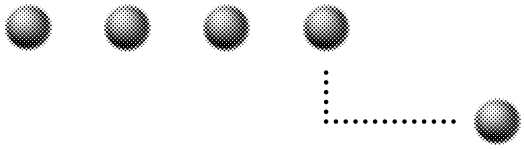


Bern University of Applied Sciences

Engineering and Information Technology

# COMPILERBAU

**J. Boillat und P. Fierz**

**Keywords:** Compilerbau, Scanner, Parser, Semantische Analyse, Codegenerierung



© J. Boillat und P. Fierz

**Zusammenfassung**

Scanning Techniken: Deterministische endliche Automaten, nichtdeterministische endliche Automaten, reguläre Ausdrücke, Scanner Generatoren. Parsing: Formale Sprachen, Top Down Parsing, Bottom Up Parsing, Parser Generatoren. Semantische Analyse: Attributgrammatiken, semantische Aktionen. Code Generierung: Elementare Methoden der Code Optimierung und Generierung, Realisation mittels JAVA Byte Code. Anwendungen der Compilerbau-Techniken: Compilerbau-Techniken und XML (Einsatz von SAX und DOM Parsern), dynamisches Parsing (Aufbau eines validierenden XML Parser), Textverarbeitung (Automatische Übersetzung von LaTeX nach XML).

# Kapitel 1

# Einführung

Dieser Skript ist entstanden aus den Unterlagen der *Compilerbau* Vorlesung, die seit etwa 20 Jahren gehalten wird.

Als begleitende Literatur empfehlen wir folgende Bücher: *Drachenbuch* von Aho, Lam, Sethi und Ullmann [ASU07], die *Theorie der Automaten und Sprachen* von Hopcroft, Motwani und Ullmann [HMU07] und das *Tigerbuch* von Appel [App98].

Wir erwarten kaum, dass unsere Studierende in der Praxis Compiler schreiben werden. Sie sind aber jedentag mit allerlei Übersetzungsproblemen konfrontiert. Aus diesem Grund liegt der Schwerpunkt der Vorlesung in Scanning- und Parsing-Techniken. Codeerzeugung wird zwar auch vollständigkeitshalber behandelt. Zur Vereinfachung wird mit JAVA-Bytecode [LY96] [MD97] gearbeitet.

Die Version 2008 des Skripts wurde mit einem Parser-Generator Projekt erweitert. Dieses Vorgehen ermöglicht es, die Theorie mit der Praxis zu verbinden.

## 1.1 Worum geht es?

Ein **Compiler** [Wir96] [ASU07] [App98] ist ein Programm für die **Übersetzung** eines in einer gegebenen Quellsprache geschriebenen Programms in eine andere Sprache (siehe Abb. 1-1). Quellprogramm und Zielprogramm sollen dabei äquivalent sein.

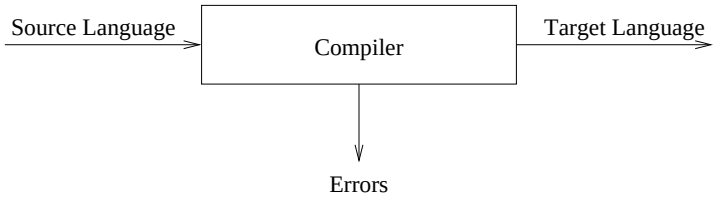



Abbildung 1-1: Compilers

In diesem Kurs ist die Quellsprache eine Programmiersprache wie C [KR90], PASCAL [Wir71], JAVA [GJS96] etc. und die Zielsprache irgendeine Maschinensprache.

**Bemerkung 1.1 [Compilerbau]** Die Ideen und Methoden des Compilerbaus können aber in sehr vielen Gebieten der Softwareentwicklung eingesetzt werden: Texteditoren, Text-

formatierung, Mustererkennung, Betriebssysteme, Entwicklung von Halbleiterbausteinen (Silicon Compiler), etc.

## 1.2 Übersicht

Die übersetzung (siehe Abb.1-2) eines Quellprogramms in ein Zielprogramm besteht logisch aus 2 Bereichen, jeweils unterteilt in mehreren Phasen:

- Analyse Phase
  - Lexikalische Analyse
  - Syntaxanalyse
  - Semantische Analyse
- Synthese Phase
  - Zwischencode Erzeugung
  - Code Optimierung
  - Code Erzeugung

**Bemerkung 1.2 [Phasen]** Die Phasen der Übersetzung sind rein logische Begriffe und sollten nicht mit der Anzahl Läufe (Passes) verwechselt werden. Ein Lauf ist ein Durchgang (Einlesen) durch das gesamte Programm, bzw. das Einlesen eines Zwischenfiles. Normalerweise können mehrere Phasen (sogar alle) in einem Lauf erledigt werden: Wirth PASCAL ist ein 1-Pass Compiler, Richie C ist ein 3-Pass Compiler, Wirth MODULA-2 [Wir89] ist ein 5-Pass, ADA95 ist ein 6-Pass Compiler, etc.

### 1.2.1 Lexikalische Analyse (Scanning)

Die lexikalische Analyse ist eine lineare Analyse. Es geht darum, die Zeichen, aus denen das Quellprogramm besteht, von links nach rechts zu lesen

1. und in gültige Wörter der Quellsprache zu gruppieren,
2. sowie **Tokens** (Symbole) zu produzieren, d.h. den gültigen Wörtern eine Bedeutung zu geben.

Sind die Wörter Bezeichner, so werden sie in eine **Symboltabelle** eingetragen, falls sie noch nicht darin enthalten sind. Diese Tabelle enthält sogenannte **lexikalische Werte**, sowie zusätzliche Informationen wie Gültigkeitsbereich, Typ, Speicherung, Parameter, etc.

### 1.2.2 Syntaxanalyse (Parsing)

Bei der **Syntaxanalyse** geht es darum zu testen, ob die Tokens grammatikalisch korrekte Sätze der (Programmier) Sprache bilden. Diese Analyse ist hierarchisch, d.h. die Sätze des Quellprogramms werden im allgemeinen durch einen Baum (Parsebaum, Ableitungabaum) implizit oder explizit dargestellt.

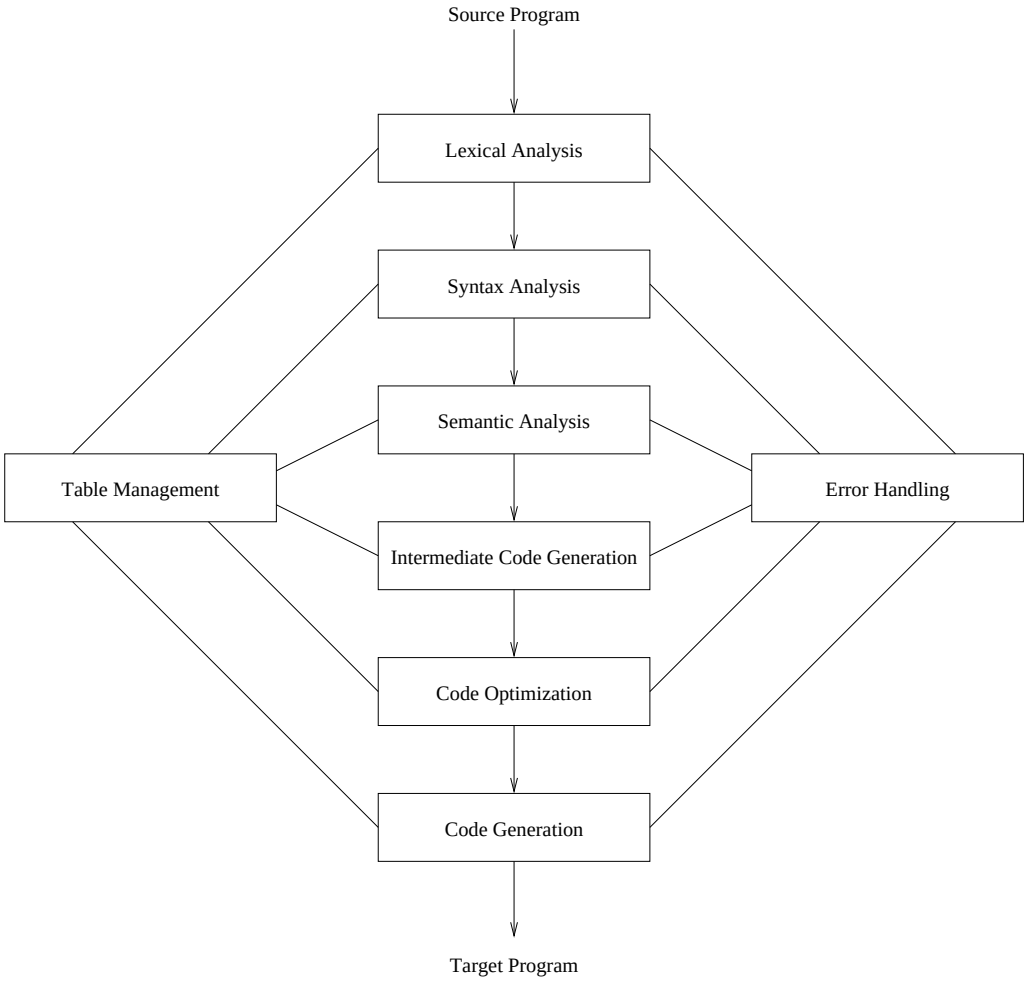



Abbildung 1-2: Phasen eines Compilers

### 1.2.3 Semantische Analyse

Bei der semantischen Analyse geht es darum, soweit wie möglich zu prüfen, ob die aus der Syntaxanalyse resultierenden Sätze sinnvoll gebildet sind. Ein wesentlicher Bestandteil der semantischen Analyse ist die Typüberprüfung.

### 1.2.4 Zwischencodeerzeugung

Die Zwischencode Darstellung ist ein Programm für eine **abstrakte Maschine**. Meistens wird dabei **Drei-Adress-Code** verwendet: eine Assemblersprache für eine Maschine, in der jeder Speicherplatz als Register fungieren kann. Drei-Adress-Code besteht aus Instruktionen, die höchstens drei Operanden besitzen.

Diese Zwischendarstellung ist besonders wichtig, falls Code für verschiedene Zielmaschinen erzeugt werden muss, denn bis und mit Zwischencode Darstellung kann der Compiler maschinenunabhängig arbeiten.

### 1.2.5 Code Optimierung

In dieser Phase wird versucht, den Zwischencode zu verbessern. Wesentliche Schritte sind: Verringerung der Anzahl Hilfsvariablen, Typumwandlung zur Compilationszeit, Ausführung von Zwischencode, Berechnungen mit Konstanten, optimale Ausnutzung des Instruktionssatzes der Zielmaschine, etc.

### 1.2.6 Code Erzeugung

Nebst der übersetzung von (optimierten) Drei-Adress-Code Instruktionen in Zielcode, geht es darum, jeder Variablen einen Speicherplatz zuzuordnen. Entscheidend ist dabei die optimale Zuordnung von Registern zu Variablen.

### 1.2.7 Fehlerbehandlung

Bei jeder Phase können Fehler auftreten. Diese Fehler sollten so behandelt werden, dass die übersetzung möglichst weit fortgeführt werden kann, somit weitere Fehler auch entdeckt und behandelt werden können. Die meisten Fehler werden während der syntaktischen und der semantischen Analyse ausfindig gemacht.

### 1.2.8 In der Praxis

**Beispiel 1.1 [Phasen]** Gegeben sei folgende Eingabe

```
float x, y;
int z;
/* ... */
x = y + z * 20;
```

Bei der lexikalischen Analyse des Deklarationsteils werden folgende Einträge in die Symboltabelle gemacht:

```
...   ...    ...   ...
452    x    float  ...
453    y    float  ...
454    z     int   ...
...   ...    ...   ...
511  const   int   20
...   ...    ...   ...
```

Ein Eintrag in der Symboltabelle enthält u.a. Informationen über die Namen, Typen, etc. der Bezeichner.

Die lexikalische Analyse der obigen Zuweisung erzeugt nachträglich eine Folge von 8 Tokens:

```
<ID_1>, <ASSIGN>, <ID_2>, <PLUS>,
<ID_3>, <MUL>, <CONST>, <SEMICOLON>
```

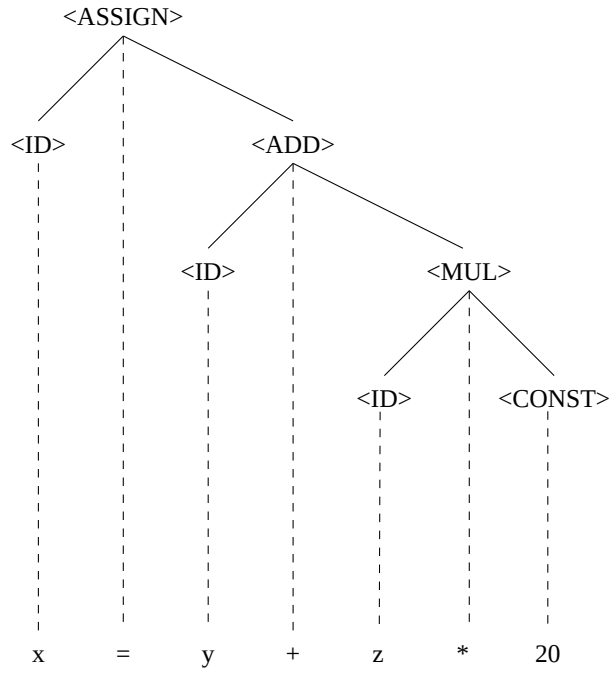
Bei Konstanten, Variablen und Labels wird noch zusätzlich nach einem entsprechenden Eintrag in der **Symboltabelle** gesucht. Falls nicht vorhanden, wird ein Eintrag gemacht. Die Symboltabelle wird in den weiteren Schritten der Compilation verwendet.

Genauer könnte die Folge

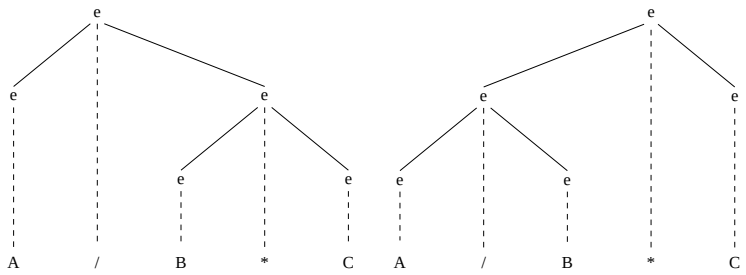
```
[<ID_1>,452], <ASSIGN>, [<ID_2>,453], <PLUS>,
[<ID_3>,454], <MUL>, [<CONST>,511], <SEMICOLON>
```

erzeugt werden, wobei die Zahlen Symboltabellenindexe sind.

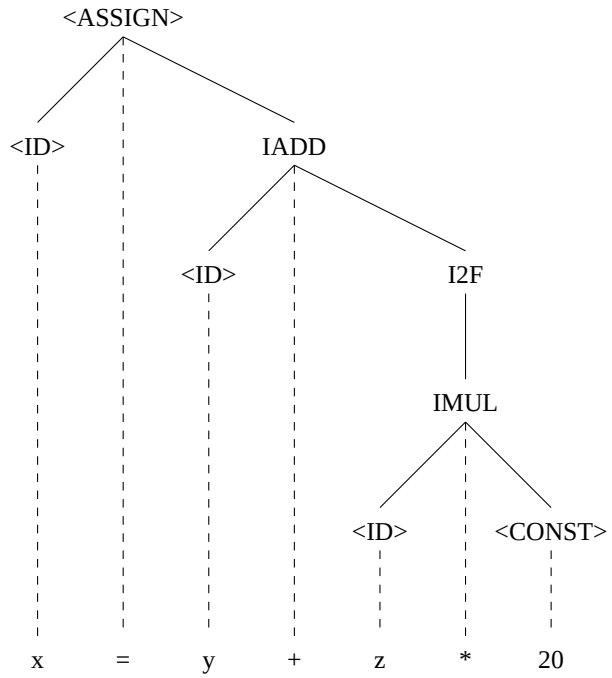
Die Syntaxanalyse erkennt einen gültigen Satz und erzeugt folgenden Ableitungsbaum:



Man bemerke, dass durch die Baumstruktur eine Hierarchie der Operationen definiert wird. Diese Hierarchie wird von der unterliegenden Grammatik der Programmiersprache bestimmt. Sie ist bei der Zuweisung `x = y + z * 20;` nicht sichtbar. So könnte etwa der Ausdruck `A / B * C` je nach Grammatik verschiedentlich interpretiert werden:




Bei der semantischen Analyse wird erkannt, dass der Wert von `z * 20` in eine Gleitkommazahl umgewandelt werden muss. Der Ableitungsbaum wird dementsprechend erweitert.




Jetzt kann Drei-Adress-Code erzeugt werden:

```
IMUL id3   20    t1
I2F  t1    NULL  t2
FADD id2   t2    t3
FMOV t3    NULL  id1
```

Je nach Methode werden jeweils temporäre Namen generiert, die die Werte der berechneten Instruktionen enthalten.

Hier ist es offensichtlich, dass die Hilfsvariable `t3` überflüssig ist. Es erfolgt somit folgende Zwischencode Optimierung.

```
IMUL id3   20    t1
I2F  t1    NULL  t2
FADD id2   t2    id1
```

Wird `id3` nach der Zuweisung `x = y + z * 20;` im Quellprogramm nicht mehr verwendet, so könnte noch auf die temporäre Variable `t1` verzichtet werden:

```
IMUL id3   20    id3
I2F  id3   NULL  t2
FADD id2   t2    id1
```

Die Code Erzeugung ist je nach Zielmaschine verschieden. Für die JAVA Stackmaschine [LY96] könnte z.B. folgender Zielcode erzeugt werden:

```
iload 3
bipush 20
imul
i2f
fload 2
fadd
fstore 1
```

Man bemerke, dass die temporären Variablen vollständig verschwunden sind; z.B. wird der Wert von `id3 * 20` auf den Stack gelegt und kann sofort mit der Instruktion `i2f` weiterverarbeitet werden.

## 1.3 Geschichtliches

- 1950: Erste Compiler für höhere Programmiersprachen (FORTRAN [BBB+57])
- 1963: Revised ALGOL60-Report [Dij60]: formale Definition der Syntax einer Programmiersprache.
- nach 1960: verschiedene Methoden der Syntaxanalyse.
- 1975: LEX [Les75] Automatischer Scannergenerator.
- 1975: YACC [LMB92] Automatischer Parsergenerator.
- 1995: JAVA [Gos96].

