static void stat() {
    switch(lookahead.kind) {
    case IDENTIFIER:
      if (symbolTable.get(lookahead.image) == null)
        symbolTable.put(lookahead.image,
                new Entry(lookahead.image,adress++));
      match(IDENTIFIER);
      match(ASSIGN);
      addExpr();
      match(SEMICOLON);
      break;
    case PRINT :
      match(PRINT);
      match(LPAREN);
      addExpr();
      match(RPAREN);
      match(SEMICOLON);
```

```
    break;
  default:
    unexpectedToken(lookahead);
  }
}

static void addExpr() {
  mulExpr();
  switch(lookahead.kind) {
  case PLUS:
    match(PLUS);
    addExpr();
    break;
  case MINUS:
    match(MINUS);
    addExpr();
    break;
  }
}

static void mulExpr() {
  unaExpr();
  switch(lookahead.kind) {
  case STAR:
    match(STAR);
    mulExpr();
    break;
  case SLASH:
    match(SLASH);
    mulExpr();
    break;
  }
}

static void unaExpr() {
  if (lookahead.kind == MINUS) {
    match(MINUS);
  }
  priExpr();
}

static void priExpr() {
  switch (lookahead.kind) {
  case LPAREN:
    match(LPAREN);
    addExpr();
    match(RPAREN);
```

```
        break;
      case IDENTIFIER:
        if (symbolTable.get(lookahead.image) == null) {
          System.out.println("> at line " +
                             lookahead.beginLine +
                             ", column " +
                             lookahead.beginColumn);
          System.out.println("> identifier \"" +
                             lookahead.image +
                             "\" not declared");
        }
        match(IDENTIFIER);
        break;
      case NUMBER:
        match(NUMBER);
        break;
      default:
        unexpectedToken(lookahead);
      }
    }

    static void unexpectedToken(Token lookahead) {
      System.out.println("> at line " +
                         lookahead.beginLine +
                         ", column " +
                         lookahead.beginColumn);
      System.out.println("> unexpected token : " +
                 tokenImage[lookahead.kind]);
      System.exit(1);
    }

    static void match(int tokenKind) {
      if (lookahead.kind == tokenKind)
        lookahead = scanner.getNextToken();
      else
        unexpectedToken(lookahead);
    }
  }
```

## 3.1 Synthetisierte Attribute

Synthetisierte Attribute werden in der Praxis häufig benutzt. Ein Parsebaum wird bewertet, indem die Semantikregeln für die Attribute an jedem Knoten von unten nach oben, also von den Blättern zur Wurzel, ausgewertet werden. Beim Top-Down-Parsing erfolgt es, indem die Methoden, die den Nichtterminalsymbole entsprechen, mit Rückgabewerte versehen werden.

**Beispiel 3.2** [Anzahl Variablen] Wir wollen nun in der Grammatik aus Beispiel 2.9 zählen,

wieviele Variablen (Wiederholungen mitgezählt) ein Ausdruck besitzt. Zu diesem Zweck verwenden wir einen synthetisierten Attribut `n`.

```
program   ::=   statl <EOF>
            ;
statl     ::=   ( stat )*
            ;
stat      ::=   <IDENTIFIER>  "=" addExpr:a ";"
                { print(a.n + 1); }
              | <PRINT> "(" addExpr:a ")" ";"
                { print(a.n); }
            ;
0addExpr:a ::=   mulExpr:m  { a.n = m.n; }
                (  "+" addExpr:a1  { a.n += a1.n; }
                 | "-" addExpr:a2  { a.n += a2.n; }
                )?
            ;
mulExpr:m ::=   unaExpr:u { m.n = u.n; }
                (  "*" mulExpr:m1 { m.n += m1.n; }
                 | "/" mulExpr:m2 { m.n += m2.n; }
                )?
            ;
unaExpr:u ::=   ( "-" )? priExpr:p { u.n = p.n; }
            ;
priExpr:p ::=   "(" addExpr:a ")"  { p.n = a.n; }
              | <IDENTIFIER>    { p.n = 1; }
              | <NUMBER>        { p.n = 0; }
        ;
```

Werte werden mittels Returnvalue der Methoden übergeben.

JAVA-Klasse `Parser`:

```
class Parser implements ScannerConstants {

  static Token  lookahead;
  static Scanner scanner;

  public static void main (String args[]) {
    scanner = new Scanner(System.in);
    lookahead = scanner.getNextToken();
    program();
    System.out.println("> parsing succesfull");
  }

  static void program() {
    statl();
```

```
  match(EOF);
}

static void statl() {
  while (lookahead.kind != EOF)
  stat();
}

static void stat() {
  int n = 0;
  switch(lookahead.kind) {
  case IDENTIFIER:
    match(IDENTIFIER);
    match(ASSIGN);
    n = 1 + addExpr();
    System.out.println("> stat<empty clipboard>ment contains "
                       + n + " variables");
    match(SEMICOLON);
    break;
  case PRINT :
    match(PRINT);
    match(LPAREN);
    n = addExpr();
    System.out.println("> statment contains "
                       + n + " variables");
    match(RPAREN);
    match(SEMICOLON);
    break;
  default:
    unexpectedToken(lookahead);
  }
}

static int addExpr() {
  int n = mulExpr();
  switch(lookahead.kind) {
  case PLUS:
    match(PLUS);
    n = n + addExpr();
    break;
  case MINUS:
    match(MINUS);
    n = n + addExpr();
    break;
  }
  return (n);
}
```

```
static int mulExpr() {
  int n = unaExpr();
  switch(lookahead.kind) {
  case STAR:
    match(STAR);
    n = n + mulExpr();
    break;
  case SLASH:
    match(SLASH);
    n = n + mulExpr();
    break;
  }
  return (n);
}

static int unaExpr() {
  int n = 0;
  if (lookahead.kind == MINUS) {
    match(MINUS);
    n = priExpr();
  }
  else
    n = priExpr();
  return (n);
}

static int priExpr() {
  int n = 0;
  switch (lookahead.kind) {
  case LPAREN:
    match(LPAREN);
    n = addExpr();
    match(RPAREN);
    break;
  case IDENTIFIER:
    match(IDENTIFIER);
    n = 1;
    break;
  case NUMBER:
    match(NUMBER);
    n = 0;
    break;
  default:
    unexpectedToken(lookahead);
  }
  return (n);
```

```
  }

  static void match(int tokenKind) {
    if (lookahead.kind == tokenKind)
      lookahead = scanner.getNextToken();
    else
      unexpectedToken(lookahead);
  }

  static void unexpectedToken(Token lookahead) {
    System.out.println("> at line " + lookahead.beginLine +
       ", column " + lookahead.beginColumn);
    System.out.println("> unexpected token : " +
       tokenImage[lookahead.kind]);
    System.exit(1);
  }
}
```

## 3.2 Ererbte Attribute

Ein ererbtes Attribut ist ein Attribut, dessen Wert an einem Knoten des Parsebaumes in Termen von Attributen des Vorgängers oder der Nachbarn des Knotens definiert ist. Ererbte Attribute sind brauchbar, um die Abhängigkeit eines Programmiersprachenkonstruktes vom Kontext, in dem es auftritt, auszudrücken.

**Beispiel 3.3** [Typdeklaration] Folgende Teilgrammatik kann für die Typengebung in einer Programmiersprache verwendet werden.

```
declaration  ::=   type idlist
                   ;
type         ::=   <INT>
                 | <REAL>
                   ;
idlist       ::=   <IDENTIFIER>; ( "," <IDENTIFIER> )* ";"
                   ;
```

Das Nichtterminal `type` hat ein synthetisiertes Attribut `t`. Die Semantikregel `{ l.in = t.type }` setzt das ererbte Attribut `in` auf den Typ `t` in der Deklaration.

```
declaration:d  ::=   type:t idlist:l { l.in = t.type }
                     ;
type:t         ::=   <INT>:i { t.type = int; }
                   | <REAL>:r { t.type = real; }
                     ;
idlist:l       ::=   <IDENTIFIER>:i1 { i1.type = l.in; }
                     (  "," <IDENTIFIER>:i2
```

```
                        { i2.type = l.in; } )* ";"
                ;
```

Werte werden mittels Parameter der Methoden übergeben.

JAVA-Klasse `Parser`:

```
import java.util.Hashtable;

class Entry {
  String lexem;
  int  type;
  Entry( String lexem, int  type) {
    this.lexem = lexem;
    this.type = type;
   }
}

class Parser implements ScannerConstants {

  static Token  lookahead;
  static Scanner scanner;
  static Hashtable symbolTable = new Hashtable();

  public static void main (String args[]) {
    scanner = new Scanner(System.in);
    lookahead = scanner.getNextToken();
    declaration();
    System.out.println("> parsing succesfull");
  }

  static void declaration() {
    int t = type();
    while (lookahead.kind != EOF) {
      idlist(t);
    }
  }

  static int type() {
    int t = -1;
    switch(lookahead.kind) {
    case INT:
      match(INT);
      t = 0;
      break;
    case REAL:
      match(REAL);
      t = 1;
```

```
    break;
  default:
    unexpectedToken(lookahead);
  }
  return(t);
}

static void idlist(int in) {
  message(lookahead,in);
  match(IDENTIFIER);
  while (lookahead.kind == COMMA) {
    match(COMMA);
    message(lookahead,in);
    match(IDENTIFIER);
  }
  match(SEMICOLON);
}

static void message (Token lookahead, int in) {
  if (lookahead.kind == IDENTIFIER) {
    System.out.print("> new ");
    switch (in) {
    case 0:
      System.out.print("int");
      break;
    case 1:
      System.out.print("real");
      break;
    default:
      System.out.print("unknown");
      break;
    }
    System.out.println(" variable " +
                       lookahead.image);
    symbolTable.put(lookahead.image,
                    new Entry(lookahead.image,in));
  }
}

static void unexpectedToken(Token lookahead) {
  System.out.println("> at line " + lookahead.beginLine +
                     ", column " + lookahead.beginColumn);
  System.out.println("> unexpected token : " +
                     tokenImage[lookahead.kind]);
  System.exit(1);
}
```

```
  static void match(int tokenKind) {
    if (lookahead.kind == tokenKind)
      lookahead = scanner.getNextToken();
    else
      unexpectedToken(lookahead);
  }
}
```

## 3.3 Abstrakte Ableitungsbäume

In den obigen Beispiele haben wir gesehen, dass beliebige semantische Operationen schon während dem parsen möglich sind. Solche Programme können aber sehr schnell unübersichtlich werden. Erwünscht ist u.U. eine Trennung zwischen den verschiedenen Phasen (lexikalische Analyse, syntaktische Analyse, semantische Analyse, übersetzung). Eine Lösung besteht in der Generierung eines Ableitungsbaumes (siehe auch [Appe98]).

**Definition 3.2 (Konkreter Ableitungsbaum)** *Es sein* G *eine Grammatik, und* $w$ *ein Satz aus* G *. Ein* ***konkreter Ableitungsbaum*** *für* $w$ *hat genau ein Blatt für jeden Token der Eingabe und einen Knoten für jede Produktion, die während dem Parsen zur Anwendung kommt.*

Ist einmal der konkrete Ableitungsbaum aufgabaut, so stellt man fest, dass er viele redundante Informationen enthält. Es sind insbesondere viele Blätter (e.g. `";"`, `","`) überflüssig, da sie keine wichtige Information mehr enthalten. Ferner sind meistens auch viele Knoten überflüssig, da sie aus Grammatiktransformationen entstanden sind (e.g. Elimination der Linksrekursion, Elimination der Mehrdeutigkeit). Solche Details sind für die semantische Analyse irrelevant.

**Definition 3.3 (Abstrakter Ableitungsbaum)** *Ein* ***abstrakter Ableitungsbaum*** *ist eine saubere Schnittstelle zwischen dem Parser und die nächste Phase der Kompilation die keine redundante Information enthält.*

Zu jedem abstrakten Ableitungsbaum gehört eine **abstrakte Syntax**. Diese abstrakte Syntaxist zwar zum Parsen ungeeignet (sie ist meistens Mehrdeutigkeiten). Sie ist aber zusammen mit dem abstrakten Ableitungsbaum für die weiteren Phasen des Kompilers bestens geeignet.

**Beispiel 3.4** [Abstrakter Ableitungsbaum] Für die Grammatik aus Beispiel 2.9, hat die zugehörige abstrakte Syntax folgende Gestalt:

```
statl      ::=   ( stat )*
              ;
stat       ::=   <IDENTIFIER> "=" expr
                | <PRINT> expr
              ;
expr       ::=   expr
                 ((  "+" | "-" |  "*" | "/"  )
```

```
            expr
           )?
        | "-" expr
        | <IDENTIFIER>
        | <NUMBER>
    ;
```

Das Kompositum [Gamm95a], [Gamm96a] ist als Entwurfsmuster besonders geeignet zur Modellierung eines Baumes. Da die Operationen auf dem Baum noch nicht definiert sind, wollen wir noch mit dem Besucher Muster arbeiten. Unserer abstrakten Syntax entspricht folgendes Programmgerüst:

Die Klasse `Node` ist die Schnittstelle aller Knoten im abstrakten Syntaxbaum. Sie definiert insbesondere die abstrakte Methode `accept()` die es dem Besucher erlaubt, die Knoten des Baumes zu verarbeiten.

JAVA-Klasse `Node`:

```
abstract class Node {
  abstract Object accept (Visitor v, Object o);
}
```

Folgende Unterklassen von `Node` implementieren die Knoten des des abstrakten Syntaxbaums. Man beachte, dass jeder Variante eine Klasse entspricht. Somit werden `if` und `switch` Anweisungen gespart (Die Applikation wird schneller).

JAVA-Klasse `Statl`:

```
import java.util.Vector;

class Statl extends Node {

  Vector v;

  Statl () {
    v = new Vector();
  }

  public void addElement (Node stat) {
    v.addElement(stat);
  }

  Object accept (Visitor v, Object o) {
    return (v.visitStatl (this,o));
  }
}
```

JAVA-Klasse `Assign`:

```
class Assign extends Node {

  Node id, addExpr;

  Assign (Node id, Node addExpr) {
    this.id = id;
    this.addExpr = addExpr;
  }

  Object accept (Visitor v, Object o) {
    return (v.visitAssign (this,o));
  }
}
```

JAVA-Klasse `Print`:

```
class Print extends Node {

  Node addExpr;

  Print (Node addExpr) {
    this.addExpr = addExpr;
  }

  Object accept (Visitor v, Object o) {
    return (v.visitPrint (this,o));
  }
}
```

JAVA-Klasse `Plus`:

```
class Plus extends Node {

  Node mulExpr, addExpr;

  Plus (Node mulExpr, Node addExpr) {
    this.mulExpr = mulExpr;
    this.addExpr = addExpr;
  }

  Object accept (Visitor v, Object o) {
    return (v.visitPlus (this,o));
  }
}
```

JAVA-Klasse `Minus`:

```
class Minus extends Node {

  Node mulExpr, addExpr;

  Minus (Node mulExpr, Node addExpr) {
    this.mulExpr = mulExpr;
    this.addExpr = addExpr;
  }

  Object accept (Visitor v, Object o) {
    return (v.visitMinus (this,o));
  }
}
```

JAVA-Klasse `Times`:

```
class Times extends Node {

  Node unaExpr, mulExpr;

  Times (Node unaExpr, Node mulExpr) {
    this.unaExpr = unaExpr;
    this.mulExpr = mulExpr;
  }

  Object accept (Visitor v, Object o) {
    return (v.visitTimes (this,o));
  }
}
```

JAVA-Klasse `Div`:

```
class Div extends Node {

  Node unaExpr, mulExpr;

  Div (Node unaExpr, Node mulExpr) {
    this.unaExpr = unaExpr;
    this.mulExpr = mulExpr;
  }

  Object accept (Visitor v, Object o) {
    return (v.visitDiv (this,o));
  }
}
```

JAVA-Klasse `Uminus`:

```
class Uminus extends Node {

  Node priExpr;

  Uminus (Node priExpr) {
    this.priExpr = priExpr;
  }

  Object accept (Visitor v, Object o) {
    return (v.visitUminus (this,o));
  }
}
```

JAVA-Klasse `Identifier`:

```
class Identifier extends Node {

  String lexem;

  Identifier (String lexem) {
    this.lexem = lexem;
  }

  Object accept (Visitor v, Object o) {
    return (v.visitIdentifier (this,o));
  }
}
```

JAVA-Klasse `Number`:

```
class Number extends Node {

  String n;

  Number (String n) {
    this.n = n;
  }

  Object accept (Visitor v, Object o) {
    return (v.visitNumber (this,o));
  }
}
```

JAVA-Klasse `Parser`:

```
import java.io.*;
```

```
class Parser implements ScannerConstants {

  static Token  lookahead;
  static Scanner scanner;

  public static void main (String args[])
       throws java.io.IOException{
    scanner = new Scanner(System.in);
    lookahead = scanner.getNextToken();
    Node node = program();
    System.out.println("> parsing succesfull");
    node.accept(new PrintVisitor(),new Integer(0));
    node.accept(new EvalVisitor(),null);
    node.accept(new PostfixVisitor(),null);
    /*
    PrintStream out =
      new PrintStream(new FileOutputStream(new File("a.j")));
    node.accept(new GenVisitor(out),null);
    */
  }

  static Node program() {
    Node op = statl();
    match(EOF);
    return (op);
  }

  static Node statl() {
    Statl n = new Statl();
    while (lookahead.kind != EOF)
      n.addElement(stat());
    return (n);
  }

  static Node stat() {
    Node id, a;
    switch(lookahead.kind) {
    case IDENTIFIER:
      id = new Identifier(lookahead.image);
      match(IDENTIFIER);
      match(ASSIGN);
      a = addExpr();
      match(SEMICOLON);
      return (new Assign(id, a));
    case PRINT :
      match(PRINT);
      match(LPAREN);
```

```
    a = addExpr();
    match(RPAREN);
    match(SEMICOLON);
    return (new Print(a));
  default:
    unexpectedToken(lookahead);
    return(null);
  }
}

static Node addExpr() {
  Node m, a;
  m = mulExpr();
  switch(lookahead.kind) {
  case PLUS:
    match(PLUS);
    a = addExpr();
    return new Plus(m, a);;
  case MINUS:
    match(MINUS);
    a = addExpr();
    return (new Minus(m, a));
  }
  return (m);
}

static Node mulExpr() {
  Node u, m;
  u = unaExpr();
  switch(lookahead.kind) {
  case STAR:
    match(STAR);
    m = mulExpr();
    return (new Times(u, m));
  case SLASH:
    match(SLASH);
    m = mulExpr();
    return (new Div(u, m));
  }
  return (u);
}

static Node unaExpr() {
  if (lookahead.kind == MINUS) {
    match(MINUS);
    return (new Uminus(priExpr()));
  }
```

```
    else
      return (priExpr());
  }

  static Node priExpr() {
    Node n;
    switch (lookahead.kind) {
    case LPAREN:
      match(LPAREN);
      n = addExpr();
      match(RPAREN);
      break;
    case IDENTIFIER:
      n = new Identifier(lookahead.image);
      match(IDENTIFIER);
      break;
    case NUMBER:
      n = new Number(lookahead.image);
      match(NUMBER);
      break;
    default:
      n = null;
      unexpectedToken(lookahead);
    }
    return (n);
  }

  static void match(int tokenKind) {
    if (lookahead.kind == tokenKind)
      lookahead = scanner.getNextToken();
    else
      unexpectedToken(lookahead);
  }

  static void unexpectedToken(Token lookahead) {
    System.out.println("> at line " +
                       lookahead.beginLine +
       ", column " +
                       lookahead.beginColumn);
    System.out.println("> unexpected token : " +
       tokenImage[lookahead.kind]);
    System.exit(1);
  }
}
```

Ein Besucher zur Manipulation des abstrakten Ableitungsbaumes hat folgende Gestalt (beachte dass jedem Knoten eine Besuchmethode entspricht).

JAVA-Klasse `Visitor`:

```
abstract class Visitor {
  abstract Object visitStatl(Statl statl, Object o);
  abstract Object visitAssign(Assign assign, Object o);
  abstract Object visitPrint(Print print, Object o);
  abstract Object visitPlus(Plus plus, Object o);
  abstract Object visitMinus(Minus minus, Object o);
  abstract Object visitTimes(Times times, Object o);
  abstract Object visitDiv(Div div, Object o);
  abstract Object visitUminus(Uminus uminus, Object o);
  abstract Object visitIdentifier(Identifier identifier,
  Object o);
  abstract Object visitNumber(Number number, Object o);
}
```

Will man mit einem Besucher den abstrakten Syntaxbaum auf den Bildschirm ausgeben (mit Indentation je nach Tiefe im Baum), so genügt es, den Baum mit dem Besucher in Preorder zu traversieren. Dabei funktioneier die Rekursion indirekt. E.g. um die beiden Operanden einer Summe zu besuchen wird die Methode `visitPlus()` zuerst ihre `accept()` Methoden aufrufen. Diese machen nichts anderes als anschliessend die geeignete Besuchermethode aufzurufen. Dabei können Parameter mit Hilfe der `Object` Argumenten und der Rückgabewerte übergeben werden.

JAVA-Klasse `PrintVisitor`:

```
import java.util.Vector;

class PrintVisitor extends Visitor {

  void indent (int j) {
    for (int i = 0; i < j; i++) System.out.print(" ");
  }

  Object visitStatl(Statl statl, Object o) {
    int i = ((Integer) o).intValue();
    indent(i);
    System.out.println("Statl");
    for (int j = 0; j < statl.v.size(); j++)
      ((Node)
       (statl.v.elementAt(j))).accept(this,new Integer(i+1));
    return(null);
  }

  Object visitAssign(Assign assign, Object o) {
    int i = ((Integer) o).intValue();
    indent(i);
    System.out.println("Assign");
```

```
  assign.id.accept(this,new Integer(i+1));
  assign.addExpr.accept(this,new Integer(i+1));
  return(null);
}

Object visitPrint(Print print, Object o) {
  int i = ((Integer) o).intValue();
  indent(i);
  System.out.println("Print");
  print.addExpr.accept(this,new Integer(i+1));
  return(null);
}

Object visitPlus(Plus plus, Object o) {
  int i = ((Integer) o).intValue();
  indent(i);
  System.out.println("Plus");
  plus.mulExpr.accept(this,new Integer(i+1));
  plus.addExpr.accept(this,new Integer(i+1));
  return(null);
}

Object visitMinus(Minus minus, Object o) {
  int i = ((Integer) o).intValue();
  indent(i);
  System.out.println("Minus");
  minus.mulExpr.accept(this,new Integer(i+1));
  minus.addExpr.accept(this,new Integer(i+1));
  return(null);
}

Object visitTimes(Times times, Object o) {
  int i = ((Integer) o).intValue();
  indent(i);
  System.out.println("Times");
  times.unaExpr.accept(this,new Integer(i+1));
  times.mulExpr.accept(this,new Integer(i+1));
  return(null);
}

Object visitDiv(Div div, Object o) {
  int i = ((Integer) o).intValue();
  indent(i);
  System.out.println("Div");
  div.unaExpr.accept(this,new Integer(i+1));
  div.mulExpr.accept(this,new Integer(i+1));
  return(null);
```

```
  }

  Object visitUminus(Uminus uminus, Object o) {
    int i = ((Integer) o).intValue();
    indent(i);
    System.out.println("Uminus");
    uminus.priExpr.accept(this,new Integer(i+1));
    return(null);
  }

  Object visitIdentifier(Identifier id, Object o) {
    int i = ((Integer) o).intValue();
    indent(i);
    System.out.println("Identifier " + id.lexem);
    return(null);
  }

  Object visitNumber(Number number, Object o) {
    int i = ((Integer) o).intValue();
    indent(i);
    System.out.println("Number " + number.n);
    return(null);
  }
}
```

### 3.3.1 Interpreter für abstrakte Syntaxbäume

Um einen Interpreter zu schreiben genügt es, den abstrakten Ableitungsbaum in postorder zu traversieren. Bei einem Blatt wird der Wert entweder direkt (`<NUMBER>`) oder aus der Symboltabelle (`<IDENTIFIER>`) abgelesen und dem Vaterknoten weitergeleitet. Bei einem Operator-Knoten werden die Werte der Operanden aus den entsprechenden Unterbäumen geholt, gemäss Operator kombiniert und anschliessend dem Vaterknoten weitergeleitet. Der Zuweisungsknoten holt den Wert des Ausdrucks aus einem Unterbaum, den Namen der Variable aus dem anderen und führt den entsprechenden Eintrag in die Symboltabelle aus. Ein Besucher zur Interpretation des Baumes hat folgende Gestalt:

JAVA-Klasse `EvalVisitor`:

```
import java.util.Hashtable;

class EvalVisitor extends Visitor {

  private Hashtable symbolTable = new Hashtable();

  Object visitStatl(Statl statl, Object o) {
    for (int j = 0; j < statl.v.size(); j++)
      ((Node) (statl.v.elementAt(j))).accept(this,null);
```

```
  return (null);
}

Object visitAssign(Assign assign, Object o) {
  Identifier id = (Identifier) assign.id;
  Integer value = (Integer) assign.addExpr.accept(this,null);
  if (symbolTable.get(id.lexem) == null)
    symbolTable.put(id.lexem,
    new Entry(id.lexem,value.intValue()));
  else {
    Entry e = (Entry) symbolTable.get(id.lexem);
    e.value =  value.intValue();
  }
  return (null);
}

Object visitPrint(Print print, Object o) {
  Integer value = (Integer) print.addExpr.accept(this,null);
  System.out.println(" > " + value.toString());
  return (null);
}

Object visitPlus(Plus plus, Object o) {
  int i1 =
    ((Integer) plus.mulExpr.accept(this,null)).intValue();
  int i2 =
    ((Integer) plus.addExpr.accept(this,null)).intValue();
  return (new Integer(i1 + i2));
}

Object visitMinus(Minus minus, Object o) {
  int i1 =
    ((Integer) minus.mulExpr.accept(this,null)).intValue();
  int i2 =
    ((Integer) minus.addExpr.accept(this,null)).intValue();
  return (new Integer(i1 - i2));
}

Object visitTimes(Times times, Object o) {
  int i1 =
    ((Integer) times.unaExpr.accept(this,null)).intValue();
  int i2 =
    ((Integer) times.mulExpr.accept(this,null)).intValue();
  return (new Integer(i1 * i2));
}

Object visitDiv(Div div, Object o) {
```

```
      int i1 =
        ((Integer) div.unaExpr.accept(this,null)).intValue();
      int i2 =
        ((Integer) div.mulExpr.accept(this,null)).intValue();
      return (new Integer(i1 / i2));
    }

    Object visitUminus(Uminus uminus, Object o) {
      int i =
        ((Integer) uminus.priExpr.accept(this,null)).intValue();
      return (new Integer(-i));
    }

    Object visitIdentifier(Identifier id, Object o) {
      int value = 0;
      if (symbolTable.get(id.lexem) == null) {
        System.out.println(" > identifier \"" + id.lexem +
  "\" not declared");
        System.exit(1);
      }
      else {
        Entry e = (Entry) symbolTable.get(id.lexem);
        value = e.value;
      }
      return (new Integer(value));
    }

    Object visitNumber(Number number, Object o) {
      return (new Integer(number.n));
    }
  }
```

## 3.4 Aufgaben

**Aufgabe 3.1** [Tracing] Erweitern Sie die Grammatik aus Beispiel 2.9 mit einem Attribut `indent`. Mit Hilfe von `indent` soll eine formatierte Ausgabe des Ableitungabaumes möglich sein.

**Lösung**

Siehe Aufgabe 2.3.

**Aufgabe 3.2** [Taschenrechner 1] Vereinfachen Sie die Grammatik aus Beispiel 2.9 indem Sie alle Variablen verbieten. Ferner ersetzen Sie `print` durch `addExpr`. Somit erhalten Sie eine Grammatik mit der Funktionalität eines (echten) Taschenrechners. Erweitern Sie die Grammatik mit einem Attribut `n` zur Berechnung der Werte der Eingaben. Nach jeder Eingabe soll das Resultat ausgegeben werden. Schreiben Sie anschliessend ein entsprechendes JAVA programm.

**Lösung**