## 1.4 Bootstrapping

Es [Wir86] wird mit Recht behauptet, dass es am einfachsten geht, Compiler für eine Sprache *L* in der Sprache *L* selber zu schreiben. Das ist nur möglich, falls ein solcher Compiler überhaupt existiert. Wie wurde der erste Compiler für die Sprache *L* überhaupt übersetzt?

Wir verwenden die folgende Notation: $C_Z(X \rightarrow Y)$ , wobei $C_Z$ ein in der Sprache $Z$ geschriebener Compiler zur übersetzung der Sprache $X$ in die Sprache $Y$ ist.

Gegeben sei eine (neue) Sprache $L$ , gesucht sind Compiler $C_A(L \rightarrow A)$ und $C_B(L \rightarrow B)$ . Dabei wird vorausgesetzt, dass Interpreter oder Compiler für die Sprachen $A$ und $B$ vorhanden sind, z.B. kann $A$ oder $B$ ein Assembler oder sogar Maschinencode sein.

Das Vorgehen erfolgt schrittweise unter Verwendung einer (oder mehrerer) Untersprachen.

1. Wähle eine Untersprache $S \subset L$
2. Schreibe $C_A(S \rightarrow A)$
3. Schreibe $C_S(L \rightarrow A)$
4. $C_S(L \rightarrow A) \rightarrow (C_A(S \rightarrow A)) \rightarrow C_A(L \rightarrow A)$ (1. Ziel erreicht)
5. Schreibe $C_L(L \rightarrow B)$
6. $C_L(L \rightarrow B) \rightarrow (C_A(L \rightarrow A)) \rightarrow C_A(L \rightarrow B)$ (Crosscompiler)
7. $C_L(L \rightarrow B) \rightarrow (C_A(L \rightarrow B)) \rightarrow C_B(L \rightarrow B)$

Die Schritte 5-7 können für beliebige andere Compiler $C_X(L \rightarrow X)$ wiederholt werden

**Bemerkung 1.3 [Bootstrapping Anwendung]** Bootstrapping ist auch bei der Überprüfung der Korrektheit von Parsern sehr nützlich: Der Parser wird mit einer sog. *Identity Transformation* erweitert, die ein Programm in sich selber transformiert.

Ein Program $X$ wird mit dem Parser nach $X'$ übersetzt, anschliessend wird $X'$ nach $X''$ übersetzt. Jetzt müssen $X'$ und $X''$ vollständig identisch sein sonst stimmt mit Sicherheit bei der Übersetzung etwas nicht.[1]

### 1.4.1 Aufgaben

**Aufgabe 1.1 [Bootstraping]** *Gegeben seien $m$ Programmiersprachen $L_1$ , $L_2$ , ... $L_m$ und $n$ Zielmaschinen $M_1$ , $M_2$ , ..., $M_n$ .*

*Gesucht sind $nm$ Compiler $C_{M_j}(L_i \rightarrow M_j)$ , $i = 1, .., m$ ; $j = 1, ..., n$ .*

*Wie können diese $nm$ Compiler mit minimalem Aufwand geschrieben werden?*

[1] Der C++ Compiler GCC wird auch bei der Installation auf eine Maschine M ähnlich überprüft: Zuerst wird der Quellcode GCC mit dem C Compiler der Maschine $M$ compiliert dies ergibt eine erste Version GCC1 des Compilers. Anschliessend wird der Quellcode GCC mit GCC1 compiliert, dies ergibt eine zweite Version GCC2. Die Installation ist nur dann erfolgreich, wenn GCC1 und GCC2 identisch sind.

# Kapitel 2

# Lexikalische Analyse (Scanning)

## 2.1 Einführung

Dieses Kapitel beschäftigt sich mit Techniken zur Spezifikation und Implementation von **Scannern** [HMU07] [ASU07]. Diese Techniken sind nicht nur im Hinblick auf den Compilerbau von Interesse, denn sie können auf andere Anwendungsbereiche wie Anfragesprachen und Informationssysteme übertragen werden. In allen Anwendungen besteht das Kernproblem in der Spezifikation und im Entwurf von Programmen, deren Aktionen durch bestimmte **Textmuster** ausgelöst werden.

Eine weitere Anwendung von Scannern ist **Pattern-Matching** das sich ganz allgemein mit dem Problembereich des Vergleichens und Erkennens von Mustern beschäftigt (also nicht nur Text).

### 2.1.1 Die Rolle des Scanners in einem Compiler

Der Scanner bildet die erste Phase des Compilers. Seine Hauptaufgabe besteht darin, Eingabezeichen einzeln zu lesen und als Ausgabe eine Folge von Symbolen (**Tokens**) zu generieren, die der Parser syntaktisch analysieren kann. Der Scanner agiert üblicherweise als Subroutine oder Coroutine des Parsers. Das heisst, der Parser gibt bei Bedarf den Befehl *nächstes Symbol* an den Scanner, worauf dieser solange Eingabezeichen liest, bis er das nächste gültige Symbol erkannt hat. Dieses Symbol wird zusammen mit einem Symboltabelleneintrag dem Parser übergeben (siehe Abb. 2-1).

Neben der Erkennung von gültigen Symbolen hat der Scanner noch die folgenden Aufgaben

- Reinigung des Quellprogrammes von Kommentaren sowie von nicht druckbaren Zeichen (Leerzeichen, Tabulatoren, Zeilenwechseln usw.)
- Festhalten der Position (Zeile und Kolonne) der einzelnen Symbole. Diese Information wird vom Parser zum Ausgeben von Fehlermeldungen verwendet.
- Wenn die Quellsprache Funktionen eines Macro-Processors anbietet, können diese Funktionen ebenfalls als Teil der lexikalischen Analyse implementiert werden.

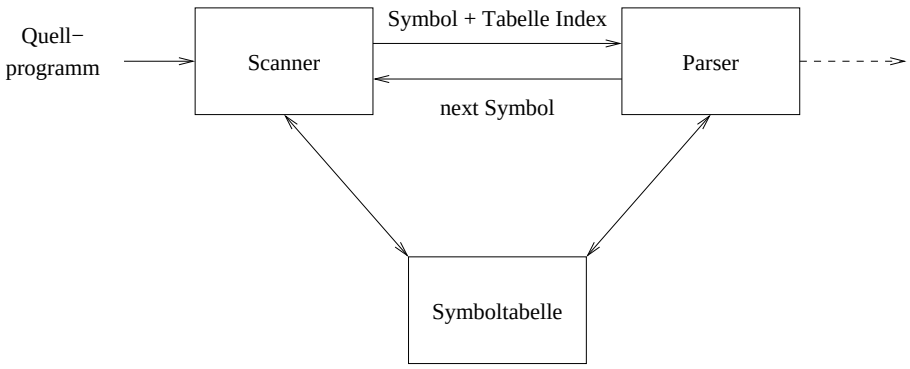



Abbildung 2-1: Interaktion des Scanners mit dem Parser

### 2.1.2 Motivation für die lexikalische Analyse

Theoretisch ist die lexikalische Analyse in einem Compiler nicht notwendig, da diese Aufgabe auch von der syntaktischen Analyse übernommen werden könnte. Es gibt jedoch gute Gründe für die Aufteilung in lexikalische und syntaktische Analyse, die nachfolgend aufgeführt sind:

- Das wichtigste Argument ist die Vereinfachung des Entwurfs.
  Durch Trennung der lexikalischen und syntaktischen Analyse wird häufig eine der beiden Phasen einfacher. Beispielsweise ist ein Parser, der die Konventionen bezüglich Kommentaren Leerzeichen und Gross- und Kleinschreibung berücksichtigen muss, um vieles komplexer als einer, der davon ausgehen kann, dass Kommentare und Leerzeichen vom Scanner entfernt wurden.
- Die Effizienz des Compilers wird verbessert.
  Ist der Scanner von den anderen Teilen getrennt, kann er spezifischer und effizienter implementiert werden.
- Die Portabilität von Compilern wird verbessert.
  Besonderheiten des Eingabealphabets und andere gerätespezifische Eigenarten können auf den Scanner beschränkt werden, da dieser als einziger den Quellcode einliest.

### 2.1.3 Symbole, Muster, Lexeme

Im Zusammenhang mit der lexikalischen Analyse haben die Begriffe **Symbol**, **Muster** und **Lexem** verschiedene Bedeutung. Im allgemeinen wird für unterschiedliche Zeichenfolgen in der Eingabe das gleiche Symbol als Ausgabe erzeugt. Eine Menge solcher Zeichenfolgen wird durch ein zum Symbol gehöriges Muster beschrieben. Das Muster kann also als eine Regel zum Konstruieren einer Menge von Zeichenfolgen aufgefasst werden. Ein Lexem ist eine Zeichenfolge im Quellprogramm, die dem Muster für ein Symbol entspricht.

**Beispiel 2.1 [Java Deklaration]** Als Beispiel betrachten wir die folgende JAVA Deklaration

```
float pi = 3.1416;
```

In der nachstehenden Tabelle sind die entsprechenden Symbole aufgeführt.

| Symbol | Beispiel Lexeme | Muster (informelle Beschreibung) |
|---|---|---|
| <FLOAT> | float | "float" |
| <ID> | pi, count d2 | Buchstabe gefolgt von Buchstaben und Ziffern |
| <NUM> | 3.1416, 73.25, 0.0 | irgendeine numerische Konstante |

### 2.1.4 Attribute von Symbolen

Wenn mehrere Lexeme dem gleichen Muster entsprechen, muss der Scanner späteren Compilerphasen für jedes Lexem zusätzlich Informationen übergeben. Das Muster zum Symbol num passt z.B. auf die beiden Zeichenketten 0 und 1. Für die Code-Erzeugung ist es jedoch wichtig zu wissen, welche Zeichenkette tatsächlich gemeint ist.

Der Scanner sammelt Informationen in den zu den Symbolen gehörigen **Attributen**. Die Symbole sind für die Syntaxanalyse wichtig, die Attribute für die Übersetzung. In der Praxis ist es üblich, einem Symbol nur ein einziges Attribut zu geben: einen Verweis auf den Symboltabelleneintrag, in dem die symbolspezifischen Informationen wie das Lexem eines Identifiers oder der Wert und der Datentyp einer Konstanten usw. stehen.

**Beispiel 2.2 [Attribute]** Die Symbole und zugehörigen Attribute der JAVA Anweisung

```
a = b + a;
```

werden folgendermassen als Folge von Symbol-Attribut Paaren dargestellt

[<ID>, 360], [<ASSIGN>], [<ID>, 361], [<ADD>], [<ID>, 360], [<SEMI>]

Man beachte, dass bei manchen Paaren ein Attribut überflüssig ist, da die erste Komponente zur Bestimmung des Lexems ausreicht.

### 2.1.5 Komplexität der lexikalischen Analyse

Sprachkonventionen beeinflussen die Komplexität der lexikalischen Analyse wesentlich. Als Beispiel wollen wir die Konventionen bezüglich der Behandlung von Leerzeichen betrachten. In der Sprache Fortran z.B. haben Leerzeichen nur in Zeichenketten eine Bedeutung, können aber nach Belieben in ein Programm eingestreut werden. Diese Konvention erschwert die Aufgabe der Symbolerkennung erheblich.

**Beispiel 2.3 [DO-Anweisung]**

Ein vielzitiertes Beispiel ist die DO-Anweisung von FORTRAN. In der Anweisung

```
DO 5 I = 1.25
```

kann man erst nach dem Lesen des Dezimalpunktes sagen, dass DO nicht ein Schlüsselwort ist, sondern Teil des Bezeichners DO5I. Die obige Anweisung muss die folgende Symbolfolge für den Parser generieren.

[<ID>, 601], [<ASSIGN>], [<CONST>, 612]

Im Gegensatz dazu besteht die Anweisung

```
DO 5 I = 1,25
```

aus einer Folge von sieben Symbolen

[<DO>], [<JMP> ,5], [<ID>,160], [<ASSIGN>], [<CONST>,161], [<COMMA>], [<CONST>,170]

### 2.1.6 Lexikalische Fehler

Nur wenige Fehler sind lokal auf der lexikalischen Ebene eindeutig identifizierbar, weil ein Scanner eine sehr beschränkte Sicht des Quellprogramms besitzt.

**Beispiel 2.4 [Fehler]** Als Beispiel betrachten wir folgende Zeichenfolge in einem C-Programm:

```
fi ( a == f(x)) ...
```

Der Scanner kann in einem solchen Fall nicht entscheiden, ob `fi` das falsch geschriebene Schlüsselwort `if` ist, oder ob `fi` ein Funktionsbezeichner ist. Weil `fi` ein gültiger Bezeichner ist, muss der Scanner das Symbol für Bezeichner übergeben und die Fehlerbehandlung anderen Compilerphasen überlassen.

Es sind aber auch Situationen möglich, in denen der Scanner nicht mehr fähig ist, weiterzufahren. Dies ist der Fall, wenn der anstehende Input mit keinem der vorgesehenen Muster übereinstimmt. Der Scanner muss in diesem Fall den Fehler selber behandeln und versuchen, die lexikalische Analyse fortzufahren. Diesen Vorgang bezeichnet man als **Recovery**. Folgende Strategien sind beim Auftreten eines Fehlers denkbar:

- überspringen von aufeinanderfolgenden Zeichen des anstehenden Inputs, bis wieder ein wohlgeformtes Symbol gefunden werden kann (**panic mode recovery**).
- Einfügen eines fehlenden Zeichens.
- Ersetzen eines falschen Zeichens durch ein korrektes.
- Vertauschen zweier benachbarter Zeichen.

## 2.2 Spezifikation von Symbolen

Um die Symbole einer Programmiersprache zu beschreiben (d.h. zur Definition der Muster) brauchen wir eine Notation (**Metasprache**). Eine wichtige Notation für die Spezifikation von Symbolen sind **reguläre Ausdrücke**.

### 2.2.1 Strings und Sprachen

Bevor wir die regulären Ausdrücke definieren können, müssen wir die Begriffe **String** und **Sprache** einführen.

**Bemerkung 2.1 [Sprache]** In diesem Abschnitt wird die allgemeinste Definition des Begriffs **Sprache** gegeben. Im Verlauf dieses Kurses werden wir spezialisierteren Begriffen

wie **reguläre Sprachen**, **kontextfreie Sprachen** usw. begegnen. Häufig wird auch für diese Begriffe das Wort **Sprache** verwendet.

**Definition 2.1 [Alphabet]** *Ein **Alphabet** $A$ ist eine endliche Menge von Zeichen.*

**Definition 2.2 [String]** *$A$ sei ein Alphabet. Ein **String** (oder **Wort**) über $A$ ist eine endliche Folge von Zeichen aus , $A$ :*

$$w = a_1 a_2 ... a_n, a_i \in A$$

*Die Länge $|w|$ eines Strings ist die Anzahl Zeichen in $w$ . Der leere String, der aus keinem Zeichen besteht, bezeichnen wir mit $\epsilon$ .*

**Definition 2.3 [Sprache]]** *Eine **Sprache** über ein Alphabet $A$ ist eine Menge von Strings über $A$ .*

Wie schon erwähnt ist diese Definition von Sprache sehr allgemein und umfasst sowohl abstrakte Sprachen wie die leere Sprache $\{\}$ und die Sprache, die nur den leeren String enthält $\{\epsilon\}$ , wie auch die Menge aller syntaktisch korrekten JAVA-Programme und die Menge aller grammatikalisch korrekten Sätze der deutschen Sprache.

### 2.2.2 Operationen auf Sprachen

**Definition 2.4 [Konkatenation]** *Wir bezeichnen mit $A^*$ die Menge aller Strings über das Alphabet $A$ . Durch Hintereinanderschreiben zweier Strings ist auf $A^*$ eine assoziative Verknüpfung definiert:*

$$\cdot : A^* \times A^* \to A^*, (a_1 a_1 ... a_n, b_1 b_2 ... b_m) \mapsto a_1 a_2 ... a_n b_1 b_2 ... b_m$$

*diese Verknüpfung heisst **Konkatenation** (oder **Produkt**) von Strings.*

In der Folge lassen wir bei der Konkatenation den $\cdot$ Operator weg und schreiben für $v, w \in A^*$ nur noch $vw$ statt $v \cdot w$.

**Definition 2.5 [Reguläre Operationen]** *Seien $L$ , $L_1$ und $L_2$ Sprachen über das Alphabet $A$ , dann definieren wir*

- *die **Vereinigung** (Auswahl) von $L_1$ und $L_2$ als*

  $$L_1 \cup L_2 = \{w | w \in L_1 \text{ oder } w \in L_2\}$$

- *die **Konkatenation** (Sequenz) von $L_1$ und $L_2$ als*

  $$L_1 L_2 = \{wv | w \in L_1 \text{ und } v \in L_2\}$$

- *und die **kleensche Hülle** (Wiederholung) $L^*$ von $L$ durch*

  $$L^0 = \{\epsilon\}$$
  $$L^i = LL^{i-1} (i > 0)$$
  $$L^* = \cup_{i \geq 0} L^i$$

### 2.2.3 Reguläre Ausdrücke

Wir sind nun in der Lage, eine Notation zur Spezifikation von Symbolen zu definieren.

**Definition 2.6 [Reguläre Ausdrücke]** *Sei $A$ ein Alphabet. Die **regulären Ausdrücke (RA)** $\alpha$ 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:*
   $L((\alpha)) = L(\alpha)$
   $L(\alpha_1|\alpha_2) = L(\alpha_1) \cup L(\alpha_2)$
   $L(\alpha_1\alpha_2) = L(\alpha_1)L(\alpha_2)$
   $L(\alpha^*) = (L(\alpha))^*$

**Bemerkung 2.2 [Reguläre Ausdrücke]** Mit den folgenden Konventionen lassen sich in regulären Ausdrücken viele Klammern vermeiden:

1. Der einstellige Operator $*$ hat die höchste Priorität und ist links-assoziativ.
2. Die Konkatenation hat die zweithöchste Priorität und ist links-assoziativ.
3. Der Operator $|$ hat die tiefste Priorität und ist links-assoziativ.

Gemäss diesen Konventionen ist z.B. $(a)|((b)^*(c))$ äquivalent zu $a|b^*c$ .