```
program ::=   statl <EOF>
          ;
statl   ::=   ( stat )*
          ;
stat    ::=   addExpr <NEWLINE>
          ;
addExpr ::=   mulExpr ( "+" addExpr | "-" addExpr )?
          ;
mulExpr ::=   unaExpr ( "*" mulExpr | "/" mulExpr )?
          ;
unaExpr ::=   ( "-" )? priExpr
          ;
priExpr ::=   "(" addExpr ")"
            | <NUMBER>
         ;
```

JAVACC-Spezifikation `Scanner`:

```
PARSER_BEGIN(Scanner)

public class Scanner { }

PARSER_END(Scanner)

TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < EOL: "\n" >
| < ASSIGN: "=" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < NUMBER:   (["0"-"9"])+ >
}

SKIP :
{
  " "
| "\t"
| "\r"
| "\f"
}
```

JAVA-Klasse `Parser`:

```
class Parser implements ScannerConstants {

  static Token  lookahead;
  static Scanner scanner;

  public static void main (String args[]) {
    scanner = new Scanner(System.in);
    System.out.print("> ");
    System.out.flush();
    lookahead = scanner.getNextToken();
    program();
    System.out.println("parsing succesfull");
  }

  static void program() {
    statl();
    match(EOF);
  }

  static void statl() {
    while (lookahead.kind != EOF) {
      stat();
      match(EOL);
    }
  }

  static void stat() {
    int n = addExpr();
    System.out.print("= " + n + "\n> ");
    System.out.flush();
  }

  static int addExpr() {
    int n =  mulExpr();
    switch(lookahead.kind) {
    case PLUS:
      match(PLUS);
      n += addExpr();
      break;
    case MINUS:
      match(MINUS);
      n -= addExpr();
      break;
    }
    return (n);
```

```
}

static int mulExpr() {
  int n = unaExpr();
  switch(lookahead.kind) {
  case STAR:
    match(STAR);
    n *= mulExpr();
    break;
  case SLASH:
    match(SLASH);
    n /= mulExpr();
    break;
  }
  return (n);
}

static int unaExpr() {
  int n = 0;
  if (lookahead.kind == MINUS) {
    match(MINUS);
    n = -priExpr();
  }
  else
    n = priExpr();
  return (n);
}

static int priExpr() {
  int n = 0;
  switch (lookahead.kind) {
  case LPAREN:
    match(LPAREN);
    n = addExpr();
    match(RPAREN);
    break;
  case NUMBER:
    n = Integer.parseInt(lookahead.image);
    match(NUMBER);
    break;
  default:
    unexpectedToken(lookahead);
  }
  return (n);
}

static void match(int tokenKind) {
```

```
    if (lookahead.kind == tokenKind)
      lookahead = scanner.getNextToken();
    else
      unexpectedToken(lookahead);
  }

  static void unexpectedToken(Token lookahead) {
    System.out.println("> at line " + lookahead.beginLine +
       ", column " + lookahead.beginColumn);
    System.out.println("> unexpected token : " +
       tokenImage[lookahead.kind]);
    System.exit(1);
  }
}
```

**Aufgabe 3.3** [Taschenrechner 2] Der Taschenrechner aus Aufgabe 3.2 liefert *falsche* Resultate. Warum? Was kann man dagegen unternehmen?

**Lösung**

```
program ::=   statl <EOF>
          ;
statl   ::=   ( stat )*
          ;
stat    ::=   addExpr <NEWLINE>
          ;
addExpr ::=   mulExpr ( "+" mulExpr | "-" mulExpr )?
          ;
mulExpr ::=   unaExpr ( "*" unaExpr | "/" unaExpr )?
          ;
unaExpr ::=   ( "-" )? priExpr
          ;
priExpr ::=   "(" addExpr ")"
            | <NUMBER>
         ;
```

JAVACC-Spezifikation `Scanner`:

```
PARSER_BEGIN(Scanner)

public class Scanner { }

PARSER_END(Scanner)

TOKEN :
{
  < LPAREN: "(" >
```

```
| < RPAREN: ")" >
| < EOL: "\n" >
| < ASSIGN: "=" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < NUMBER:   (["0"-"9"])+ >
}

SKIP :
{
  " "
| "\t"
| "\r"
| "\f"
}
```

JAVA-Klasse `Parser`:

```
class Parser implements ScannerConstants {

  static Token  lookahead;
  static Scanner scanner;

  public static void main (String args[]) {
    scanner = new Scanner(System.in);
    System.out.print("> ");
    System.out.flush();
    lookahead = scanner.getNextToken();
    program();
    System.out.println("parsing succesfull");
  }

  static void program() {
    statl();
    match(EOF);
  }

  static void statl() {
    while (lookahead.kind != EOF) {
      stat();
      match(EOL);
    }
  }

  static void stat() {
```

```
  int n = addExpr();
  System.out.print("= " + n + "\n> ");
  System.out.flush();
}

static int addExpr() {
  int n =  mulExpr();
  while (lookahead.kind==PLUS || lookahead.kind==MINUS) {
    switch(lookahead.kind) {
    case PLUS:
      match(PLUS);
      n += mulExpr();
      break;
    case MINUS:
      match(MINUS);
      n -= mulExpr();
      break;
    }
  }
  return (n);
}

static int mulExpr() {
  int n = unaExpr();
  while (lookahead.kind==STAR || lookahead.kind==SLASH) {
    switch(lookahead.kind) {
    case STAR:
      match(STAR);
      n *= unaExpr();
      break;
    case SLASH:
      match(SLASH);
      n /= unaExpr();
      break;
    }
  }
  return (n);
}

static int unaExpr() {
  int n = 0;
  if (lookahead.kind == MINUS) {
    match(MINUS);
    n = -priExpr();
  }
  else
    n = priExpr();
```

```
    return (n);
  }

  static int priExpr() {
    int n = 0;
    switch (lookahead.kind) {
    case LPAREN:
      match(LPAREN);
      n = addExpr();
      match(RPAREN);
      break;
    case NUMBER:
      n = Integer.parseInt(lookahead.image);
      match(NUMBER);
      break;
    default:
      unexpectedToken(lookahead);
    }
    return (n);
  }

  static void match(int tokenKind) {
    if (lookahead.kind == tokenKind)
      lookahead = scanner.getNextToken();
    else
      unexpectedToken(lookahead);
  }

  static void unexpectedToken(Token lookahead) {
    System.out.println("> at line " + lookahead.beginLine +
       ", column " + lookahead.beginColumn);
    System.out.println("> unexpected token : " +
       tokenImage[lookahead.kind]);
    System.exit(1);
  }
}
```

**Aufgabe 3.4** [Taschenrechner 3] Erweitern Sie den Taschenrechner aus Aufgaben 3.2 und 3.2 mit Variablen.

Die Variablen müssen in einer Tabelle verwaltet werden. In der Tabelle sollen die Werte und Namen der Variablen enthalten sein. Ist eine Variable nicht deklariert, so soll die Applikation eine Warnung erzeugen. Zusätzlich soll die Variable in diesem Falle mit Defaultwert `0` in die Tabelle eingetragen werden (und es wird mit diesem Wert weitergerechnet). Eine Variable gilt als deklariert, wenn sie auf der linken Seite einer Zuweisung steht.

Passen Sie auf, obige Grammatik ist eine LL(2) Grammatik. Um das übernächste Token zu testen, können Sie die JAVACC Methode `getToken()` (Siehe JAVACC Dokumentation) verwenden.

**Lösung**

```
program ::=   statl <EOF>
          ;
statl   ::=   ( stat )*
          ;
stat    ::=   <IDENTIFIER> "=" addExpr <NEWLINE>
            | addExpr <NEWLINE>
          ;
addExpr ::=   mulExpr ( "+" mulExpr | "-" mulExpr )*
          ;
mulExpr ::=   unaExpr ( "*" unaExpr | "/" unaExpr )*
          ;
unaExpr ::=   ( "-" )? priExpr
          ;
priExpr ::=   "(" addExpr ")"
            | <IDENTIFIER>
            | <NUMBER>
         ;
```

JAVA-Klasse `Entry`:

```
class Entry {
    String lexem;
    int  value;
    Entry( String lexem, int value) {
      this.lexem = lexem;
      this.value = value;
   }
}
```

JAVACC-Spezifikation `Scanner`:

```
PARSER_BEGIN(Scanner)

public class Scanner { }

PARSER_END(Scanner)

TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < EOL: "\n" >
| < ASSIGN: "=" >
| < PLUS: "+" >
| < MINUS: "-" >
```

```
| < STAR: "*" >
| < SLASH: "/" >
| < EXP: "exp" >
| < LOG: "log" >
| < IDENTIFIER:  ["a"-"z","A"-"Z"]
                 ( ["a"-"z","A"-"Z","0"-"9"] )* >
| < NUMBER:   (["0"-"9"])+ >
}

SKIP :
{
  " "
| "\t"
| "\r"
| "\f"
}
```

JAVA-Klasse `Parser`:

```
class Parser implements ScannerConstants {

  static java.util.Hashtable symbolTable = new java.util.Hashtable();
  static Token  lookahead;
  static Scanner scanner;

  public static void main (String args[]) {
    scanner = new Scanner(System.in);
    System.out.print("> ");
    System.out.flush();
    lookahead = scanner.getNextToken();
    program();
    System.out.println("parsing succesfull");
  }

  static void program() {
    statl();
    match(EOF);
  }

  static void statl() {
    while (lookahead.kind != EOF) {
      stat();
      match(EOL);
    }
  }

  static void stat() {
```

```
    int n = 0;
    switch(lookahead.kind) {
    case IDENTIFIER:
      String s = lookahead.image;
      if (scanner.getToken(1).kind == ASSIGN) {
        match(IDENTIFIER);
        match(ASSIGN);
        n = addExpr();
        if (symbolTable.get(s) == null)
         symbolTable.put(s,new Entry(s,n));
else ((Entry) symbolTable.get(s)).value = n;
      }
      else {
        n = addExpr();
      }
      break;
    default:
      n = addExpr();
    }
    System.out.print("= " + n + "\n> ");
    System.out.flush();
  }

  static int addExpr() {
    int n =  mulExpr();
    while (lookahead.kind==PLUS || lookahead.kind==MINUS) {
      switch(lookahead.kind) {
      case PLUS:
        match(PLUS);
        n += mulExpr();
        break;
      case MINUS:
        match(MINUS);
        n -= mulExpr();
        break;
      }
    }
    return (n);
  }

  static int mulExpr() {
    int n = unaExpr();
    while (lookahead.kind==STAR || lookahead.kind==SLASH) {
      switch(lookahead.kind) {
      case STAR:
        match(STAR);
        n *= unaExpr();
```

```
      break;
    case SLASH:
      match(SLASH);
      int m = unaExpr();
      if (m==0) {
System.out.println("> division by 0");
n = 0;
      }
      else {
n /= m;
      }
      break;
    }
  }
  return (n);
}

static int unaExpr() {
  int n = 0;
  if (lookahead.kind == MINUS) {
    match(MINUS);
    n = -priExpr();
  }
  else
    n = priExpr();
  return (n);
}

static int priExpr() {
  int n = 0;
  switch (lookahead.kind) {
  case LPAREN:
    match(LPAREN);
    n = addExpr();
    match(RPAREN);
    break;
  case IDENTIFIER:
    if (symbolTable.get(lookahead.image) == null) {
      System.out.println("> identifier \"" +
                         lookahead.image +
                         "\" not declared");
      symbolTable.put(lookahead.image,
              new Entry(lookahead.image,0));
    }
    n = ((Entry) symbolTable.get(lookahead.image)).value;
    match(IDENTIFIER);
    break;
```

```
      case NUMBER:
        n = Integer.parseInt(lookahead.image);
        match(NUMBER);
        break;
      default:
        unexpectedToken(lookahead);
      }
      return (n);
    }

    static void match(int tokenKind) {
      if (lookahead.kind == tokenKind)
        lookahead = scanner.getNextToken();
      else
        unexpectedToken(lookahead);
    }

    static void unexpectedToken(Token lookahead) {
      System.out.println("> at line " + lookahead.beginLine +
         ", column " + lookahead.beginColumn);
      System.out.println("> unexpected token : " +
         tokenImage[lookahead.kind]);
      System.exit(1);
    }
  }
```

**Aufgabe 3.5** [Taschenrechner 4] Erweitern Sie den Taschenrechner aus Aufgabe 3.4 so, dass die Applikation bei einer Division durch `0` das Resultat `0` ergibt.

**Lösung**

JAVA-Klasse `Entry`:

```
class Entry {
    String lexem;
    int  value;
    Entry( String lexem, int value) {
      this.lexem = lexem;
      this.value = value;
   }
}
```

JAVACC-Spezifikation `Scanner`:

```
PARSER_BEGIN(Scanner)

public class Scanner { }
```

```
PARSER_END(Scanner)

TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < EOL: "\n" >
| < ASSIGN: "=" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < EXP: "exp" >
| < LOG: "log" >
| < IDENTIFIER:  ["a"-"z","A"-"Z"]
                 ( ["a"-"z","A"-"Z","0"-"9"] )* >
| < NUMBER:   (["0"-"9"])+ >
}

SKIP :
{
  " "
| "\t"
| "\r"
| "\f"
}
```

JAVA-Klasse `Parser`:

```
class Parser implements ScannerConstants {

  static java.util.Hashtable symbolTable = new java.util.Hashtable();
  static Token  lookahead;
  static Scanner scanner;

  public static void main (String args[]) {
    scanner = new Scanner(System.in);
    System.out.print("> ");
    System.out.flush();
    lookahead = scanner.getNextToken();
    program();
    System.out.println("parsing succesfull");
  }

  static void program() {
    statl();
    match(EOF);
```

```
  }

  static void statl() {
    while (lookahead.kind != EOF) {
      stat();
      match(EOL);
    }
  }

  static void stat() {
    int n = 0;
    switch(lookahead.kind) {
    case IDENTIFIER:
      String s = lookahead.image;
      if (scanner.getToken(1).kind == ASSIGN) {
        match(IDENTIFIER);
        match(ASSIGN);
        n = addExpr();
        if (symbolTable.get(s) == null)
         symbolTable.put(s,new Entry(s,n));
else ((Entry) symbolTable.get(s)).value = n;
      }
      else {
        n = addExpr();
      }
      break;
    default:
      n = addExpr();
    }
    System.out.print("= " + n + "\n> ");
    System.out.flush();
  }

  static int addExpr() {
    int n =  mulExpr();
    while (lookahead.kind==PLUS || lookahead.kind==MINUS) {
      switch(lookahead.kind) {
      case PLUS:
        match(PLUS);
        n += mulExpr();
        break;
      case MINUS:
        match(MINUS);
        n -= mulExpr();
        break;
      }
    }
```

```
  return (n);
}

static int mulExpr() {
  int n = unaExpr();
  while (lookahead.kind==STAR || lookahead.kind==SLASH) {
    switch(lookahead.kind) {
    case STAR:
      match(STAR);
      n *= unaExpr();
      break;
    case SLASH:
      match(SLASH);
      int m = unaExpr();
      if (m==0) {
System.out.println("> division by 0");
n = 0;
      }
      else {
n /= m;
      }
      break;
    }
  }
  return (n);
}

static int unaExpr() {
  int n = 0;
  if (lookahead.kind == MINUS) {
    match(MINUS);
    n = -priExpr();
  }
  else
    n = priExpr();
  return (n);
}

static int priExpr() {
  int n = 0;
  switch (lookahead.kind) {
  case LPAREN:
    match(LPAREN);
    n = addExpr();
    match(RPAREN);
    break;
  case IDENTIFIER:
```

```
        if (symbolTable.get(lookahead.image) == null) {
          System.out.println("> identifier \"" +
                             lookahead.image +
                             "\" not declared");
          symbolTable.put(lookahead.image,
                  new Entry(lookahead.image,0));
        }
        n = ((Entry) symbolTable.get(lookahead.image)).value;
        match(IDENTIFIER);
        break;
      case NUMBER:
        n = Integer.parseInt(lookahead.image);
        match(NUMBER);
        break;
      default:
        unexpectedToken(lookahead);
      }
      return (n);
    }

    static void match(int tokenKind) {
      if (lookahead.kind == tokenKind)
        lookahead = scanner.getNextToken();
      else
        unexpectedToken(lookahead);
    }

    static void unexpectedToken(Token lookahead) {
      System.out.println("> at line " + lookahead.beginLine +
         ", column " + lookahead.beginColumn);
      System.out.println("> unexpected token : " +
         tokenImage[lookahead.kind]);
      System.exit(1);
    }
  }
```

**Aufgabe 3.6** [Taschenrechner 5] Erweitern Sie den Taschenrechner aus Aufgabe 3.5 mit der Exponential- und der Logarithmus-Funktion (Dies bedeutet eine entsprechende Erweiterung der Grammatik).

**Lösung**

```
program ::=   statl <EOF>
          ;
statl   ::=   ( stat )*
          ;
stat    ::=   <IDENTIFIER> "=" addExpr <NEWLINE>
```

```
            | addExpr <NEWLINE>
          ;
addExpr ::=   mulExpr ( "+" mulExpr | "-" mulExpr )*
          ;
mulExpr ::=   unaExpr ( "*" unaExpr | "/" unaExpr )*
          ;
unaExpr ::=   ( "-" )? priExpr
          ;
priExpr ::=   "(" addExpr ")"
     | <EXP> "(" addExpr ")"
     | <LOG> "(" addExpr ")"
             | <IDENTIFIER>
             | <NUMBER>
         ;
```

JAVA-Klasse `Entry`:

```
class Entry {
    String lexem;
    double  value;
    Entry( String lexem, double value) {
      this.lexem = lexem;
      this.value = value;
   }
}
```

JAVACC-Spezifikation `Scanner`:

```
PARSER_BEGIN(Scanner)

public class Scanner { }

PARSER_END(Scanner)

TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < EOL: "\n" >
| < ASSIGN: "=" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < EXP: "exp" >
| < LOG: "log" >
| < IDENTIFIER:  ["a"-"z","A"-"Z"]
```

```
                    ( ["a"-"z","A"-"Z","0"-"9"] )* >
| < NUMBER:   (["0"-"9"])+ ("." (["0"-"9"])*)?
            | ((["0"-"9"])* ".")? (["0"-"9"])+ >
}

SKIP :
{
  " "
| "\t"
| "\r"
| "\f"
}
```

JAVA-Klasse `Parser`:

```
class Parser implements ScannerConstants {

  static java.util.Hashtable symbolTable = new java.util.Hashtable();
  static Token  lookahead;
  static Scanner scanner;

  public static void main (String args[]) {
    scanner = new Scanner(System.in);
    System.out.print("> ");
    System.out.flush();
    lookahead = scanner.getNextToken();
    program();
    System.out.println("parsing succesfull");
  }

  static void program() {
    statl();
    match(EOF);
  }

  static void statl() {
    while (lookahead.kind != EOF) {
      stat();
      match(EOL);
    }
  }

  static void stat() {
    double n = 0;
    switch(lookahead.kind) {
    case IDENTIFIER:
      String s = lookahead.image;
```

```
      if (scanner.getToken(1).kind == ASSIGN) {
        match(IDENTIFIER);
        match(ASSIGN);
        n = addExpr();
        if (symbolTable.get(s) == null)
         symbolTable.put(s,new Entry(s,n));
else ((Entry) symbolTable.get(s)).value = n;
      }
      else {
        n = addExpr();
      }
      break;
    default:
      n = addExpr();
    }
    System.out.print("= " + n + "\n> ");
    System.out.flush();
  }

  static double addExpr() {
    double n =  mulExpr();
    while (lookahead.kind==PLUS || lookahead.kind==MINUS) {
      switch(lookahead.kind) {
      case PLUS:
        match(PLUS);
        n += mulExpr();
        break;
      case MINUS:
        match(MINUS);
        n -= mulExpr();
        break;
      }
    }
    return (n);
  }

  static double mulExpr() {
    double n = unaExpr();
    while (lookahead.kind==STAR || lookahead.kind==SLASH) {
      switch(lookahead.kind) {
      case STAR:
        match(STAR);
        n *= unaExpr();
        break;
      case SLASH:
        match(SLASH);
        double m = unaExpr();
```

```
      if (m==0) {
System.out.println("> division by 0");
n = 0;
      }
      else {
n /= m;
      }
      break;
    }
  }
  return (n);
}

static double unaExpr() {
  double n = 0;
  if (lookahead.kind == MINUS) {
    match(MINUS);
    n = -priExpr();
  }
  else
    n = priExpr();
  return (n);
}

static double priExpr() {
  double n = 0;
  switch (lookahead.kind) {
  case EXP:
    match(EXP);
    match(LPAREN);
    n = Math.exp(addExpr());
    match(RPAREN);
    break;
  case LOG:
    match(LOG);
    match(LPAREN);
    n = Math.log(addExpr());
    match(RPAREN);
    break;
  case LPAREN:
    match(LPAREN);
    n = addExpr();
    match(RPAREN);
    break;
  case IDENTIFIER:
    if (symbolTable.get(lookahead.image) == null) {
      System.out.println("> identifier \"" +
```

```
                              lookahead.image +
                              "\" not declared");
            symbolTable.put(lookahead.image,
                    new Entry(lookahead.image,0));
          }
          n = ((Entry) symbolTable.get(lookahead.image)).value;
          match(IDENTIFIER);
          break;
        case NUMBER:
          n = Double.parseDouble(lookahead.image);
          match(NUMBER);
          break;
        default:
          unexpectedToken(lookahead);
        }
        return (n);
      }

      static void match(int tokenKind) {
        if (lookahead.kind == tokenKind)
          lookahead = scanner.getNextToken();
        else
          unexpectedToken(lookahead);
      }

      static void unexpectedToken(Token lookahead) {
        System.out.println("> at line " + lookahead.beginLine +
           ", column " + lookahead.beginColumn);
        System.out.println("> unexpected token : " +
           tokenImage[lookahead.kind]);
        System.exit(1);
      }
    }
```

**Aufgabe 3.7** [Taschenrechner 6] Der Taschenrechner aus Beispiel 3.4 rechnet von rechts nach links. Modifizieren sie die JAVA-Klasse `Parser` so, dass der Taschenrechner wie üblich von links nach rechts rechnet.

**Lösung**

JAVA-Klasse `Parser`:

```
import java.io.*;

class Parser implements ScannerConstants {

  static Token  lookahead;
  static Scanner scanner;
```

```
public static void main (String args[])
     throws java.io.IOException{
  scanner = new Scanner(System.in);
  lookahead = scanner.getNextToken();
  Node node = program();
  System.out.println("> parsing succesfull");
  node.accept(new PrintVisitor(),new Integer(0));
  node.accept(new EvalVisitor(),null);
}

static Node program() {
  Node op = statl();
  match(EOF);
  return op;
}

static Node statl() {
  Statl n = new Statl();
  while (lookahead.kind != EOF)
    n.addElement(stat());
  return n;
}

static Node stat() {
  Node id, a;
  switch(lookahead.kind) {
  case IDENTIFIER:
    id = new Identifier(lookahead.image);
    match(IDENTIFIER);
    match(ASSIGN);
    a = addExpr();
    match(SEMICOLON);
    return new Assign(id, a);
  case PRINT :
    match(PRINT);
    match(LPAREN);
    a = addExpr();
    match(RPAREN);
    match(SEMICOLON);
    return new Print(a);
  default:
    unexpectedToken(lookahead);
    return null;
  }
}
```

```
static Node addExpr() {
  Node m1, m2;
  m1 = mulExpr();
  while (lookahead.kind==PLUS || lookahead.kind==MINUS) {
    switch(lookahead.kind) {
    case PLUS:
      match(PLUS);
      m2 = mulExpr();
      m1 = new Plus(m1, m2);
      break;
    case MINUS:
      match(MINUS);
      m2 = mulExpr();
      m1 = new Minus(m1, m2);
      break;
    }
  }
  return m1;
}

static Node mulExpr() {
  Node u1, u2;
  u1 = unaExpr();
  while (lookahead.kind==STAR || lookahead.kind==SLASH) {
    switch(lookahead.kind) {
    case STAR:
      match(STAR);
      u2 = unaExpr();
      u1 = new Times(u1, u2);
      break;
    case SLASH:
      match(SLASH);
      u2 = unaExpr();
      u1 = new Div(u1, u2);
      break;
    }
  }
  return u1;
}

static Node unaExpr() {
  if (lookahead.kind == MINUS) {
    match(MINUS);
    return new Uminus(priExpr());
  }
  else
    return priExpr();
```

```
    }

    static Node priExpr() {
      Node n;
      switch (lookahead.kind) {
      case LPAREN:
        match(LPAREN);
        n = addExpr();
        match(RPAREN);
        break;
      case IDENTIFIER:
        n = new Identifier(lookahead.image);
        match(IDENTIFIER);
        break;
      case NUMBER:
        n = new Number(lookahead.image);
        match(NUMBER);
        break;
      default:
        n = null;
        unexpectedToken(lookahead);
      }
      return n;
    }

    static void match(int tokenKind) {
      if (lookahead.kind == tokenKind)
        lookahead = scanner.getNextToken();
      else
        unexpectedToken(lookahead);
    }

    static void unexpectedToken(Token lookahead) {
      System.out.println("> at line " +
                         lookahead.beginLine +
         ", column " +
                         lookahead.beginColumn);
      System.out.println("> unexpected token : " +
         tokenImage[lookahead.kind]);
      System.exit(1);
    }
  }
```

**Aufgabe 3.8** [Taschenrechner 7] Erweitern Sie den Taschenrechner aus Beispiel 3.4 mit einem Postfix-Visitor. Dabei soll jede Anweisung (`stat`) in Postfix-Notation ausgegeben werden.

**Lösung**

JAVA-Klasse `Parser`:

```
import java.util.Vector;

class PostfixVisitor extends Visitor {

  Object visitStatl(Statl statl, Object o) {
    for (int j = 0; j < statl.v.size(); j++)
      ((Node)
       (statl.v.elementAt(j))).accept(this,null);
    return(null);
  }

  Object visitAssign(Assign assign, Object o) {
    assign.addExpr.accept(this,null);
    assign.id.accept(this,null);
    System.out.println("=");
    return(null);
  }

  Object visitPrint(Print print, Object o) {
    print.addExpr.accept(this,null);
    System.out.println("print");
    return(null);
  }

  Object visitPlus(Plus plus, Object o) {
    plus.mulExpr.accept(this,null);
    plus.addExpr.accept(this,null);
    System.out.print("+ ");
    return(null);
  }

  Object visitMinus(Minus minus, Object o) {
    minus.mulExpr.accept(this,null);
    minus.addExpr.accept(this,null);
    System.out.print("- ");
    return(null);
  }

  Object visitTimes(Times times, Object o) {
    times.unaExpr.accept(this,null);
    times.mulExpr.accept(this,null);
    System.out.print("* ");
    return(null);
  }
```

```
  Object visitDiv(Div div, Object o) {
    div.unaExpr.accept(this,null);
    div.mulExpr.accept(this,null);
    System.out.print("/ ");
    return(null);
  }

  Object visitUminus(Uminus uminus, Object o) {
    uminus.priExpr.accept(this,null);
    System.out.print("neg ");
    return(null);
  }

  Object visitIdentifier(Identifier id, Object o) {
    System.out.print(id.lexem + " ");
    return(null);
  }

  Object visitNumber(Number number, Object o) {
    System.out.print(number.n + " ");
    return(null);
  }
}
```

# 4 Parser Generatoren

## 4.1 Einführung

Ein Parser Generator ist ein Programm, das aus einer syntaktischen Spezifikation automatisch einen Parser (Syntaxanalyser) erzeugt. Die Spezifikation basiert meistens auf kontextfreien Grammatiken, die wir in den vorangehenden Abschnitten besprochen haben.

## 4.2 Bekannte Parser Generatoren

In diesem Abschnitt wollen wir ganz kurz die Parser Generatoren BISON, JAVACC und CUP anhand einiger Beispiele vorstellen. Für Details verweisen wir auf die Dokumentation der jeweiligen Produkten.

### 4.2.1 Bison

Der wohl bekannteste Parser Generator ist YACC (yet another compiler compiler) ein Befehl, der auf dem UNIX-System verfügbar ist [LeMa92a]. In diesem Abschnitt werden wir aber BISON von der Free Software Foundation verwenden. Im allgemeinen ist BISON mit YACC kompatibel. BISON hat einfach noch einige Optionen, die YACC nicht kennt. Die Unterschiede zwischen YACC und BISON können etwa in [LeMa92a] nachgelesen werden. Die komplette Beschreibung von BISON ist in [Donn92a] zu finden.

BISON erzeugt einen *shift*/*reduce* Parser(Bottom-Up Parser).

### 4.2.2 JavaCC

JAVACC [6] ist ein Compilerbau-Werkzeug ähnlich FLEX und BISON. Im Unterschied zu FLEX und BISON erzeugt JAVACCJAVA Quellcode (arbeitet also objektorientiert). Im Gegensatz zu BISON erzeugt JAVACC einen rekursiven Parser. Zusätzlich ist der Scanner direkt in JAVACC integriert.

Ausserdem wurde bei der Entwicklung der Syntax für die Grammatik darauf geachtet, diese soweit möglich an die Java Syntax anzulehnen. JJTREE ist eine Erweiterung des JAVACC Pakets um die Fähigkeit, abstrakte Syntaxbäume zu erzeugen. Die hierfür verwendeten Klassen können leicht erweitert werden, um Aktionen auf den Knoten des Syntaxbaumes durchzuführen.

### 4.2.3 CUP

CUP [7] (Constructor of Useful Parsers) wurde 1996 von Scott E. Hudson entwickelt. CUP ist vollständig in der Programmiersprache JAVA geschrieben. CUP ist ein LALR-Parser und

[6] JAVACC urde 1996 von Sriram Sankar (Sun Microsystems) und Sreenivasa Viswanadha (SUNY at Albany) entwickelt. JAVACC ist frei erhältlich unter `http://www.metamata.com`

[7] CUP ist frei erhältlich unter `http://www.cs.princeton.edu/~appel/modern/java/CUP`

besitzt im Wesentlichen dieselbe Funktionalität als BISON.

## 4.3 Tischrechner 1

Als Beispiel wollen wir folgendeGrammatik benutzen.