**Beispiel 2.5 [Reguläre Ausdrücke]** Sei $A = \{a, b\}$

- $L(\alpha) = \{a, b\}$ ist die dem regulären Ausdruck $\alpha = a|b$ zugeordnete reguläre Sprache.
- $L(\alpha) = \{aa, ab, ba, bb\}$ ist die dem regulären Ausdruck $\alpha = (a|b)(a|b)$ zugeordenete reguläre Sprache. Dies ist gerade die Menge aller Strings der Länge zwei, die aus $a$ 's und $b$ 's bestehen. Ein anderer regulärer Ausdruck für diese Menge ist $aa|ab|ba|bb$ .
- Der regulären Ausdruck $\alpha = (a|b)*$ bezeichnet die Menge aller Strings, die null oder mehr $a$ 's oder $b$ 's enthalten. Ein anderer regulärer Ausdruck für diese Menge ist $(a * b*)*$ .

Wie aus dem Beispiel hervorgeht, können zwei verschiedene reguläre Ausdrücke $\alpha_1$ und $\alpha_2$ ein und dieselbe Sprache bezeichnen. In diesem Fall heissen $\alpha_1$ und $\alpha_2$ **äquivalent**. Man schreibt dafür $\alpha_1 = \alpha_2$ .

Für die Umformung regulärer Ausdrücke gelten eine Reihe algebraischer Regeln. Im nächsten Satz sind einige davon angegeben.

**Satz 2.1 [Eigenschaften regulärer Ausrücke]**

*Sind $\alpha, \alpha_1, \alpha_2, \alpha_3$ reguläre Ausdrücke, dann gilt:*

- $\alpha_1|\alpha_2 = \alpha_2|\alpha_1$ , *d.h.* | *ist kommutativ.*
- $(\alpha_1|\alpha_2)|\alpha_3 = \alpha_1|(\alpha_2|\alpha_3)$ , *d.h.* | *ist assoziativ.*
- $(\alpha_1\alpha_2)\alpha_3 = \alpha_1(\alpha_2\alpha_3)$ , *d.h. die Konkatenation ist assoziativ.*
- $\alpha_1(\alpha_2|\alpha_3) = \alpha_1\alpha_2|\alpha_1\alpha_3$
  $(\alpha_1|\alpha_2)\alpha_3 = \alpha_1\alpha_3|\alpha_2\alpha_3$ , *d.h. Die Konkatenation ist bzgl.* | *distributiv.*
- $\epsilon\alpha = \alpha\epsilon = \alpha$ , *d.h.* $\epsilon$ *ist das neutrale Element der Konkatenation.*
- $\alpha^{**} = \alpha^*$ , *d.h.* $*$ *ist idempotent.*

### 2.2.4 Reguläre Definitionen

Zur Vereinfachung der Schreibweise ist es vorteilhaft, regulären Ausdrücken Namen zu geben und diese zur Definition von weiteren regulären Ausdrücken zu verwenden.

**Definition 2.7 [Reguläre Definition]** *Eine* ***reguläre Definition*** *über das Alphabet A ist eine Folge von Audrücken der Form*

$d_1 ::= r_1$ ;
$d_2 ::= r_2$ ;

...
$d_n ::= r_n$ ;

*dabei ist jedes* $d_i$ *ein eindeutiger Name und jedes* $r_i$ *ein regulärer Ausdruck über den Symbolen aus* $A \cup \{d_1, d_2 ... d_{i-1}\}$ .

Reguläre Definitionen entsprechen Definitionen von Macros in Assembler Sprachen.

**Beispiel 2.6 [Bezeichner]** Wir wollen die Menge der JAVA-Bezeichner (Identifier) mit Hilfe einer regulären Definition spezifizieren. Ein JAVA-Bezeichner besteht aus Buchstaben und Ziffern, wobei das erste Zeichen ein Buchstabe sein muss [1].

```
letter ::= A|B|...|Z|a|b|...|z ;
digit  ::= 0|1|2|3|4|5|6|7|8|9 ;
id     ::= letter (letter|digit)* ;
```

**Beispiel 2.7 [Ganze Zahl]** Eine ganze Zahl mit optionalem Vorzeichen kann folgendermassen definiert werden

```
digit         ::= 0|1|2|3|4|5|6|7|8|9 ;
digits        ::= digit digit* ;
optional_sign ::= (+|-|"") ;
int           ::= optional_sign digits ;
```

[1] Dies beschreibt eigentlich eine Teilmenge der gültigen JAVA-Bezeichner.

### 2.2.5 Abkürzungen

Manche Ausdrücke kommen in regulären Ausdrücken so häufig vor, dass es sinnvoll ist, Abkürzungen für sie einzuführen. In der Folge sei $\alpha$ ein regulärer Ausdruck über das Alphabet $A$ .

- Ein- oder mehrmaliges Auftreten ( + Operator).

  $\alpha^+ = \alpha\alpha^*$

- Null- oder einmaliges Auftreten ( ? Operator).

  $\alpha^? = \alpha|\epsilon$

- Zeichenklassen ( [] Operator)

  - Seien $a_1, a_2, ...a_n \in A$ . Dann ist
    $[a_1a_2...a_n] = a_1|a_2|...|a_n$
  - Wir nehmen nun an, dass auf dem Alphabet $A$ eine totale Ordnung definiert $\leq$ definiert ist (z.B.ASCII-Zeichensatz). Dann ist
    $[x - y] = a_{i_1}|a_{i_2}...|a_{i_n}$ wobei $a_{i_1}...a_{i_n} = \{a|a \geq x \text{ und } a \leq y\}$

**Beispiel 2.8 [Zeichenklassen]** Mit den eben eingeführten Abkürzungen können die ganzen Zahlen aus dem Beispiel 2.7 folgendermassen definiert werden.

```
digit         ::= [0-9] ;
digits        ::= digit+ ;
optional_sign ::= (+|-)? ;
int           ::= optional_sign digits ;
```

oder auch

```
int ::= (+|-)? [0-9]+ ;
```

### 2.2.6 Nichtreguläre Mengen

Nicht jede beliebige Menge von Zeichenketten über einem Alphabet kann mit Hilfe eines regulären Ausdrucks beschrieben werden. Um dies zu illustrieren, geben wir zwei exemplarische (in Programmiersprachen vorkommende) Konstrukte, die nicht mit Hilfe eines regulären Ausdrucks beschrieben werden können.

1. Reguläre Ausdrücke sind unbrauchbar zur Beschreibung gepaarter oder verschachtelter Konstrukte. So kann die Menge aller korrekt geklammerten Ausdrücke nicht durch einen regulären Ausdruck beschrieben werden (ein regulärer Ausdruck kann nicht zählen).
2. Wiederholtes Auftreten eines Teil-Strings ist durch reguläre Ausdrücke nicht beschreibbar. Für die Menge

   $\{wcw|w$ ist ein String bestehend aus $a$ 's und $b$ 's $\}$

   lässt sich kein regulärer Ausdruck angeben.

### 2.2.7 RA Implementierung

Wir werden später (Abschnitt 2.3.9) ein Programm zur Erkennung von RA entwickeln. Als vorarbeit wollen wir nun Zeigen, wie RA implementiert werden können. Für die Implementierung verwenden wir das *Besucher* Muster:

```
public abstract class RegularExpression {

  public abstract Object accept(RegularExpressionVisitor visitor,
                                Object udata);
}
```

Für jede Konstruktion aus Definition 2.6 ergibt sich eine entsprechende Unterklasse von `RegularExpression`:

```
public class CharacterExpression extends RegularExpression {
  private final char c;

  public CharacterExpression(char c) {
    this.c = c;
  }

  public Object accept(RegularExpressionVisitor visitor,
                       Object udata) {
    return visitor.visitCharacterExpression(this, udata);
  }

  public char getChar() {
    return c;
  }
}

public class EmptyExpression extends RegularExpression {

  public Object accept(RegularExpressionVisitor visitor,
                       Object udata) {
    return visitor.visitEmptyExpression(this, udata);
  }
}

public class ChoiceExpression extends RegularExpression {

  private final RegularExpression e1;
  private final RegularExpression e2;

  public ChoiceExpression(RegularExpression e1,
                          RegularExpression e2) {
    this.e1 = e1;
    this.e2 = e2;
```

```
  }

  public Object accept(RegularExpressionVisitor visitor,
                       Object udata) {
    return visitor.visitChoiceExpression(this, udata);
  }

  public RegularExpression getSubAlternative(int nr) {
    if (nr == 1)
      return e1;
    else if (nr == 2)
      return e2;
    else
      return null;
  }
}

public class SequenceExpression extends RegularExpression {

  private final RegularExpression e1;
  private final RegularExpression e2;

  public SequenceExpression(RegularExpression e1,
                            RegularExpression e2) {
    this.e1 = e1;
    this.e2 = e2;
  }

  public Object accept(RegularExpressionVisitor visitor,
                       Object udata) {
    return visitor.visitSequenceExpression(this, udata);
  }

  public RegularExpression getSubExpression(int nr) {
    if (nr == 1)
      return e1;
    else if (nr == 2)
      return e2;
    else
      return null;
  }
}

public class IterateExpression extends RegularExpression {
  private final RegularExpression e;

  public IterateExpression(RegularExpression e) {
    this.e = e;
  }
```

```
  public Object accept(RegularExpressionVisitor visitor,
                       Object udata) {
    return visitor.visitIterateExpression(this, udata);
  }

  public RegularExpression getIterateExpression() {
    return e;
  }
}

public abstract class RegularExpressionVisitor {


  public abstract Object visitCharacterExpression(CharacterExpression e,
                                                  Object udata);

  public abstract Object visitEmptyExpression(EmptyExpression e,
                                              Object udata);

  public abstract Object visitChoiceExpression(ChoiceExpression e,
                                               Object udata);

  public abstract Object visitSequenceExpression(SequenceExpression e,
                                                 Object udata);

  public abstract Object visitIterateExpression(IterateExpression e,
                                                Object udata);
}
```

### 2.2.8 Aufgaben

**Aufgabe 2.1 [Reguläre Ausdrücke 1]** *Gegeben sei das Alphabet $A = \{a, b\}$ . Gesucht sind reguläre Ausdrücke, die folgende Mengen beschreiben:*

1. *Die Menge aller Strings, die mit $a$ beginnen und mit $a$ enden.*
2. *Die Menge aller Strings, die den Teil-String $aba$ nicht enthalten.*
3. *Die Menge $\{a^n b^n; n \in N\}$*

**Aufgabe 2.2 [Reguläre Ausdrücke 2]** *Gegeben sei das Alphabet $A = \{a, b, c\}$ . Gesucht sind reguläre Ausdrücke, die die folgenden Mengen beschreiben:*

1. *Alle Strings, die den Teil-String $aba$ nicht enthalten.*
2. *Alle Strings, in denen die Buchstaben lexikographisch aufsteigend geordnet sind.*
3. *Alle Strings, in denen jeder Buchstabe höchstens einmal vorkommt.*

**Aufgabe 2.3 [Float]** *Geben Sie eine reguläre Definition an, welche die Floatkonstanten einer Sprache beschreibt (mit optionalem Exponent).*

**Aufgabe 2.4 [Strings]** *Gegeben sei das Alphabet $A = \{a, b\}$ . Welche String-Mengen werden von den folgenden regulären Ausdrücken beschrieben?*

1. `(a|b)*a(a|b)(a|b)`
2. `(( ϵ |a)b*)*`
3. `b*ab*`

**Aufgabe 2.5 [Java Strings]** *Geben Sie eine reguläre Definition für* JAVA*-Strings.*

**Aufgabe 2.6 [Implementierung]** *Implementieren Sie für jede* JAVA *Klasse aus Abschnitt 2.2.7 die Methode* `toString()` *und schreiben Sie ein entsprechendes Testprogramm.*

**Aufgabe 2.7 [Zeichenklassen]** *Erweitern Sie die RA Implementierung mit der Unterstützung von Zeichenklassen gemäss folgender Spezifikation:*

| | |
|---|---|
| `[x-yz]` | *Gegebene Intervalle / Einzelzeichen* |
| `[^x-yz]` | *Alle Zeichen ausser angegebene Intervalle / Einzelzeichen* |

## 2.3 Erkennen von Symbolen

Im vorigen Abschnitt haben wir die regulären Ausdrücke (und Sprachen) zur Spezifikation von Symbolen eingeführt. In diesem Abschnitt wollen wir uns dem Problem zuwenden, wie diese Symbole in einem Eingabestrom erkannt werden. Zu diesem Zweck führen wir nun **Automaten** ein.

### 2.3.1 Automaten

Wir wollen die Automaten mit Hilfe eines konkreten Beispiels einführen.

**Beispiel 2.9 [Getränkeautomat]** Ein Getränkeautomat funktioniert etwa wie folgt: Nach Einwurf eines Geldstückes kann der Kunde eines von mehreren Getränken durch Drücken der entsprechenden Auswahltaste herauslassen. Anstatt eines Getränkes kann auch das eingeworfene Geldstück durch Drücken der Rückgabetaste zurückverlangt werden. Wird eine Getränkeauswahltaste gedrückt, ohne dass vorher Geld eingeworfen wurde, so ertönt ein Signalton. Wird ein zweites Geldstück eingeworfen, bevor ein Getränk ausgewählt ist, so wird das zweite Geldstück einfach ausgeworfen.

Für jeden Automaten sind die drei folgenden Begriffe von grundlegender Bedeutung:

- **Eingabe:** Ein Automat wird von aussen bedient, d.h. er wird mit Eingabedaten versorgt. Es gibt also eine endliche Menge $A$ von Zeichen, die der Automat versteht. Wir bezeichnen $A$ als **Eingabealphabet** des Automaten.
- **Zustand:** Ein Automat befindet sich stets in einem bestimmten Zustand. Unter Einwirkung der Eingabe kann er von einem Zustand in einen anderen übergehen. Die endliche Menge $Z$ der möglichen Zustände heisst **Zustandsmenge**.
- **Ausgabe:** Im Laufe seiner Arbeit gibt der Automat auch Ausgabedaten aus. Die endliche Menge $O$ der produzierten Ausgabedaten heisst **Ausgabealphabet**. .

Die Arbeitsweise des Automaten kann durch ein **Zustandsdiagramm** oder durch eine **Zustandstabelle** beschrieben werden.

**Beispiel 2.10 [Zustandstabelle]** Für den Getränkeautomaten setzen wir

$A = \{G(eldstueck), K(affetaste), T(eetaste), R(ueckgabetaste)\}$
$Z = \{b(ereit), a(ktiv)\}$
$O = k(affe), t(ee), g(eld), s(ignalton)\}$

Das Zustandsdiagramm für den Getränkeautomaten ist in der Abb. 2-2 dargestellt. Zustände werden als Kreise und Zustandsübergänge als Pfeile dargestellt.

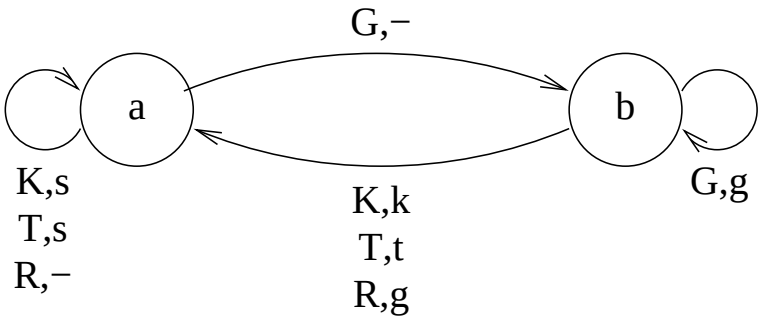



Abbildung 2-2: Zustandsdiagramm des Getränkeautomaten

In der folgender Tabelle ist die Zustandstabelle für den Getränkeautomaten dargestellt.

| | $G$ | $K$ | $T$ | $R$ |
|---|---|---|---|---|
| $a$ | $a, g$ | $b, k$ | $b, t$ | $b, g$ |
| $b$ | $a, -$ | $b, s$ | $b, s$ | $b, -$ |

**Bemerkung 2.3 [Automat ohne Ausgabe]** Für die Erkennung von Symbolen braucht mann nur Automaten, bei denen $O = \emptyset$ die leere Menge ist. Solche Automaten heissen **Automaten ohne Ausgabe**. In der Folge werden wir aber dafür weiterhin den Begriff Automat verwenden.

### 2.3.2 Deterministische endliche Automaten (DEA)

**Definition 2.8 [DEA]** *Ein **deterministischer endlicher Automat** (DEA) ist ein 5-Tuppel*

$DEA = (A, Z, \delta, z_0, F)$ *wobei*

$A = \{a_1, ..., a_n\}$ *das Eingabealphabet,.*
$Z = \{z_0, ..., z_n\}$ *die Zustandsmenge,*
$\delta : Z \times A \rightarrow Z$ *die Übergangsfunktion,*
$z_0 \in Z$ *der Anfangszustand und*
$F \subseteq Z$ *die Menge der akzeptierenden Zustände ist.*

#### Arbeitsweise des Automaten

Am Anfang befindet sich der Automat $DEA$ im Zustand $z_0$ und ein Lesepointer zeigt auf das erste Zeichen eines Strings $w \in A^*$ . Befindet sich der Automat während der

Verarbeitung von $w$ im Zustand $z_i$ und zeigt der Lesepointer auf das Eingabezeichen $a_j$ , so geht der Automat in den Zustand $z' = \delta(z_i, a_j)$ über und der Lesezeiger wird um ein Zeichen nach rechts veschoben.

Geht der Automat DEA bei der Verarbeitung eines Strings $w \in A^*$ vom Zustand $z$ in den Zustand $z'$ über, so schreiben wir dafür $z' = \tilde{\delta}(z, w)$.

Die Funktion $\tilde{\delta}$ wird wie folgt rekursiv definiert.

**Definition 2.9 [Delta Tilde]**

1. $\tilde{\delta}(q, \epsilon) = q$
2. $\tilde{\delta}(q, wa) = \delta(\tilde{\delta}(q, w), a)$ *wobei* $w$ *ein String ist und* $a$ *ein Symbol ist.*

**Bemerkung 2.4 [Pfad]** Falls $w = a_1 a_2 \ldots a_n$ und $\delta(zi, a_i) = z_{i+1}$ für alle $i = 0, \ldots n-1$ so gilt $\tilde{\delta}(z_0, w) = z_n$

**Definition 2.10 [DEA Sprache]** *Ein deterministischer endlicher Automat* $DEA = (A, Z, \delta, z_0, F)$ ***akzeptiert*** *ein String* $w \in A^*$ *, falls*

$\tilde{\delta}(z_0, w) \in F$

*Die Menge aller Strings, die der Automat* $DEA$ *akzeptiert, heisst die* ***Sprache*** *von* $DEA$

$L(DEA) = \{w \in A^* | \tilde{\delta}(z_0, w) \in F\}$

**Beispiel 2.11 [DEA]** Sei $DEA = (A, Z, \delta, z_0, F)$ mit $A = \{a, b\}$ , $Z = \{z_0, z_1, z_2\}$ , $F = \{z_2\}$ und der folgenden Übergangstabelle $\delta$

| | $a$ | $b$ |
|---|---|---|
| $z_0$ | $z_0$ | $z_1$ |
| $z_1$ | $z_0$ | $z_2$ |
| $z_2$ | $z_0$ | $z_2$ |

Der Automat ist in der Abb. 2-3 dargestellt. Man beachte, dass akzeptierende Zustände mit zwei konzentrischen Kreisen dargestellt sind.