```
statlist   ::=   ( statement )* <EOF>
               ;
statement  ::=   expr <SEMICOLON>
               ;
expr       ::=   expr <PLUS> expr
               | expr <MINUS> expr
               | expr <STAR> expr
               | expr <SLASH> expr
               | <MINUS> expr
               | <LPAREN> expr <RPAREN>
               | <NUMBER>
               ;
```

Je nach Parser Generator muss diese Grammatik transformiert werden. Z.B. ist in BISON Linksrekursionen erlaubt, da BISON einen botom-up Parser erzeugt, dafür ist sie in JAVACC verboten.

Der in diesem Beispiel generierte Parser testet die Syntax eines Programms und meldet entweder `parsing successful` oder `parse error`.

### 4.3.1 JavaCC Implementierung

JAVACC-Spezifikation `Parser`:

```
PARSER_BEGIN(Parser)

public class Parser {
  public static void main (String args []) {
    Parser parser = new Parser(System.in);
    try {
      parser.program();
    }
    catch (Exception e) {
      System.out.println("parse error");
    }
  }
}

PARSER_END(Parser)
```

```
TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < SEMICOLON: ";" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < NUMBER: ( ["0"-"9"] )+ >
}

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

void program():
{}
{
  ( statement() )* <EOF>
  { System.out.println("parsing successful"); }
}

void statement():
{}
{
  addExpr() <SEMICOLON>
}

void addExpr():
{}
{
  mulExpr() ( ( <PLUS> | <MINUS> ) addExpr() )?
}

void mulExpr():
{}
{
  unaExpr() ( ( <STAR> | <SLASH> ) mulExpr() )?
}
```

```
void unaExpr():
{}
{
  ( <MINUS> )? priExpr()
}


void priExpr():
{}
{
  <LPAREN> addExpr() <RPAREN>
| <NUMBER>
}
```

## 4.4 Tischrechner 2

Bis jetzt ist unser Parser nicht sehr nützlich. Er kann uns nur sagen, ob das eingegebene Programm syntaktisch korrekt ist oder nicht. In diesem Abschnitt wollen wir nun Aktionen und Symbolwerte einführen und so einen kleinen Tischrechner implementieren.

### 4.4.1 JavaCC Implementierung

JAVACC erlaubt keine direkte Umsetzung der Grammatik, da sie linksrekursiv ist.

JAVACC-Spezifikation `Parse`:

```
PARSER_BEGIN(Parser)

public class Parser {
  public static void main (String args []) {
    Parser parser = new Parser(System.in);
    try {
      parser.program();
      System.out.println("parsing successful");
    }
    catch (Exception e) {
      System.out.println("parse error");
    }
  }
}

PARSER_END(Parser)

TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
```

```
| < SEMICOLON: ";" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < NUMBER: ( ["0"-"9"] )+ >
}

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

void program():
{
  int a = 0;
}
{
  ( a = addExpr()
   <SEMICOLON>
   {System.out.println("> " + a);}
  ) *
  <EOF>
}

int addExpr():
{
  int a =0;
  int m = 0;
  Token tok = null;
}
{
  m = mulExpr() ( ( tok=<PLUS> | tok=<MINUS> ) a = addExpr() ) ?
  {
    if (tok == null) return (m);
    else if (tok.kind == PLUS) return (m + a);
    else return (m - a);
  }
}

int mulExpr():
{
  int u = 0;
```

```
  int m = 0;
  Token tok = null;
}
{
  u = unaExpr() ( ( tok=<STAR> | tok=<SLASH> ) m = mulExpr() ) ?
  {
    if (tok == null) return (u);
    else if (tok.kind == STAR) return (u * m);
    else return (u / m);
  }
}

int unaExpr():
{
  int p = 0;
  Token tok = null;
}
{
  ( tok=<MINUS> )? p = priExpr()
  {
    if (tok == null) return (p);
    else return (-p);
  }
}

int priExpr():
{
  int e = 0;
}
{
  <LPAREN> e = addExpr() <RPAREN> {return (e); }
| <NUMBER>
  {
    e=Integer.parseInt(token.image);
    return (e);
  }
}
```

Das Programm läuft korrekt, liefert aber *falsche* Resultate. Die Rechtsrekursion bewirkt, dass von rechts nach links gerechnet wird.

## 4.5 Tischrechner 3

Wir wollen nun unseren Tischrechner so definieren, dass richtig (von links nach rechts) gerechnet wird:

### 4.5.1 JavaCC Implementierung

JAVACC-Spezifikation `Parser`:

```
PARSER_BEGIN(Parser)

public class Parser {
  public static void main (String args []) {
    Parser parser = new Parser(System.in);
    try {
      parser.program();
      System.out.println("parsing successful");
    }
    catch (Exception e) {
      System.out.println("parse error");
    }
  }
}

PARSER_END(Parser)

TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < SEMICOLON: ";" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < NUMBER: ( ["0"-"9"] )+ >
}

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

void program():
{
  int a = 0;
}
{
```

```
  ( a = addExpr()
   <SEMICOLON>
   {System.out.println("> " + a);}
  ) *
  <EOF>
}

int addExpr():
{
  int m1=0, m2 = 0;
}
{
  m1 = mulExpr() ( <PLUS> m2 = mulExpr()  {m1 += m2;}
                 | <MINUS> m2 = mulExpr() {m1 -= m2;}
                 ) * {return (m1);}
}

int mulExpr():
{
  int u1 = 0, u2 = 0;
}
{
  u1 = unaExpr() ( <STAR> u2 = unaExpr() {u1 *= u2;}
                 | <SLASH> u2 = unaExpr() {u1 /= u2;}
                 ) ? {return (u1);}
}

int unaExpr():
{
  int p = 0;
  Token tok = null;
}
{
  ( tok=<MINUS> )? p = priExpr()
  {
    if (tok == null) return (p);
    else return (-p);
  }
}

int priExpr():
{
  int e = 0;
}
{
  <LPAREN> e = addExpr() <RPAREN> {return (e); }
| <NUMBER> {return (Integer.parseInt(token.image)); }
```

```
}
```

## 4.6 Fehlerbehandlung

In allen bisherigen Beispielen haben wir immer Parser ohne Fehlerbehandlung erzeugt. Das heisst, beim ersten Syntaxfehler wird der Parser eine Fehlerfunktion aufrufen und anschliessend das Programm verlassen. Die Fehlerfunktion macht in diesen Beispielen nicht viel anderes als die Meldung `parse error` auszugeben. Man weiss also nicht an welcher Stelle der Fehler passiert ist und auch nicht was für einen Fehler.

Die erzeugte Parser brechen ihre Arbeit ab, sobald der erste Fehler im Eingabestrom erkannt wird. Dies ist in den meisten Fällen nicht erwünscht. Bei einem Compiler möchte man erreichen, dass möglichst alle Fehler in einem Lauf erkannt werden. Das Problem dabei ist, dass nach einem Fehler der Parser sich in einem *unbestimmten* Zustand befindet (Stackzustand, Lookahaed usw). Wird nach einem Fehler mit dem Parsing einfach fortgefahren, so ist die Gefahr gross, dass sehr viele *Folgefehler* entdeckt und gemeldet werden. Um dies zu verhindern, versucht man den Parser zu *resynchronisieren*. Das heisst, man versucht im Eingabestrom einen Punkt zu finden, wo man glaubt, wieder eine korrekte Eingabe zu finden, so dass mit der Syntaxanalyse fortgefahren werden kann.

### 4.6.1 Fehlerbehandlung in JavaCC

Gibt es beim Parsen einen Fehler, so wirft JAVACC automatisch ein `ParseException`. Es genügt dieses Exception in einem `try` abzufangen und im `catch`-Teil alle Tokens bis zum nächsten `<SEMICOLON>` zu konsummieren.

JAVACC-Spezifikation `Parser`:

```
// options { DEBUG_PARSER=true;}

PARSER_BEGIN(Parser)

public class Parser {

  public static void main (String args []) throws ParseException {
    Parser parser = new Parser(System.in);
    parser.program();
  }
}

PARSER_END(Parser)

JAVACODE
void skipTo(int kind) {
  System.out.println("> error (skipping to next " + tokenImage[kind]
  Token t;
  do {
```

```
    t = getNextToken();
  } while (t.kind != kind && t.kind != EOF);
}

TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < SEMICOLON: ";" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < NUMBER: ( ["0"-"9"] )+ >
}

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

void program():
{}
{
  (
    try {
      statement()
    }
    catch (ParseException e) {
      skipTo(SEMICOLON);
    }
  )* <EOF>
}

void statement():
{
  int a;
}
{
  a = addExpr()
  <SEMICOLON>
  { System.out.println("> " + a); }
}
```

```
int addExpr():
{
  int m1 = 0 , m2 = 0;
}
{
  m1 = mulExpr()
  (
    <PLUS>  m2 = mulExpr() {m1 += m2;}
  |
    <MINUS> m2 = mulExpr() {m1 -= m2;}
  ) *
  {return (m1);}

}

int mulExpr():
{
  int u1 = 0, u2 = 0;
}
{
  u1 = unaExpr()
  (
    <STAR>  u2 = unaExpr() {u1 *= u2;}
  |
    <SLASH> u2 = unaExpr(){u1 /= u2;}
  ) *
  {return (u1);}
}

int unaExpr():
{
  int p = 0;
}
{
  <MINUS> p = priExpr() {return (-p); }
|
  p = priExpr() {return (p); }
}

int priExpr():
{
  int e;
}
{
  <LPAREN> e = addExpr() <RPAREN> {return (e); }
|
```

```
  <NUMBER> {return (Integer.parseInt(token.image)); }
}
```

## 4.7 Abstrakte Syntaxbäume

Moderne Compilerbauwerkzeuge unterstützen die Generierung von abstrakten Ableitungsbäume (AST). Dies ist insbesondere der Fall bei JAVACC. JJTREE ist ein JAVACC-Preprozessor zur Erzeugung von AST's.

Um einen abstrakten Syntaxbaum aufbauen zu können, benötigt man Objekte, die als Knoten verwendet werden können. JJTREE stellt verschiedene Möglichkeiten bereit, aus welchen Klassen solche Knotenobjekte erzeugt werden können. Das voreingestellte Verhalten von JJTREE ist es, für jedes Nonterminal einen Knoten vom Typ `SimpleNode` zu erzeugen und in den Syntaxbaum einzuhängen.

- Das Interface `Node` Ein von JJTREE erzeugter Knoten muss immer dieses Interface implementieren. Hier werden die wichtigsten Funktionen eines Knoten vereinbart, wie z.B. `jjtGetParent` oder `jjtAddChild(Node n, int i)`.
- Die Klasse `SimpleNodeSimpleNode` implementiert das Interface `Node`. Man kann mit Hilfe des Knotens `SimpleNode` einen Syntaxbaum erzeugen. Es ist auch möglich, diesen Baum dann mit der Methode `dump()` auszugeben. Jeder `SimpleNode` speichert den Namen des Nonterminals, aus dem er erzeugt wurde.
- Selbstdefinierte Knoten Wenn es für die Abarbeitung des Syntaxbaums wichtig ist, dass die Knoten verschiedenes Verhalten zeigen (z.B. in einer Programmiersprache), dann verwendet man am einfachsten von `SimpleNode` abgeleitete Knoten, die die gewünschte Funktionalität bieten. Man kann in der Grammatikdefinitionsdatei bei jedem Nonterminal angeben, welche Art von Knoten erzeugt werden soll.

Der von JJTREE generierte Code bewirkt beim parsen der Eingabe, dass für jedes Nonterminal ein Knoten gegebenen Typs erzeugt wird. Durch Schachtelung von Nonterminalen entstehen so Baumstrukturen dadurch, dass für jedes geschachtelte Nonterminal ein Unterknoten zum Knoten des umgebenden Nonterminals hinzugefügt wird.

### 4.7.1 JavaCC Implementierung

JAVACC-Spezifikation `Parser`:

```
options {
  MULTI = true;
  VISITOR = true;
}

PARSER_BEGIN(Parser)
  public class Parser {
    public static void main (String args []) {
```

```
      Parser parser = new Parser(System.in);
        try {
        program();
        ((SimpleNode) jjtree.rootNode()).dump(" ");
        } catch (Exception e) {
          e.printStackTrace(); System.exit(1);
        }
    }
  }
PARSER_END(Parser)

TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < SEMICOLON: ";" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < NUMBER: ( ["0"-"9"] )+ >
}

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

void program() #Program :
{}
{
  (statement())*
}

void statement() #Statement :
{}
{
  addExpr() <SEMICOLON>
}

void addExpr() #void :
{}
{
```

```
  ( mulExpr() (( <PLUS> | <MINUS> ) mulExpr())* ) #Add(>1)
}

void mulExpr() #void :
{}
{
  ( unaExpr() (( <STAR> | <SLASH> ) unaExpr() )* ) #Mult(>1)
}

void unaExpr() #void :
{}
{
  <MINUS> priExpr() #Uminus
|
  priExpr()
}


void priExpr() #void :
{
  Token t;
}
{
  <LPAREN> addExpr() <RPAREN>
|
  t=<NUMBER> { jjtThis.setName(t.image); } #Number
}
```

Achtung! Nach dem Lauf mit JJTREE muss die Klasse `ASTNumber` nacheditiert werden. Dabei müssen die Methode `setName` sowie das Attribut `name` hinzugefügt werden.

## 4.8 Aufgaben

**Aufgabe 4.1** [JavaCC] Überlegen Sie sich, wie die Taschenrechner-Programme aus Abschnitt 3 mittels JAVACC spezifiziert werden können.

# 5 Eine Anwendung: XML

## 5.1 Einführung

### 5.1.1 Was ist XML?

- XML steht für eXtensible Markup Language.
- XML ist ein Formalismus, der die Definition HTML-ähnlicher Markup-Sprachen bzw. Daten-Formate erlaubt.
- XML ist eine vereinfachte Fassung von SGML (Standard Generalized Markup Language).
- XML ist ein neues Datenmodell, das in Konkurrenz zum relationalen Datenmodell steht
- Mit XML kann die Verarbeitung von Daten (in Sinne von Datenbanken) und Dokumenten integriert werden.
- XML ist die Grundlage des zukünftigen *semantischen Web*.
- XML ist ein universelles Datenaustausch-Format.

### 5.1.2 Markup-Sprachen

Früher wurde ein Manuskript mit der Schreibmaschine getippt, aber dann für den Setzer mit handschriftlichen Markierungen versehen. Z.B. wurde etwas unterstrichen oder eingekreist und dann am Rand *kursiv* Vermerkt.

Später wurden solche handschriftlichen Markierungen von Textteilen durch entsprechende Kommandos für Programme ersetzt, etwa `\texttt{ ... }` für TEX oder LATEX .

Mit leuchtend-gelben *Textmarkern* werden häufig besonders wichtige Textteile hervorgehoben. Markierungen in Texten dienen also nicht nur der Formatierung beim Drucken.

Charles Goldfarb, der Vater von SGML, sagt, er hätte den Begriff *Markup Language* 1969 erfunden, damit die Abkürzung GML (*Generalized Markup Language*) zu den Erfindern Goldfarb, Mosher und Lorie passt. Dieser Vorläufer von SGML ist in einem Projekt bei IBM entstanden, in dem ein System zur Verwaltung von juristischen Dokumenten entwickelt wurde.

### 5.1.3 Darstellungs-orientiertes und inhaltliches Markup

Zunächst war die Markierungen in den Texten nur Anweisungen für den Schriftsatz, z.B. 18pt-Schrift, kursiv, Einrückung.

Dies ist vernünftig, solange die einzige Verwendung des Textes das Ausdrucken in genau diesem Format ist. Wenn man den Text aber auch auf andere Weisen verarbeiten möchte, reicht dies nicht mehr aus. Zum Beispiel könnte man Kursiv-Schrift für Hervorhebungen

verwenden, aber auch für definierende Vorkommen von Begriffen, und für Namen. Für eine Rechtschreib-Prüfung ist es nun wichtig, dass das Programm Namen erkennt. Für die Erstellung eines Glossars oder Indexes müssen müssen die Definitionen erkannt werden. Für einen Menschen ist es nicht schwierig, diese verschiedenen Verwendungen kursiver Schrift zu unterscheiden, aber Computer sind dumm und brauchen explizite Markierungen. Indem für inhaltlich verschiedene Dinge die gleiche Markierung verwendet wurde, ist es zu einem Informationsverlust gekommen

Texte werden heute also nicht nur zum Ausdrucken verwendet, sondern es sind auch andere Verarbeitungen möglich. Zum Beispiel erfordert auch das automatische Erstellen eines Inhaltsverzeichnisses, dass Kapitelüberschriften klar als solche gekennzeichnet sind, und nicht einfach nur als gross und fett. Besondere Warnungen im Text könnten auch gross und fett sein.

Man muss sich bei der Erstellung des Textes Gedanken über mögliche Verwendungen machen. Alle Textbestandteile, die man gesondert behandeln will, müssen bei der Erfassung entsprechend markiert werden. Die Verwendung des XML-Formates alleine ist noch kein Zaubermittel.

Auch muss man Texte heute häufig auf verschiedenen Medien ausgeben, was unterschiedliche Formatierungen des gleichen Textes erfordert. Z.B. Darstellung im Web-Browser (mit / ohne Java, Grafiken, Frames), Drucken auf Papier (Seiten-Einteilung wichtig, besser ein grösseres Dokument als viele kleine), automatisches Vorlesen für blinde Nutzer (oder Autofahrer), Kurzfassung für Handys (WAP).

Manchmal möchte man auch nur Auszüge eines Dokumentes ausgeben, nicht den ganzen Text. Interaktive Lehrbücher: Textteile werden je nach Kenntnisstand des Lesers weggelassen. *Document Processing*-Kurs in Pittsburgh: Tabelle mit Studenten- Information. Stylesheet druckt Email-Liste. Notizen, Anmerkungen, Versions-Information im Text selbst wird normalerweise nicht mit ausgegeben.

WYSIWYG (*what you see is what you get*): *What you see is all you've got*.

> As programming legend Brian Kernighan once noted, the problem with *What you see is what you get* is that what you see is all you've got [Bos99]

*Write once, use everywhere*.

### 5.1.4 XML als universelles Daten-Austauschformat

Für relationale Datenbanken gibt es bisher kein allgemein akzeptiertes Austausch-Format.

Es wäre viel einfacher, wenn jedes Programm den Export und Import von Daten im XML-Format unterstützen würde. Mit einem Standard-Austauschformat benötigt man für $n$ verschiedene interne Formate nur $2n$ Konvertierungsprogramme. Ansonsten benötigt man $n(n-1)$ Programme, um von jedem Format direkt in jedes andere konvertieren zu können.

Zwar ist XML alleine schon eine Hilfe (weil es viele XML-zu-XML Transformations-Programme gibt, z.B. XSLT-Implementierungen), aber ein vollständige Lösung erfordert auch die Normung der *Tags*. Es gibt zur Zeit viele Bestrebungen, *Document Type Definitions*

(oder *Schemas*) für verschiedene Anwendungsbereiche zu entwickeln und zu standardisieren. Es gibt schon einige DTD-Sammlungen im Web. Siehe [DTD01], [DTD02], [DTD03] ,[DTD04]. Auch für Datenbank-Entwurf interessant: DTDs sind Datenbank-Schemata.

### 5.1.5 XML als Syntax-Analyse-Werkzeug

Bei der Programmierung muss man häufig kleine Sprachen entwerfen und implementieren. Wenn man z.B. ein Programm für Multiple-Choice Tests entwickelt, muss man sich darüber Gedanken machen, wie man die Fragen und möglichen Antworten abspeichern will.

Man kann bei solchen Gelegenheiten immer sein eigenes Datenformat definieren. Aber wenn man es als Anwendung von XML auslegt, hat man gleich Zugriff auf eine grosse Anzahl von XML-Werkzeugen, wie etwa Parser (zur Syntaxanalyse), Syntax-gestützte Editoren, Abfrage- und Transformations-Werkzeuge, und natürlich Werkzeuge zur ansprechenden Darstellung der Daten auf verschiedenen Ausgabemedien, inklusive der Darstellung im Web.

### 5.1.6 Schwächen / Probleme von HTML

HTML hat viele Bestandteile, die nur die Darstellung des Textes im Browser betreffen (etwa Schriftart-Angaben wie `<B>`), und relativ wenige Inhalts-orientierte *Tags*. `<quote><par>` Wie soll z.B. `<center>` in einer Sprachausgabe dargestellt werden? Und wie soll `<blink>` ausgedruckt werden?

HTML ist nur eine einzige Anwendung von SGML, d.h. ein fester Satz von *Tags*. Dies kann natürlich nicht Inhalts-orientiertes Markup für beliebige Anwendungen erlauben. Die möglichen Anwendungen sind einfach zu verschieden. HTML hat einige inhaltliche Tags wie etwa `<address>`, `<title>` und `<kbd>`, auch die verschiedenen Überschriften, Paragraphen usw. können so verstanden werden. Aber Dinge wie `<product>` und `<price>`, die im heutigen Web durchaus häufiger vorkommen, fehlen. Es ist einfach nicht möglich, mit einem festen Satz von Tags beliebige Anwendungen zu unterstützen. Man gelangt dann zu einem kleinsten- gemeinsamen Nenner von Tags (z.B. sollte jedes Dokument einen Titel haben). Die einzige gemeinsame Funktion aller Arten von Dokumenten ist das Ausdrucken, daher führt der feste Satz von Tags auch zu stärker darstellungs-orientiertem Markup.

HTML hat eine sehr lockere DTD (Grammatik), die nur wenig Struktur vorschreibt. Bekannte Struktur ist für die Weiterverarbeitung aber sehr nützlich. Je mehr man über die Struktur weiss, umso komplexere Anfragen sind möglich. Wenn ein Dokument nur eine Folge von Zeichen ist, kann man nur nach Teilstrings suchen. Relationale Datenbanken sind stark strukturiert, und entsprechend ist SQL auch eine viel mächtigere Sprache.

Aber selbst die wenigen Syntax-Regeln, die HTML vorschreibt, werden sehr häufig verletzt, und Browser zeigen den Text normalerweise dennoch ohne Fehlermeldung dar.

### 5.1.7 SGML, HTML, und XML (Historisches)

1969 entwickelten Goldfarb, Mosher und Lorie die Generalized Markup Language (GML) bei IBM (in einem Projekt zur Verwaltung juristischer Dokumente).

1974 entwickelte Goldfarb ein Programm zur Syntaxanalyse, das eine DTD (Document Type Definition, Grammatik) einliest und dann ein Dokument auf syntaktische Korrektheit bezüglich dieser DTD prüft.

1978-1986 war Goldfarb technischer Leiter eines Komitees, das den ISO-Standard 8879 für SGML entwickelte. SGML wurde auch ein de facto Standard zum Austausch grosser, komplexer Dokumente, z.B. Wartungsunterlagen für Flugzeuge oder Test-Unterlagen zur Zulassung neuer Medikamente.

1989 hat Tim Berners-Lee beim CERN ein Projekt vorgeschlagen, das die Grundlage des World Wide Web bildete.> Sein Kollege Anders Berglund hat ihn auf SGML hingewiesen, aber sie entwickelten die Hypertext Markup Language (HTML) zunächst nur anhand von Beispielen. Als schliesslich eine formale DTD für HTML definiert wurde, gab es schon Tausende von inkorrekten HTML-Dokumenten. DTD für HTML 4.01 siehe [HTML], [HTML]. DTD für HTML 3.2 siehe [HTML]. Syntaxprüfer für HTML siehe [HTML].

Die erste HTML-Version hatte überwiegend inhalts-orientierte Tags. Später wurden von den Browser-Herstellern mehr darstellungs-orientierte Tags hinzugefügt. Durch solche Erweiterungen sollten die Anwender an einen bestimmten Browser gebunden werden.

Die Implementierung von SGML war den Browser-Herstellern aber zu kompliziert. Nach langen Verhandlungen einigte man sich auf eine (leicht modifizierte) SGML-Teilmenge, die XML (eXtensible Markup Language) genannt wurde. Die W3C Arbeitsgruppe zur Definition von XML wurde von Jon Bosak (SUN Microsystems) gegründet und geleitet.

Am 10. Februar 1998 wurde die XML 1.0 Empfehlung veröffentlicht (Siehe [XML]).Während XML 1.0 heute stabil ist, sind viele der darauf aufbauenden Standards, z.B. für Stylesheets, noch in Entwicklung. Bücher, die 1998 or 1999 erschienen sind, weisen darauf hin, dass ihre Darstellung von XSL nur auf Vor- Informationen basiert, und die entgültige XSL Norm sich davon recht deutlich unterscheiden kann. Der Implementierung von XSLT im Internet Explorer 5 merkt man manchmal auch an, dass sie sich noch auf einen Vorentwurf bezieht (und mit der schliesslich endgültig verabschiedeten XSLT-Spezifikation nicht genau übereinstimmt).

## 5.2 Well-Formed XML

In diesem und den nächsten Abschnitte wollen wir erklären (definieren), welche Zeichenfolgen syntaktisch korrekte XML-Dokumente sind.

Wir werden die XML-Syntax hier mit Prosa *halb-formal* definieren. Es sei aber ausdrücklich auf die im Web verfügbare XML-Spezifikation verwiesen, die eine kontextfreie Grammatik für XML enthält (plus einige zusätzliche Einschränkungen in Prosa) (Siehe [XML]). Die Grammatik hat insbesamt 89 Produktionen und die ganze XML Spezifikation hat etwa 36 Seiten.

### 5.2.1 Zwei Stufen Syntaktischer Korrektheit

Der XML-Standard definiert zwei Stufen syntaktischer Korrektheit.

**Definition 5.1 (Well-Formed)** *Ein* XML-*Document heisst* ***Well-Formed****, wenn es sich sich ohne* Document Type Definition *(Grammatik) prüfen lässt.*

*Dies beinhaltet u.a. die korrekte Klammerstruktur der Tags*

**Definition 5.2 (Valid)** *Ein* XML*-Document heisst* ***Well-Formed****, wenn die Verwendung der Tags gemäss der* DTD *korrekt ist.*

Viele existierende Parser prüfen nur die Wohlgeformtheit. Dies ist zum Beispiel ausreichend, um die DOM-Datenstruktur aufzubauen, und auch für die Darstellung im Browser (zum Beispiel mit XSL).

Die Gültigkeit bezüglich einer Grammatik ist ein zusätzlicher Integritäts-Test.

Bei der Entwicklung von DTDs schreibt man häufig zunächst ein Beispiel- Dokument auf. Dies ist auch ohne DTD schon syntaktisch korrektes XML.

Theoretisch könnte man XML-Dateien ganz ohne DTDs verwenden, aber die Weiterverarbeitung der Daten wird sehr unsicher und schwierig, wenn man keine DTD hat.

In der XML-Spezifikation definiert Abschnitt 5.1 [XML] die Anforderungen an *validating* und *non- validating*XML-Prozessoren. Die Anforderungen für *well-formed* und *valid*XML sind dadurch geregelt, dass zwar beide die gleiche kontext-freie Grammatik verwenden, aber bei den (kontext-sensitiven) Zusatz-Bedingungen zwischen *well-formedness constraints* (muss jedes XML Dokument erfüllen) und *validity constraints* (müssen nur *valid*XML-Dokumente erfüllen) unterschieden wird.

### 5.2.2 Zeichensatz

Ein XML-Dokument ist eine Folge von Zeichen.

Gültige Zeichen sind `TAB` (tabulator), `CR` (carriage return), `LF` (line feed), und die grafischen Zeichen des Unicode Standards und ISO / IEC 10646 Standards (*Universal Multiple-Octed Coded Character Set*). Zum Beispiel sind Escape und BEL nicht erlaubt.

Man muss die Menge der erlaubten Zeichen von ihrer Codierung trennen. XML erlaubt durchaus verschiedene Codierungen für die XML-Dokumente. Alle XML Prozessoren müssen die UTF-8 und UTF-16 Codierungen des ISO / IEC 10646 Standards verarbeiten können.

Will man z.B. mit ISO Latin 8859 / 1 arbeiten, so schreibt man ganz am Anfang des Dokumentes:

```
<?xml version="1.0" encoding="ISO-8859-1"?>.
```

Allerdings kann es sein, dass ein XML- Prozessor diese Codierung nicht kennt und das Dokument zurückweist.

Zeichensätze werden in der XML-Spezifikation in den Abschnitten 2.2 und 4.3.3 sowie in Anhang B und Anhang F (Siehe [XML]). diskutiert.

### 5.2.3 Namen von Tags (Attributen, Entities, etc.)

Namen (von Elementen oder Attributen etc.) müssen mit einem Buchstaben, einem Unterstrich `"_"` oder einem Doppelpunkt beginnen, und kann danach Buchstaben, Ziffern, einen Unterstrich `"_"`, einen Bindestrich `"-"`, einen Punkt `"."`, oder einen Doppelpunkt `":"`

enthalten. *Combining Characters* und *Extenders* im Sinne des Unicode Standards sind auch erlaubt, nur nicht als erste Zeichen. Der Anhang B des XML Standards enthält eine lange Liste der Unicode-Bereiche, die als Buchstaben etc. gelten. Die deutschen Umlaute gehören natürlich dazu.

Im Gegensatz zu HTML sind Gross- und Kleinschreibung wichtig. `<A>` und `<a>` sind verschiedene Elemente.

Namen (und andere häufiger verwendete syntaktische Konstrukte) werden in der XML Spezifikation in [XML] Abschnitt 2.3 definiert.

### 5.2.4 Syntax und Schachtelung von Tags

**Definition 5.3 (XML-Dokument)** *Ein* XML *Dokument besteht aus Text-Daten und Markup. Die wichtigste Art von Markup sind Tags. Ausserdem gibt es noch Entity-Referenzen, Zeichen-Referenzen, Kommentare, Begrenzungen von* `CDATA` *Abschnitten, Document Type Deklarationen, und Processing Instructions.*

Es gibt öffnende, schliessende, und leere Tags.

**Notation 5.1 (Tags)** *Ein* ***öffnendes Tag*** *besteht aus einem kleiner-Zeichen* `"<"` *(*spitze Klammer auf*), einem Namen wie oben definiert (Name des Element-Typs), optional Attributen (s.u.), und einem grösser-Zeichen* `">"` *(*spitze Klammer zu*).*

*Ein* ***schliessendes Tag*** *besteht aus einem kleiner-Zeichen* `"<"`*, einem Schrägstrich* `"/"`*, einem Namen, und einem grösser-Zeichen* `">"`*.*

*Ein* ***leeres Tag*** *besteht aus einem kleiner-Zeichen* `"<"`*, einem Namen, optionalen Attributen, einem Schrägstrich* `"/"`*, und einem grösser- Zeichen* `">"`*.*

Die Syntax von öffnenden und schliessenden Tags sollte von HTML her bekannt sein (es gibt nur kleine Unterschiede bei den Attributen). Im Unterschied zu HTML muss es in XML zu jedem öffnenden Tag ein passendes schliessendes Tag geben. Öffnende und schliessende Tags treten also immer in Paaren auf. In HTML werden dagegen Tags wie etwa `<br>` und `<IMG>` typischerweise ohne schliessende Tags verwendet.

Öffnende und schliessende Tags müssen korrekt geschachtelt werden: Wenn ein Tag `xx` nach einem Tag `yy` geöffnet wurde, so muss `xx` vor `yy` wieder geschlossen werden. Zum Beispiel ist `<a> <b> </b> </a>` zulässig, aber `<a> <b> </a> </b>` ist syntaktisch nicht korrekt. Das heisst, das zugehörige schliessende Tag ist ein Tag mit dem gleichen Namen und derart, dass gleich viele öffnende und schliessende Tags zwischen den beiden Tags stehen.

Eine weitere Syntax-Regel von XML ist, dass es ein Paar von äussersten Tag geben muss, die alle anderen einschliessen. Vor diesem öffnenden Tag und nach dem zugehörigen schliessenden Tag sind keine anderen Tags und kein Text erlaubt.

Die grundlegenden Syntax-Regeln für Tags sind in der XML Spezifikation in [XML] Abschnitt 2.1 und Abschnitt 3.1 festgelegt.

### 5.2.5 Elemente, Baum-Darstellung von Dokumenten

**Definition 5.4 (Element)** *Öffnendes und zugehöriges schliessendes Tags und alles, was dazwischen steht, ist ein **Element** in* XML. *Entsprechend ist auch jedes leere Tag für sich ein Element*

*Der Name des Tags wird auch **Typ** des Elementes genannt. Der **Inhalt** des Elementes ist das, was zwischen öffnendem und schliessendem Tag steht. Leere Tags heissen so, weil der Inhalt des durch sie beschriebenen Elementes leer ist.*

Die äussersten Tags, die den gesamten Text des Dokumentes und alle anderen Tags einschliessen, entsprechen dem sogenannten Wurzel-Element des Dokumentes. Es wird auch Dokument-Element genannt.

Die Schachtelung der Tags führt dazu, dass die Elemente als Baum angeordnet werden können. Die Wurzel des Baumes ist das Wurzel- oder Dokument-Element (dessen Tags den gesamten Text einschliessen). Jedes Element hat als Kinder die direkt darin geschachtelten Elemente.

Wir können die Baumdarstellung noch etwas verfeinern, indem wir auch Knoten für den Text des Dokumentes einführen.

### 5.2.6 Entity-Referenzen

**Definition 5.5 (Entity)** *Eine **Entity** ist eine Art Macro-mechanismus, mit dem Abkürzungen für* XML*-Text deklariert werden können.*

**Notation 5.2 (Predefined Entities)** XML *hat fünf vordefinierte Entities:* `lt` *steht für ein kleiner-Zeichen* `"<"`, `gt` *steht für ein grösser-Zeichen* `">"`, `amp` *steht für ein kaufmännisches und-Zeichen* `"&"`, `quot` *steht für ein doppeltes Anführungszeichen '"', und* `apos` *steht für ein einfaches Anführungszeichen (Apostroph)* `"'"`.

Der Ersetzungstext eines Entities wird eingefügt, wenn man ein kaufmännisches und-Zeichen, den Namen des Entities, und ein Semikolon `";"` schreibt. Zum Beispiel kann man ein kleiner-Zeichen als `<` schreiben.

Entity-Referenzen sind sehr ähnlich zu den oben schon erklärten Zeichen- Referenzen: Dort folgt nach dem *und*-Zeichen aber noch ein Nummer- Zeichen, um anzudeuten, dass man ein Zeichen über seine Nummer im Unicode angibt: Z.B. `&#169;` (dezimal) oder `&#xA9;` (hexadezimal) für das Copyright-Zeichen `"l'"`.

Entity-Referenzen werden in der Spezifikation in [XML] Abschnitt 4.1 behandelt

### 5.2.7 Attribute

In Start-Tags und leeren Tags können nach dem Element-Typ auch Attribut- Werte für dieses Element definiert werden, z.B.:

```
<publ year="1999" month="11">
```

Attributwerte müssen immer in Anführungszeichen eingeschlossen werden.

Jede einzelne Attribut-Angabe besteht aus dem Namen des Attributes, einem Gleichheitszeichen =, und dem Wert des Attributes.

Der Wert des Attributes muss in einfache oder doppelte Anführungszeichen eingeschlossen sein, d.h. `year='1999'` und `year="1999"` sind beide zulässig. Die beiden Anführungszeichen müssen aber übereinstimmen.

In Attribut-Werten sind keine kleiner-Zeichen `"<"` erlaubt (ausser als `<`). Dadurch kann ein Parser ein eventuell vergessenes Anführungszeichen recht einfach finden.

Die Reihenfolge von Attribut-Angaben in einem Tag ist unwesentlich.

Kein Attribut darf zweimal in einem Tag definiert werden.

### 5.2.8 XML Deklaration

Ganz am Anfang eines XML-Dokumentes sollte normalerweise die XML- Deklaration stehen, z.B.:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Die XML-Deklaration enthält noch eine dritte Klausel (*Standalone Declaration*), die wie die Zeichencodierungs-Klausel optional ist, und im Zusammenhang mit DTDs unten erklärt wird.

Die XML Deklaration wird in [XML] Abschnitt 2.8 der Spezifikation behandelt. Siehe auch Abschnitt 4.3.1 für die verwandte Text-Deklaration am Anfang von *Include-Dateien*. Die Bedeutung der *Standalone Deklaration* ist in Abschnitt 2.9 erklärt.

### 5.2.9 Kommentare

**Definition 5.6 (Kommentare)** ***Kommentare*** *beginnen mit der Zeichenfolge `"<!--"` und enden mit der Zeichenfolge `"-->"`. Kommentare dürfen nicht die Zeichenfolge `"--"` (einen doppelten Anführungsstrich) enthalten, dafür sind die ansonsten speziellen Zeichen `"<"` und `"&"` kein Problem.*

Kommentare sind überall ausserhalb von Markup in Dokumenten erlaubt, zusätzlich sind sie innerhalb der Dokument Type Definition (die ein grosser Block von Markup ist) zwischen den Einzeldeklarationen erlaubt.

Kommentare sind in [XML] Abschnitt 2.5 der Spezifikation behandelt.

### 5.2.10 Processing-Instructions

Processing-Instruktions sind Hinweise im Dokument für bestimmte Programme, die das Dokument verarbeiten. Zum Beispiel ist die Angabe des Stylesheets für XSLT-Prozessoren wichtig:

```
<?xml-stylesheet type="text/xsl" href="notes.xsl"?>
```

Processing Instructions sind im [XML] Abschnitt 2.6 der Spezifikation definiert.

## 5.3 Document Type Definitions

### 5.3.1 Ziel

**Definition 5.7 (DTD)** ***Document Type Defintions (DTDs)*** *legen zusätzliche Einschränkungen für* XML*-Dokumente fest, die über die reine Wohlgeformtheit hinaus gehen.*

Zum Beispiel definieren DTDs,

1. welche Element-Typen (Tags) es gibt,
2. welche Elemente im Inhalt welcher anderen Elemente vorkommen können (und ob sie wiederholbar oder optional sind, welche Reihenfolge eingehalten werden muss),
3. welche Elemente Text-Daten enthalten können
4. welche Attribute ein Element-Typ hat und
5. was mögliche Werte für ein Attribut sind.

Im Prinzip ist eine DTD einfach eine kontextfreie Grammatik für die Daten (in EBNF-ähnlicher Notation). Man kann eine gegebene DTD direkt in eine kontextfreie Grammatik (CFG) übersetzen. Dabei führt man für jeden Element-Typ ein entsprechendes Nichtterminal-Symbol ein. Die Regel, dass Attribute in beliebiger Reihenfolge angegeben werden können, aber keines mehrfach, ist in der CFG allerdings nur umständlich auszudrücken (Ausserdem können ID / IDREF-Attribute nicht kontextfrei dargestellt werden.). Umgekehrt kann man eine kontextfreie Grammatik auch in eine DTD übersetzen (wieder mit der Entsprechung Nichtterminalsymbol - Elementtyp), aber man behält dann mit den Start- und End-Tags Markierungen zurück, die den ganzen Ableitungsbaum im Ergebniswort codieren. Man könnte also vielleicht sagen, dass man eine beliebige kontextfreie *abstrakte Syntax* als DTD darstellen kann, aber die konkrete Syntax dann von XML vorgeschrieben wird.

Eine DTD kann im Dokument selbst definiert werden (*internal subset*), in einer (oder mehreren) anderen Dateien definiert werden (*external subset*), oder kann aus internem und externem Anteil bestehen.

Beispiel

```
<!-- Document Type Definition für Literaturlisten -->
<!ELEMENT booklist (book)*>
<!ELEMENT book     (author+, title, publ?, note?)>
<!ATTLIST book     isbn CDATA #REQUIRED
                   pages CDATA #IMPLIED>
<!ELEMENT author   EMPTY>
<!ATTLIST author   first CDATA #IMPLIED
                   mi CDATA #IMPLIED
                   last CDATA #REQUIRED>
<!ELEMENT title    (#PCDATA)>
<!ELEMENT publ     (#PCDATA)>
```

```
<!ATTLIST publ     year CDATA #IMPLIED
           mo CDATA #IMPLIED>
<!ELEMENT note     (#PCDATA)>
```

Folgende Datei basiert auf dieser DTD und ist *valid*:

```
<?xml version="1.0"?>
<!-- Kommentar: Einige Bücher zur XML-Definition -->
<!DOCTYPE booklist SYSTEM "books1.dtd" >
<booklist>
  <book isbn="0-13-014714-1" pages="1074">
    <author first="Paul" last="Prescod"/>
    <author first="Charles" mi="F." last="Goldfarb"/>
    <title>The XML Handbook - 2nd Edition</title>
    <publ year="1999" mo="11">Prentice Hall</publ>
    <note>Contains CD.</note>
  </book>
  <book isbn="1-56592-709-5" pages="107">
     <author first="Robert" last="Eckstein"/>
     <title>XML Pocket Reference</title>
     <publ year="1999" mo="10">O'Reilly</publ>
  </book>
  <book isbn="0-13-082676-6" pages="368">
    <author first="Bob" last="DuCharme"/>
    <title>XML: The Annotated Specification</title>
    <publ year="1998" mo="12" >Prentice Hall</publ>
 </book>
</booklist>
```

### 5.3.2 Element-Deklarationen

**Definition 5.8 (Element Deklaration)** *Eine Element-Deklaration besteht aus*

1. *der Zeichenfolge* `"<!ELEMENT"`*,*
2. *dem Namen des Element-Typs*
3. *einer Spezifikation über das Format des Inhaltes des Elementes (was also zwischen öffnendem und schliessendem Tag stehen darf,* content model*) und*
4. *dem Zeichen* `">"`*.*

Alle Deklarationen in XML beginnen mit `"<!"`.

**Definition 5.9 (Inhalt Spezifikation)** *Für die Inhalts-Spezifikation gibt es vier Möglichkeiten:*

1. *kein Inhalt erlaubt,*
2. *beliebiger Inhalt,*
3. *nur Elemente (mit vorgeschriebener Struktur) (*element content*),*
4. *Elemente und Text (mit Angabe welche Elemente zulässig sind) (*mixed content*).*

Natürlich kann ein Element bei der dritten Möglichkeit (*element content*) indirekt Text enthalten, nämlich innerhalb dieser Elemente. Aber die Kind-Knoten in der Baum-Darstellung sollen dann sämtlich Element-Knoten sein, Qkeine Text-Knoten.

Die Festlegung, dass ein Element leer sein muss, geschieht mit dem Schlüsselwort `EMPTY`:

```
<!ELEMENT author EMPTY>
```

Ein anderer häufiger Spezialfall ist, dass ein Element nur Text enthalten kann, aber keine weiteren Elemente:

```
<!ELEMENT title (#PCDATA) >
```

Oft enthält ein Element im wesentlichen Text, von dem aber Teile in weitere Tags eingeschlossen sind. Zum Beispiel könnte es sein, dass man definierende Vorkommen von Worten im Text markieren möchte (`def`), sowie auch Namen (`name`). Dann hat man ein echtes *mixed content model*. In der Baum-Darstellung kann ein solcher Knoten eine Folge von Text- und Element-Knoten als Kinder haben.

Beliebiger Inhalt wird mit dem Schlüsselwort `ANY` erlaubt:

```
<!ELEMENT example ANY >
```

Man beachte schliesslich noch, dass kein Element-Typ mehr als einmal in einer DTD definiert sein darf.

### 5.3.3 Inhalts-Spezifikation für *Element Content*

Wenn ein Element-Typ als Inhalt nur andere Elemente haben kann, aber nicht direkt Text, spricht man von *Element Content*. Für diesen Fall bietet XML recht mächtige Definitionsmöglichkeiten für die Struktur des Inhalts.

**Definition 5.10 (Element Content)** *Man kann Auswahl `"|"` und Sequenz `","` verwenden, diese auch (fast) beliebig schachten, und für jede Gruppe definieren, ob sie*

1. *optional ist (`"?"`, null oder einmal),*
2. *optional und wiederholbar (`"*"`, null oder mehrfach), oder*
3. *notwendig und wiederholbar (`"+"`, einmal oder mehrfach).*
4. *Gibt man keines der drei Zeichen an, so muss das Element genau einmal vorkommen (notwendig und nicht wiederholbar).*

Zum Beispiel legt folgende Definition fest, dass jedes BOOK-Element genau drei Kinder haben muss, die die Typen AUTHOR, TITLE, und PUBLISHER haben, und in dieser Reihenfolge angegeben sein müssen:

```
<!ELEMENT book (author, title, publisher) >
```

Sequenz hat also die allgemeine Syntax `(..., ..., ...)`. Sie erzwingt eine Reihenfolge für $n$ , $n \geq 1$ , Elemente.

Für jedes Element der Sequenz (wie auch für die Sequenz als ganzes) kann man Optionalität und Wiederholbarkeit definieren, z.B.:

```
<!ELEMENT book (author+, title, publisher?) >
```

Dies bedeutet, dass es mehr als einen Autoren geben kann (aber mindestens einen geben muss), und dass der Publisher entfallen kann. Es muss genau einen Titel geben, und kann nicht mehr als einen Verlag geben. Die Reihenfolge ist: Erst die Autoren, dann der Titel, und zum Schluss der Verlag (falls vorhanden). Möchte man erlauben, dass es auch gar keine Autoren kann, so muss man das `+` durch ein `*` ersetzen.

Man kann auch die ganze Liste wiederholbar machen, z.B.:

```
<!ELEMENT authors (first, last)+ >
```

Dies erlaubt mehrere Paare von Vorname und Nachname, z.B.:

```
<authors>
  <first>Udo</first><last>Lipeck</last>
  <first>Stefan</first><last>Brass</last>
</authors>
```

Inhalts-Spezifikationen für *Element Content* bekommen ihre Mächtigkeit auch dadurch, dass man die Konstrukte schachteln kann. Überall, wo ein Name stehen kann, kann auch selbst wieder eine Auswahl oder Sequenz stehen. Zum Beispiel ist folgendes möglich:

```
<!ELEMENT book ((first, last)+, title, publisher?) >
```

Es müssen dann zunächst Paare von Vornamen und Nachnamen angegeben werden (mindestens einmal), dann ein Titel, und zum Schluss möglicherweise ein Verlag.

Neben der mit `,` markierten Sequenz gibt es auch die mit `|` markierte Auswahl. Zum Beispiel bedeutet

```
<!ELEMENT address (postal | email | phone) >
```

dass jedes `address`-Element genau eines der drei Elemente `postal`, `email` oder `phone` enthalten muss.

Es sind beliebige Schachtelungen von Auswahl und Sequenz möglich, z.B.:

```
<!ELEMENT book ((first, last)+, title, (publisher|uni)?) >
```

Dies erlaubt anstelle der Angabe eines Verlages auch die Angabe einer Universität (etwa für technische Berichte). Aufgrund des `?` kann die Angabe aber auch ganz entfallen.

Jede Inhaltsspezifikation für *Element Content* muss mindestens ein Paar Klammern enthalten. Selbst wenn jedes Element `a` das Element `b` genau einmal enthalten soll, muss man folgendes schreiben:

```
<!ELEMENT a (b) >
```

Wiederholungs- / Optionalitäts-Angaben sind sowohl nach Element-Typen als auch nach der schliessenden Klammer einer Auswahl oder Sequenz möglich. Zum Beispiel ist folgendes äquivalent:

```
<!ELEMENT a (b*) >
<!ELEMENT a (b)* >
```

Dagegen ist folgendes natürlich nicht äquivalent:

```
<!ELEMENT a (b*, c*) >
<!ELEMENT a (b, c)* >
```

Element-Typ Deklarationen sind in [XML] Abschnitt 3.2 der XML Spezifikation behandelt

### 5.3.4 Deterministic Content Models

Schliesslich fordert XML*deterministic content models*. Das bedeutet, dass der XML Parser allein durch ansehen des nächsten Elementes in der Eingabe entscheiden kann, welchen Weg er bei einer Auswahl verfolgen muss.

Zum Beispiel wäre folgendes nicht zulässig:

```
<!ELEMENT example ((a, b) | (a, c)) >
```

Wenn der Parser das `a` liest, muss er sich zwischen den beiden Alternativen entscheiden, ohne zu wissen, ob danach ein `b` oder ein `c` folgt.

Es ist in der Informatik durchaus bekannt, wie man das Problem lösen kann (man kann jeden nichtdeterministischen endlichen Automaten in einen deterministischen endlichen Automaten umwandeln). Aber der XML Standard fordert aus Kompatibilitätsgründen mit existierenden SGML Systemen diese Einschränkung. Es macht die Entwicklung von Parsern etwas einfacher, wenn diese Umwandlung nicht nötig ist.

Dafür muss der XML Benutzer die Inhaltsspezifikation faktorisieren:

```
<!ELEMENT EXAMPLE (a, (b | c)) >
```

Dies beschreibt das gleiche Format für den Inhalt, ist aber ein *deterministic content model*.

Anhang E der XML Spezifikation [XML] beschäftigt sich mit *Deterministic Content Models*

### 5.3.5 Definition von Attributen

Man kann Attribute für einen Elementtyp in der folgenden Form definieren:

```
<!ATTLIST author first CDATA #IMPLIED
                mi CDATA #IMPLIED
                last CDATA #REQUIRED >
```

**Definition 5.11 (Attribut Deklaration)** *Die Deklaration von Attributen eines Element-Typs besteht aus*

1. *der Zeichenkette* `"<!ATTLIST"`,

2. *dem Namen des Element-Typs,*
3. *ein oder mehreren Attribut-Deklarationen und*
4. *dem Zeichen ">".*

*Jede einzelne Attribut Deklaration besteht aus drei Teilen:*

1. *dem Namen des Attributes,*
2. *dem Typ des Attributes und*
3. *einer Default Deklaration.*

Ein wichtiger Typ von Attributen ist `CDATA`. Es sind dann beliebige Zeichenketten erlaubt (die aber kein `"<"` enthalten können, und `"&"` nur für Entity-Referenzen). Weitere Typen von Attributen werden unten erläutert.

Default-Deklarationen haben vier Formen:

1. `#REQUIRED`,
2. `#IMPLIED`,
3. ein Default-Wert,
4. `#FIXED` mit der Angabe eines konstanten Wertes.

Attribut-Deklarationen werden in [XML] Abschnitt 3.3 der Spezifikation behandelt

### 5.3.6 Daten-Typen für Attribute

`CDATA`: Beliebiger String (*character data*).

Aufzählungen, zum Beispiel `(yes|no)`: Das Attribut muss einen der angegebenen Werte haben. Die Werte müssen Folgen von Buchstaben, Zifferen, Unterstrich _, Bindestrich `-`, Punkt `.` und Doppelpunkt `:` sein. Dies sind die gleichen Zeichen, die in Namen verwendet werden dürfen, aber es gibt keine Beschränkung bezüglich des ersten Zeichens.

`ID`: Dies sind eindeutige Namen für das Element. Attribut-Werte müssen die gleichen Einschränkungen wie Namen erfüllen, zusätzlich kann jeder Name im ganzen XML-Dokument aber nur einmal in einem Attribut dieses Typs verwendet werden. Dadurch identifiziert jeder solche Name eindeutig ein Element. Kein Element-Typ kann mehr als ein Attribut dieses Typs haben. Es ist üblich, aber nicht Vorschrift, Attribute dieses Typs auch `ID` zu nennen: `<!ATTLIST book id ID #IMPLIED >`

`IDREF`: Attribute dieses Typs verweisen auf andere Elemente. Attribut-Werte müssen wieder zulässige Namen sein, und sie müssen ausserdem im XML Dokument als Wert eines Attributes des Typs `ID` vorkommen.

`IDREFS`: Attribute dieses Typs verweisen auf eine Menge anderer Elemente. Attribut-Werte müssen eine durch Leerplatz (Leerzeichen etc.) getrennte Liste von Namen sein.

NMTOKEN: Attribut-Werte müssen Folgen von Buchstaben, Ziffern, und den vier Zeichen _, -, . und : sein.

NMTOKENS: Dies ist entsprechend eine durch Leerzeichen etc. getrennte Liste von NMTOKEN-Werten. Zum Beispiel: <box color="0 50 100" >

Daneben gibt es noch die Typen ENTITY, ENTITIES, und Aufzählungen von Notationen. Diese werden im nächsten Kapitel erläutert.

[XML] Abschnitt 3.3.1 behandelt Datentypen für Attribute.

### 5.3.7 Beispiel für ID / IDREF

Wenn man noch weitere Informationen über Autoren erfassen will, z.B. eine EMail-Adresse oder Homepage (soweit bekannt), dann wären diese Informationen redundant dargestellt, wenn einige Autoren mehrere Bücher geschrieben haben. Im Datenbank-Entwurf, besonders bei der Normalisierung, versucht man solche Redundanzen zu vermeiden. Redundanzen führen häufig zu Inkonsistenzen und doppelter Arbeit.

Eine Lösung ist, die Information über Autoren getrennt von der Information über Bücher zu erfassen und in den Buch-Elementen auf die Autor-Elemente zu verweisen:

```
<?xml version="1.0"?>
<!DOCTYPE booklist SYSTEM "books3.dtd">
 <booklist>
  <author id="PP" first="Paul" last="Prescod"/>
  <author id="CG" first="Charles" mi="F." last="Goldfarb"/>
  <author id="RE" first="Robert" last="Eckstein"/>
  <author id="BC" first="Bob" last="DuCharme"/>
  <book authors="PP CG" isbn="0-13-014714-1" pages="1074">
   <title>The XML Handbook - 2nd Edition</title>
   <publ year="1999" mo="11">Prentice Hall</publ>
   <note>Contains CD.</note>
  </book>
  <book authors="RE" isbn="1-56592-709-5" pages="107">
   <title>XML Pocket Reference</title>
   <publ year="1999" mo="10">O'Reilly</publ>
  </book>
  <book authors="BC" isbn="0-13-082676-6" pages="368">
   <title>XML: The Annotated Specification</title>
   <publ year="1998" mo="12" >Prentice Hall</publ>
  </book>
 </booklist>
```

Die DTD dazu sieht so aus:

```
<!ELEMENT booklist (author | book)*>
<!ELEMENT author   EMPTY>
<!ATTLIST author   id ID #REQUIRED
```

```
                    first CDATA #IMPLIED
                    mi CDATA #IMPLIED
                    last CDATA #REQUIRED>
<!ELEMENT book      (title, publ?, note?)>
<!ATTLIST book      authors IDREFS #IMPLIED
                    isbn CDATA #REQUIRED
                    pages CDATA #IMPLIED>
<!ELEMENT title     (#PCDATA)>
<!ELEMENT publ      (#PCDATA)>
<!ATTLIST publ      year CDATA #IMPLIED
                    mo CDATA #IMPLIED>
<!ELEMENT note      (#PCDATA)>
```

### 5.3.8 Dokument-Typ Deklaration

Die Dokument-Typ Deklaration ist optional. Wenn ein Dokument keine solche Deklaration enthält, kann es allerdings nur *well-formed* sein. Um *valid* zu sein, muss es eine Document Type Deklaration enthalten, und Elemente und Attribute müssen im Dokument gemäss den Einschränkungen dieses Dokumenttyps verwendet werden.

Die Dokument-Typ Deklaration muss vor dem Start-Tag des äussersten Elementes stehen, und nach der XML-Deklaration.

Die Dokument-Typ Deklaration kann auf eine Datei verweisen, die die einzelnen Markup Deklarationen (für Elemente, Attribute, etc.) enthält, oder diese Deklarationen können auch direkt in der Dokument-Typ Deklaration enthalten sein. Beide Möglichkeiten können auch kombiniert sein.

Zum Beispiel würde die Bücherliste in nur einer Datei so aussehen:

```
<?xml version="1.0"?>
<!-- Kommentar: Einige Buecher zur XML-Definition -->
<!DOCTYPE booklist [
  <!ELEMENT booklist (book)*>
  <!ELEMENT book     (author+, title, publ?, note?)>
  <!ATTLIST book     isbn CDATA #REQUIRED
                     pages CDATA #IMPLIED>
  <!ELEMENT author   EMPTY>
  <!ATTLIST author   first CDATA #IMPLIED
                     mi CDATA #IMPLIED
                     last CDATA #REQUIRED>
  <!ELEMENT title    (#PCDATA)>
  <!ELEMENT publ     (#PCDATA)>
  <!ATTLIST publ     year CDATA #IMPLIED
                     mo CDATA #IMPLIED>
  <!ELEMENT note     (#PCDATA)>
]>
<booklist>
```

```
  <book isbn="0-13-014714-1" pages="1074">
    <author first="Paul" last="Prescod"/>
    <author first="Charles" mi="F." last="Goldfarb"/>
    <title>The XML Handbook - 2nd Edition</title>
    <publ year="1999" mo="11">Prentice Hall</publ>
    <note>Contains CD.</note>
  </book>
  <book isbn="1-56592-709-5" pages="107">
    <author first="Robert" last="Eckstein"/>
    <title>XML Pocket Reference</title>
    <publ year="1999" mo="10">O'Reilly</publ>
  </book>
  <book isbn="0-13-082676-6" pages="368">
    <author first="Bob" last="DuCharme"/>
    <title>XML: The Annotated Specification</title>
    <publ year="1998" mo="12">Prentice Hall</publ>
  </book>
</booklist>
```

**Definition 5.12 (Dokument Typ Deklaration)** *Die Dokument-Typ Deklaration besteht also aus:*

1. *Der Zeichenkette* `"<!DOCTYPE"`.
2. *Dem Namen des Dokumenttyps. Dieser muss mit dem Namen des Wurzel-Elementes übereinstimmen (im Beispiel:* `BOOKLIST`*).*
3. *Optional einem Verweis auf eine Datei, die Markup-Deklarationen enthält (*external subset*).*
4. *Optional Markup Deklarationen, die in eckige Klammern eingeschlossen sind (*internal subset*).*
5. *Einer schliessenden spitzen Klammer:* `">"`.

Der Verweis auf eine externe Datei kann aus zwei Teilen bestehen:

1. Dem Schlüsselwort `SYSTEM`.
2. Einer URL (Dies kann auch eine relative URL sein, z.B. einfach ein Dateiname.).

Auf eine externe Datei kann aber auch mit einem `PUBLIC`-Identifier verwiesen werden. Dies ist Gegenstand des nächsten Abschnittes.

Die Syntax von Markup-Deklarationen im *internal subset* ist etwas eingeschränkt. Der Grund dafür ist, dass dieser Teil auch von einem *non-validation parser* verarbeitet werden muss, der nur die Wohlgeformtheit prüft.

Werden Markup-Deklarationen im *internal subset* angegeben, und gibt es ausserdem einen Verweis auf eine Datei mit Markup-Deklarationen, so werden die im Dokument selbst angegebenen Deklarationen zuerst verarbeitet.

## 5.4 Entities

### 5.4.1 Ziel

Entities entsprechen Macros (und auch Include-Dateien) in Programmiersprachen wie C. Der Name eines Entities ist eine Abkürzung, die vom XML-Prozessor automatisch durch den Inhalt des Entities ersetzt wird.

Man kann ein Entity wie in folgendem Beispiel definieren:

```
<!ENTITY email "gugus@bfh.ch">
```

Dann würde überall im Dokument die Entity-Referenz `&email;` automatisch durch den Text `gugus@bfh.ch` ersetzt.

### 5.4.2 Nutzen von Entities

Wenn sich die Email-Adresse ändert, muss man sie dann nur an einer Stelle ändern (in der Entity-Deklaration). Ansonsten müsste man das ganze Dokument durchsuchen, um alle Vorkommen zu ersetzen.

Ausserdem können Entities Tipparbeit sparen.

Entities erhöhen auch die Einheitlichkeit und Konsistenz.

Entities wie `<` sind wesentlich lesbarer und leichter zu merken als die Zeichen-Referenz `&#60;`, für die sie stehen.

Zum Beispiel in Dokumenten, die Benutzerschnittstellen von Programmen beschreiben, wird die übersetzung in andere Sprachen leichter, wenn alle Ausgabe-Texte als Entities definiert sind.

Mit Entities kann man ein Dokument in mehrere wiederverwendbare Komponenten zerlegen, bzw. umgekehrt modular zusammensetzen.

### 5.4.3 External Entities

Mit externen parsed Entities kann man den Text aus einer Datei in das Dokument einfügen. Dies ist besonders für grosse Dokumente nützlich, zum Beispiel kann man jedes Kapitel in eine eigene Datei schreiben. Es erlaubt auch einen modulareren Aufbau, so kann man etwa die DTD aus mehreren Modulen in unterschiedlichen Dateien zusammensetzen.

Externe Entities gibt es als General Entities (für Text) und als Parameter Entities (für DTDs). Externe General Entities werden folgendermassen deklariert:

```
<!ENTITY lic SYSTEM "license.xml">
```

Der Inhalt der Datei `license.xml` kann dann mit `&lic;` eingefügt werden (es ist also eine normale Entity-Referenz). Will man eine Datei in die DTD einbinden, so geschieht das folgendermassen:

```
<!ENTITY % tables SYSTEM "tab.xml"> %tables;
```

Externe Entities dürfen nicht in Attribut-Werten referenziert werden.

### 5.4.4 Unparsed Entities

Unparsed Entities sind Dateien, die nicht im XML Format vorliegen, aber zum Dokument gehören. Ein Beispiel wären etwa Bilder im GIF Format.

In solchen Fällen muss man zuerst einen Namen für das Format definieren. Das geschieht mit einer Notations-Deklaration:

```
<!NOTATION gif SYSTEM "gifmagic.exe">
```

Die Angabe nach `SYSTEM` könnte z.B. ein Programm sein, dass dieses Format anzeigen kann, aber das ist keine Bedingung. Auch eine Quelle für weitere Informationen über das Format (wie das Postscript Referenz Handbuch) kann zur Identifizierung dienen.

Anschliessend kann man ein Entity dieses Formates deklarieren:

```
<!ENTITY clown SYSTEM "clown.gif" NDATA GIF>
```

Durch den Zusatz `NDATA` mit der Format-Angabe ist klar, dass es sich hier um ein unparsed Entity handelt.

## 5.5 SAX (Simple API for XML)

SAX ist eine Abkürzung für *Simple Api for XML*. Ein SAX-Parser traversiert ein XML Dokument und ruft für jedes Element eine benutzer-definierte Handler-methode auf.

Das SAX API ist einfach zu benutzen und eignet sich gut für das prozessorientierte Verarbeiten von XML-Daten. Es ist auf die Verwendung mit JAVA als Programmiersprache zugeschnitten, es gibt jedoch analoge API's für andere Programmiersprachen.

Jedes XML-Dokument besitzt eine Baumstruktur. SAX realisiert im wesentlichen das traversieren des XML-Baumes. Genauer ist es eine in-order Traversierung: Sobald der traversierungsmechanismus in einen Element eintritt, wird die Callback-Funktion `startElement()` aufgerufen. Anschliessend wird in weiteren Aufrufen von Callback-Funktionen der Inhalt des Elements an die Applikation übergeben. Danach wird das erste, weiter unten im Baum liegende Element Knoten bearbeitet. Sind alle Kind-Elemente bearbeitet, wird beim Elternknoten die Austrittsfunktion `endElement()` aufgerufen.

Weitere Callback-Funktionen werden beim Starten und Verlassen des Dokumentes, beim Wechseln des namensraumes, beim Auftreten von Fehlern, für die Verarbeitung von Processing Instructions, etc.

Der SAX-Parser kann das verarbeitete XML-Dokument validieren. Letzteres muss dafür eine DTD-Deklaration beinhalten, und beim SAX-Parser muss die `setValidating()` Funktion aufgerufen werden.

Im Moment aktuell ist die Version 2 des SAX API. Die entsprechenden JAVA Packages sind bei SUN unter JAXP 1.1 zu finden. SAX ist nicht ein W3C Standard und wird auch nicht vom W3C Konsortium gepflegt.

### 5.5.1 Interfaces

Ein SAX-Parser muss u.a. folgende Interfaces implementieren.

**ContentHandler** Das interface `ContentHandler` beschreibt die eigentliche verarbeitungsmethoden für ein XML-Dokument. Es sind u.a. die Methoden `startDocument()`, `endDocument()`, `startElement()`, `endElement()`, `characters()` und `processingInstruction()`.

**ErrorHandler** Das Interface `Errorhandler` behandelt Parsing-Fehler. Beachte, dass z.B. validierungsfehler nicht als Exception geworfen werden, sondern die Methode `error()` aufgerufen und dieser ein Exception-Objekt übergeben wird.

Lesen Sie bitte die API-Dokumentation für mehr Details.

**Bemerkung 5.1** [Default Implementierung] Die meisten SAX-Distributionen stellen eine Default Implementierungen (`Defaulthandler`) zur Verfügung.

**Beispiel 5.1** [SAX Baum] Folgendes Programm validiert (mittels Methode `setValidating()` ein XML-Dokument. Die Ausgabe ist eine Baumdarstellung des Dokuments. `argv[0]` enthält die einzulesende Datei.

JAVA-Klasse `SAXTree`:

```
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

/**
 * Prints the parse tree of an XML
 * document to stdout.
 *
 * Tested with Xerces2 Java Parser 2.0.1
 * and with SUN java compiler version 1.3.0
 *
 * Required libraries on the classpath:
 *   \xerces-2_0_1\xmlParserAPIs.jar
 *   \xerces-2_0_1\xercesImpl.jar
 *
 * @author Michael Dürig HTA-BE
 * 2002.06.12
 */
public class SAXTree extends DefaultHandler {

  private StringBuffer _Indent = new StringBuffer("");
  private StringBuffer _Characters = new StringBuffer("");
  public SAXTree() { }

  /** Print a string with proper indentination to stdout */
  protected void emit(String s) {
    System.out.println(_Indent + s);
  }
```

```
 /** Add character to the character buffer */
 protected void emit(char c) {
   _Characters.append(c);
 }

 /** Print accumulated characters to stdout */
 protected void flushCharacters() {
   if(_Characters.length() != 0) {
     emit("Characters: (" + _Characters + ")");
     _Characters = new StringBuffer("");
   }
 }

 /** Indentination helper function */
 protected void indent() {
   _Indent.append("  ");
 }

 /** Indentination helper function */
 protected void dedent() {
   _Indent.delete(0, 2);
 }

 /** Prints an error message */
 protected void printError(String type,
                           SAXParseException e) {
   System.err.println(type + ": " + e.getMessage() +
                      " Line: " + e.getLineNumber() +
      " Column: " +
                      e.getColumnNumber());
   System.err.flush();
 }

 public void processingInstruction(String target,
   String data)
                                    throws SAXException {
   emit("Processing Instruction: (" + target + " " +
data + ")");
 }

 public void startElement(String namespaceURI,
                          String localName,
                          String qName, Attributes atts)
                         throws SAXException {
   emit("Start Element: (" + qName + ")");
   indent();
```

```
    for(int i = 0; i < atts.getLength(); i++)
      emit("Attribute: (" + atts.getQName(i) +
   "=" + atts.getValue(i) + ")");
  }

  public void characters(char ch[],
int start,
int length) throws SAXException {
    for(int i = start; i < start + length; i++)
      emit(ch[i]);
  }

  public void endElement(String namespaceURI,
                         String localName,
                         String qName)
                        throws SAXException {
    flushCharacters();
    dedent();
    emit("End Element: (" + qName + ")");
  }

  public void warning(SAXParseException e)
                     throws SAXException {
    printError("Warning", e);
  }

  public void error(SAXParseException e)
                   throws SAXException {
    printError("Error", e);
  }

  public void fatalError(SAXParseException e)
                        throws SAXException {
    printError("Fatal Error", e);
    throw e;
  }

  public static void main(String args[]) {
    if (args.length == 0) {
      System.out.println("No file URI specified");
      System.exit(1);
    }

    // Request a SAX-parser instance from the SAXParserFactory
    SAXParserFactory ParserFactory =
      SAXParserFactory.newInstance();
    ParserFactory.setValidating(true);
```

```
    ParserFactory.setNamespaceAware(false);
    // and parse the given document using an instance of
    // SAXTree as handler for the various notifications
    try {
      SAXParser Parser = ParserFactory.newSAXParser();
      Parser.parse(args[0], new SAXTree());
    }
    catch(ParserConfigurationException e) {
      System.err.println("Error initializing parser: " +
 e.getMessage());
      System.exit(1);
    }
    catch(Exception e) {
      System.err.println("Parse error: " + e.getMessage());
    }
  }
}
```

## 5.6 DOM (Document Object Model)

DOM erlaubt es, XML-Dokumente in Objektbäume zu transformieren. DOM besitzt standard Schnittstellen für den Zugriff auf Attribute. Ferner stellt DOM standard Methoden für die Traversierung und die Modifikation des Baumes zur Verfügung.

Während SAX ein Verarbeitungspaket ist, dient DOM sowohl der Bearbeitung von XML Dateien, ist aber gleichzeitig auch ein generisches Datenmodell. SAX erlaubt eine streamartige Verarbeitung, DOM baut ein ganzes Dokument als speicherresidente Struktur auf.

### 5.6.1 Interfaces

Ein DOM-Parser muss u.a. folgende Interfaces implementieren:

`Node` und die von `Node` abgeleitete Interfaces `Attr`, `CharacterData`, `CDATASection`, `Comment`, `Text`, `Document`, `DocumentFragment`, `DocumentType`, `Element`, `Entity`, `EntityReference`, `Notation` und `ProcessingInstruction`.

Lesen Sie bitte die API-Dokumentation für mehr Details.

**Beispiel 5.2** [DOM Baum] Folgendes Programm validiert (mittels Methode `setValidating()` ein XML-Dokument. Die Ausgabe ist eine Baumdarstellung des Dokuments. `argv[0]` enthält die einzulesende Datei.

JAVA-Klasse `DOMTree`:

```
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
```

```
/**
 * Prints the parse tree of an XML
 * document to stdout.
 *
 * Tested with Xerces2 Java Parser 2.0.1
 * and with SUN java compiler version 1.3.0
 *
 * Required libraries on the classpath:
 *   \xerces-2_0_1\xmlParserAPIs.jar
 *   \xerces-2_0_1\xercesImpl.jar
 *
 * @author Michael Dürig HTA-BE
 * 2002.06.13
 */
public class DOMTree implements ErrorHandler{

  private StringBuffer _Indent = new StringBuffer("");

  public DOMTree() { }

  /** Print a string with proper indentination to stdout */
  protected void emit(String s) {
    System.out.println(_Indent + s);
  }

  /** Indentination helper function */
  protected void indent() {
    _Indent.append("  ");
  }

  /** Indentination helper function */
  protected void dedent() {
    _Indent.delete(0, 2);
  }

  /** Prints an error message */
  protected void printError(String type,
    SAXParseException e) {
    System.err.println(type + ": " + e.getMessage() +
       " Line: " + e.getLineNumber() +
       " Column: " + e.getColumnNumber());
    System.err.flush();
  }

   /** Print a document node */
  public void print(Document n) {
    emit("Document:");
```

```
}

/** Print an element node */
public void print(Element n) {
  emit("Element: " + n.getNodeName());
  if(n.hasAttributes()){
    indent();
    NamedNodeMap Attributes = n.getAttributes();
    for(int i = 0; i < Attributes.getLength(); i++) {
      Attr Attribute = (Attr)Attributes.item(i);
      emit("Attribute: " + Attribute.getName() +
   "=" + Attribute.getValue());
    }
    dedent();
  }
}

/** Print a comment node */
public void print(Comment n) {
  emit("Comment: " + n.getNodeValue());
}

/** Print a text node */
public void print(Text n) {
  emit("Text: " + n.getNodeValue());
}

/** Print a processing instruction node */
public void print(ProcessingInstruction n) {
  emit("Processing Instruction: " + n.getNodeName() +
"=" + n.getNodeValue());
}

/** Print a document type node */
public void print(DocumentType n) {
  emit("Document Type: " + n.getNodeName());
}

/**
 * Print an arbitrary node by dispatching
 * to the proper print method. This
 * methode prints children of the current
 * node by calling itself recursively.
 */
public void print(Node n) {
  switch(n.getNodeType()) {
    case Node.DOCUMENT_NODE:
```

```
      print((Document)n);
      break;
    case Node.ELEMENT_NODE:
      print((Element)n);
      break;
    case Node.COMMENT_NODE:
      print((Comment)n);
      break;
    case Node.TEXT_NODE:
      print((Text)n);
      break;
    case Node.PROCESSING_INSTRUCTION_NODE:
      print((ProcessingInstruction)n);
      break;
    case Node.DOCUMENT_TYPE_NODE:
      print((DocumentType)n);
      break;
    default:
      emit("Not supported Node Type: " +
   n.getNodeName());
      break;
  }
  // Print the nodes children
  if(n.hasChildNodes()) {
    NodeList nl = n.getChildNodes();
    for(int i = 0; i < nl.getLength(); i++) {
      indent();
      print(nl.item(i));
      dedent();
    }
  }
}

public void warning(SAXParseException e)
                   throws SAXException {
  printError("Warning", e);
}

public void error(SAXParseException e)
                 throws SAXException {
  printError("Error", e);
}

public void fatalError(SAXParseException e)
                       throws SAXException {
  printError("Fatal Error", e);
  throw e;
```

```
    }

     public static void main(String args[]) {
      if (args.length == 0) {
        System.out.println("No file URI specified");
        System.exit(1);
      }
      // Request a DOM-parser instance from
      // the DocumentBuilderFactory
      DocumentBuilderFactory ParserFactory =
        DocumentBuilderFactory.newInstance();
      ParserFactory.setValidating(true);
      ParserFactory.setNamespaceAware(false);
      // and parse the given document.
      try {
        DocumentBuilder Parser =
           ParserFactory.newDocumentBuilder();
        DOMTree DOMTraversor = new DOMTree();
        Parser.setErrorHandler(DOMTraversor);
        Document DocumentNode = Parser.parse(args[0]);
        DOMTraversor.print((Node)DocumentNode);
      }
      catch(Exception e) {
        System.err.println("Parse error: " + e.getMessage());
      }
    }
  }
```

## 5.7 Aufgaben

**Aufgabe 5.1** [Syntax] Die Syntax-Prüfung geschieht folgendermassen: Man liest das Dokument von vorne nach hinten. Immer, wenn man ein öffnendes Tag sieht, legt man es oben auf den Stack. Immer, wenn man ein schliessendes Tag liest, nimmt man das oberste Tag vom Stapel, und vergleicht, dass die Namen übereinstimmen. Diese beiden Tags gehören dann zusammen. Stimmen die Namen nicht überein, so liegt ein Syntaxfehler vor. Ein Syntaxfehler liegt auch vor, wenn man an ein schliessendes Tag gerät, und der Stapel aber schon leer ist. Schliesslich liegt ein Syntaxfehler auch dann vor, wenn man am Ende des Dokumentes ist, aber der Stack noch nicht leer ist, es also noch öffnende Tags ohne zugehörige schliessende Tags gibt. Leere Tags sind für die Syntaxprüfung unkritisch, da sie ja automatisch sofort wieder geschlossen werden.

Führen Sie diese Prüfung an einem Beispiel durch. Geben Sie jeweils ein Beispiel für jede der drei Fehlermöglichkeiten (Siehe Abschnitt 5.2.4). Diese Beispiele können künstlich sein (mit <a>, <b>, etc.).

**Aufgabe 5.2** [Syntaxbaum] Was ist die Baumdarstellung für das Beispiel- Dokument? Aus [XML] Abschnitt 2.1 der Spezifikation: *As a consequence of this, for each non- root element*

*C in the document, there is another element P in the document such that C is in the content of P, but not in the content of any other element that is in the content of P. P is referred to as parent of C, and C as child of P.*

**Aufgabe 5.3** [Syntaxbaum] Man kann zu einem gegebenen Dokument einen Baum aufbauen, indem man wieder einen Stack verwendet. Dieser Stack enthält nun Knoten des Baumes. Man des Dokument wieder von vorne nach hinten linear durch. Wenn man an ein öffnendes Tag gerät, erzeugt man einen Element-Knoten, und ordnet ihn als Kind-Knoten dem derzeit obersten Knoten auf dem Stack zu. (Eine Ausnahme ist das Dokument-Element, das bei dieser Baum-Variante keinen Vater-Knoten hat. Später wird der Dokument-Knoten diese Rolle spielen, so dass dann der Stack niemals leer ist.) Anschliessend legt man den neu erzeugten Knoten auf den Stack. Wenn man ein schliessendes Tag erreicht, kontrolliert man, dass der Element-Typ mit dem Typ des obersten Knoten auf dem Stack übereinstimmt, und nimmt diesen Knoten dann vom Stack. Bei einem leeren Tag geht man wie bei einem öffnenden Tag vor, aber legt den Knoten nicht auf den Stack (man würde ihn ja sozusagen sofort wieder herunternehmen). Für jeden Abschnitt von Text, der selbst keine weiteren Tags enthält, erzeugt man einen Text-Knoten, und ordnet ihm dem obersten Knoten des Stacks als Kind-Knoten zu. Auch diese Knoten werden nicht auf den Stack gelegt, da Text-Knoten niemals selbst Kind-Knoten haben können.

Führen Sie diesen Algorithmus für das Beispiel- Dokument durch.

**Aufgabe 5.4** [XML] Beschaffen Sie sich einige XML-Dateien und schauen Sie sich den Inhalt an (die DTD am Anfang können Sie noch nicht verstehen, aber die folgenden Daten schon).

`xmlTree`[DTD02] ist eine Art YAHOO! für XML-Dateien im Web. Eine andere Quelle sind Jon Bosak's Shakespeare-Dateien [Bosak].

**Aufgabe 5.5** [XML] Schreiben Sie selbst eine kleine XML-Datei und lassen Sie diese Datei von einem Parser auf Wohlgeformtheit prüfen (z.B. Komponisten und Stücke, Angestellte und Abteilungen).

Internet Explorer 5 enthält einen *non-validating* Parser für XML, und ist sogar für verschiedene UNIX-Betriebssysteme (inklusive Solaris) erhältlich (Siehe [IE5]). Auch Netscape 6 und Mozilla sollen XML unterstützen. Es gibt im Web auch einen XML-Validierungs-Service von mehreren Anbietern, zum Beispiel [XML]. Dort müssen Sie nur die URL Ihrer XML-Datei eintragen. Sie können aber auch einen Parser lokal installieren, z.B. Apache Xerces [Apache] (Xerces ist im wesentlichen eine Bibliothek zum Zugriff auf XML-Daten in eigenen Programmen, aber die beigelegten Beispiel-Programme wie etwa DOMCount können zur Syntax-Prüfung verwendet werden.) Weitere Verweise finden Sie auf der WWW-Seite dieses Kurses [Bra00].

**Aufgabe 5.6** [Fehler] Entwickeln Sie fünf kurze Beispiele für nicht-wohlgeformtes XML, die bei dem Parser Ihrer Wahl fünf unterschiedliche Fehlermeldungen erzeugen.

**Aufgabe 5.7** [DTD] Wie kann man definieren, dass ein Element `a` die Elemente `b` und `c` enthalten muss, aber in beliebiger Reihenfolge (entweder erst `b`, dann `c`, oder erst `c` dann `b`). In SGML gibt es dafür eine eigene Notation, die aber zur Vereinfachung in XML weggelassen wurde.

**Aufgabe 5.8** [Nichtdeterminismus] Das folgende Beispiel ist nicht zulässig:
`<!ELEMENT example ((a, b)*, (a, c)) >`

Geben Sie eine äquivalente deterministische Inhalts-Spezifikation an.

**Aufgabe 5.9** [DOM und SAX] Definieren Sie eine DTD `matrix1.dtd` für quadratische Matrizen. Dabei soll eine Matrix Zeilenweise abgelegt werden. Definieren Sie auch eine zweite DTD `matrix2.dtd`, wobei Matrizen spaltenweise abgelegt werden.

Schreiben Sie mit DOM und mit SAX Programme die eine bezüglich `matrix1.dtd` gültige XML-Datei in eine bezüglich `matrix2.dtd` gültige XML-Datei transformieren.

# A Top-Down Parsing

## A.1 Code generierung

### A.1.1 Semantische Aktionen

Aus Portabilitätsgründen, wird JAVAJASMIN-Assembler [MeDo97a] Code erzeugt [8]

Der Erzeugte Code hat viel ähnlichkeiten mit der Postfix Notation [VanL92b]. Nehmen wir an, es müsse Code für `term "+" expr` generiert werden. Nach Definition ist die Postfix Notation von `term "+" expr` die Konkatenation der Postfix Notation von `term`, der Postfix Notation von `expr` und `"+"`. Der Code für eine Stackmaschine wird ganz ähnlich erzeugt. Der Parser mit rekursivem Abstieg generiert zuerst Code (in Postfix Notation) für `term`, anschliessend für `expr`. Schliesslich wird die Instruktion `iadd` generiert:

| Code für `term` |
|---|
| Code für `expr` |
| `iadd` |

Für die Operationen `"-"`, `"*"` und `"-"` ist die Generierung analog.

Bei einer Zuweisung `<IDENTIFIER»<=>expr` wird zuerst Code für `expr` generiert. Bei der Auswertung dieses Codes wird am Schluss das Resultat auf dem Stack liegen. Es muss also nur noch mittels `istore` gespeichert werden.

| Code für `expr` |
|---|
| `istore o` |

Beim `print` Befehl, e.g. `<PRINT>"("expr")"";"` wird das Resultat ausgegeben, somit nicht gespeichert.

| Code für `expr` |
|---|
| `invokevirtual MCCLib/puti(I)V`<br>`bipush 10`<br>`invokevirtual MCCLib/putc(I)V` |

Die Speicherzuordnung wird vom Parser vorgenommen. Jedesmal, wenn der Scanner einen Identifier erkennt, testet er, ob der Name in der Symboltabelle schon vorhanden ist. Ist dies nicht der Fall, so wird diesem Identifier eine Stackadresse zugeordnet, und er wird in die Symboltabelle eingetragen.

**Beispiel A.1** [Code Generierung] Code Generierung für das Programm

[8] JASMIN kann an der Internetdresse `http://mrl.nyu.edu/~meyer/jvm/jasmin.html` geladen werden.