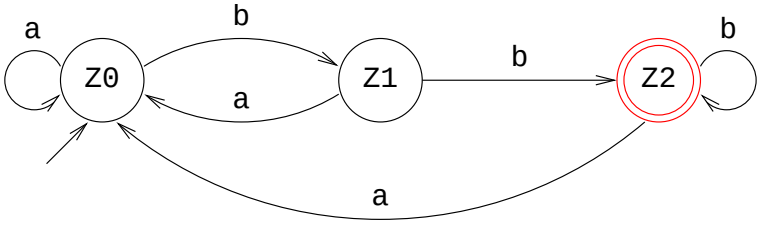



Abbildung 2-3: Darstellung eines deterministischen endlichen Automaten

Der Automat in diesem Beispiel akzeptiert alle Strings, die aus $a$ 's und $b$ 's bestehen und mit $bb$ enden.

$L(DEA) = \{w \in A^* | w \quad \text{endet auf} \quad bb\}$

**Definition 2.11 [Nicht vollständiger Automat]** *Ist bei einem endlichen Automaten* $DEA = (A, Z, \delta, z_0, F)$ *die Übergangsfunktion* $\delta$ *nicht auf der ganzen Menge* $Z \times A$ *definiert, so handelt es sich um einen* ***nicht-vollständig definierten*** *Automaten.*

**Arbeitsweise**

Liest ein nicht-vollständig definierter Automat $DEA$ im Zustand $z$ das Zeichen $a$ vom Eingabe-String $w$ und ist die Übergangsfunktion $\delta$ für das Paar $(z,a)$ nicht definiert, so hält der Automat $DEA$ an. In diesem Fall wird $w \in A^*$ nicht akzeptiert.

**Beispiel 2.12 [Nicht vollständiger Automat]** Sei $DEA = (A, Z, \delta, z_0, F)$ mit $A = \{a, b, c\}$ , $Z = \{z_0, z_1, z_2\}$ , $F = \{z_0, z_2\}$ und der folgenden Übergangstabelle $\delta$

| | $a$ | $b$ | $c$ |
|---|---|---|---|
| $z_0$ | $z_1$ | $z_0$ | $z_0$ |
| $z_1$ | - | $z_2$ | - |
| $z_2$ | $z_1$ | - | $z_0$ |

Der Automat ist in der Abb. 2-4 dargestellt.

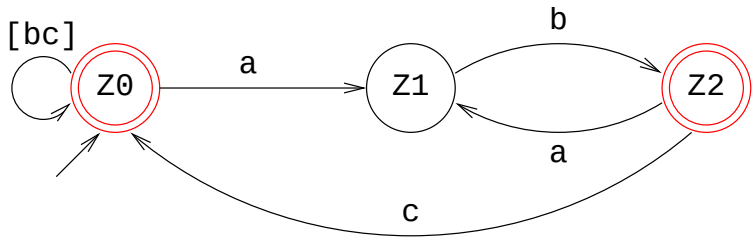



Abbildung 2-4: Nicht-vollständiger deterministischer endlicher Automat

Für diesen Automaten gilt:

$L(DEA) = \{w \in A^* | w$ enthält nach jedem $a$ genau ein $b\}$

**Bemerkung 2.5 [Vervollständigung]** Jeder nicht-vollständige Automat kann durch Hinzufügen eines neuen, nicht akzeptierenden Zustands $z_e$ in einen äquivalenten vollständig definierten Automaten transformiert werden. In Abb. 2-5 ist der zum Automat aus Abb. 2-4 äquivalente vollständige Automat dargestellt.

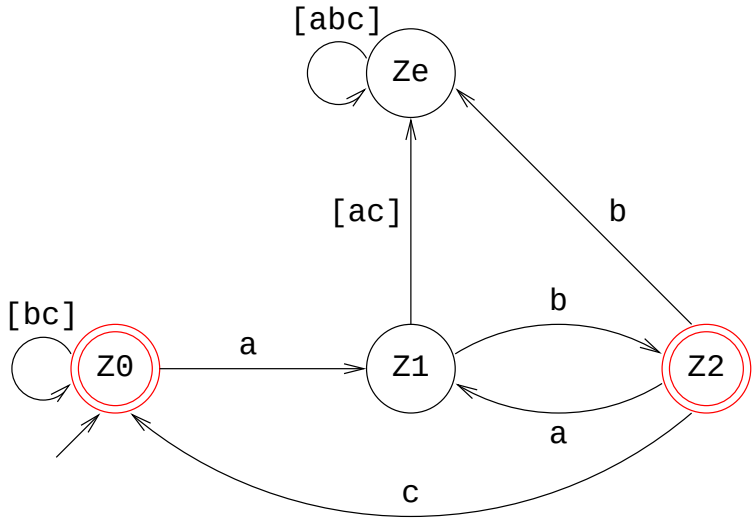



Abbildung 2-5: Vollständiger deterministischer endlicher Automat

### 2.3.3 Reguläre Ausdrücke und endliche Automaten

Wir wollen nun den Zusammenhang zwischen regulären Ausdrücken und deterministischen endlichen Automaten herstellen.

**Satz 2.2 [Reguläre Ausdrücke und Automaten]** *Zu jedem regulären Ausdruck $\alpha$ existiert ein Automat $DEA$ , sodass $L(\alpha) = L(DEA)$ gilt.*

Wir wollen diesen Satz nicht formal beweisen, sondern ihn an Beispielen illustrieren.

**Beispiel 2.13 [Reguläre Ausdrücke und Automaten]** Wir übernehmen die reguläre Definition aus dem Beispiel 2.7

```
digit          ::= [0-9] ;
digits         ::= digit+ ;
optional_sign  ::= (+|-)? ;
int            ::= optional_sign digits ;
```

Wir definieren nun den Automaten für diese reguläre Definition. $DEA$ sei ein Automat mit $A = \{digit, +, -\}$ , $Z = \{z_0, z_1, z_2\}$ , $F = \{z_2\}$ und der folgenden Übergangstabelle $\delta$ :

| | $digit$ | $+$ | $-$ |
|---|---|---|---|
| $z_0$ | $z_2$ | $z_1$ | $z_1$ |
| $z_1$ | $z_2$ | - | - |
| $z_2$ | $z_2$ | - | - |

Der entsprechende Automat ist in der Abb. 2-6 dargestellt.

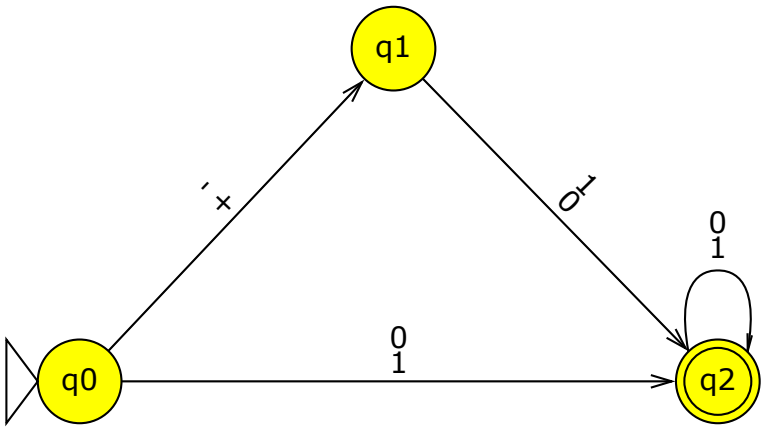



Abbildung 2-6: Automat, der eine ganze Zahl erkennt

**Beispiel 2.14 [DEA]** Als zweites Beispiel betrachten wir die reguläre Menge aus der Übung 2.1. Wir wollen alle Strings über das Alphabet $A = \{a, b\}$ , die mit einem $a$ beginnen und mit einem $a$ enden, beschreiben. Der reguläre Ausdruck für diese Menge ist $\alpha = (a(a|b)^*a)|a$

Wir definieren nun den Automaten für diesen regulären Ausdruck. $DEA$ sei ein Automat mit $A = \{a, b\}$ , $Z = \{z_0, z_1, z_2\}$ , $F = \{z_1, z_2\}$ und der folgenden Übergangstabelle $\delta$ :

| | $a$ | $b$ |
|---|---|---|
| $z_0$ | $z_1$ | - |
| $z_1$ | $z_1$ | $z_2$ |
| $z_2$ | $z_1$ | $z_2$ |

Der entsprechende Automat ist in der Abb. 2-7 dargestellt.

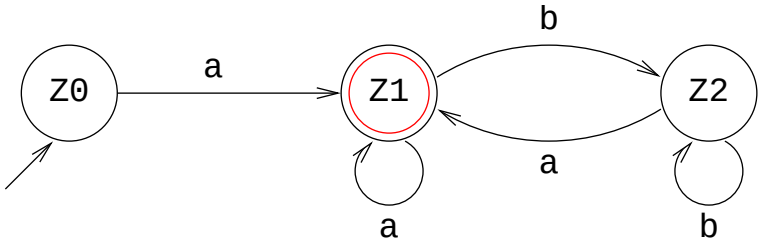



Abbildung 2-7: Der Automat zum regulären Ausdruck $(a(a|b)^*a)|a$

### 2.3.4 DEA Simulation

Wir wollen nun einen Algorithmus entwickeln, der entscheidet, ob ein String (oder der Anfang eines Strings) zur Sprache eines gegebenen endlichen Automaten gehört oder nicht. Dieser Vorgang wird als **Simulation** des Automaten bezeichnet.

**Bemerkung 2.6 [Longest Match]** Die Simulation wird so gestaltet, dass immer der **längste** Anfangsabschnitt eines Strings gefunden wird, der zur Sprache des Automaten gehört.

Für die Simulation werden folgende Funktionen verwendet:

| Funktion | Beschreibung |
|---|---|
| move$(s,a)$ | DEA-Zustand, der von einem DEA-Zustand $s$ über eine mit $a$ markierte Transition erreichbar ist. |
| nextChar() | Gibt das nächste Zeichen aus der Eingabe |

**Algorithmus 2.1 [DEA Simulation]** $s_0$ ist der Startzustand des DEA's

$s = s_0$
$c$ = nextChar()
**while** ($c \neq$ **eof**)
    $s$ = move$(s,c)$
    $c$ = nextChar()
**endwhile**
**if** ($s \in F$)
    **return** `true`
**else**
    **return** `false`
**endif**

### 2.3.5 DEA Implementierung

#### Implementierung mit Übergangstabelle

Bei dieser Methode wird zuerst die Übergangstabelle aufgebaut und anschliessend in einer Schleife mit der Funktion move$(s, c)$ abgearbeitet. Für nicht existierende Übergänge wird -1 in die Tabelle eingetragen. . Die Methode `execute()` versucht immer den **längsten** Anfangsabschnitt des Eingabestream zu findet. Als Resultat gibt sie den erreichten akzeptierenden Zustand oder -1 falls nichts akzeptiert wird.

JAVA-Klasse DFA

```
1  import java.io.IOException;
2  import java.io.PushbackReader;
3
4  public class DFA {
5
6    public int[][] table;
7    public int start;
8    public boolean[] accepting;
9    private int states = 0;
10   public final int SE = -1;
11   private final int INPUTS = 128;
12
13   public DFA(int states) {
14     super();
15     init(states);
16   }
17
18   protected void init(int size) {
19     states = 0;
20     table = new int[size][INPUTS];
21     accepting = new boolean[size];
22     for (int i = 0; i < size; i++) {
23       for (int j = 0; j < INPUTS; j++) {
24         table[i][j] = SE;
25       }
26       accepting[i] = false;
27     }
28   }
29
30   public int execute(PushbackReader reader) {
31     int state = start;
32     String lexem = "";
33     int matchedState = SE;
34     try {
35       int i = reader.read();
36       String s = "";
37       while (i != -1 && state != SE) {
38         m.debug("reading [" + (char) i + "] state [" + state + "]");
39         char c = (char) i;
```

```
40          s += "" + c;
41          state = table[state][c];
42          if (state != SE && accepting[state]) {
43            lexem += s;
44            s = "";
45            matchedState = state;
46          }
47          i = reader.read();
48        }
49        if (s.length() != 0) {
50          reader.unread(s.toCharArray());
51        }
52      } catch (IOException e) {}
53      return matchedState;
54    }
55  }
56
```

Diese Implementierungsvariante wird in der Regel bei Scanner-Generatoren verwendet[2].

**Bemerkung 2.7 [Komplexität]** Die Komplexität ist proportional zur Länge des Eingabestrings. Dafür ist die Tabellengrösse proportional zur Anzahl Zustände des Automaten (bei gegebener Anzahl Eingabezeichen).

### 2.3.6 Nichtdeterministische endliche Automaten NEA

Wir haben gesehen, wie Symbole mit Hilfe von endlichen Automaten erkannt werden können. Eine weitere Methode, die in diesem Abschnitt vorgestellt wird, benutzt **nichtdeterministische endliche Automaten**.

**Definition 2.12 [NEA]** *Ein **nichtdeterministischer endlicher Automat** (NEA) ist ein 5-Tupel*
$NEA = (A, Z, \delta, z_0, F)$ *, wobei*

$A = \{a_1, ..., a_n\}$ *das Eingabealphabet,*
$Z = \{z_0, ..., z_n\}$ *die Zustandsmenge,*
$\delta : Z \times A \rightarrow P(Z)$ *die Übergangsfunktion,*

[2] **Rekursive Implementierung:** Man kann die rekursive Definition der Übergangsfunktion eins zu eins codieren (siehe `prog/la/Automaton.java`).
**Implementierung ohne Übergangstabelle:** Bei dieser Methode entspricht jedem Zustand ein Code-Segment im Programm. Wenn Kanten aus dem Zustand herausführen, dann liest der zugehörige Code das nächste Zeichen und wählt, falls möglich, eine Kante aus. Nun wird die Kontrolle an das Code-Segment desjenigen Zustands übergeben, auf den die Kante zeigt (siehe `prog/la/deaimp1.c`).
**Implementierung mit `goto`:** Bei dieser Methode entspricht jedem Zustand ein Label im Programm. Wenn Kanten aus dem Zustand herausführen, dann liest der zugehörige Code das nächste Zeichen und wählt, falls möglich, eine Kante aus. Nun wird die Kontrolle an das Code-Segment desjenigen Zustands übergeben, auf den die Kante zeigt. Dies erfolgt mit einem Sprung `goto`) zum entsprechenden Code-Segment (siehe `prog/la/deaimp2.c` oder auch `prog/la/dea.java`).
**Implementierung mit dem Zustandsmuster (State Pattern):** Bei dieser methode ergibt sich folgendes JAVA-Programm. Die `goto` Instruktionen in der dritten Implementierung wurden durch Instanzierung von neuen Zustandsobjekten in den jeweiligen Methoden `bearbeite()` der Unterklassen der abstrakten Klasse `Zustand` realisiert. Die Klassenstruktur des JAVA-Programms basiert auf dem klassischen Entwurfsmuster *Zustand (State)*[GHJV95], [GHJV96] (siehe `prog/la/deaimp4.java`).

$z_0 \in Z$ *der Anfangszustand und*
$F \subseteq Z$ *die Menge der akzeptierenden Zustände ist.*

$P(Z)$ *bezeichnet die Potenzmenge von* $Z$ .

**Arbeitsweise**

Am Anfang befindet sich der Automat $NEA$ im Zustand $z_0$ und ein Lesepointer zeigt auf das erste Zeichen eines Strings $w \in A^*$ . Befindet sich der Automat während der Verarbeitung von $w$ im Zustand $z_i$ und zeigt der Lesepointer auf das Eingabezeichen $a_j$ , so ist der Folgezustand nicht eindeutig festgelegt, sondern es kann aus der Menge $\delta(z_i, a_j) \subseteq Z$ einer ausgewählt werden. Ein String $w \in A^*$ wird genau dann vom Automaten $NEA$ akzeptiert, wenn es unter den verschiedenen Möglichkeiten der Verarbeitung von $w$ mindestens eine gibt, die in einen Endzustand führt.

**Leerübergänge**

In der Folge ergänzen wir das Alphabet immer um das Zeichen $\epsilon$, das den leeren String bezeichnet. Ist nun ein Automat im Zustand $z_i$ und ist $\delta(z_i, \epsilon) \neq \emptyset$, so kann der Automat in einen beliebigen Zustand $z \in \delta(z, \epsilon)$ übergehen, ohne dass der Lesepointer nach rechts verschoben wird.

**Definition 2.13 [$\epsilon$_closure]**

1. *Jeder Zustand* $q$ *gehört zu* $\epsilon$*_closure*$(q)$
2. *Falls* $p \in \epsilon$*_closure*$(q)$ *und falls es einen mit* $\epsilon$ *markierten Zustandsübergang von* $p$ *nach* $r$ *gibt, dann gehört* $r$ *zu* $p \in \epsilon$*_closure*$(q)$*.*

Die Funktion $\delta\tilde{}$ wird wie folgt rekursiv definiert.

**Definition 2.14 [Delta Tilde]**

1. $\delta\tilde{}(q, \epsilon) = \epsilon$*_closure*$(q)$
2. *Es sei* $w = xa$ *ein String,* $a$ *das letzte Zeichen von* $w$*. Dann lässt sich* $\delta\tilde{}(q, w)$ *wie folgt berechnen:*
   (a) *Es sei* $\delta\tilde{}(q, x) = \{p_1, p_2, \ldots, p_k\}$
   (b) *Es sei* $\cup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, \ldots, r_m\}$
   (c) *Dann gilt* $\delta\tilde{}(q, w) = \epsilon$*_closure*$(\{r_1, r_2, \ldots, r_m\})$

Die Sprache $L(NEA)$ des NEA's wird somit wie folgt definiert:

$$L(NEA) = \{w \in A^* | \delta\tilde{}(z_0, w) \cap F \neq \emptyset\}$$

**Beispiel 2.15 [NEA]** Gegeben sei der $NEA = (A, Z, \delta, z_0, F)$ mit $A = \{a, b, c, \epsilon\}$ , $Z = \{z_0, z_1, z_2, z_3\}$ , $F = \{z_3\}$ und der folgenden Übergangsfunktion $\delta$

| | $a$ | $b$ | $c$ | $\epsilon$ |
|---|---|---|---|---|
| $z_0$ | $\{z_1, z_2\}$ | $\varnothing$ | $\varnothing$ | $\varnothing$ |
| $z_1$ | $\varnothing$ | $\{z_0, z_3\}$ | $\varnothing$ | $\varnothing$ |
| $z_2$ | $\varnothing$ | $\varnothing$ | $\{z_3\}$ | $\{z_3\}$ |
| $z_3$ | $\varnothing$ | $\varnothing$ | $\varnothing$ | $\varnothing$ |

Der Automat ist in der Abb. 2-8 dargestellt.