```
x = 2;
y = 3;
z = x * y + 2;
print(z*z);
print(1);
```

Jeder einfachen Zuweisung (e.g. `x = 2;` entspricht eine Push- und eine Storeoperation.

JASMIN-Programm `T0`:

```
bipush 2
istore_0                        ; x = 2;
bipush 3
istore_1                        ; y = 3;
iload_0
iload_1
imul
bipush 2
iadd
istore_2                        ; z = x * y + 2;
iload_2
iload_2
imul
invokestatic MCCLib/puti(I)V
sipush 10
invokestatic MCCLib/putc(I)V    ; print (z * z);
bipush 1
invokestatic MCCLib/puti(I)V
sipush 10
invokestatic MCCLib/putc(I)V    ; print (1);
return
```

### A.1.2 Semantische Analyse

Für die Grammatik aus Beispiel 2.9 reduziert sich die semantische Analyse auf die Kontrolle, ob Identifier, die auf der rechten Seite einer Zuweisung, oder als Argumente von `print` vorkommen, vorher schon einmal deklariert worden sind, d.h. schon auf der linken Seite einer Zuweisung vorgekommen sind.

**Beispiel A.2** [Code Generierung] Für die Grammatik aus Beispiel 2.9 hat ein Programmgerüst (Codeerzeugungsbesucher) folgende Gestalt:

JAVA-Klasse `GenVisitor`:

```
import java.util.Hashtable;
import java.io.*;

class GenVisitor extends Visitor {
```

```
private Hashtable symbolTable = new Hashtable();

private static int maxStack  = 0;
private static int actStack  = 0;
private static int address = 0;

private PrintStream out;

GenVisitor (PrintStream out) {
  this.out = out;
}

private void emit (String s) {
    out.print(s + "\n");
}

private static void adjustStack (int i) {
  actStack += i;
  if (actStack > maxStack) maxStack = actStack;
}

Object visitStatl(Statl statl, Object o) {

  /* output jasmin header */
  emit(".class public a");
  emit(".super A");
  emit(".method public <init>()V");
  emit("        aload_0");
  emit("        invokespecial A/<init>()V");
  emit("        return");
  emit(".end method\n");
  emit(".method public exec()V");

  for (int j = 0; j < statl.v.size(); j++)
    ((Node) (statl.v.elementAt(j))).accept(this,null);

  /* output jasmin footer */
  emit("      return");
  emit("   .limit locals " + String.valueOf(address+1));
  emit("   .limit stack " + String.valueOf(maxStack));
  emit(".end method");

  /* translate and execute code */
  Exec e = new Exec();
  e.exec();
```

```
  return (null);
}

Object visitAssign(Assign assign, Object o) {
  Identifier id = (Identifier) assign.id;
  int a;
  if (symbolTable.get(id.lexem) == null) {
    a = address++;
    symbolTable.put(id.lexem,
    new Entry(id.lexem,a));
  }
  else {
    Entry e = (Entry) symbolTable.get(id.lexem);
    a = e.value;
  }
  assign.addExpr.accept(this,null);
  emit("      istore " + a + " ; " + id.lexem);
  adjustStack(-1);
  return (null);
}

Object visitPrint(Print print, Object o) {
  print.addExpr.accept(this,null);
  emit("      invokestatic MCCLib/puti(I)V");
  emit("      sipush 10");
  emit("      invokestatic MCCLib/putc(I)V");
  adjustStack(-1);
  return (null);
}

Object visitPlus(Plus plus, Object o) {
  plus.mulExpr.accept(this,null);
  plus.addExpr.accept(this,null);
  emit("      iadd");
  adjustStack(-1);
  return (null);
}

Object visitMinus(Minus minus, Object o) {
  minus.mulExpr.accept(this,null);
  minus.addExpr.accept(this,null);
  emit("      isub");
  adjustStack(-1);
  return (null);
}

Object visitTimes(Times times, Object o) {
```

```
    times.unaExpr.accept(this,null);
    times.mulExpr.accept(this,null);
    emit("      imul");
    adjustStack(-1);
    return (null);
  }

  Object visitDiv(Div div, Object o) {
    div.unaExpr.accept(this,null);
    div.mulExpr.accept(this,null);
    emit("      idiv");
    adjustStack(-1);
    return (null);
  }

  Object visitUminus(Uminus uminus, Object o) {
    uminus.priExpr.accept(this,null);
    emit("      ineg");
    adjustStack(-1);
    return (null);
  }

  Object visitIdentifier(Identifier identifier, Object o) {
    int a = 0;
    if (symbolTable.get(identifier.lexem) == null) {
      System.out.println(" > identifier \"" +
 identifier.lexem +
 "\" not declared");
      System.exit(1);
    }
    else {
      Entry e = (Entry) symbolTable.get(identifier.lexem);
      a = e.value;
        emit("      iload " + a + " ; " + identifier.lexem);
        adjustStack(1);
    }
    return (null);
  }

  Object visitNumber(Number number, Object o) {
    emit("      sipush " + number.n);
    adjustStack(1);
    return (null);
  }
}
```

## A.2 Prädiktive Parser

Ein prädiktiver Parser ist eine nicht rekursive Version eines Parsers mit rekursivem Abstieg. Dabei wird ein Stack explizit verwaltet.

### A.2.1 Funktionsweise

Ein prädiktiver Parser besteht aus einem Eingabepuffer, einem Ausgabestrom, einem Stack, sowie einem deterministischen Automat: die Parse-Tabelle. Der Eingabestring wird mit der Endmarkierung `"$"` abgeschlossen. Der Stack enthält eine Folge von Grammatiksymbolen mit `"$"` abgeschlossen (zwecks Kennzeichnung vom Ende des Stacks). Am Anfang enthält der Stack `"$"` und darüber das Startsymbol der Grammatik. Die Parse-Tabelle ist eine Matrix $M[\alpha, A]$ , wobei $\alpha$ , ein Nichtterminal und $A$ ein Terminal oder `"$"` ist (siehe Abb. A-1).

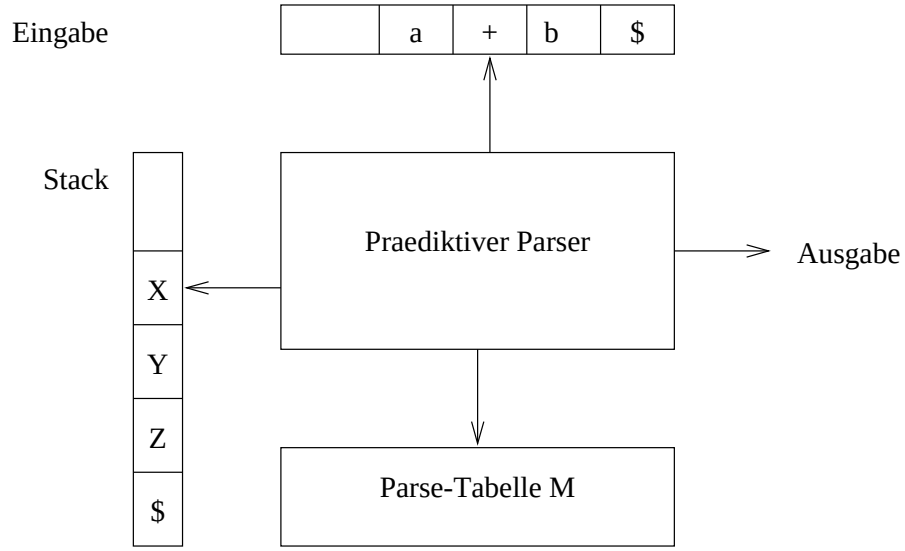



Abbildung A-1: Prädiktiver Parser

Die Steuerung des Parsers übernimmt ein Programm das sich folgendermassen verhält: Er schaut sich $x$ , das oberste Stackelement, und $A$ das aktuelle Eingabesymbol, an. Diese beiden Symbolen entscheiden darüber, was zu tun ist:

1. Falls $x = A =$ `"$"` , stoppt der Parser und meldet den erfolgreichen Abschluss der Syntaxanalyse.
2. Falls $x = A \neq$ `"$"` , entfernt der Parser $x$ vom Stack und setzt den Eingabezeiger auf das nächste Element der Eingabekette.
3. Falls $x$ Nichtterminal, so konsultiert der Parser die Parse-Tabelle $M$ beim Eintrag $[x, A]$ . Dieser Eintrag ist entweder die rechte Seite einer $x$ -Produktion oder ein Fehleintrag. Ist $[x, A] = x ::= uvw$; so setzt der Parser $u$ , $v$ und $w$ auf den Stack ( $u$ oben). Als Ausgabe erfolgt z.B. die angewendete Produktion.

**Algorithmus A.1 (Nichtrekursive prädiktive Syntaxanalyse) Eingabe** *Eine Parse-Tabelle* $M$ *für eine Grammatik* $G$ *und ein String* $w$ *.*

**Ausgabe** *Eine Links-Herleitung von* $w$ *falls* $w$ *in* $L(G)$ *ist; andernfalls eine Fehlermeldung.*

**Methode** *Zu Beginn befindet sich der Parser in einer Konfiguration in der "$" auf dem Stack liegt (mit* $s$ *, dem Startsymbol der Grammatik als oberstes Element) und in der der Eingabepuffer* $w$\$ *enthält. Das Programm zur Durchführung der Syntaxanalyse hat folgende Gestalt:*

*Die Variable* $I_p$ *zeige auf das erste Symbol von* $w$\$ *.*

***repeat***
*sei* $x =$ `top()` *und* $A$ *das Symbol, auf das* $I_p$ *zeigt*
***if*** $x$ *ist Terminal oder "$"**then***
***if*** $x = A$ ***then***
`pop()` *und rücke* $I_p$ *vor*
***else*** `error()`
***else if*** $M[x, A] = x ::= y_1 y_2 ... y_k$ ***;then begin***
`pop()`
`push( `$y_i$` )`, $i = k, ..., 1$
*gib dir Produktion x::=* $y_1 y_2 ... y_k$ *; aus*
***end***
***else*** `error()`
***until*** $x =$ `"$"`

**Beispiel A.3** [Parse Tabelle] Die Grammatik aus Bsp. 2.5 besitzt folgende Parse Tabelle (Leereinträge kennzeichnen Fehler):

```
M[e ,<ID>] = e ::= t e’;
M[e ,"(" ] = e ::= t e’;

M[e’,"+" ] = e’ ::= "+" t e’;
M[e’,")" ] = e’ ::= "";
M[e’,"$" ] = e’ ::= "";

M[t ,<ID>] = t ::= f t’;
M[t ,"(" ] = t ::= f t’;

M[ t’,"+"] = t’ ::= "";
M[ t’,"*"] = t’ ::= "*" f t’;
M[ t’,")"] = t’ ::= "";
M[ t’,"$"] = t’ ::= "";

M[f ,<ID>] = f ::= <ID>;
M[f ,"(" ] = f ::= "(" e ")";
```

Bei der Eingabe <ID>"+"<ID>"*"<ID> führt der Parser folgende Aktionen durch:

Die erste Spalte ist der Stackinhalt, die zweite die anstehend Eingabe und die dritte die Ausgabe des Parsers.

```
"$" e             <id> "+" <id> "*" <id> "$"
"$" e' t          <id> "+" <id> "*" <id> "$"   e ::= t e';
"$" e' t' f       <id> "+" <id> "*" <id> "$"   t ::= f t';
"$" e' t' <ID >   <id> "+" <id> "*" <id> "$"   f  ::= <ID>;
"$" e' t'              "+" <id> "*" <id> "$"
"$" e'                 "+" <id> "*" <id> "$"   t'::= ";
"$" e' t "+"           "+" <id> "*" <id> "$"   e'::= "+" t e';
"$" e' t                   <id> "*" <id> "$"
"$" e' t' f                <id> "*" <id> "$"   t ::= f t';
"$" e' t' <ID>             <id> "*" <id> "$"   f ::= <ID>;
"$" e' t'                       "*" <id> "$"
"$" e' t' f "*"                 "*" <id> "$"   t' ::= "*" f t';
"$" e' t' f                         <id> "$"
"$" e' t' <ID>                      <id> "$"   f ::= <ID>;
"$" e' t'                                "$"
"$" e'                                   "$"   t' ::= "";
"$"                                      "$"   e' ::= "";
```

Man merke, dass die Aktionen des Parsers einer Links-Herleitung der Eingabe entsprechen.

Um die Parsetabelle zu berechnen, werden die Hilfsfunktionen $\mathtt{First}$ und $\mathtt{follow}$ verwendet. Zusätzlich sind die Elemente von $\mathtt{follow}$ sehr nützlich, als synchronisierende Symbole bei Fehler-Recovery.

### A.2.2 Konstruktion von M[x,A]

Eine prädiktive Parse-Tabelle für eine Grammatik G kann mit folgendem Algorithmus erstellt werden. Die Idee dabei ist die folgende: Angenommen $\alpha ::= w; \in P$ und $A \in \mathtt{first}(w)$ A>. Dann expandiert der Parser $\alpha$ zu $w$ , wenn $A$ aktuelles Eingabesymbol ist. Zu Komplikationen kann es kommen, wenn $w = \epsilon$ oder $w \Rightarrow \epsilon$ gilt. In diesem Fall muss $\alpha$ erneut zu $w$ expandiert werden, wenn das aktuelle Eingabesymbol in $\mathtt{follow}(\alpha)$ ist oder wenn in der Eingabe die Endmarkierung "$" erreicht wurde und "$" in $\mathtt{follow}(\alpha)$ enthalten ist.

**Algorithmus A.2 (Parse-Tabelle)** *Konstruktion einer prädiktiven Parse-Tabelle*

**Eingabe** *Grammatik* G .

**Ausgabe** *Parse-Tabelle* M .

**Methode**
1. *Führe für jede Produktion* $\alpha ::= w$; *die Schritte 2 und 3 durch.*
2. *Trage für jedes Terminal* $A$ *aus* $\mathtt{first}(w)$ *die Produktion* $\alpha ::= w$; *in* $M[\alpha, A]$ *ein.*

3. *Wenn* $\epsilon \in \mathit{first}(w)$ *enthalten ist, trage* $a ::= w$; *für jedes Terminal* B *aus* $\mathit{follow}(a)$ *an der Stelle* $M[a, B]$ *ein.Ist* $\epsilon \in \mathit{first}(w)$ *und* `"$"` $\in \mathit{follow}(a)$ , *so trage* $a ::= w$; *in* $M[a,$ `"$"`$]$ *ein.*
4. *Trage in jedem undefinierten Eintrag* `error()` *ein.*

**Beispiel A.4** [Prädiktiver Parser] Für die Grammatik G aus Beispiel 2.5: Weil $\mathit{first}(te') = \mathit{first}(t) = \{$`"("`, `<ID>`$\}$ ist, wird aufgrund der Produktion $e ::= te'$; an den Stellen $M[e,$ `"("`$]$ und $M[e,$ `<ID>`$]$ jeweils $e ::= te'$; eingetragen.

Produktion $e' ::=$ `"+"`$te'$; bewirkt, dass $e' ::=$ `"+"`$te'$; in $M[e',$ `"+"`$]$ eingetragen wird. Produktion $e' ::= \epsilon$ ; bewirkt, dass $e' ::= \epsilon$ ; sowohl in $M[e',$ `")"`$]$ als auch in $M[e',$ `"$"`$]$ eingetragen wird, da $\mathit{follow}(e') = \{$`")"`, `"$"`$\}$ ist.

Die vollständige Tabelle ist in Beispiel 2.5 gezeigt.

**Beispiel A.5** [Prädiktiver Parser] Für die Grammatik G aus Beispiel 2.5 sieht ein Programmgerüst wie folgt aus:

JAVA-program `Prädictiver Parser`:

```
interface ParserConstants {

  final public static int  ID        = 257;
  final public static int  ADDOP     = 258;
  final public static int  MULOP     = 259;
  final public static int  LEFTPAR   = 260;
  final public static int  RIGHTPAR  = 261;
  final public static int  END       = 262;
  final public static int  ENDSIGN   = 263;

  final public static int  EXPR      =   1;
  final public static int  EXPR1     =   2;
  final public static int  TERM      =   3;
  final public static int  TERM1     =   4;
  final public static int  FACTOR    =   5;

  final public static int  ERR       =   0;
  final public static int  P1        =   1;
  final public static int  P2        =   2;
  final public static int  P3        =   3;
  final public static int  P4        =   4;
  final public static int  P5        =   5;
  final public static int  P6        =   6;
  final public static int  P7        =   7;
  final public static int  P8        =   8;
}

class PredictiveParser implements ParserConstants {
```

```
  int production[][] = {
    {       0,      0,         0}, /* ERR */
    {    TERM,  EXPR1,         0}, /*  P1 */
    {   ADDOP,   TERM,     EXPR1}, /*  P2 */
    {       0,      0,         0}, /*  P3 */
    {  FACTOR,  TERM1,         0}, /*  P4 */
    {   MULOP, FACTOR,     TERM1}, /*  P5 */
    {       0,      0,         0}, /*  P6 */
    { LEFTPAR,   EXPR, RIGHTPAR}, /*  P7 */
    {      ID,      0,         0}  /*  P8 */
  };

  int parseTable[][] = {
    {  P1, ERR, ERR,  P1, ERR, ERR}, /*   EXPR */
    { ERR,  P2, ERR, ERR,  P3,  P3}, /*  EXPR1 */
    {  P4, ERR, ERR,  P4, ERR, ERR}, /*   TERM */
    { ERR,  P6,  P5, ERR,  P6,  P6}, /*  TERM1 */
    {  P8, ERR, ERR,  P7, ERR, ERR}  /* FACTOR */
  };

  private int lookahead;
  private int stackPointer = -1;
  private int stack[1000];

  public static void main (String args[]) {
    push(END);
    push(EXPR);
    lookahead = scanner.getNextToken();
    parse();
  }

  void parse () {
    int topOfStack;
    boolean symbolIsTerminal;
    int actualProduction;
    int i;
    do {
      topOfStack = stack[stackPointer];
      if ((topOfStack >= ID) && (topOfStack <= END))
symbolIsTerminal = true;
      else symbolIsTerminal = false;
      if (symbolIsTerminal) {
        if (lookahead == topOfStack) {
          pop();
          if (lookahead != END)
    lookahead = scanner.getNextToken();
```

```
          }
          else error ();
        }
        else {
          actualProduction =
    parseTable[topOfStack-1][lookahead-257];
          if (actualProduction == ERR)
            error ();
          pop();
          for (i=2;i>=0;i--) {
            if (production[actualProduction][i] != 0)
              push(production[actualProduction][i]);
          }
        }
      }
      while (topOfStack != END);
    }
  }
```

## A.3 Aufgaben

**Aufgabe A.1** [Codeerzeugung] Betten Sie den Codegenerator aus Beispiel A.2 direkt in einem Top-Down Parser mit rekursiven Abstieg ein.

**Aufgabe A.2** [Modulo] Erweitern Sie die Grammatik aus Beispiel 2.9 mit dem Modulo Operator, Vergleichsoperatoren und boolsche Operatoren. Erweitern Sie dementsprechend den Parser (oder zugehöriger Besucher) aus Beispiel A.2. Schreiben Sie schliesslich einen Codegenerator für diese erweiterte Grammatik.

**Aufgabe A.3** [Parser Generator] Schreiben Sie einen Parser, der aus einer Menge von Produktionen in BNF Schreibweise eine JAVACC-Spezifikation zur überprüfung der zugehörigen Grammatik erzeugt.

# B Bottom-up Parsing

Die bottom-up Syntaxanalyse versucht, für einen Eingabestring einen Ableitungsbaum zu konstruieren, wobei man an den Blättern (bottom) beginnt und sich zur Wurzel (top) hocharbeitet. Bei dieser Methode wird der Eingabestring $ schrittweise auf das Startsymbol einer Grammatik *reduziert*. Bei jedem Reduktionsschritt wird ein Substring, der mit der rechten Seite einer Produktion übereinstimmt, durch das Terminalsymbol auf der linken Seite ersetzt. Bei richtiger Wahl der Reduktionsschritte ergibt sich eine Rechtsableitung in umgekehrter Reihenfolge.

**Beispiel B.1** [Bottom-up Parsing] Gegeben sei folgende Grammatik

```
s ::= <A> a b <E>
   ;
a ::=   a <B> <C>
      | <B>
   ;
b ::= <D>
   ;
```

Der Satz<A><B><B><C><D><E> kann durch folgende Schritte auf s reduziert werden:

```
<A> <B> <B> <C> <D> <E>
<A> a <B> <C> <D> <E>
<A> a <D> <E>
<A> a b <E>
s
```

Unklar ist dabei wie die Reduktionsschritte gewählt werden müssen. Im ersten Schritt sind nämlich die Substrings<B> und <D> reduzierbar. Gewählt wurde das am weitesten links stehende <B>. Im zweiten Schritt sind a<B><C>, <B> und <D> reduzierbar.

## B.1 LR-Parsing: Übersicht

Wir wollen nun einen Algorithmus vorstellen, der analog zur prädiktiven top-down Analyse aus Stack, Parsetabelle und einem zentralen Automat besteht.

Am Anfang des Analysevorgangs ist der Stack leer. Dann werden die folgenden Schritte durchgeführt, bis das Startsymbol auf dem Stack steht oder ein Syntaxfehler erkannt wird.

- Falls die rechte Seite einer Produktion auf dem Stack steht, so ersetze diese durch das Nichtterminal auf der linken Seite. Diese Operation wird als **reduce**-Operation bezeichnet.
- Im anderen Fall wird das Lookaheadsymbol auf den Stack geschoben. Diese Operation wird als **shift**-Operation bezeichnet.

Im folgenden werden wir mit der erweiterten Grammatik aus Beispiel B.2 arbeiten.

**Beispiel B.2** [Erweiterte Grammatik] Gegeben sei folgende erweiterte Grammatik $G'$

```
P0:  s ::= e "$";
P1:  e ::= e "+" t;
P2:  e ::= e "-" t;
P3:  e ::= t;
P4:  t ::= t "*" f;
P5:  t ::= t "/" f;
P6:  t ::= f;
P7:  s ::= "(" e ")";
P8:  s ::= <ID>;
```

Um das Ende der Eingabe in die Syntaxbeschreibung zu integrieren wurde der Grammatik ein neues Startsymbol `s` hinzugefügt. Die neue Produktion $P_0$ mit `s` auf der linken Seite leitet das alte Startsymbol `e` gefolgt vom Endetoken `"$"` her.

Die Arbeitsweise des Automates soll nun anhand folgenden Beispiels erläutert werden:

**Beispiel B.3** [LR-Parsing] Gegeben sei der Eingabestring `<ID>"*""("<ID>"+"<ID>")""$"`. Zu jedem Schritt ist die Aktion, die Eingabe und der Stack (vor der Aktion) aufgeführt. Mögliche Aktionen sind

**shift** Das Lookaheadsymbol wird auf den Stack geschoben.

**reduce n** Die rechte Seite der Produktion mit Nummer `n` steht auf dem Stack. Diese rechte Seite wird zu den Nichtterminal auf der linken Seite reduziert.

**accept** Die Eingabe wurde zum neuen Startsymbol `s` reduziert. Werden alle ausgeführten Reduktionen zusammengefasst, so ergibt sich eine umgekehrte Rechtsableitung des Eingabestrings.

```
Aktion    Eingabe                                  Stack

shift     <ID> "*" "(" <ID> "+" <ID> ")" "$"
reduce 8       "*" "(" <ID> "+" <ID> ")" "$"   <ID>
reduce 6       "*"" (" <ID> "+" <ID> ")" "$"    f
shift          "*"" (" <ID> "+" <ID> ")" "$"    t
shift              "(" <ID> "+" <ID> ")" "$"    t "*"
shift                  <ID> "+" <ID> ")" "$"    t "*" "("
reduce 8                    "+" <ID> ")" "$"    t "*" "("<ID>
reduce 6                    "+" <ID> ")" "$"    t "*" "(" f"
reduce 3                    "+" <ID> ")" "$"    t "*" "(" t
shift                       "+" <ID> ")" "$"    t "*" "(" e
shift                           <ID> ")" "$"    t "*" "(" e"+"
reduce 8                             ")" "$"    t "*" "(" e"+" <ID>
reduce 6                             ")" "$"    t "*" "(" e"+" f"
```

```
reduce 1                          ")" "$"    t "*" "(" e"+" t
shift                             ")" "$"    t "*" "(" e
reduce 7                              "$"    t "*" "(" e")"
reduce 4                              "$"    t "*" f"
reduce 3                              "$"    t
shift                                 "$"    e
reduce 0                                     e "$"
accept                                       s
```

Wird Beispiel B.3 näher betrachtet, so kann folgendes festgestellt werden:

Steht die rechte Seite einer Produktion auf dem Stack, so ist es an einigen Stellen notwendig, dass eine *shift*-Operation ausgeführt wird, obwohl zugleich auch eine Reduktion durchgeführt werden könnte. Diese Notwendigkeit resultiert daraus, dass bei einer *reduce*- oder *shift*-Operation der weitere Aufbau des Ableitungsbaums bis zum Start sichergestellt werden muss.

Aus diesem Grund und damit der Parser nicht bei jeder Operation den gesamten Stack auf die Anwesenheit der rechten Seite einer Produktion untersuchen muss, werden jeweils teile des Stackinhalts durch Zustände eines Automaten kodiert. In der Tat enthält der Stack nur Zustände. Die Strings sind hier nur aus Illustrationsgründen angegeben.

## B.2 LR-Syntaxanalysealgorithmus

Im Abschnitt B.1 wurde die Syntax mittels dem sogenannten LR-Syntaxanalysealgorithmus analysiert. Wir wollen nun diesen Algorithmus vorstellen, ohne auf die Konstruktion des zugehörigen Automaten einzugehen.

**Definition B.1 (Konfiguration)** *Eine **Konfiguration** eines LR-Parsers ist ein Paar, dessen erste Komponente der Stackinhalt und dessen zweite Komponente die unverbrauchte Eingabe ist:*

$(s_0, x_1, s_1, x_2, s_2, ..., x_m, s_m; A_i, A_{i+1}, ..., A_n\$)$

*Diese Konfiguration repräsentiert die rechtsabgeleitete Satzform* $x_1x_2...x_mA_iA_{i+1}...A_n\$$

Der LR-Parser arbeitet wie folgt: Nach dem Lesen des aktuellen Eingabesymbol $A_i$ , und $s_m$ , dem Zustand an der Spitze des Stacks wird der Eintrag `action`$[ s_m, A_i]$ in der Aktionstabelle aufgesucht. Dabei sind vier verschiedene Aktionen möglich

1. Wenn `action` $[s_m, A_i]$ = `shift` $s$ dann führt der parser eine Schiebeaktion durch und erreicht die Konfiguration $(s_0, x_1, s_1, x_2, s_2, ..., x_m, s_m, A_i, s; A_{i+1}, ..., A_n\$)$
2. Wenn `action` $[s_m, A_i]$ = `reduce` $a ::= w$; dann führt der parser eine Reduzieraktion durch und erreicht die Konfiguration $(s_0, x_1, s_1, x_2, s_2, ..., x_{m-r}, s_{m-r}, a, s; A_i, A_{i+1}, ..., A_n\$)$ wobei $s$ = `goto` $[s_{m-r}, a]$ und $r$ die Länge von $w$ (rechte Seite der Produktion) ist. Die reduzierte Produktion wird dabei ausgegeben, oder die zugehörige semantische Aktion wird ausgeführt.

3. Wenn `action` $[s_m, A_i]$ = `accept` dann terminiert der Parser erfolgreich.

4. Wenn `action` $[s_m, A_i]$ = `error` dann wird eine Fehlerroutine ausgeführt.

**Algorithmus B.1 (LR-Parser) Eingabe** *Ein Eingabestring $w$ und eine LR-Syntaxanalysetabelle mit den Funktion* `action` *und* `goto` *für eine Grammatik* $G$ .

**Ausgabe** *Wenn $w \in L(G)$ , eine Bottom-Up-Syntaxanalyse für $w$ . andernfalls eine Fehleranzeige.*

**Methode** *Anfangs hat der Parser den Startzustand $s_0$ auf seinem Stack, und* \$ *im Eingabepuffer. Der Parser führt dann folgendes Programm aus, bis eine* `accept` *Aktion oder eine* `error` *Aktion erreicht wird.*

*setze $i_p$ , auf das erste Symbol von $w$\$ ;*
***repeat forever begin***
  *sei $s$ der Zustand an der Spitze des Stacks*
    *und $A$ das Symbol, auf das $i_p$ zeigt;*
  ***if*** `action` $[s, A]$ = `shift` $s'$ ***then begin***
    *lege zuerst $A$ , dann $s'$ auf dem Stack;*
    *rücke $i_p$ auf das nächste Eingabesymbol vor;*
  ***end***
  ***else if*** `action` $[s, A]$ = `reduce` $\alpha ::= w$; ***then begin***
    *hole $2|w|$ Symbole vom Stack;*
    *sei $s'$ der Zustand, der nun als oberstes Element*
      *im Stack steht;*
    *lege zuerst $\alpha$ , dann* `goto` $[s', \alpha]$ *auf den Stack;*
    *gebe die Produktion $\alpha ::= w$; aus;*
  ***end***
  ***else if*** `action` $[s, A]$ = `accept` ***then return***
  ***else*** `error()`
***end***

**Bemerkung B.1** [Nichtterminale] Konfigurationen enthalten Nichtterminale. Wir werden später sehen, dass diese Nichtterminale überflüssig sind, d.h. dass es genügt, die Zustände alleine zu speichern. Die Angabe der Terminale macht aber den Algorithmus anschaulicher.

## B.3 Handle

Ein **Handle** (**Henkel**) eines Strings ist ein Substring, der mit der rechten Seite einer Produktion übereinstimmt und dessen Reduktion zum Nichtterminalen auf der linken Seite der Produktion einen Schritt einer inversen Rechtsableitung entspricht. Handles sind also wichtige Instrumente, um richtige Reduktionen ausfindig zu machen.

**Definition B.2 (Handle)** *Ein **Handle** einer rechtsabgeleiteten Satzform $w$ ist eine Produktion $\alpha ::= w_2$ ; und eine Position in $w$ . An dieser Position wird der String $w_2$ gefunden,*

*d.h. $w = w_1w_2w_3$ . Weiter kann $w_2$ in $w$ durch $\alpha$ ersetzt werden, um die in einer Rechtsableitung von $w$ unmittelbar vorherige rechtsabgeleitete Satzform zu erzeugen. Das bedeutet, dass unter der Voraussetzung $S \Rightarrow w_1\alpha w_3 \Rightarrow w_1w_2w_3$ die Produktion $\alpha$ ::= $w_2$ ; an der $w_1$ folgenden Position ein Handle von $w_1w_2w_3$ ist. Der String $w_3$ rechts vom Handle enthält nur Terminalsymbole.*

**Beispiel B.4** [bu:exa:handle] Betrachte die Grammatik aus Beispiel B.2 und die Rechtsableitung