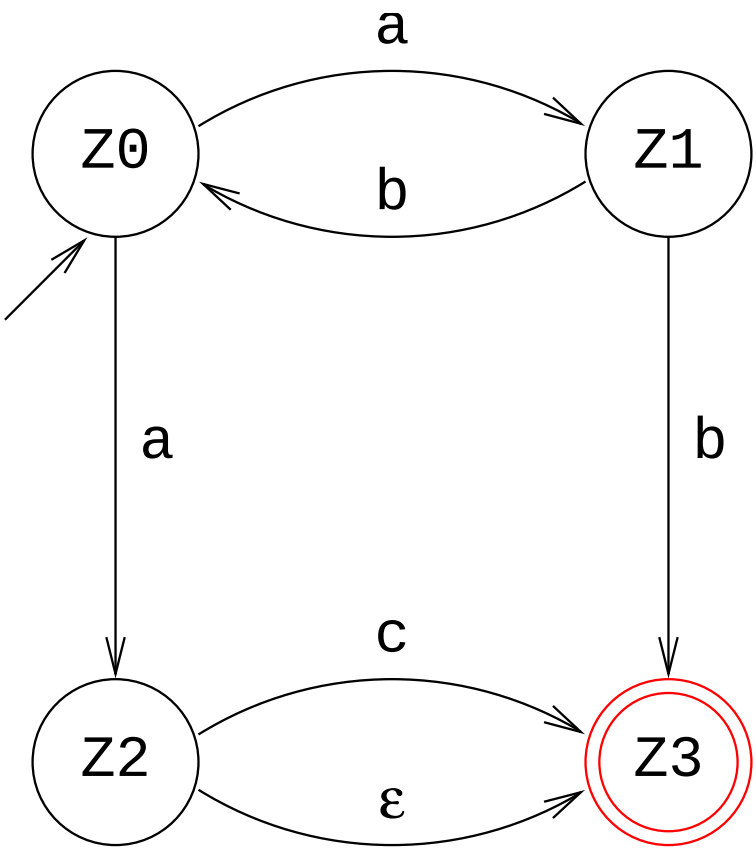



Abbildung 2-8: Darstellung eines nichtdeterministischen endlichen Automaten

Der Automat akzeptiert die durch $(ab)^*|((ab)^*a(c|\epsilon))$ definierte Sprache.

### 2.3.7 NEA Simulation

Wir wollen nun einen Algorithmus entwickeln, der entscheidet, ob ein String (oder der Anfang eines Strings) zur Sprache eines gegebenen nichtdeterministischen endlichen Automaten gehört oder nicht. Dieser Vorgang wird als **Simulation** des Automaten bezeichnet.

**Bemerkung 2.8 [Longest Match]** Die Simulation wird so gestaltet, dass immer der **längste** Anfangsabschnitt eines Strings gefunden wird, der zur Sprache des Automaten gehört. Als Beispiel betrachten wir den Automaten aus der Abb. 2-13 und der String $ababbabbbb$ . Bei diesem String gehören die Anfangsabschnitte $a$ , $aba$ und $ababba$ alle zur Sprache des Automaten. Die hier vorgestellte Simulation wird also den String $ababba$ akzeptieren.

Wir verwenden zur Simulation folgende Hilfsfunktionen:

| Funktion | Beschreibung |
|---|---|
| $\epsilon$_closure($T$) | Die Menge $T$ vereinigt mit der Menge der NEA-Zustände, die von einem NEA-Zustand $s$ aus $T$ allein über $\epsilon$-Transitionen erreichbar sind. |
| move ($T, c$) | Menge der NEA-Zustände, die von einem NEA-Zustand $s$ aus $T$ über eine mit $c$ markierte Transition erreichbar sind. |
| nextChar() | Gibt das nächste Zeichen aus der Eingabe |

Bevor das erste Eingabezeichen eingelesen wird, kann sich der NEA in irgend einem Zustand in $\epsilon$_closure($\{s_0\}$) befinden ( $s_0$ ist der Startzustand des NEA). Nehmen wir nun an, dass von , $s_0$ aus für eine gegebene Folge von Eingabezeichen genau die Zustände der Menge $T$ erreichbar sind und dass $a$ das nächste Eingabezeichen ist. Durch Lesen von $a$ kann der NEA in einen beliebigen Zustand der Menge $\epsilon$_closure( move ($T, a$)) übergehen. Diese überlegungen führen zum Simulationsalgorithmus.

**Algorithmus 2.2 [NEA Simulation]** Dstates ist die Zustandsmenge des DEA's Dtrans die Übergangstabelle.

$T = \epsilon$_closure($s_0$)
$c$ = nextChar()
**while** ($c \neq$ **eof**)
    $U = \emptyset$
    **forall** ($s \in T$)
        $U = U \cup$ move($s, c$)
    **endforall**
    $T = \epsilon$_closure($U$)
    $c$ = nextChar()
**endwhile**
**if** ($T \cap F \neq \emptyset$)
    **return** `true`
**else**
    **return** `false`
**endif**

### 2.3.8 NEA Implementierung

JAVA-Klasse `NFA`

```
 1  import java.io.IOException;
 2  import java.io.PushbackReader;
 3  import java.util.BitSet;
 4  import java.util.Vector;
 5
 6  public class NFA {
 7
 8    protected BitSet[][] table;
 9    protected BitSet[] epsilon;
10    protected BitSet accept;
11    protected int start;
12    protected int states;
```

```
13    private final int INPUTS = 128;
14
15    public NFA(int size) {
16      super();
17      init(size);
18    }
19
20    protected void init(int size) {
21      states = size;
22      table = new BitSet[size][INPUTS];
23      accept = new BitSet();
24      epsilon = new BitSet[size];
25      for (int i = 0; i < size; i++) {
26        for (int j = 0; j < INPUTS; j++) {
27          table[i][j] = new BitSet();
28        }
29        epsilon[i] = new BitSet();
30      }
31    }
32
33    public BitSet closure(BitSet from) {
34      BitSet open = (BitSet) from.clone();
35      BitSet closed = new BitSet(states);
36      while (!open.isEmpty()) {
37        int index = open.nextSetBit(0);
38        open.clear(index);
39        closed.set(index);
40        BitSet e = epsilon[index];
41        if (e != null) {
42          for (int i = e.nextSetBit(0);
43               i >= 0; i = e.nextSetBit(i + 1)) {
44            if (!closed.get(i) && !open.get(i)) {
45              open.set(i);
46            }
47          }
48        }
49      }
50      return closed;
51    }
52
53    public BitSet closure(int state) {
54      BitSet b = new BitSet(states);
55      b.set(state);
56      return closure(b);
57    }
58
59    public BitSet move(BitSet from, char input) {
60      BitSet closed = new BitSet(states);
61      for (int i = from.nextSetBit(0);
62           i >= 0; i = from.nextSetBit(i + 1)) {
```

```
63        BitSet e = table[i][input];
64        if (e != null) {
65          closed.or(e);
66        }
67      }
68      return closure(closed);
69    }
70
71    public BitSet execute(PushbackReader reader) {
72      String lexem = "";
73      BitSet b = new BitSet(states);
74      b.set(start);
75      BitSet state = closure(b);
76      int pos = 0;
77      BitSet matchedState = null;
78      try {
79        int i = reader.read();
80        String s = "";
81        while (i != -1 && state != null) {
82          char c = (char) i;
83          s += "" + c;
84          state = closure(move(state, c));
85          if (state != null) {
86            boolean found = false;
87            for (int j = 0; j < states; j++) {
88              if (accept.get(j) && state.get(j)) {
89                found = true;
90                break;
91              }
92            }
93            if (found) {
94              lexem += s;
95              s = "";
96              matchedState = state;
97            }
98          }
99          i = reader.read();
100       }
101       if (s.length() != 0) {
102         reader.unread(s.toCharArray());
103       }
104     } catch (IOException e) {}
105     return matchedState;
106   }
107 }
```

### 2.3.9 Konstruktion eines NEA aus einem regulären Ausdruck

Ein regulärer Ausdruck $\alpha$ kann in einen nichtdeterministischen endlichen Automaten $NEA$ transformiert werden, der genau die durch $\alpha$ beschriebene Sprache erkennt. Der dafür verwendete Algorithmus heisst **Thompsons Konstruktion**.

**Atomare Automaten** Die $NEA$ 's für den Leer-String und für einzelne Symbole des Alphabets sind in der Abbildung 2-9 angegeben. Sie bestehen aus nur zwei Zuständen und einer Transition, die vom Startzustand zum akzeptierenden Zustand führt und mit dem entsprechenden Symbol markiert ist. Für jedes Zeichen des Alphabets, das im regulären Ausdruck vorkommt, konstruieren wir einen solchen atomaren Automaten. Wichtig ist, dass für ein mehrfach in $\alpha$ vorkommendes Zeichen $a \in A$ mehrere $NEA$ 's erstellt werden. Die atomaren Automaten entsprechen dem Punkt 1 der Definition von regulären Ausdrücken (siehe Definition 2.6).

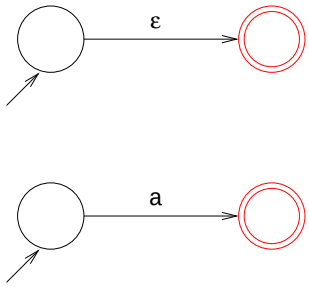



Abbildung 2-9: Thompson Konstruktion: Atomare Automaten

**Konkatenation** Gegeben seien zwei $NEA$ 's $N_1$ und $N_2$ , die die regulären Ausdrücke $\alpha_1$ und $\alpha_2$ repräsentieren. Der Automat $N$ , der die Konkatenation $\alpha_1\alpha_2$ repräsentiert, ist in der Abbildung 2-10 dargestellt.

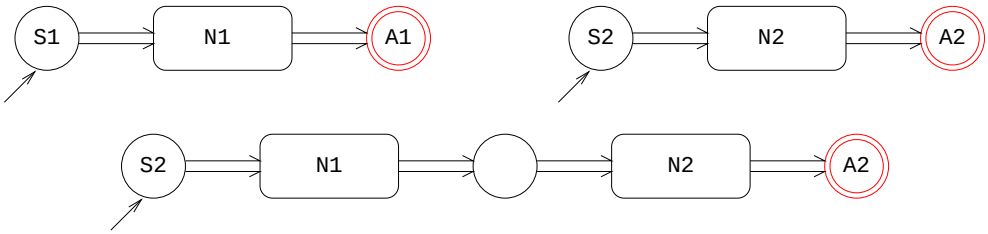



Abbildung 2-10: Thompson Konstruktion: Konkatenation

Der Startzustand $s_1$ des Automaten $N_1$ wird zum Startzustand von $N$ , der akzeptierende Zustand $a_2$ von $N_2$ wird zum akzeptierenden Zustand von $N$ . Der akzeptierende Zustand $a_1$ von $N_1$ wird mit dem Startzustand $s_2$ von $N_2$ identifiziert.

**Auswahl** Gegeben seien zwei $NEA$ 's $N_1$ und $N_2$ , die die regulären Ausdrücke $\alpha_1$ und $\alpha_2$ repräsentieren. Der Automat $N$ , der die Auswahl $\alpha_1|\alpha_2$ repräsentiert, ist in der Abbildung 2-11 dargestellt.

Wir kreieren für $N$ einen neuen Startzustand $s_{neu}$ mit einer Transition auf den Startzustand $s_1$ von $N_1$ und einer Transition auf den Startzustand $s_2$ von $N_2$ . Beide Transitionen sind mit $\epsilon$ markiert. Vom akzeptierenden Zustand $a_1$ von $N_1$ zum akzep-

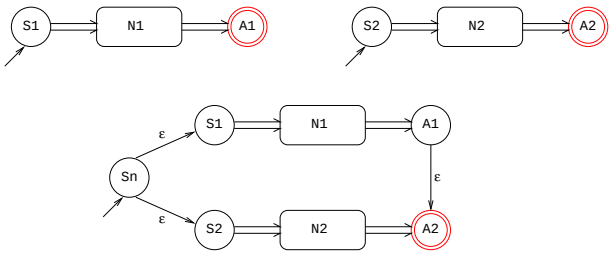



Abbildung 2-11: Thompson Konstruktion: Auswahl

tierenden Zustand $a_2$ von $N_2$ wird eine neue mit $\epsilon$ markierte Transition eingeführt. Der akzeptierende Zustand $a_2$ von $N_2$ wird zum akzeptierenden Zustand von $N$ .

**Wiederhohlung** Gegeben sei der $NEA$ $N_1$ , der den regulären Ausdruck $\alpha$ repräsentiert. Der Automat $N$ , der die Wiederhohlung $\alpha*$ repräsentiert, ist in der Abbildung 2-12 dargestellt.

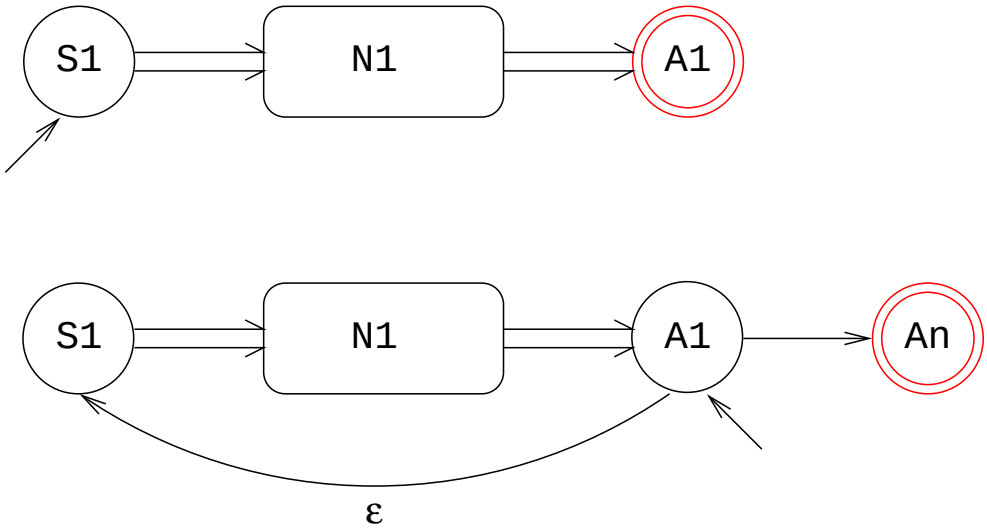



Abbildung 2-12: Thompson Konstruktion: Wiederholung

Wir kreieren für $N$ einen neuen akzeptierenden Zustand $a_{\text{neu}}$ und eine mit $\epsilon$ markierte Transition, die vom akzeptierenden Zustand $a_1$ von $N_1$ zum neuen akzeptierenden Zustand führt. Von $a_1$ zum Startzustand $s_1$ von $N_1$ wird eine mit $\epsilon$ markierte Transition eingeführt und $a_1$ wird zum neuen Startzustand $s_{neu}$ von $N$ .

Mit diesen 4 Regeln kann aus einem regulären Ausdruck schrittweise der entsprechende Automat aufgebaut werden.

### Maximale Anzahl Zustände

Gegeben sei ein regulärer Ausdruck $\alpha$ der Länge $|\alpha|$ . Wir wollen die maximale Anzahl Zustände des mit Hilfe von Thompsons Konstruktion erstellten Automaten berechnen. Pro Zeichen in $\alpha$ entsteht ein atomarer Automat mit zwei Zuständen. Für die Operatoren $*$ und $|$ wird ein zusätzlicher Zustand kreiert. Also gilt:

Anzahl Zustände $\leq 2|\alpha|$

**Beispiel 2.16 [Thompson Konstruktion]** In der Abbildung 2-13 ist Thompsons Konstruktion für den regulären Ausdruck $(a(a|b)^*a)|a$ dargestellt.

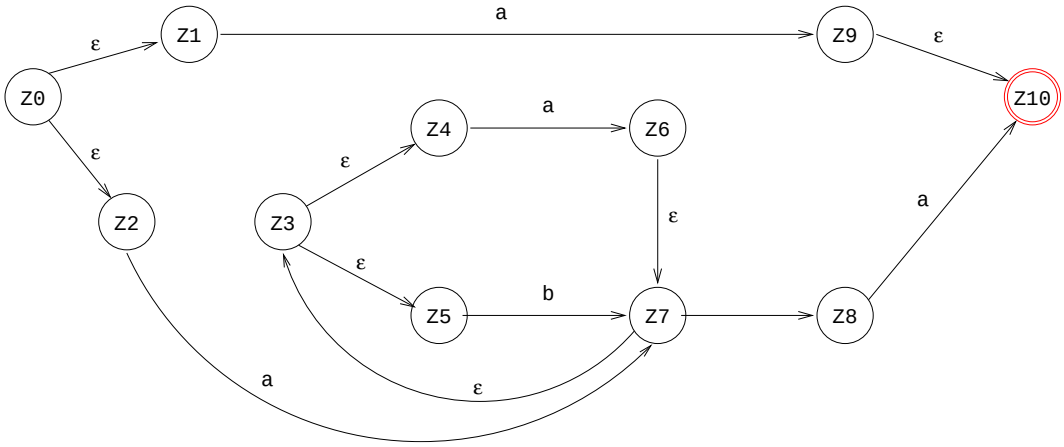



Abbildung 2-13: Thompson Konstruktion

### 2.3.10 Umwandlung des NEA in einen DEA

**Satz 2.3 [Von NEA zu DEA]** *Zu jedem nichtdeterministischen endlichen Automaten NEA existiert ein deterministischer endlicher Automat DEA , der die gleiche Sprache erkennt. Das heisst,* $L(DEA) = L(NEA)$ .

**Beweis:** Die Idee ist eigentlich dieselbe wie die der Simulation eines NEA. Das heisst, wir simulieren den NEA für alle Zeichen des Alphabets $A$ gleichzeitig. Jede mögliche Konfiguration dieser Simulation wird zu einem Zustand des DEA. Wir verwenden dazu folgende Hilfsfunktionen:

| Funktion | Beschreibung |
|---|---|
| $\epsilon$_closure$(T)$ | Die Menge $T$ vereinigt mit der Menge der NEA-Zustände, die von einem NEA-Zustand $s$ aus $T$ allein über $\epsilon$-Transitionen erreichbar sind. |
| move $(T, a)$ | Menge der NEA-Zustände, die von einem NEA-Zustand $s$ aus $T$ über eine mit $a$ markierte Transition erreichbar sind. |

Bevor das erste Eingabezeichen eingelesen wird, kann sich der NEA in irgend einem Zustand in $\epsilon$_closure$(\{z_0\})$ befinden ( $z_0$ ist der Startzustand des NEA). Nehmen wir nun an, dass von , $z_0$ aus für eine gegebene Folge von Eingabezeichen genau die Zustände der Menge $T$ erreichbar sind und dass $a$ das nächste Eingabezeichen ist. Durch Lesen von $a$ kann der NEA in einen beliebigen Zustand der Menge $\epsilon$_closure( move $(T, a)$) übergehen. Diese überlegungen führen zum Simulationsalgorithmus.

**Algorithmus 2.3 [Umwandlung]** Dstates ist die Zustandsmenge des DEA's Dtrans die Übergangstabelle.

DStates = $\{\epsilon$_closure$(z_0)\}$
**while** (es existiert ein unmarkierter Zustand $T$ in Dstates )

markiere $T$
**for** jedes Eingabesymbol $a$ im Alphabet $A$
$U = \epsilon$_closure(move$(T, a)$)
**if** ($U$ ist nicht in $Dstates$ )
füge $U$ als (unmarkierten) Zustand zu $Dstates$ hinzu
Dtran$[T, a] = U$
**endfor**
**endwhile**

**Bemerkung 2.9 [Akzeptierender Zustand]** Ein Zustand aus $Dtrans$ ist genau dann ein akzeptierender Zustand, wenn er mindestens einen akzeptierenden Zustand des NEA enthält.

**Beispiel 2.17 [Umwandlung]** Als Beispiel betrachten wir wieder den regulären Ausdruck $(a(a|b)^*a)|a$ über das Alphabet $A = \{a, b\}$ (Alle Strings die mit $a$ beginnen und mit $a$ enden). Der entsprechende Automat ist in der Abb. 2-13 dargestellt. Wir wollen nun die angegebene Konstruktion für den Automaten angeben. Die neuen Zustände bezeichnen wir mit $D_i$ .

$D_0 = \epsilon$_closure$(\{z_0\}) = \{z_0, z_1, z_2\}$

$D_1 = \epsilon$_closure(move$(D_0, a)) = \epsilon$_closure$(\{z_7, z_9\}) = \{z_3, z_4, z_5, z_7, z_8, z_9, z_{10}\}$

$D_2 = \epsilon$_closure(move$(D_1, a)) = \epsilon$_closure$(\{z_6, z_{10}\}) = \{z_3, z_4, z_5, z_6, z_7, z_8, z_{10}\}$

$D_3 = \epsilon$_closure(move$(D_1, b)) = \epsilon$_closure$(\{z_7\}) = \{z_3, z_4, z_5, z_7, z_8\}$

Man kann nun kontrollieren, dass keine weiteren Zustände dazukommen. Wir berechnen noch die noch nicht bestimmten Transitionen.

$\epsilon$_closure(move$(D_0, b)) = \epsilon$_closure$(\emptyset) = \emptyset$

$\epsilon$_closure(move$(D_2, a)) = \epsilon$_closure$(\{z_6, z_{10}\}) = D_2$

$\epsilon$_closure(move$(D_2, b)) = \epsilon$_closure$(\{z_7\}) = D_3$

$\epsilon$_closure(move$(D_3, a)) = \epsilon$_closure$(\{z_6, z_{10}\}) = D_2$

$\epsilon$_closure(move$(D_3, b)) = \epsilon$_closure$(\{z_7\}) = D_3$

Aus diesen Berechnungen ergibt sich die folgende Übergangstabelle:

| | $a$ | $b$ |
|---|---|---|
| $D_0$ | $D_1$ | - |
| $D_1$ | $D_2$ | $D_3$ |
| $D_2$ | $D_2$ | $D_3$ |
| $D_3$ | $D_2$ | $D_3$ |

Die Zustände $D_1$ und $D_2$ enthalten den (einzigen) akzeptierenden Zustand $z_{10}$ des NEA und sind damit akzeptierend. Der entsprechende Automat ist in der Abb. 2-14 dargestellt.

Der konstruierte Automat ist nicht optimal bezüglich der Anzahl Zustände (Vergleiche dazu das Beispiel 2.14).

### 2.3.11 Vergleich von NEA und DEA

Wir kennen nun zwei Methoden, um für einen gegebenen regulären Ausdruck $\alpha$ und einem Eingabe-String $w$ zu entscheiden, ob $w \in L(\alpha)$ ist.

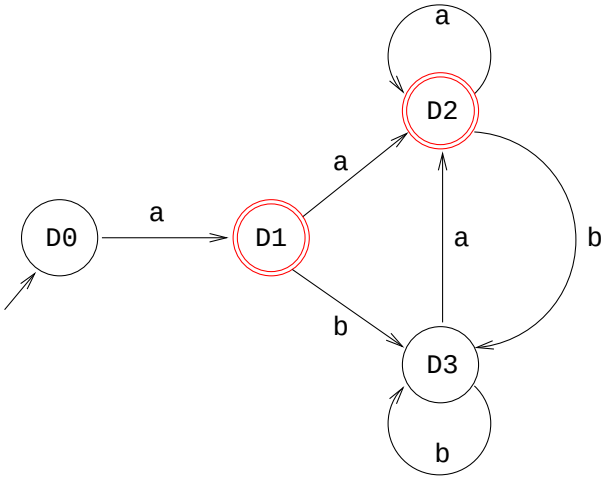



Abbildung 2-14: Konstruktion eines DEA aus einem NEA f $(a(a|b)^*a)|a$

1. Wir können mit Hilfe der Thompson Konstruktion aus $\alpha$ einen NEA konstruieren und mit Hilfe des Simulationsalgorithmus den NEA simulieren.
2. Wir können den NEA konstruieren, mit Hilfe des Algorithmus in der Abb. 2.3 in einen DEA umwandeln und anschliessend den DEA simulieren.

Wir wollen den Zeit- und Platzbedarf für beide Methoden abschätzen. In den folgenden Überlegungen steht $m$ für die Länge des regulären Ausdrucks $\alpha$ und $n$ für die Länge des Strings $w$ . Für die erste Methode wissen wir schon, dass die Anzahl Zustände $O(m)$ ist und der Zeitbedarf der Simulation $O(mn)$ .

Der Zeitbedarf zur Simulation des DEA ist linear also $O(n)$ . Der Platzbedarf eines DEA kann aber exponentiell mit der Grösse des regulären Ausdrucks wachsen.

**Beispiel 2.18 [Komplexität]** Gegeben sei der reguläre Ausdruck

$$(a|b)^*a(a|b)(a|b)...(a|b)$$

bei dem $r-1$ $(a|b)$ 's am Ende stehen. Es gibt für diesen regulären Ausdruck keinen DEA mit weniger als $2^r$ Zuständen.

Platz- und Zeitbedarf für NEA und DEA sind in folgender Tabelle zusammengefasst.

| Automat | Platz | Zeit |
|---|---|---|
| NEA | $O(m)$ | $O(mn)$ |
| DEA | $O(2^m)$ | $O(n)$ |

### 2.3.12 Aufgaben

**Aufgabe 2.8 [Reelle Zahlen]**

1. *Finden Sie einen Automaten, der Floatkonstanten und ganze Zahlen erkennt. Benutzen Sie dazu den in der Übung 2.3 definierten regulären Ausdruck für Floatkonstanten.*

2. *Ändern Sie das Programm zur Erkennung von ganzen Zahlen mit Hilfe einer Uebergangstabelle so, dass es sowohl Floatkonstanten wie auch ganze Zahlen erkennen kann. (Statt* `success` *soll das Programm* `integer` *oder* `float` *ausgeben).*

3. *Ergänzen Sie das Programm so, dass mehrere durch Leerzeichen getrennte Zahlen nacheinander erkannt werden können.*

**Aufgabe 2.9 [Thompson Konstruktion]** *Ergänzen Sie Thompsons Konstruktion so, dass auch die Operatoren + und ? berücksichtigt werden.*

## 2.4 Scanner Generatoren

### 2.4.1 Einführung

Ein **Scanner Generator** ist ein Programm, das aus einer lexikalischen Spezifikation automatisch einen Scanner (Programm zum Erkennen von Symbolen) erzeugt. Die Spezifikation basiert meistens auf regulären Ausdrücken. Von der Struktur her sind die erzeugten Scanner nichts anderes als endliche Automaten. Sie sind aber meistens mächtiger als reine endliche Automaten (Aktionen und Kontextsensitivität).

#### Der Einsatzgebiet von Scannergeneratoren

Scannergeneratoren sind Software-Werkzeuge, die sich auf vielen Anwendungsgebieten erfolgreich einsetzen lassen. Als Beispiele können angeführt werden:

**Compilerbau:** Erzeugen von Scanner für die lexikalische Analyse.

**Textverarbeitung:** überprüfen der Rechtschreibung in einem Text.

**Kommandointerpreter:** Mit Scannergeneratoren können einfache Kommandointerpreter erzeugt werden.

**Chiffrierung:** Es können Strings nach bestimmten Regeln in andere Strings umgewandelt werden.

**Filter:** Im Abschnitt 2.4.2 ist ein Filter angegeben, der alle Leerzeilen aus einer Datei herausfiltert. Eine weitere Anwendung ist ein Filter welches alle HTML-Tags aus einer Textdatei entfernt.

Der bekannteste Scannergenerator LEX wurde in den 70er Jahren von M.E. Lesk [Les75] und E. Schmidt als Ergänzung zu YACC (einem Parser Generator) [LMB92] entwickelt. Eine Public Domain Version (GNU) von LEX. existiert unter dem Namen FLEX.

JLEX wurde von E. Berk, einem Studenten der Princeton Universität im Jahre 1996 entwickelt. JLEX ist ein Scanner Generator, der JAVA Code erzeugt. Im Gegensatz zu FLEX ist JLEX nur als Coroutine einsetzbar. Seine Funktionalität ist gegenüber FLEX ein bisschen eingeschränkt. Dafür erzeugt JLEX JAVA Klassen. Die erzeugte Scanner Klasse kann mehrfach instanziert werden. Dies ergibt die Möglichkeit, rekursive Scanner zu entwickeln [3]

[3] Die C++ Version FLEX++ von FLEX erzeugt C++ Klassen und erlaubt auch rekursives Scanning.

### 2.4.2 JavaCC

JAVACC [4] ist ein Compilerbau-Werkzeug ähnlich FLEX und BISON. Im Unterschied zu FLEX und BISON erzeugt JAVACC JAVA Quellcode und arbeitet objektorientiert. Ausserdem wurde bei der Entwicklung der Syntax für die Grammatik darauf geachtet, diese soweit möglich an die JAVA Syntax anzulehnen.

Wir beschränken uns hier auf dem Scanneraspekt vom JAVACC.

#### Arbeitsweise

Grundsätzlich bestehen alle Scanner Generatoren aus folgen 3 Teilen: **Definitionen** (Reguläre Definitionen, Startbedingungen, Import- oder Include-Anweisungen), **Regeln** (Muster und zugehörige Aktion) und **Benutzerprozeduren** (oder Methoden).

Jede Regel setzt sich aus einem regulären Ausdruck (dem Muster) und einer zugehörigen Aktion zusammen. Die Aktion ist einfach ein Stück C oder JAVA Code. Die Aktion wird dann ausgeführt, wenn im Eingabetext ein Lexem gefunden wird, das durch den entsprechenden regulären Audruck beschrieben wird.

Die Klasse `Tokenmanager` stellt den eigentlichen Scanner dar. Sie kümmert sich um das Scannen der Eingabe und das Erkennen von Token abhängig vom lexikalischen Zustand, in dem sich der Tokenmanager gerade befindet. Der Tokenmanager versucht immer, aus der Eingabe soviele Zeichen wie möglich zu extrahieren und aus dieser Zeichenkette ein Token zu erzeugen. Wenn er ein Token erkannt hat, wechselt er evtl. in einen neuen lexikalischen Zustand und sucht dann das nächste Token in der Eingabe.

Der `Tokenmanager` befindet sich immer in genau einem **lexikalischen Zustand**. Vom jeweiligen Zustand hängt es ab, welche Token erkannt werden können. Wenn vom Sprachdesigner nichts anderes angegeben wurde, bleibt der Tokenmanager immer im Zustand `DEFAULT`. Es ist möglich, für jeden Zustand eigene Token zu definieren.

Bei der Suche nach dem nächsten Symbol hält sich JAVACC an die folgenden 3 Vorschriften:

1. Wenn im Eingabetext ein String steht, der von keinem der regulären Ausdrücke erkannt wird, beendet der Scanner mit einer Fehlermeldung.
2. Es wird das längste mögliche Lexem aus dem Eingabetext gesucht, das durch einen der regulären Ausdrücke erkannt wird.
3. Wird ein Lexem durch zwei oder mehr reguläre Ausdrücke gleichzeitig erkannt, so wird die zuerst angegebene Regel ausgewählt und damit deren Aktion ausgeführt.

#### Einfache Anwendung

Wir wollen eine Spezifikation angeben, welche die Schlüsselwörter `while` und `if` sowie JAVA Bezeichner erkennt und auf die Standardausgabe ausgibt. Die übrigen Lexeme sollen einfach übersprungen werden.

JAVACC-Spezifikation `Scanner1`

[4] JAVACC wurde 1996 von Sriram Sankar (Sun Microsystems) und Sreenivasa Viswanadha (SUNY at Albany) entwickelt. JAVACC ist frei erhältlich unter `https://javacc.dev.java.net`

```
1   PARSER_BEGIN(Scanner1)
2
3   public class Scanner1 {
4
5     public static void main(String args[])
6       throws ParseException {
7       Scanner1 lexer = new Scanner1(System.in);
8       Token t = lexer.getNextToken();
9       while (t.kind != Scanner1Constants.EOF) {
10        t = lexer.getNextToken();
11      }
12    }
13  }
14
15  PARSER_END(Scanner1)
16
17  TOKEN :
18  {
19    < KEYWORD: "if" | "while" >
20    { System.out.println("KW : " + matchedToken.image); }
21  }
22
23  TOKEN :
24  {
25    < ID: ["a"-"z","A"-"Z","_"]
26          ( ["a"-"z","A"-"Z","_","0"-"9"] )* >
27    { System.out.println("ID : " + matchedToken.image); }
28  }
29
30  SKIP :
31  {
32    <~[]>
33  }
```

**Bemerkung 2.10 [Verhalten des Scanners]** Wir wollen nun einige Eingabe-Strings betrachten und bestimmen, was der Scanner in den verschiedenen Fällen tut:

1. `ab111`
   Dieser String wird nur als Bezeichner erkannt, also wird die Meldung `Identifier: ab111` ausgegeben.

2. `while`
   Dieser String wird sowohl als Bezeichner als auch als Schlüsselwort erkannt. Die zuerst angegebene Regel (und damit deren Aktion) wird somit ausgewählt und die Meldung `Keywort: while`. wird ausgegeben.

3. `iffi`
   Dieser String wird als Bezeichner erkannt. Der Substring `if` wird als Schlüsselwort erkannt. Da immer das längst mögliche Lexem aus dem Eingabetext gesucht wird erfolgt die Meldung `Identifier: iffi`

**Kontextsensitivität**

Das folgende Beispiel ersetzt Leerzeilen und Zeilen, die nur aus Blanks und Tabs bestehen durch einen entsprechenden Text. Man merke, dass eine Leerzeile aus dem einzigen Charakter \n besteht, d.h. \n kommt unmittelbar am Zeilenanfang vor. Bei einer aus Blanks und Tabs bestehenden Zeile kommt \n am Zeilenende vor.

JAVACC-Spezifikation Scanner2

```
1  PARSER_BEGIN(Scanner2)
2
3  public class Scanner2 {
4
5    public static void main(String args[])
6      throws ParseException {
7      Scanner2 lexer = new Scanner2(System.in);
8      Token t = lexer.getNextToken();
9      while (t.kind != Scanner2Constants.EOF) {
10       t = lexer.getNextToken();
11     }
12   }
13 }
14
15 PARSER_END(Scanner2)
16
17 TOKEN :
18 {
19   < BLANK: ([" ","\t"])+ ("\r")? "\n" >
20   { System.out.println(" *** blank line"); }
21 }
22
23 TOKEN :
24 {
25   < ANY: (~["\n","\r"] )+ ("\r")? "\n" >
26   { System.out.print(matchedToken.image); }
27 }
28
29 TOKEN :
30 {
31   < EMPTY: ("\r")? "\n" >
32   { System.out.println(" *** empty line"); }
33 }
34
35 TOKEN :
36 {
37   < LAST: ~[]>
38   { System.out.print(matchedToken.image); }
39 }
```

### Startbedingungen

Bei der Verwendung von mehreren Regeln in einer Spezifikation ist es manchmal erforderlich, dass zu einem bestimmten Zeitpunkt der Verarbeitung nur ein bestimmter Teil der Regeln aktiv ist. Diese Aufgabe kann mit Hilfe von **Startbedingungen** gelöst werden.

Wir wissen, dass ein regulärer Ausdruck keine geschachtelten Konstrukte (Klammerungen) erkennen kann. Mit Hilfe von Startbedingungen und Aktionen ist jedoch diese Aufgabe lösbar.

JAVACC-Spezifikation `Scanner3`

```
1   PARSER_BEGIN(Scanner3)
2
3   public class Scanner3 {
4
5     public static void main(String args[])
6       throws ParseException {
7       Scanner3 lexer = new Scanner3(System.in);
8       Token t = lexer.getNextToken();
9       while (t.kind != Scanner3Constants.EOF) {
10        t = lexer.getNextToken();
11      }
12      System.out.println("\n/* depth = "
13                           + token_source.depth + " */");
14    }
15  }
16
17  PARSER_END(Scanner3)
18
19  TOKEN_MGR_DECLS :
20  {
21    static int depth;
22  }
23
24  SKIP :
25  {
26    "/*" { depth = 1; } : COMMENT
27  }
28
29  <COMMENT> SKIP :
30  {
31    "/*"
32    {
33      ++depth;
34    }
35  }
36
37  <COMMENT> SKIP :
38  {
39    "*/"
40    {
```

```
41      --depth;
42      if (depth == 0) SwitchTo(DEFAULT);
43    }
44  }
45
46  <COMMENT> SKIP :
47  {
48    <~[]>
49  }
50
51  TOKEN :
52  {
53    <PROG: ~[]>
54    {
55      System.out.print(matchedToken.image);
56    }
57  }
```

Diese Spezifikation ist natürlich nicht korrekt, da sie /* auch innerhalb eines Strings als Anfang eines Kommentars interpretiert. Im folgenden Beispiel werden geschachtelte Kommentare aus einem Programm entfernt. Gleichzeitig wird getestet, ob Strings und Kommentare korrekt abgeschlossen sind.