```
e  ::=  e "+" e
   ::=  e "+" e "*" e
   ::=  e "+" e "*" <ID>3
   ::=  e "+" <ID>2 "*" <ID>3
   ::=  <ID>1 "+" <ID>2 "*" <ID>3
```

`<ID>`$_1$ ist ein Handle der rechtsabgeleiteten Satzform `<ID>`$_1$+`<ID>`$_2$`"*"<ID>`$_3$ weil `<ID>` die rechte Seite der Produktion $e$ ::= `<ID>`; ist und das ersetzen von `<ID>`$_1$ durch $e$ die unmittelbar vorher abgeleitete Satzform $e$`"+"<ID>`$_2$`"*"<ID>`$_3$ erzeugt.

Ferner gilt

## B.4 Items

Mit dem Begriff Handle ist eine Situation definiert, die beim Analyseprozess eine Reduktion zulässt. Beim Analysevorgang treten Situationen auf, in denen etwa für einige Nichtterminale auf der rechten Seite einer Produktion bereits ein Teilbaum aufgebaut ist, für andere Nichtterminale aber noch nicht. Diese Situationen werden durch Kodierung des Stackinhaltes ausgedrückt. Diese Kodierung entsteht in Zusammenhang mit **Items**.

**Definition B.3 (Item)** *Ein **Item** ist eine Erweiterung einer Produktion mit einem Punkt auf der rechten Seite. Zum besseren Unterscheiden werden Items in eckige Klammern geschrieben.*

*Der Punkt soll so interpretiert werden, dass für die Grammatiksymbole links vom Punkt bereits der zugehörige Teil des Syntaxbaumes aufgebaut wurde, der Aufbau für die Symbole rechts vom Punkt noch aussteht.*

**Beispiel B.5** [bu:exa:items] Menge der Items zur Produktion `e::=t "+" f;`

| | |
|---|---|
| `[e::=. t "+" f;]` | Die Produktion e::=t `"+"` f; kann potentiell angewendet werden. |
| `[e::=t . "+" f;]` | Ein Teilbaum für `t` wurde bereits aufgebaut. |
| `[e::= t "+" . f;]` | Ein Teilbaum für `t` wurde bereits aufgebaut und das Terminal `"+"` wurde gele |
| `[e::= t "+" f .;]` | Es wurden Teilbäume für `t` und `f` aufgebaut. |

Mit solchen Kodierungen in Form von Items ist es möglich, die Operationen des Automates in Abhängigkeit vom aktuellen Zustand und dem Lookaheadsymbol genau zu definieren. Zustände werden durch Mengen von Items dargestelt.

Es sei $z_\alpha$ , der durch das Item $[\alpha ::= w_1.xw_2]$ beschrieben ist.

- Ist $x$ ein Terminalsymbol und ist das Lookahead auch $x$ , so wird eine `shift`-Operation durchgeführt und $[\alpha ::= w_1x.w_2]$ beschreibt den neuen Zustand.
- Ist $x$ ein Nichtterminal, so wird die $x$ bearbeitende Funktion aufgerufen, d.h. der entsprechende Teilautomat zur Behandlung von $x$ wird aufgerufen.
- Ist $xw_2 = \epsilon$ , so wird eine Reduktion nach der Produktion $\alpha ::= w_1xw_2$; durchgeführt. Sei $s_p$ der Vorgängerzustand von $s_\alpha$ , so wird im Anschluss an die Reduktion ein Zustandsübergang nach $s_p$ durchgeführt. Dieser Zustandsübergang wird als `goto`-Operation oder Jumpoperation bezeichnet.

## B.5 Konstruktion des Automaten

Um einen Bottom-Up-Parser zu konstruieren, muss die Parsetabelle konstruiert werden. Dazu werden zwei Funktionen `Hülle` und `goto` benötigt.

### B.5.1 Hülle

Zustände des Analyseautomates werden durch Items beschrieben. Wird jedem Item ein Zustand zugeordnet, so ist der Automat im Allgemeinen nicht deterministisch. Für die erweiterte Grammatik aus Beispiel B.2 gäbe es $\epsilon$ -Zustandsübergänge vom Zustand $[e' ::= .e]$ zu $[e ::= .e\texttt{"+"}t]$ oder zu $[e ::= .t]$ , etc. Will man einen deterministischen Automaten konstruieren, so ist eine Hüllenoperation notwendig.

**Definition B.4 (Hülle)** *Es sei* $I$ *eine Menge von Items einer Grammatik* $G$ *, dann ist* `Hülle` $(I)$ *wie folgt definiert:*

1. $x \in I \Rightarrow x \in$ `Hülle` $(I)$
2. *Aus* $[\alpha ::= w_1.bw_2] \in$ `Hülle` $(I)$ *und* $b ::= w_3 \in P$ *folgt* $[b ::= .w_3] \in$ `Hülle`*Hülle>* $(I)$

**Beispiel B.6** [bu:exa:huelle] Es sei $G'$ die Grammatik aus Beispiel B.2. Ferner sei $I = \{[e' ::= .e]\}$ . Dann enthält `Hülle` $(I)$ die Elemente:

```
[e' ::= . e]
[e ::= . e "+" t]
[e ::= . e "-" t]
[e ::= . t]
[t ::= . t "*" f]
[t ::= . t "/" f]
[t ::= . f]
[f ::= . "(" e ")"]
[f ::= . <ID>]
```

### B.5.2 Goto

Um die Zustandsübergänge zu beschreiben, wird die Funktion `goto` benötigt. Die Zustände sind Hüllen und die Zustandsübergänge sind mit Terminalsymbolen und Nichtterminalsymbolen der Grammatik beschriftet.

**Definition B.5 (Goto)** *Es sei* I *eine Menge von Items und* x$ *ein Symbol der Grammatik (Terminal oder Nichtterminal). Dann ist*
`goto` (I) = `Hülle` ({[a ::= $w_1$x > x.$w_2$]; [a ::= $w_1$.x$w_2$] ∈ I}

**Beispiel B.7** [Goto] Es sei G′ die Grammatik aus Beispiel B.2. Ferner sei I =}[e′ ::= e.][e ::= e."+"t]} . Dann enthält `goto` (I, `"+"`) die Elemente:

```
[e ::= e "+" . t]
[t ::= . t "*" f]
[t ::= . t "/" f]
[t ::= . f]
[f ::= . "(" E ")"]
[f ::= . <ID>]
```

### B.5.3 Berechnung der LR(0) Mengen

Die Zustände des Automats heissen **LR(0)-Mengen** und werden wie folgt berechnet:

Es sei G′ eine Erweiterung der Grammatik G = (N, T, P, $s_{old}$) mit dem neuen Startsymbol $s_{new}$ . Der Startzustand wird als `Hülle` ([$s_{new}$ ::= .$s_{old}$]) und die Zustände werden wie folgt bestimmt:

**Algorithmus B.2 (LR(0)-Mengen)** *Es sei Z die Menge der Zustände.*

*Setze* Z = {$I_0$ = `Hülle` ([$s_{new}$ ::= .$s_{old}$])}
***repeat***
  *Es sei* Z = {$I_0$, ..., $I_n$}
  ***for*** x ∈ V ***do***
    ***for*** i = 1 ***to*** n ***do***
      ***if*** `goto` ($I_i$, x) ≠ ∅ *und*
        `goto` ($I_i$, x) *nicht in* Z
      ***then*** *nehme* `goto` ($I_i$, x) *in* Z *auf;*
    ***end for***
  ***end for***
***until*** *in* Z *wird keine neue Menge mehr aufgenommen*

**Beispiel B.8** [bu:exa:lr0] Es sei G′ die Grammatik aus Beispiel B.2.

```
I0  = Hülle([s ::= .e])
    = {
        [s ::= .e]
        [e ::= .e  "+"  t]
        [e ::= .e  "-"  t]
        [e ::= .t]
        [t ::= .t  "*"  f]
        [t ::= .t  "/"  f]
        [t ::= .f]
        [f ::= . "(" e ")" ]
        [f ::= .<ID>]
      }

I1  = goto(I0,e) = Hülle(I1)
    = {
        [s ::= e.]
        [e ::= e. "+"  t]
        [e ::= e. "-"  t]
      }

I2  = goto(I0,t) = Hülle(I2)
    = {
        [e ::= t.]
        [t ::= t. "*"  f]
        [t ::= t. "/"  f]
      }

I3  = goto(I0,f) = Hülle(I3)
    = {
        [t ::= f.]
      }

I4  = goto(I0,"(")
    = Hülle([f ::=  "(" .e  ")"])
    = {
        [f ::=  "(" .e ")"]
        [e ::= .e "+" t ")"]
        [e ::= .e "-" t ")"]
        [e ::= .t]
        [t ::= .t "*" f ")"]
        [t ::= .t "/" f ")"]
        [t ::= .f ")"]
        [f ::= . "(" e ")"]
        [f ::= .<ID>]
      }

I5  = goto(I0, <ID>) = Hülle(I5)
```

```
    = {
        [f ::= <ID>.]
      }

I6  = goto(I1,"+") = Hülle([e ::= e "+" .t])
    = {
        [e ::= e "+" .t ")"]
        [t ::= .t "*" f ")"]
        [t ::= .t "/" f ")"]
        [t ::= .f ")"]
        [f ::= . "(" e ")"]
        [f ::= .<ID>]
      }

I7  = goto(I1,"-") = Hülle([e ::= e "-" .t])
    = {
        [e ::= e "-" .t ")"]
        [t ::= .t "*" f ")"]
        [t ::= .t "/" f ")"]
        [t ::= .f ")"]
        [f ::= . "(" e ")"]
        [f ::= .<ID>]
      }

I8  = goto(I2,"*") = Hülle([t ::= t "*" .f])
    = {
        [t ::= t "*" .f ")"]
        [f ::= . "(" e ")"]
        [f ::= .<ID>]
      }

I9 = goto(I2,"/") = Hülle([t ::= t "/" .f])
    = {
        [t ::= t "/" .f  ")"]
        [f ::= . "(" e ")"]
        [f ::= .<ID>]
      }

I10 = goto(I4,e) = Hülle(I10)
    = {
        [f ::=  "(" e. ")"]
        [e ::= e. "+" t]
        [e ::= e. "-" t]
      }

I11 = goto(I6,t) = Hülle(I11)
    = {
```

```
          [e ::= e "+" t.]
          [t ::= t. "*" f]
          [t ::= t. "/" f]
           }


I12 = goto(I7,t) = Hülle(I12)
    = {
        [e ::= e "-" t.]
        [t ::= t. "*" f]
        [t ::= t. "/" f]
      }


I13 = goto(I8,f) = Hülle(I13)
    = {
        [t ::= t "*" f.]
      }

I14 = goto(I9,f) = Hülle(I14)
    = {
        [t ::= t "/" f.]
      }

I15 = goto(I10,")")  = Hülle(I{15})
    = {
        = [f ::=  "(" e ")" .]
      }
```

Somit besitzt der LR(0)-Automat zu Grammatik $G'$ 16 Zustände. Die Zustandsübergänge sind durch die `goto` Mengen beschrieben, d.h. ist $I_j$ = `goto` $(I_k, x)$ so ist $x$ eine Beschriftung des übergangs von Zustand $I_j$ zu Zustand $I_k$ .

### B.5.4 SLR Syntaxanalysetabellen

Es gibt verschiedene Methoden, Syntaxanalysetabellen aufzubauen. Wir zeigen hier die SLR-Methode (Simpel Left Right):

**Algorithmus B.3 (SLR-Syntaxanalysetabelle)** *Konstruktion einer SLR-Syntaxanalysetabelle*

**Eingabe:** *Eine erweiterte Grammatik* $G'$

**Ausgabe:** *Die SLR-Syntaxanalysetabellenfunktionen* `action` *und* `goto`

**[Methode:** 1. *Konstruiere die Menge* $C = \{I_0, I_1, ..., I_n\}$ *der* $LR(0)$ *Mengen für* $G'$ .

2. *Zustand* $i$ *wird aus* $I_i$ *konstruiert. Die Aktionen für den Zustand* $i$ *werden wie folgt bestimmt:*

*(a) Wenn* $[a ::= w_1.Aw_2] \in I_i$ *und* `goto` $(I_i, A) = I_j$ *, dann setze* `action` $[I_i, A]$ *auf* `shift` $j$ *. Hier muss* $A$ *ein Terminal sein.*

*(b) Wenn* $[a ::= w_1.] \in I_i$ *, dann setze* `action` $[I_i, A]$ *für alle* $A$ *aus* $\mathrm{follow}(a)$ *auf* `reduce` $a ::= w_1$ *. Hier darf* $a$ *nicht* $s'$ *sein.*

*(c) Wenn* $[s' ::= s.] \in I_i$ *, dann setze* `action` $[i, \$]$ *auf* `accept`.

*Wenn durch die obigen Regeln irgendwelche Konfliktaktionen generiert werden, sagen wir, die Grammatik ist nicht* `SLR(1)`*. Der Algorithmus produziert in diesem Fall keinen Parser.*

3. *Die Zustandsübergänge für Zustand* $i$ *werden für alle Nichtterminale* $a$ *unter Verwendung der folgenden Regel konstruiert: Wenn* `goto` $[I_i, a] = I_j$ *setze* `goto` $[i, a] = j$ .
4. *Alle Einträge, die nicht durch die Regeln (2) und (3) definiert sind, werden auf* `error` *gesetzt.*
5. *Der Anfangszustand des Parsers ist* $[s' ::= .s]$

**Beispiel B.9** [SLRtabellen] Es sei $G'$ die Grammatik aus Beispiel B.2.

Die `shift`- und `goto`-Tabelle hat folgende Gestalt:

```
 s | <ID> "+" "-" "*" "/" "(" ")" "$"  e   t   f
-------------------------------------------------
 0 |  s5                  s4           1   2   3
 1 |      s6  s7                   acc
 2 |              s8  s9
 3 |
 4 |  s5                  s4          10   2   3
 5 |
 6 |  s5                  s4              11   3
 7 |  s5                  s4              12   3
 8 |  s5                  s4                  13
 9 |  s5                  s4                  14
10 |      s6  s7              s15
11 |              s8  s9
12 |              s8  s9
13 |
14 |
15 |
```

Um die Reduktionen festzulegen, müssen noch die $\mathrm{follow}$ Mengen zu den Nichtterminalen bestimmt werden:

```
first(e)  = { "(", <ID> }
first(t)  = { "(", <ID> }
first(f)  = { "(", <ID> }
follow(e) = { "+", "-", ")",  $}
follow(t) = { "+", "-", "*", "/", ")", "$" }
follow(f) = { "+", "-", "*", "/", ")", "$" }
```

Daraus ergibt sich folgende `reduce`-Tabelle. Dabei beteutet `ri`, dass nach Produktion `i` reduziert werden soll.

```
 s | <ID> "+" "-" "*" "/" "(" ")" "$"
-------------------------------------
 0 |
 1 |
 2 |      r3  r3              r3  r3
 3 |      r6  r6  r6  r6      r6  r6
 4 |
 5 |      r8  r8  r8  r8      r8  r8
 6 |
 7 |
 8 |
 9 |
10 |
11 |      r1  r1              r1  r1
12 |      r2  r2              r2  r2
13 |      r4  r4  r4  r4      r4  r4
14 |      r5  r5  r5  r5      r5  r5
15 |      r7  r7  r7  r7      r7  r7
```

**Bemerkung B.2** [Tabellenaufbau] Beim Tabellenaufbau im Beispiel B.9 sind keine Konflikte aufgetreten. Werden die `shift`- und die `reduce`-Tabelle zusammengefügt, so gibt es höchstens einen Eintrag pro Stelle in der Tabelle.

**Beispiel B.10** [bu:exa:slrimpl] Es sei G′ die Grammatik aus Beispiel B.2. Ein C-Programmgerüst für die Bottom-Up-Syntaxanalyse hat folgende Gestalt. Dabei wurden die `shift` und `goto` Tabellen auseinandergenommen.

C-Programm `LR-Parser`:

```
#define S           0
#define E           1
#define T           2
#define F           3
#define ID          4
#define ADD         5
#define SUB         6
#define MUL         7
#define DIV         8
#define LPAR        9
#define RPAR       10
#define END        11

#define P0          0 /* s := E $                */
#define P1          1 /* E := E ADD T            */
```

```
#define P2          2 /* E := E SUB T            */
#define P3          3 /* E := T                  */
#define P4          4 /* T := T MUL F            */
#define P5          5 /* T := T DIV F            */
#define P6          6 /* T := F                  */
#define P7          7 /* F := LPAR E RPAR        */
#define P8          8 /* F := ID                 */

int rightSide[] = {2,3,3,1,3,3,1,3,1};

int shiftTable[16][8] =
{
  {5, 0, 0, 0, 0, 4, 0, 0}, {0, 6, 7, 0, 0, 0, 0, 0},
  {0, 0, 0, 8, 9, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0},
  {5, 0, 0, 0, 0, 4, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0},
  {5, 0, 0, 0, 0, 4, 0, 0}, {5, 0, 0, 0, 0, 4, 0, 0},
  {5, 0, 0, 0, 0, 4, 0, 0}, {5, 0, 0, 0, 0, 4, 0, 0},
  {0, 6, 7, 0, 0, 0,15, 0}, {0, 0, 0, 8, 9, 0, 0, 0},
  {0, 0, 0, 8, 9, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0}
};

int reduceTable[16][8] =
{
  {0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0},
  {0,P3,P3, 0, 0, 0,P3,P3}, {0,P6,P6,P6,P6, 0,P6,P6},
  {0, 0, 0, 0, 0, 0, 0, 0}, {0,P8,P8,P8,P8, 0,P8,P8},
  {0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0}, {0,P1,P1, 0, 0, 0,P1,P1},
  {0,P2,P2, 0, 0, 0,P2,P2}, {0,P4,P4,P4,P4, 0,P4,P4},
  {0,P5,P5,P5,P5, 0,P5,P5}, {0,P7,P7,P7,P7, 0,P7,P7}
};

int gotoTable[16][3] =
{
  { 1, 2, 3}, { 0, 0, 0}, { 0, 0, 0}, { 0, 0, 0},
  {10, 2, 3}, { 0, 0, 0}, { 0,11, 3}, { 0,12, 3},
  { 0, 0,13}, { 0, 0,14}, { 0, 0, 0}, { 0, 0, 0},
  { 0, 0, 0}, { 0, 0, 0}, { 0, 0, 0}, { 0, 0, 0}
};

main ()
{
  push(0);
  lookahead = yylex();
  parse();
```

```
}

int shift (int z)
{
  push(z);
  lookahead = yylex();
}

int jump (int z)
{
  push(z);
}

int reduce(int prod)
{
  int i;
  for (i=1;i<=rightSide[prod];i++) pop();
}

int parse ()
{
  while (1) {
    int false = 1;
    state = top();
    if ((state == 1) && (lookahead == END)) {
      exit(0);
    }
    if ((lookahead >= ID) && (lookahead <= END)) {
      if (shiftTable[state][lookahead-ID] != 0) {
        shift(shiftTable[state][lookahead-ID],lookahead);
        false = 0;
      }
      if (reduceTable[state][lookahead-ID] != 0) {
        state = reduce(reduceTable[state][lookahead-ID],
                       lookahead);
        false = 0;
        if (gotoTable[state][symbol-1] != 0)
          jump(gotoTable[state][symbol-1]);
        else {
          error();
        }
      }
    }
    if (false == 1) error();
  }
}
```

## B.6 Bemerkungen

**Definition B.6 (SLR(1)-Grammatik)** *Eine Grammatik* G *heisst* ***SLR(1)****, wenn für jede der mit* G *herleitbaren Satzform das Handle eindeutig durch*

1. *den Inhalt des Stacks*
2. *das Lookaheadsymbol*

*definiert ist, wobei die Konstruktion der Parsingtabellen nach dem SLR-Algorithmus B.3 erfolgt.*

**Bemerkung B.3** [Konflikte] Das Verfahren ist nur anwendbar, falls die Grammatik eindeutig ist. Bei mehrdeutigen Grammatiken entstehen Konfliktsituationen:

1. `shift/reduce` Konflikte
2. `reduce/reduce` Konflikte

`shift/reduce` Konflikte entstehen, wenn es Produktionen gibt, die einen gemeinsamen echten Präfix besitzen. `reduce/reduce` Konflikte entstehen wenn es Produktionen gibt, die einen gemeinsamen echten Suffix besitzen. `reduce/reduce` Konflikte sind durch Rechtsfaktorisierung zu vermeiden.

**Bemerkung B.4** [LALR-Technik] Als Syntaxanalysemethode soll auch die **LALR**-Technik (lookahead-LR) erwähnt werden. Sie ist sehr ähnlich zur SLR-Technik, produziert aber bedeutend kleinere Zustandstabellen. Diese Methode wird bei YACC und BISON verwendet [LeMa92a]. Die manuelle Erzeugung eines Bottom-Up-Parsers, selbst für eine sehr einfache Grammatik ist ziemlich komplex und fehleranfällig. Zu diesem Zweck sollen auf jeden Fall Parsergeneratoren verwendet werden.

## B.7 Aufgaben

**Aufgabe B.1** [Shift/Reduce-Tabellen] Erweitere die Grammatik aus Beispiel B.2 mit dem Modulooperator und berechne die `shift`- und `reduce`-Tabellen.

**Aufgabe B.2** [SLR-Tabellen] Betrachte folgende Grammatik G

```
s ::= a s | "B" ;
a ::= s a | "A" ;
```

Konstruiere die SLR-Syntaxanalysetabelle von G .

# C Parser Generatoren

## C.1 Bison

Wir wollen nun die Beispiele aus Kapitel 4 mit BISON implementieren:

### C.1.1 Tischrechner 1

BISON-Spezifikation `b1.y`:

```
%token NUMBER

%%
statlist  :  /* empty */
             { printf("Programm ist korrekt\n"); }
          | statlist statement
             { printf("Programm ist korrekt\n"); }
          ;
statement : expr ';'
          ;
expr      : expr '+' expr
          | expr '-' expr
          | expr '*' expr
          | expr '/' expr
          | '-' expr
          | '(' expr ')'
          | NUMBER
          ;
%%
main()
{
  yyparse();
}

yyerror(char *s)
{
  printf("%s\n", s);
}
```

Der von BISON erzeugte Parser ruft die Funktion `yylex` auf, welche die Aufgabe hat, den nächsten Token zurückzuliefern. Entweder wird `yylex` von Hand erstellt oder mit FLEX erzeugt. Hier folgt die FLEX-Spezifikation:

FLEX-Spezifikation `s1.l`:

```
%{
#include "b1.tab.h"
```

```
%}
number [0-9]+
%%
" "|\t          ;
"\n"|"\r"       ;
{number}        {return NUMBER;}
.               {return yytext[0];}
%%
```

Im Falle eines Fehlers ruft BISON die Funktion `yyerror` auf. Diese Funktion, sowie die `main` Funktion müssen vom Anwender zur Verfügung gestellt werden.

### C.1.2 Tischrechner 2

Zu jeder Produktion der Grammatik kann eine Aktion in der Sprache C angegeben werden. Falls im Laufe der Syntaxanalyse eine Reduktion mit Hilfe der entsprechenden Produktion vorgenommen wird, so wird der Code der Aktion unmittelbar nach der Reduktion ausgeführt.

BISON-Spezifikation `b2.y`:

```
%token NUMBER

%%
statlist  :  /* empty */
          | statlist statement
          ;
statement : expr ';' { printf("= %d\n", $1); }
          ;
expr      : expr '+' expr {$$ = $1 + $3;}
          | expr '-' expr {$$ = $1 - $3;}
          | expr '*' expr {$$ = $1 * $3;}
          | expr '/' expr {$$ = $1 / $3;}
          | '-' expr      {$$ = -$2;}
          | '(' expr ')'  {$$ = $2;}
          | NUMBER        {$$ = $1;}
          ;
%%
main()
{
  yyparse();
}

yyerror(char *s)
{
  printf("%s\n", s);
}
```

Die BISON-Spezifikation benutzt sogenannte Symbolwerte (alle Variablen, die mit $ beginnen). Jedes Symbol hat in einem BISON-Parser einen Wert. Dieser Wert enthält zusätzliche Informationen zu einem Symbol. Der Wert eines Tokens kann zum Beispiel eine Zahl, ein String oder ein Zeiger sein.

Der zugehörige Scanner hat folgende Gestalt:

FLEX-Spezifikation `s2.l`:

```
%{
#include "b2.tab.h"
%}
number [0-9]+
%%
" "|\t          ;
"\n"|"\r"       ;
{number}        {yylval = atoi(yytext);
                 return NUMBER;}
.               {return yytext[0];}
%%
```

Wir wollen nun untersuchen, wie sich der Parser verhält.