JAVACC-Spezifikation Scanner4

```
 1  PARSER_BEGIN(Scanner4)
 2
 3  public class Scanner4 {
 4
 5    public static void main(String args[])
 6      throws ParseException {
 7      Scanner4 lexer = new Scanner4(System.in);
 8      Token t = lexer.getNextToken();
 9      while (t.kind != Scanner4Constants.EOF) {
10        t = lexer.getNextToken();
11      }
12      if (token_source.openstring == 1)
13        System.out.println(" *** illegal string");
14      System.out.println("\n *** maximal comment depth : " +
15                         token_source.maxDepth);
16      if (token_source.depth > 0)
17        System.out.println("\n *** illegal comment depth " +
18                           "at end : " + token_source.depth);
19    }
20  }
21
22  PARSER_END(Scanner4)
23
24  TOKEN_MGR_DECLS :
25  {
26    static int depth = 0;
```

```
27    static int maxDepth =0;
28    static int openstring = 0;
29  }
30
31  SKIP :
32  {
33    "/*"
34    {
35      depth = 1;
36      if (maxDepth < 1) maxDepth = 1;
37    } : COMMENT
38  }
39
40  TOKEN :
41  {
42    <STRBEGIN: "\"">
43    {
44      System.out.print(matchedToken.image);
45    } : STRING
46  }
47
48  <STRING> TOKEN :
49  {
50    <STRCHAR: ((~["\n","\r","\""]) | "\\\"")+>
51    {
52      openstring = 1;
53      System.out.print(matchedToken.image);
54    }
55  }
56
57  <STRING> TOKEN :
58  {
59    <STRERROR: ["\n","\r"]>
60    {
61      System.out.println(" *** illegal string");
62    }
63  }
64
65  <STRING> TOKEN :
66  {
67    <STREND: "\"">
68    {
69      openstring = 0;
70      System.out.print(matchedToken.image);
71    } : DEFAULT
72  }
73
74  <COMMENT> SKIP :
75  {
76    "/*"
```

```
77     {
78       ++depth;
79       if (depth > maxDepth) maxDepth = depth;
80      }
81   }
82
83   <COMMENT> SKIP :
84   {
85     "*/"
86     {
87       --depth;
88       if (depth == 0) SwitchTo(DEFAULT);
89     }
90   }
91
92   <COMMENT> SKIP :
93   {
94     <~[]>
95   }
96
97   TOKEN :
98   {
99     <PROG: ~[]>
100    {
101      System.out.print(matchedToken.image);
102    }
103  }
```

**JavaCC: Klassendiagram**

### 2.4.3 Aufgaben

**Aufgabe 2.10 [Klammerung]** *Beim Programmieren in der* JAVA*-Sprache kommt es vor, dass eine öffnende oder schliessende Klammer vergessen wird oder ein Kommentar nicht abgeschlossen wird. Dies kann zu schwer auffindbaren Syntaxfehlern führen.*

*Entwerfen Sie mit* JAVACC *ein Programm, das ein* JAVA*-Program liest und auf jeder Zeile die aktuelle Schachtelungstiefe für Klammern und den Kommentar anzeigt. Eine Zeile des Outputs hat also folgendes Format:*

```
10: {2} (1) /*0*/ System.out.println("hello world");
```

*Dies bedeutet, dass die Verschachtelungstiefe für "{" zwei ist, für ,"(" eins und für den Kommentar null.*

*Am Ende muss eine Warnung für jede nicht abgeschlossene Klammerung erscheinen.*

**Aufgabe 2.11 [Crossreferencer]** *Erstellen Sie mit Hilfe von* JAVACC *ein Programm, das für* C*-Module eine Cross-Referenz-Liste erstellt. Das heisst, für jeden gefundenen Namen, der nicht in der folgenden Keyword Tabelle vorhanden ist, soll der Filename, der Identifier*

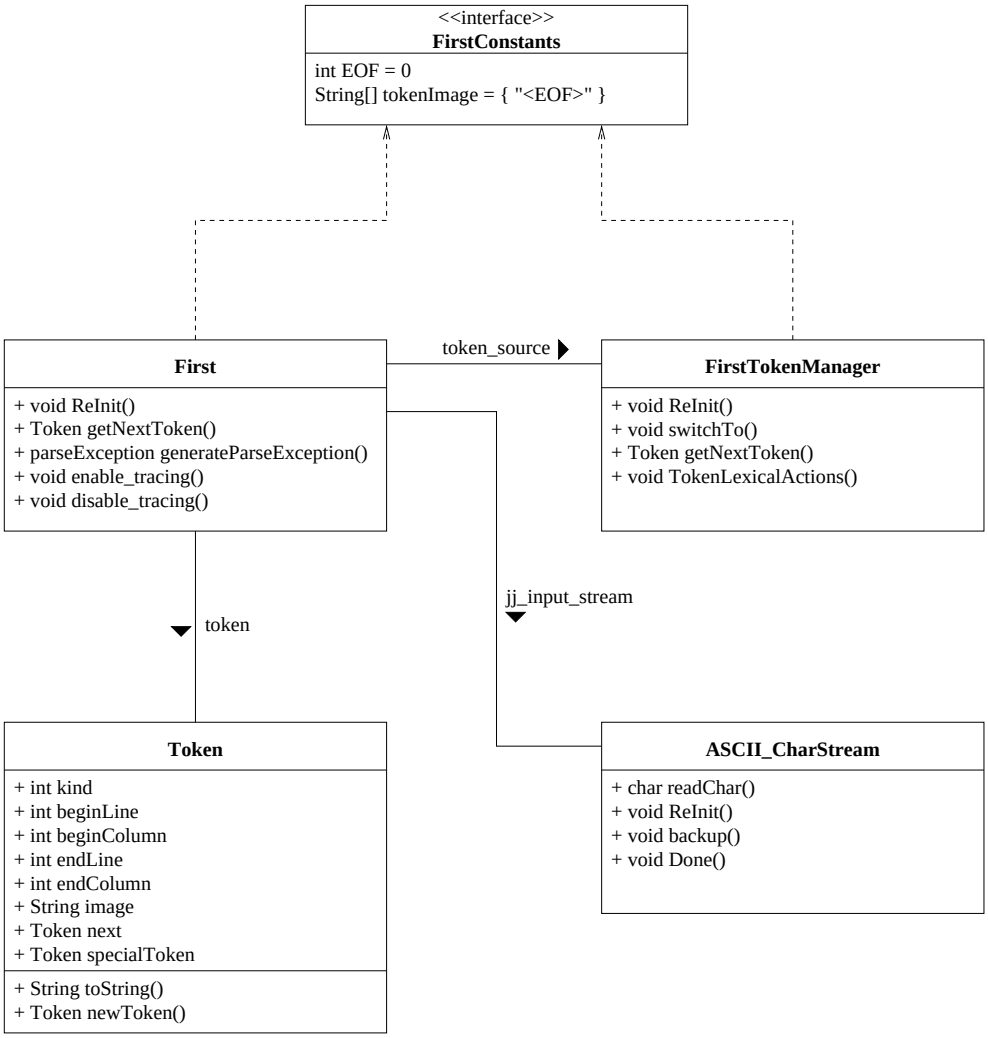



Abbildung 2-15: Struktur von JAVACC

*Name und die Zeilennummern ausgedruckt werden, in denen der Name vorkommt (Ein Name darf in der Liste nur einmal vorkommen).*

*Schlüsselwörter:*

| | | | | | |
|---|---|---|---|---|---|
| *auto* | *break* | *case* | *char* | *const* | *continue* |
| *default* | *do* | *double* | *else* | *enum* | *extern* |
| *float* | *for* | *goto* | *if* | *int* | *long* |
| *register* | *return* | *short* | *signed* | *sizeof* | *static* |
| *struct* | *switch* | *typedef* | *union* | *unsigned* | *void* |
| *volatile* | *while* | *define* | *include* | *NULL* | |

**Aufgabe 2.12 [Kommentare]** *Das letzte Beispiel aus Abschnitt 2.4.2 ist noch nicht korrekt. Was ist in diesem Beispiel noch falsch? ändern Sie die Spezifikation so, dass die Kommentare richtig entfernt werden.*

**Aufgabe 2.13 [HTML]** *Schreiben Sie eine* JavaCC *Spezifikation, die alle* HTML*-Befehle aus einer* HTML*-Datei entfernt. Die Applikation soll auch mit* HTML*-Kommentare umgehen können.*

**Aufgabe 2.14 [HTML]** *Erweitern Sie das Programm aus Aufgabe 2.13 so, dass alle* HTML *Headers (z.B. `h1`) auf einer separaten Zeile ausgegeben werden. Anschliessend soll noch eine Leerzeile generiert werden. Ferner sollten die Standard* XML *Entitäten durch die entsprechenden Zeichen ersetzt werden.*

**Aufgabe 2.15 [LaTeX]** *Schreiben Sie eine* JavaCC *Prototyp Spezifikation, für die Übersetzung von* LaTeX *nach* ASCII*. Im wesentlichen sollen alle* LaTeX *Befehle entfernt werden (mit Ausnahme der Spezialzeichen). Sie dürfen annehmen, dass sich im Dokument keine mathematische Zeichen befinden. Die Applikation soll auch mit* LaTeX*-Kommentare umgehen können.*

*Alles was vor dem Befehl `\begin{document}` vorkommt kann ignoriert werden*[5]*.*

## 2.5 Projekt

Ziel des Projekts ist die Implementierung eines einfachen Parsergenerators. Wir sind nun nicht soweit! Wir haben wir gesehen, wie sich DEA's implementieren lassen. Wir fangen also an mit der Implementierung eines Scannergenerators. In diesem Kapitel haben wir fast alle Komponenten Zusammen:

- Im Abschnitt 2.2.7 haben wir gesehen, wie sich RA (manuell) Implementieren lassen.
- Im Abschnitt 2.3.5 haben wir gesehen, wie sich DEA's Implementieren lassen.
- Im Abschnitt 2.3.8 haben wir schliesslich gesehen, wie sich NEA's Implementieren lassen.

[5] Man indet eine gute LaTeX Einführung unter `http://elib.uni-stuttgart.de/opus/doku/gestaltung/latexkurz`

### 2.5.1 Vorgehen

Wir verfügen noch nicht über folgende Programme

1. Erzeugung von RA Objekten (gemäss Abschnitt 2.2.7) aus einem Regulären Ausdruck
2. Erzeugung von einem NEA aus einem RA
3. Erzeugung eines DEA's aus einem NEA (hier kennen wir vorläufig nur den Algorithmus).

Das erste Problem kann erst einfach gelöst werden mit Hilfe der Theorie aus Kapitel 3. Wir werden uns also damit später beschäftigen.

Um das zweite Problem zu lösen müssen wir einen NEA aus einem RA konstruieren. Wir müssen also die Thompson konstruktion aus Abschnitt 2.3.9 Implementieren. Zu diesem Zweck werden wir eine entsprechende Besucherklasse `RegularExpressionToNFA` implementieren.

Um das dritte Problem zu lösen. Müssen wir den Algorithmus 2.3 implementieren. Wir werden zu diesem Zweck die Klasse `DFA` mit der Methode `toDFA()` erweitern.

### 2.5.2 Test Beispiel

Als Beispiel wollen wir den Regulären Ausdruck $\alpha$ = `a(a|b)*a|b` verwendet. Die Sprache dieses NEA's besteht aus alle Strings über das Alphabet $A = \{a, b\}$ , die mit einem $a$ beginnen und mit einem $a$ enden, beschreiben.

Ein Testprogramm für $\alpha$ soll wie folgt aussehen:

```
// regular expression a(a|b)*a|b
RegularExpression a = new CharacterExpression('a');
RegularExpression b = new CharacterExpression('b');
RegularExpression r1 = new ChoiceExpression(a, b);
RegularExpression r2 = new IterateExpression(r1);
RegularExpression r3 = new SequenceExpression(a,
  new SequenceExpression(r2, a));
RegularExpression r4 = new ChoiceExpression(r3, a);
RegularExpressionToNFA visitor = new RegularExpressionToNFA();
// transform a(a|b)*a|b to NFA
r.accept(visitor, new FromToPair(0, 1));
NFA nfa = visitor.getNfa();
nfa.setStart(0);
nfa.setAccept(1);
// transform to DFA
DFA dfa = nfa.toDFA();
```

### 2.5.3 Von RA zu NEA

Die Thompson Konstruktion erzeugt einen NEA mit einem Startzustand und einem akzeptierenden Endzustand. Wir werden die Besucherklasse `RegularExpressionToNFA` so implementieren, dass Zustand `0` Startzustand und Zustand `1` akzeptierender Zustand sind. Jede Besuchermethode soll via Parameter `udata` ein Paar (StartZustand, Endzustand) erhalten. Dies kann mit folgender Hilfsklasse `FromToPair` erfolgen:

```
public class FromToPair {

  protected int from;
  protected int to;

  public FromToPair(int start, int end) {
    super();
    this.from = start;
    this.to = end;
  }
}
```

Die Implementierung der Methode `visitCharacterExpression()` ist somit wie folgt möglich:

```
public Object visitCharacterExpression(CharacterExpression re,
                                       Object udata) {
  FromToPair ft = (FromToPair) udata;
  nfa.addTransition(ft.getFrom(), re.getChar(), ft.getTo());
  return null;
}
```

Bei der Erzeugung des NEA's kennen wir die Anzahl Zustände nicht zum Voraus. Am besten geht es mit einer dynamischen Erzeugung der NEA Zustände und Zustandsübergänge. Zu diesem Zweck wird die Klasse `NFA` mit folgenden Methoden erweitert:

Die Methode `adjustSize()` vergrössert die Zustandstabellen des NEA's (falls nötig). `STATES` ist eine ganzzahlige Konstante mit Wert `100`.

```
private void adjustSize(int num) {
  int length = epsilon.length;
  // adjust number of states
  if (num >= states) {
    states = num;
  }
  // adjust array sizes
  if (num < length) {
    return;
  } else {
    while (num > length) {
      length += STATES;
    }
```

```
      BitSet[][] newTable = new BitSet[length][INPUTS];
      BitSet[] newEpsilon = new BitSet[length];
      System.arraycopy(epsilon, 0, newEpsilon, 0, states);
      System.arraycopy(table, 0, newTable, 0, states);
      epsilon = newEpsilon;
      table = newTable;
    }
  }
```

Die Methode `addState()` fügt einen neuen Zustand hinzu.

```
  public void addState(int state) {
    adjustSize(state + 1);
  }
```

Die Methode `addTransition()` fügt einen neuen Zustandsübergang hinzu.

```
  public void addTransition(int from, int input, int to) {
    int max = Math.max(from, to) + 1;
    adjustSize(max);
    if (table[from][input] != null) {
      table[from][input].set(to);
    } else {
      table[from][input] = new BitSet();
      table[from][input].set(to);
    }
    if (epsilon[from] == null) {
      epsilon[from] = new BitSet();
    }
    if (epsilon[to] == null) {
      epsilon[to] = new BitSet();
    }
  }
```

Die Methode `addEpsilonTransition()` fügt einen neuen Leerübergang hinzu.

```
  public void addEpsilonTransition(int from, int to) {
    int max = Math.max(from, to) + 1;
    adjustSize(max);
    if (epsilon[from] != null) {
      epsilon[from].set(to);
    } else {
      epsilon[from] = new BitSet();
      epsilon[from].set(to);
    }
  }
```

### 2.5.4 Von NEA zu DEA

Die Methode `toDFA()` lässt sich aus derjenigen der Methode `execute()` ableiten. Dabei können die Mengen $\epsilon$_closure(move($T, a$)) in einem JAVA-Vektor gespeichert. Das Vektorindex ergibt die Nummer des entsprechenden DEA Zustand. Achtung, Sie brauchen einen zusätzlichen Vektor um die Zustände, von denen die Nachfolgezustände noch nicht bekannt sind, zu speichern.

Bei der Erzeugung des DEA's kennen wir die Anzahl Zustände nicht zum Voraus. Am besten geht es mit einer dynamischen Erzeugung der DEA Zustände und Zustandsübergänge. Zu diesem Zweck wird die Klasse `DFA` mit folgenden Methoden erweitert:

Die Methode `adjustSize()` vergrössert die Zustandstabellen des DEA's (falls nötig). `STATES` ist eine ganzzahlige Konstante mit Wert `100`.

```
private void adjustSize(int state) {
  states = Math.max(state + 1, states);
  int length = accepting.length;
  if (state < length) {
    return;
  } else {
    while (length <= state)
      length += STATES;
    boolean[] newAccepting = new boolean[length];
    int[][] newTable = new int[length][INPUTS];
    System.arraycopy(accepting, 0, newAccepting, 0, accepting.length);
    System.arraycopy(table, 0, newTable, 0, accepting.length);
    for (int i = accepting.length; i < length; i++) {
      for (int j = 0; j < INPUTS; j++) {
        newTable[i][j] = SE;
      }
      accepting[i] = false;
    }
  }
}
```

Die Methode `addState()` fügt einen neuen Zustand hinzu.

```
public void addState(int state) {
  adjustSize(state);
}
```

Die Methode `addTransition()` fügt einen neuen Zustandsübergang hinzu.

```
public void addTransition(int from, char input, int to) {
  adjustSize(Math.max(from, to));
  table[from][input] = to;
}
```

### 2.5.5 Aufgaben

**Aufgabe 2.16 [Projekt RA zu DEA]**

1. *Schreiben sie ein* JAVA*-Programm zur Transformation eines RA's in einen NEA gemäss vorgehen aus Abschnitt 2.5.3.*
2. *Schreiben sie ein* JAVA*-Programm zur Transformation eines NEA's in einen DEA gemäss vorgehen aus Abschnitt 2.5.4.*
3. *Schreiben sie ein* JAVA*-Programm zur Transformation eines RA's in einen DEA indem Sie die Programme aus den ersten zwei Teilaufgaben zusammenfügen. Testen Sie Ihr Programm gemäss Abschnitt 2.5.2.*

# Kapitel 3

# Syntaktische Analyse (Parsing)

## 3.1 Formale Sprachen

### 3.1.1 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.

### 3.1.2 Theoretische Konzepte

**Definition 3.1 [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 3.2 [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[1] $v \rightarrow w$ wobei $v$ und $w$ Strings über $V = T \cup N$ sind. Mit Hilfe der Produktionen wird ein **Syntaxbaum** entwickelt. 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 3.1 [Wörter]** $V$ ist die Vereinigungsmenge der terminalen ($T$) und nichtterminalen ($N$) Symbole.