```
10 + 20;
= 30
10 - 3 + 5;
= 2
3 + 3 * 4;
= 15
3 * 4 + 3;
= 21
(3 * 4) + 3;
= 15
a;
parse error
```

Wir sehen, dass unserer Tischrechner nicht nach den üblichen Regeln rechnet. Alle Operationen haben dieselbe Priorität und es wird rechtsassoziativ gerechnet. Das heisst, `a + b + c` wird als `a + (b + c)` interpretiert und nicht wie üblich als `(a + b) + c`.

Die Probleme mit der Auswertung rühren daher, dass die Grammatik keine eindeutige Rechtsableitung besitzt (siehe etwa das Beispiel 1.7). Es ist auch so, dass BISON beim Erstellen des Parsers für die Grammatik die Meldung `<file> contains 20 shift/reduce conflicts` ausgibt.

### C.1.3 Tischrechner 3

Die folgende Deklaration bewirkt, dass die Operatoren `"+"`, `"-"`, `"*"` und `"/"` linksassoziativ sind und dass `"*"` und `"/"` höhere Priorität als `"+"` und `"-"` besitzen. Das Token \texttt{NEG} hat die höchste Priorität und keine Assoziativität.

```
%left '+' '-'
%left '*' '/'
%nonassoc NEG
```

BISON-Spezifikation `b3.y`:

```
%token NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc NEG         /* unary minus    */
%%
statlist  :  /* empty */
          | statlist statement
          ;
statement : expr ';' {printf("= %d\n", $1);}
          ;
expr      : expr '+' expr          {$$ = $1 + $3;}
          | expr '-' expr          {$$ = $1 - $3;}
          | expr '*' expr          {$$ = $1 * $3;}
          | expr '/' expr          {$$ = $1 / $3;}
          | '-' expr %prec NEG     {$$ = -$2;}
          | '(' expr ')'           {$$ = $2;}
          | NUMBER                 {$$ = $1;}
          ;
%%
main()
{
  yyparse();
}

yyerror(char *s)
{
  printf("%s\n", s);
}
```

### C.1.4 Tischrechner 4

Die folgende Definition bewirkt, dass das Token `NUMBER` eine ganze Zahl als Symbolwert annehmen kann und das Token `IDENTIFIER` einen String.

```
%union {
  int    val;
  char   lexem[BSIZE+1];
}

%token <val>NUMBER, <lexem>IDENTIFIER
```

Es ist die Sache der Scannerfunktion in der Variablen `yylval` die richtige Variante abzulegen.

Nun haben wir alle Elemente, die wir brauchen, um den JAVA Assembler Code zu generieren. Die folgende BISON Spezifikation erzeugt einen Parser, der JAVA Assembler Code für die Grammatik aus dem Beispiel 2.9 generiert.

BISON-Spezifikation `b4.y`:

```
%{
#include "glodef.h"
#include "glovar.h"

int actStack = 0;
int regNr = 0;
int maxStack = 0;


%}
%union {
  int val;
  char lexem[BSIZE+1];
}
%token <val>NUMBER, <lexem>IDENTIFIER, PRINT
%left '-' '+'
%left '*' '/'
%left NEG         /* unary minus    */
%%
program   : '{' statlist '}'
          ;

statlist  :  /* empty */
          | statlist statement
          ;

statement : IDENTIFIER
            {int adr;
             adr = lookup($1);
             if (adr < 0)
             {
               ++regNr;
               adr = insert($1, regNr);
             }
             $<val>$ = adr;
            }

            '=' expr ';'            {printf("\nistore %d",
                  symbolTable[$<val>2].address);
             adjustStack(-1);
```

```
            }

          | PRINT '(' expr ')' ';'
            { printf(
               "\ninvokestatic MCCLib/puti(I)V");
              printf("\nsipush 10");
              printf(
               "\ninvokestatic MCCLib/putc(I)V");
             adjustStack(-1);
            }
          ;

expr      : expr '+' expr
            {printf("\niadd");
             adjustStack(-1);
            }
          | expr '-' expr
            {printf("\nisub");
             adjustStack(-1);
            }
          | expr '*' expr
            {printf("\nimul");
             adjustStack(-1);
            }
          | expr '/' expr
           {printf("\nidiv");
            adjustStack(-1);
            }

          | '-' expr %prec NEG
            {printf("\nineg");}

          | '(' expr ')'

          | NUMBER
            {printf("\nsipush %d", $1);
             adjustStack(1);
            }

          | IDENTIFIER
            {int adr;
             adr = lookup($1);
             if (adr < 0)
             {
               yyerror("Semantik error\n");
               YYABORT;
             }
```

```
              else
                printf("\niload %d",
                     symbolTable[adr].address);
              adjustStack(1);
             }
          ;
%%

main (int argc, char *argv[])
{
  if (argc < 2)
    in = stdin;
  else
  {
    in = fopen(argv[1], "r");
    if (in == NULL)
    {
      printf("Illegal File %s\n", argv[1]);
      exit(1);
    }
  }
  header();
  yyparse();
  footer();
}

yyerror(char *s)
{
  printf("%s\n", s);
}

int adjustStack (int i) {
  actStack += i;
  if (actStack > maxStack) maxStack = actStack;
}
```

Die Funktion `adjustStack` berechnet die maximale Höehe des Java Stacks. Die Java Methoden `void puti(int i)` und `void putc (int c)` werden in der Klasse `MCCLib` zur Verfügung gestellt. Sie dienen zur Ausgabe von ganzzahligen, bzw ASCII Werte.

### C.1.5 Fehlerbehandlung

Um den Parser nach einem Fehler zu resynchronisieren, stellt BISON das spezielle Token `error` zur Verfügung. Dieses Token kann an beliebiger Stelle auf der rechten Seite einer Produktion stehen. Die Resynchronisation funktioniert nun folgendermassen: Beim Auftreten eines Fehlers macht der Parser `shift/reduce` Aktionen rückgängig bis er einen Zu-

stand findet, in dem er das Token `error` auf dem Stack shiften kann. Anschliessend werden soviele Tokens im Eingabestrom überlesen bis ein Token kommt, das in der Grammatik dem `error` Token folgen kann. Der Parser versucht nun nach diesem Token die Syntaxanalyse fortzusetzen.

BISON-Spezifikation `b5.y`:

```
%{
#define MELDUNG(m) { \
  printf("%s at line: %d at position: %d\n", \
         m, tokline, tokpos); \
}

extern int tokline,
           tokpos;

%}

%union {
  int val;
  char lexem[33];
}

%token IF NUMBER IDENTIFIER

%%
statlist      : /* empty */
              | statlist statement
              | error ';'
                {MELDUNG("unknown error");}
              ;
statement     : assignstmt
              | ifstmt
              ;
assignstmt    : IDENTIFIER '=' expr ';'
              | IDENTIFIER error ';'
                {MELDUNG("illegal statement");}
              | IDENTIFIER '=' error ';'
                {MELDUNG("illegal expression");}
              ;
ifstmt        : IF '(' expr ')' statement
              | IF error ')'
                {MELDUNG("missing (");}
              | IF '(' expr error ';'
                {MELDUNG("missing )");}
              | IF '(' error ')'
                {MELDUNG("error in expression");}
              | IF error ';'
```

```
                {MELDUNG("illegal if");}
              ;
expr          : NUMBER
              ;
%%
main()
{
  yyparse();
}

yyerror(char *s)
{
  ;
}
```

# D Uebungen

## D.1 First und Follow

Wir haben schon gelernt, dass es eine BNF-Grammatik für BNF-Grammatiken gibt.

```
bnf        ::=   production bnf
                | ""
                ;
production ::=   <NONTERMINAL> "::=" choice ";"
                ;
choice     ::=   sequence
                | sequence "|" choice
                ;
sequence   ::=   symbol
                | symbol sequence
                ;
symbol     ::=   <NONTERMINAL>
                | <TERMINALTOKEN>
                | <TERMINALSTRING>
                ;
```

Ferner haben wir im Script gesehen, wie die Mengen $first$ und $follow$ berechnet werden können. Wir wollen nun ein Programm zur Berechnung von $first$ und $follow$ schreiben.

### D.1.1 Datenstrukturen für Symbole

Die atomare Bestandteile einer Produktion sind Terminal- und Nichtterminalsymbole . Für Terminalsymbole sind folgende Formate erlaubt: Terminalstring (`"a"`, und Terminaltoken (`<T>`). Für jeden Symboltyp gibt es eine Klasse, sowie eine Gemeinsame Oberklasse `Symbol`.

JAVA-Klasse `Symbol`:

```
class Symbol {

  protected String name;
  protected boolean nullable = false;
  protected java.util.Hashtable fi = new java.util.Hashtable();
  protected java.util.Hashtable fo = new java.util.Hashtable();

  protected java.util.Hashtable first () {
    return (fi);
  }

  protected java.util.Hashtable follow () {
    return (fo);
  }
```

```
  public String toString() {
    String s = new String();
    java.util.Enumeration e;
    s = s.concat(name + "\n");
    s = s.concat("  FIRST    : ");
    if (nullable) s = s.concat("\"\" ");
    e = fi.keys();
    while (e.hasMoreElements()) {
      s = s.concat((String) e.nextElement() + " ");
    }
    s = s.concat("\n  FOLLOW   : ");
    e = fo.keys();
    while (e.hasMoreElements()) {
      s = s.concat((String) e.nextElement() + " ");
    }
    return(s);
  }

  private boolean add (java.util.Hashtable h1, java.util.Hashtable h2
    boolean result = false;
    java.util.Enumeration e = h2.elements();
    while (e.hasMoreElements()) {
      Symbol s =  (Symbol) e.nextElement();
      if (! h1.containsKey(s.name)) {
h1.put(s.name,h2.get(s.name));
result = true;
      }
    }
    return (result);
  }

  protected boolean addFirst (java.util.Hashtable h) {
    return (add(fi,h));
  }

  protected boolean addFollow (java.util.Hashtable h) {
    return (add(fo,h));
  }
}
```

Die Attribute `fi` und `fo` der Klasse `Symbol` implementieren die Mengen $first$ und $follow$ mittels einer Hashtabelle. Kann aus einem Symbol der Nullstring `""` abgeleitet werden, so gehört $\epsilon$ zu seiner $first$ -Menge. Diese Eigenschaft wird mit Hilfe des boolschen Attributs `nullable` implementiert.

JAVA-Klasse `TerminalSymbol`:

```
class TerminalSymbol extends Symbol {

  TerminalSymbol (String name) {
    this.name = name;
    if (name.equals("\"\"")) {
      // the empty String "" is our symbol for epsilon
      nullable = true;
    }
    else {
      // init with FIRST(X) = {X}
      fi.put(name,this);
    }
  }
}
```

Der Konstruktor der Klasse `TerminalSymbol` setzt `nullable` auf `true` beim der Instanzierung des leeren Strings `""`.

JAVA-Klasse `NonTerminalSymbol`:

```
class NonTerminalSymbol extends Symbol {

  NonTerminalSymbol (String name) {
    this.name = name;
  }
}
```

### D.1.2 Datenstrukturen für Productionen

In einem ersten Schritt werden alle Produktionen, die eine Auswahl enthalten in mehreren Produktionen zerlegt. Z.B. p::=a b | c; wird in p::=a b; und p::=c; zerlegt. Somit ist die rechte Seite einer Produktion immer eine Sequenz von Terminal- und Nichtterminalsymbolen.

JAVA-Klasse `Production`:

```
class Production {

  protected NonTerminalSymbol lhs;
  protected java.util.Vector rhs = new java.util.Vector();

  Production(NonTerminalSymbol lhs, java.util.Vector rhs) {
    this.lhs = lhs;
    this.rhs =rhs;
  }

  public final String toString() {
    String s = new String(lhs.name + " ::=");
    for (int i = 0; i < rhs.size(); i++) {
```

```
        s = s.concat(" " + ((Symbol) rhs.elementAt(i)).name);
      }
      s = s.concat(" ;");
      return (s);
    }

  }
```

Das Attribut `lhs` ist die linke Seite der Produktion und ist somit einen Instanz von `NonTerminalSymbol`. Das Attribut `rhs` implementiert die rechte Seite der Produktion in Form eines JAVA-`Vector`.

### D.1.3 Datenstrukturen für die Grammatik

Die Klasse `Context` ist für die Implementierung der Grammatik und des Algorithmus zur Berechnung von *first* und *follow* verantwortlich

JAVA-Klasse `Context`:

```
  class Context {

    protected java.util.Vector productions = new java.util.Vector();
    private java.util.Hashtable terminals = new java.util.Hashtable();
    private java.util.Hashtable nonTerminals = new java.util.Hashtable(
    private ProductionReader productionReader = new ProductionReader(th

    protected NonTerminalSymbol ntInstance (String s) {
      // singleton
      NonTerminalSymbol nts = (NonTerminalSymbol)nonTerminals.get(s) ;
      if (nts == null) {
        nts = new NonTerminalSymbol(s);
        nonTerminals.put(s,nts);
      }
      return (nts);
    }

    protected TerminalSymbol tInstance (String s) {
      // singleton
      TerminalSymbol ts = (TerminalSymbol)terminals.get(s) ;
      if (ts == null) {
        ts = new TerminalSymbol(s);
        terminals.put(s,ts);
      }
      return (ts);
    }

    private boolean isNullable(java.util.Vector v, int i, int j) {
      if (j < i) return true;
```

```
    if (i < 0) return true;
    if (j >= v.size()) return true;
    boolean nullable = true;
    for (int k = i; k <= j; k++) {
      nullable = nullable && ((Symbol) v.elementAt(k)).nullable;
    }
    return (nullable);
  }

  protected void readProductions () {
    productionReader.readProductions ();
  }

  protected void computeFirstAndFollow () {
    boolean changed;
    do {
      changed = false;
      java.util.Enumeration e = productions.elements();
      while (e.hasMoreElements()) {
Production p = (Production) e.nextElement();
        if (p.rhs.size() == 0 && p.lhs.nullable == false) {
  p.lhs.nullable = true;
  changed = true;
}
for (int i = 0; i < p.rhs.size(); i++) {
  if (isNullable(p.rhs,0,p.rhs.size()-1)) {
    if (!p.lhs.nullable) {
      p.lhs.nullable = true;
      changed = true;
    }
  }
  if (isNullable(p.rhs,0,i-1)) {
    Symbol s = (Symbol) p.rhs.elementAt(i);
    changed = changed || p.lhs.addFirst(s.first());
  }
         if (i == p.rhs.size() -1) {
    Symbol s = (Symbol) p.rhs.elementAt(i);
    changed = changed || s.addFollow(p.lhs.follow());
  }
  for (int j = i+1; j < p.rhs.size(); j++) {
    if (isNullable(p.rhs,i+1,p.rhs.size()-1)) {
      Symbol s = (Symbol) p.rhs.elementAt(i);
      changed = changed || s.addFollow(p.lhs.follow());
    }
    if (isNullable(p.rhs,i+1,j-1)) {
      Symbol s = (Symbol) p.rhs.elementAt(i);
      changed = changed ||
```

```
s.addFollow(((Symbol) p.rhs.elementAt(j)).first());
    }
  }
}
      }
    }
    while (changed);
  }

  protected void printFirstAndFollow() {
    java.util.Enumeration e;
    System.out.println("> productions\n");
    e = productions.elements();
    while (e.hasMoreElements()) {
      Production p = (Production) e.nextElement();
      System.out.println(p.toString());
    }
    System.out.println("\n> first and follow\n");
    e = nonTerminals.elements();
    while (e.hasMoreElements()) {
      Symbol s = (Symbol) e.nextElement();
      System.out.println(s.toString());
    }
  }
}
```

Die Menge der Terminalsymbole ist mittels der Hashtabelle `terminals` implementiert. Die Menge der Nichtterminalsymbole ist mittels der Hashtabelle `nonterminals` implementiert (Eindeutigkeit der Symbole). Die Produktionen werden in einem JAVA-`Vectorproductions` verwaltet.

Man beachte, dass Terminal- und Nichtterminalsymbole nur dann instanziert werden, wenn sie noch nicht in der entsprechenden Hashtabelle eingetragen wurden. Dies erfolgt mittels Methoden `tInstance()` und `ntInstance()`

Die Methode (Algorithmus) `computeFirstAndFollow()` ist im Skript beschrieben.

### D.1.4 Datenstrukturen für das Einlesen der Produktionen

Die Produktionen (in BNF-Notation, d.h. eventuell mit "|"-Operatoren) werden mit einem JAVACC-Parser eingelesen. Beim Einlesen müssen die Produktionen in Einzelteile zerlegt (Produktionen ohne "|"-Operatore) werden. Hier folgt ein Gerüst des JAVACC-Parsers:

JAVACC-Spezifikation `ProductionReader`:

```
PARSER_BEGIN(ProductionReader)
import java.util.Vector;

public class ProductionReader{
```

```
  private static Context context;

  ProductionReader (Context context) {
    this(System.in);
    this.context = context;
  }

  public static void readProductions () {
    try {
      bnf();
    }
    catch (Exception e) {
      System.out.println("parse error");
    }
  }
}

PARSER_END(ProductionReader)

TOKEN :
{
  < SEMICOLON: ";" >
| < IS: "::=" >
| < OR: "|" >
| < EMPTY: "\"\"" >
| < NONTERMINAL: ["a"-"z"] ( ["a"-"z","A"-"Z","0"-"9"] )* >
| < TERMINALTOKEN: "<" ["A"-"Z"] ( ["A"-"Z","0"-"9"] )* ">" >
| < TERMINALSTRING: "\""
                    (   (~["\"","\\","\n","\r"])
                      | (   "\\"
                            (   ["n","t","b","r","f","\\","'","\""]
                              | ["0"-"7"] ( ["0"-"7"] )?
                              | ["0"-"3"] ["0"-"7"] ["0"-"7"]
                            )
                        )
                    )+
                    "\""
  >
}

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
```

```
| "\f"
}

void bnf() :
{}
{
  ( production() )+ <EOF>
}

void production():
{}
{
  <NONTERMINAL> <IS> choice() <SEMICOLON>
}

void choice() :
{}
{
  sequence() ( <OR> sequence() ) *
}

void sequence() :
{}
{
  ( symbol() )+
}

Symbol symbol() :
{}
{
  <NONTERMINAL>
| <TERMINALTOKEN>
| <TERMINALSTRING>
| <EMPTY>
}
```

### D.1.5 Main

Die Klasse `Main` instanziert die `Context`-Klasse, liest die Produktionen ein (vom der standard Eingabe), Berechnet die first und follow Mengen und gibt das Resultat auf die standard Ausgabe aus.

JAVA-Klasse `Main`:

```
class Main {

  public static void main (String args[]) {
```

```
    Context c = new Context();
    c.readProductions();
    c.computeFirstAndFollow();
    c.printFirstAndFollow();
  }
}
```

### D.1.6 Aufgabe

Vervollständigen Sie die JAVACC-Spezifikation `ProductionReader` mit semantischen Aktionen, damit Terminal-, Nichtterminalsymbole und Produktionen instanziert werden. Es entsteht somit ein lauffähiges Programm zur Berechnung von $first$ und $follow$ .

### D.1.7 Testdateien

Verwenden Sie zum Testen folgende Testdateien `prog/ue/ff/bnf.txt`, `prog/ue/ff/g3.15.txt` und `prog/ue/ff/g3.16.txt`.

### D.1.8 Lösung

JAVACC-xSpezifikation `ProductionReader`:

```
PARSER_BEGIN(ProductionReader)

import java.util.Vector;

public class ProductionReader{

  private static Context context;
  private static Production p;
  private static Vector rhs = new Vector();
  private static NonTerminalSymbol lhs;

  ProductionReader (Context context) {
    this(System.in);
    this.context = context;
  }

  public static void readProductions () {
    try {
      bnf();
    }
    catch (Exception e) {
      System.out.println("parse error");
    }
  }
```

```
}

PARSER_END(ProductionReader)

TOKEN :
{
  < SEMICOLON: ";" >
| < IS: "::=" >
| < OR: "|" >
| < EMPTY: "\"\"" >
| < NONTERMINAL: ["a"-"z"] ( ["a"-"z","A"-"Z","0"-"9"] )* >
| < TERMINALTOKEN: "<" ["A"-"Z"] ( ["A"-"Z","0"-"9"] )* ">" >
| < TERMINALSTRING: "\""
                    (   (~["\"","\\","\n","\r"])
                      | (   "\\"
                            (   ["n","t","b","r","f","\\","'","\""]
                              | ["0"-"7"] ( ["0"-"7"] )?
                              | ["0"-"3"] ["0"-"7"] ["0"-"7"]
                            )
                        )
                    )+
                    "\""
  >
}

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

void bnf() :
{}
{
  production() ( bnf() )? <EOF>
}

void production():
{
  Token t;
  NonTerminalSymbol lhs;
  Vector rhs;
}
{
```

```
  t=<NONTERMINAL>
  {
    lhs = context.ntInstance(t.image);
  }
  <IS> choice(lhs)
  <SEMICOLON>
}

void choice(NonTerminalSymbol lhs) :
{
  Vector rhs;
  Production p;
}
{
  rhs = sequence()
  {
    p = new Production(lhs,rhs);
    context.productions.addElement(p);
  }
  (
    <OR> rhs = sequence()
  {
    p = new Production(lhs,rhs);
    context.productions.addElement(p);
  }
  ) *
}

Vector sequence() :
{
  Symbol s;
  Vector rhs = new Vector();
}
{
  ( s = symbol() { rhs.addElement(s); } )+
  { return rhs; }
}

Symbol symbol() :
{
  Token t;
  NonTerminalSymbol nts;
  TerminalSymbol ts;
}
{
  t = <NONTERMINAL>
  {
```

```
    nts = context.ntInstance(t.image);
    return nts;
  }
| t = <TERMINALTOKEN>
  {
    ts = context.tInstance(t.image);
    return ts;
  }
| t = <TERMINALSTRING>
  {
    ts = context.tInstance(t.image);
    return ts;
  }
| t = <EMPTY>
  {
    ts = context.tInstance(t.image);
    return ts;
  }
}
```

# Literatur


[AhUl77a] Aho, A.V., Ullman, J.D. Principles of Compiler Design. Addison-Wesley. Reading, Massachusetts. 1977.

[Apache] `http:// xml.apache.org/xerces-c/index.html`.

[Appe98] Appel, A.W. Modern Compiler Implementation in Java. Cambridge University Press. 1998.

[ASU86a] Aho, A.V., Sethi, R., Ullman, J.D. Compilers: Principles, Techniques and Tools. Addison-Wesley. Reading, Massachusetts. 1986.

[ASU92a] Aho, A.V., Sethi, R. , Ullman, J.D. Compilerbau. Addison-Wesley. Bonn. 1992. Band 1.

[ASU92b] Aho, A.V., Sethi, R., Ullman, J.D. Compilerbau. Addison-Wesley. Bonn. 1992. Band 2.

[Balz96a] Balzert, H. Lehrbuch der Software-Technik. Spektrum Akademischer Verlag. Heidelberg. 1996. ISBN 3-8274-0042-2.

[BaRe76a] Bauer, F.L., De Remer, F.L., Ershov, A.P., Gries, D., Griffiths, M., Hill, U., Horning, J.J., Koster, C.H.A., Mc Keeman W.M., Poole, P.C. , Waite, W.M. Compiler Construction, An Advanced Course. Springer. New York. 1976. 2. Auflage.

[Bos99] `http://www.sciam.com/1999/0599issue/0599bosak.html`.

[Bosak] `http://sunsite.unc.edu/pub/sun-info/standards/xml/eg/shakespeare`

[Bra00] `http://www.informatik.uni-giessen.de/staff/brass/`.

[Bra00] `http://www.informatik.uni-giessen.de/staff/brass/xml00`.

[Cuni80a] Cunin, P.Y., Griffiths, M., Voiron, J. Comprendre la Compilation. Springer. Berlin. 1980.

[DOM] `http://www.w3.org/TR/REC-DOM-Level-1/`.

[Donn92a] Donnelly, C., Stallman, R. Bison, the YACC-compatible Parser Generator. 1992.

[DTD01] `http://www.dtd.com/`.

[DTD02] `http://www.xmltree.com/`.

[DTD03] `http://www.biztalk.org/`.

[DTD04] `http://xml.org/`.

[Gamm95a] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns, Elements of Reusable Object-Oriented Software . Addison-Wesley. Reading, Massachusetts. 1995.

[Gamm96a] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Entwurfsmuster, Elemente wiederverwendbarer objektorintierter Software . Addison-Wesley. Bonn. 1996.

[GJS96a] Goslin, J., Joy, B., Steele, G. The Java Language Specification. Addison-Wesley. Reading, Massachusetts. 1996.

[Gold84a] Goldschlager, L., Lister, A. Informatik, Eine moderne Einführung. Hanser. München. 1984.

[Gosl96d] Gosling, J. The Java Programming Language. Addison-Wesley. Reading, Massachusetts. 1996.

[GoYe96b] Gosling, J., Yellin, F., The Java Team The Java Application Programming Interface, Core Packages . Addison-Wesley. Reading, Massachusetts. 1996. Band 1.

[GoYe96c] Gosling, J., Yellin, F., The Java Team The Java Application Programming Interface, Window Toolkit and Applets . Addison-Wesley. Reading, Massachusetts. 1996. Band 2.

[Hero92a] Herold, H. lex und yacc, Lexikalische und syntaktische Analyse. Addison-Wesley. Bonn. 1992.

[HoUl88a] Hopcroft, J.E., Ullman, J.D. Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie . Addison-Wesley. Bonn. 1988.

[HTML] `http://www.w3.org/TR/REC-html40/sgml/dtd.html`.

[HTML] `http://www.w3.org/TR/REC-html40/strict.dtd`.

[HTML] `http://www.w3.org/TR/REC-html32#dtd`.

[HTML] `http://validator.w3.org/`.

[IE5] `http://www.microsoft.com/windows/ie/download/ie5.htm`.

[KeRi90a] Kernighan, B.W., Ritchie, D.M. Programmieren in C: mit dem C-Reference Manual in deutscher Sprache . Carl Hanser und Prentice-Hall International. München. 1990. 2. Auflage.

[LeMa92a] Levine, J.R., Mason, T., Brown, D. lex & yacc. O'Reilly & Associates. Inc.. Sebastopol, CA 95472. 1992. 2. Auflage.

[Lesk75a] Lesk, M.E. Lex - a lexical analyser generator. Murray Hill, N.J.. 1975.

[LiYe96a] Lindholm, T., Yellin, F The Java Virtual Machine Specification. Addison-Wesley. Reading, Massachusetts. 1996.

[MeDo97a] Meyer, J., Downing, T. Java Virtual Machine. O'Reilly. 1997.

[Schm92a] Schmitt, F.J. Praxis des Compilerbaus. Hanser. München. 1992.

[SGML] `http://www.w3.org/TR/NOTE-sgml-xml-971215`.

[UNICODE] http://www.unicode.org/.

[VanL90a] van Leeuwen, J. Handbook of Theoretical Computer Science, Algorithms and Complexity . Elsevier. Amsterdam. 1990. Band A.

[VanL92b] van Leeuwen, J. Handbook of Theoretical Computer Science, Formal Models and Semantics . Elsevier. Amsterdam. 1990. Band B.

[WiMa92a] Wilhelm, R., Maurer, D. Übersetzerbau. Springer. Berlin. 1992.

[Wirt86a] Wirth, N. Compilerbau. Teubner. Stuttgart. 1986. 4. Auflage.

[Wirt96a] Wirth, N. Grundlagen und Techniken des Compilerbaus. Addison-Wesley. Bonn. 1996.

[XLi] http://www.w3.org/TR/xlink/.

[XML] http://www.w3.org/TR/REC-xml.

[XML] http://www.w3.org/TR/REC-xml-names/.

[XML] http://www.w3.org/XML/Query.

[XML] http://www.w3.org/XML/Schema.

[XML] http://www.stg.brown.edu/service/xmlvalid/.

[XMTML] http://www.w3.org/TR/xhtml1/.

[XPa] http://www.w3.org/TR/xpath.

[XPo] http://www.w3.org/TR/xptr.

[XSL] http://www.w3.org/TR/xsl/.

[XSLT] http://www.w3.org/TR/xslt.