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

[1] An Stelle von $v \rightarrow w$ wird auch oft die Notation $v ::= w$; verwendet.

$V^0$ 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>)  oder 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 [2] 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.

**Beispiel 3.1 [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 ::= ""
                   ;
```

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

```
   par ::= <TEXT>
         ;
stitle ::= <TEXT>
         ;
```

**Definition 3.3 [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* $a \rightarrow bA$ *bzw.* $arightarrowA$ *oder die Form* $a \rightarrow Ab$ *bzw.* $a \rightarrow A$ *wobei* $a$ *und* $b$ *Nichtterminalsymbole sind.*

**Kontextfreie Grammatik (Typ 2 Grammatik)** *alle Produktionen haben die Form* $a \rightarrow w$ *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* $v \rightarrow w$ *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* $v \rightarrow w$ *wobei* $v$ *und* $w$ *Strings aus* $V$ *sind.*

**Bemerkung 3.1 [Kontextfrei]** Eine Produktion $a \rightarrow w$ 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 3.2 [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 3.4 [Herleitung]** *Ein String* $v$ *kann aus dem String* $w$ ***direkt hergeleitet*** *werden wenn es Strings* $\alpha$ *,* $\beta$ *und* $w'$*, ein nonterminal* $v'$*, sowie eine Produktion* $p : v' \rightarrow w'$ *so gibt, dass* $v = \alpha v' \beta b$ *und* $w = \alpha a w' \beta b$ *. Man sagt dann, dass die Produktion* $p$ *in* ***Kontext*** *von* $a$ *und* $b$ *Anwendung findet (* $a$ *und* $b$ *dürfen leer sein). Wir schreiben* $w \Rightarrow v$ *oder auch* $\alpha v' \beta \Rightarrow \alpha w' \beta$

*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 3.5 [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) kontextfrei.

**Beispiel 3.2 [Taschenrechner]** Gegeben sei Folgende Grammatik $G = (T, N, P, s)$ wobei

$T = \{+, -, *, /, (, ), \text{<ID>}\}$

$N = \{e\}$

$P$

$$\begin{aligned} e &\rightarrow e + e \\ e &\rightarrow e - e \\ e &\rightarrow e * e \\ e &\rightarrow e/e \\ e &\rightarrow (e) \\ e &\rightarrow -e \\ e &\rightarrow \text{<ID>} \end{aligned}$$

$s = e$

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

$$\begin{aligned} e &\Rightarrow -e \\ &\Rightarrow -(e) \\ &\Rightarrow -(e + e) \\ &\Rightarrow -(e + \text{<ID>}) \\ &\Rightarrow -(\text{<ID>} + \text{<ID>}) \end{aligned}$$

oder als Linksableitung

$$\begin{aligned} e &\Rightarrow -e \\ &\Rightarrow -(e) \\ &\Rightarrow -(e + e) \\ &\Rightarrow -(\text{<ID>} + e) \\ &\Rightarrow -(\text{<ID>} + \text{<ID>}) \end{aligned}$$

herleiten.

### 3.1.3 Backus-Naur Form

**Notation 3.2 [BNF]** Gib es mehrere Produktionen mit dem selben Symbol auf der linken Seite, so wird oft die sog. Backus-Naur Form (BNF) verwendet: $v \rightarrow w_1$, $v \rightarrow w_2$, ..., $v \rightarrow w_n$, $v \rightarrow w_1|w_2...|w_n$, abgekürzt.

**Beispiel 3.3 [Postfix]** Grammatik $G = (T, N, P, s)$ für arithmetische Ausdrücke in Postfix Notation:

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

$N = \{\text{num}, \text{digit}, \text{op}, \text{addop}, \text{subop}, \text{mulop}, \text{divop}, \text{e}\}$

$P$

$$\begin{aligned} \text{num} &\rightarrow \text{num digit} \mid \text{digit} \\ \text{digit} &\rightarrow 0|1|2|3|4|5|6|7|8|9 \\ \text{e} &\rightarrow \text{e e op} \mid \text{num} \\ \text{op} &\rightarrow \text{addop} \mid \text{subop} \mid \text{mulop} \mid \text{divop} \\ \text{addop} &\rightarrow + \\ \text{subop} &\rightarrow - \\ \text{mulop} &\rightarrow * \\ \text{divop} &\rightarrow / \end{aligned}$$

$s = e$

In der Praxis wird die Grammatik aus Beispiel 3.3, unter Verwendung von Tokens, wie folgt vereinfacht:

**Beispiel 3.4 [Postfix]** Grammatik $G = (T, N, P, s)$ für arithmetische Ausdrücke in Postfix Notation:

$T = \{$<DIGIT>, <OP>$\}$

$N = \{$num, op$\}$

$P$

num → num <DIGIT> | <DIGIT>
e → e e op | num

$s = e$

Die BNF-Notation lässt sich selbstverständlich in BNF darstellen.

**Beispiel 3.5 [BNF in BNF]**

<NT> ist ein Nonterminal- und <T> ist ein Terminalsymbol:

bnf → production bnf | $\epsilon$
production → <NT> " → " choice
choice → sequence | sequence "|" choice
sequence → factor | factor sequence
factor → <NT> | <T>

Reguläre Ausdrücke lassen sich auch ahnlich wie die BNF-Notation mit einer Grammatik beschreiben:

**Beispiel 3.6 [RA in BNF]**

<Z> ist ein Zeichen aus dem Alphabet des RA's.

ra → choice
choice → sequence | sequence "|" choice
sequence → iterate | iterate sequence
iterate → factor " $*$ " | factor
factor → <Z> | "(" ra ")"

### 3.1.4 Ableitungen

Im Folgenden ist die Grammatik kontextfrei.

**Definition 3.6 [Ableitung]** *Eine Herleitung heisst Linksableitung, bzw. Rechtsableitung, wenn jeweils das am weitesten links, bzw. rechts stehende nichtterminale Symbol ersetzt wird (Siehe Bsp. 3.2).*

**Definition 3.7 [Ableitungsbaum]** *Ein Ableitungsbaum ist eine Graphische Darstellung von Ersetzungsschritten. Für jede Produktion $a \rightarrow bcd$ existiert ein Teilableitungsbaum mit Wurzel $a$ und Unterbäume $b$, $c$ und $d$ (Siehe Abb. 3-1 und 3-2).*

**Definition 3.8 [Mehrdeutigkeit]** *Eine Zeichenkette $x$ heisst **mehrdeutig**, falls sie $n > 1$ verschiedene Ableitungsbäume bezüglich einer Grammatik $G$ besitzt (Siehe Abb. 3-3).*

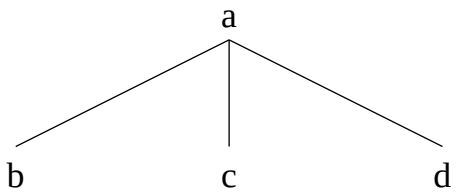



Abbildung 3-1: Ableitungsbaum für die Produktion $a \rightarrow bcd$

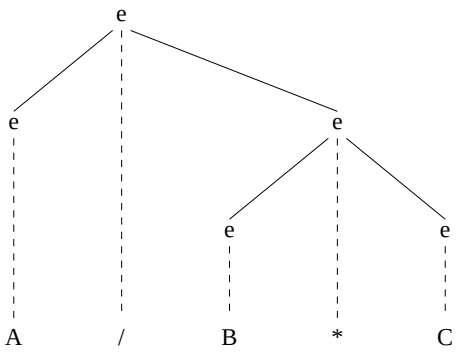



Abbildung 3-2: Ableitungsbaum für die Eingabe `A / B * C`

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

**Beispiel 3.7 [Mehrdeutig]** Die Grammatik aus Bsp. 3.2 ist mehrdeutig (Siehe Abb. 3-3).

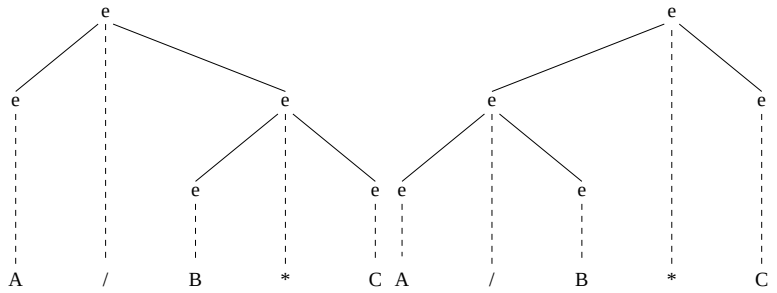



Abbildung 3-3: Mehrdeutige Grammatik

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

**Beispiel 3.8 [Mehrdeutig]** Die Grammatik $G = (T, N, P, s)$ ist nicht eindeutig

$T = \{+, *, (, ), \text{<ID>}\}$

$N = \{e\}$

$P$

$$\begin{aligned} e &\rightarrow e + e \\ e &\rightarrow e * e \\ e &\rightarrow (e) \\ e &\rightarrow \text{<ID>} \end{aligned}$$

$s = e$

Sie lässt sich aber einfach in eine eindeutige Grammatik $G_1 = (T_1, {}_1 N, P_1, s_1)$ umwandeln, nämlich:

$T_1 = \{+, *, (, ), \text{<ID>}\}$

$N_1 = \{a, m, p\}$

$P_1$

$$\begin{array}{rcl} a & \rightarrow & m + a \mid m \\ m & \rightarrow & p * m \mid p \\ p & \rightarrow & (a) \mid \text{<ID>} \end{array}$$

$s_1 = a$

### 3.1.5 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 3.9 [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    ::=   "{" varDecls 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
          ;
```

### 3.1.6 Erweiterte Backus-Naur Form

**Notation 3.3 [EBNF]** $v \to (w)*$ bzw. $v \to \{w\}$ Repetition von einem String $w$ ($v \to wv|\epsilon$)

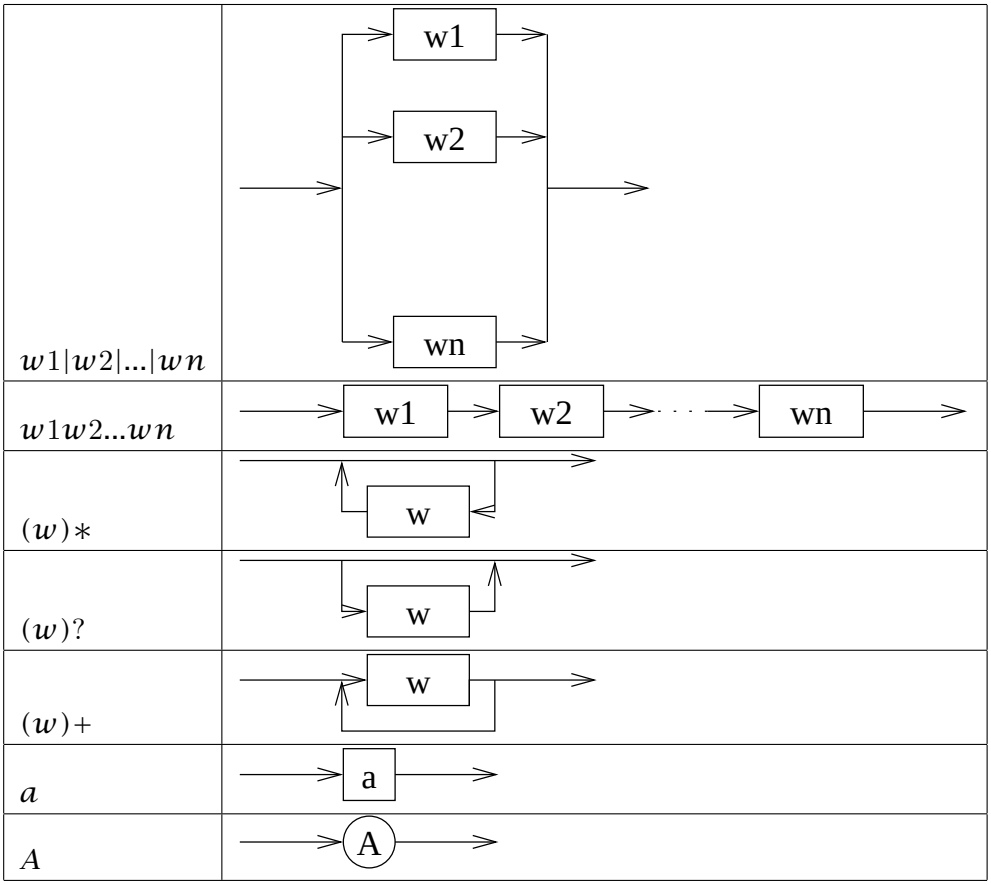
$v \to (w)?$ bzw. $v \to [w]$ optionales String ( $v \to w|\epsilon$)

$v \to (w)+$ Mindestens eine Repetition von w ($v \to wv|w$)

### 3.1.7 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 3.9 [Syntaxdiagramme]** *Die EBNF-Konstrukte werden wie folgt in* ***Syntaxdiagramme*** *umgesetzt:*




**Beispiel 3.10 [Reelle Zahl]**

### 3.1.8 Implementierung

Eine Grammatik lässt sich einfach in JAVA implementieren. Zuerst brauchen wir eine abstrakte Klasse für Symbole.

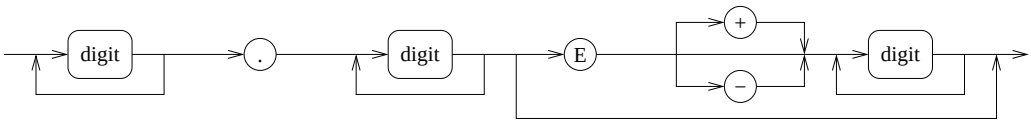



Abbildung 3-4: Syntaxdiagramm für eine Vorzeichenlose reele Zahl

```
public abstract class Symbol {

  public String name;
}
```

Terminal Symbole werden mit dem Singleton-Muster implementiert.

```
public class TerminalSymbol extends Symbol {

  private static Hashtable<String, TerminalSymbol> instances =
    new Hashtable<String, TerminalSymbol>();
  private static int count = 0;

  private TerminalSymbol(String name) {
    this.name = name.intern();
  }

  public static TerminalSymbol getInstance(String name) {
    TerminalSymbol ts = instances.get(name.intern());
    if (ts != null) {
      return ts;
    } else {
      ts = new TerminalSymbol(name);
      instances.put(name, ts);
      return ts;
    }
  }

  public static Hashtable<String, TerminalSymbol> getInstances() {
    return instances;
  }
}
```

Nonterminal Symbole werden mit dem Singleton-Muster implementiert.

```
public class NonTerminalSymbol extends Symbol {

  private static Hashtable<String, NonTerminalSymbol> instances =
    new Hashtable<String, NonTerminalSymbol>();
  public boolean start = false;

  private NonTerminalSymbol(String name) {
    this.name = name.intern();
  }

  public static NonTerminalSymbol getInstance(String name) {
    NonTerminalSymbol nts = instances.get(name.intern());
    if (nts != null) {
      return nts;
```

```
    } else {
      nts = new NonTerminalSymbol(name);
      instances.put(name, nts);
      return nts;
    }
  }

  public static Hashtable<String, NonTerminalSymbol> getInstances() {
    return instances;
  }
}
```

Eine Produktion besteht aus einer linken Seite (Nonterminal) und einer rechten Seite (Folge von Symbolen).

```
public class Production {

  public NonTerminalSymbol lhs;
  public Vector<Symbol> rhs;

  public Production(NonTerminalSymbol lhs, Vector<Symbol> rhs) {
    this.lhs = lhs;
    this.rhs = rhs;
  }
}
```

Schliesslich verwaltet die Klasse `Grammar` eine Liste von Produktionen.

```
public class Grammar {

  public Vector<Production> table = new Vector<Production>();
  private NonTerminalSymbol start;

  public void addProduction(Production p) {
    table.add(p);
  }
}
```

### 3.1.9 Aufgaben

**Aufgabe 3.1 [Sprachhierarchie]** *Zeigen Sie, dass eine Sprache vom Typ $i$ auch eine Sprache von Typ $m-1$ ist, $i = 1...3$.*

**Aufgabe 3.2 [Reguläre Ausdrücke und Grammatiken]** *Zeigen Sie, dass reguläre Ausdrücke und Typ 3 Grammatiken äquivalent sind.*

**Aufgabe 3.3 [Klammerpaare]** *Zeigen Sie, dass kontextfreie Grammatiken zur Beschreibung von Klammerpaaren geeignet sind, z.B.* `begin - end`.

**Aufgabe 3.4 [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.*

**Aufgabe 3.5 [Eindeutigkeit]** *Gegeben sei eine Kontextfreie Grammatik G mit Produktionen* $s \rightarrow s\ s$ `"+"` $|\ s\ s$ `"*"` $|$ `"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?*

**Aufgabe 3.6 [Vereinfachung]** *Vereinfachen Sie die Grammatik aus Beispiel 3.9 unter Verwendung der EBNF-Notation. Eliminieren Sie insbesondere alle Literals.*

**Aufgabe 3.7 [EBNF in EBNF]** *Entwerfen Sie eine EBNF-Grammatik für EBNF-Grammatiken.*

**Aufgabe 3.8 [Römischen Zahlen]** *Konstruieren Sie eine Grammatik für die römischen Zahlen im Bereich* 1 *bis* 2000 *.*

**Aufgabe 3.9 [Reguläre Ausdrücke]** *Erweitern Sie die Grammatik aus Beispiel 3.6 mit Zeichenklassen gemäss folgender Spezifikation:*

| | |
|---|---|
| `[x-yz]` | *Gegebene Intervalle / Einzelzeichen* |
| `[^x-yz]` | *Alle Zeichen ausser angegebene Intervalle / Einzelzeichen* |

## 3.2 Top-down Parsing

### 3.2.1 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.

#### 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.

### 3.2.2 Top-down Parsing

**Beispiel 3.11 [Wirth]** Aus [Wir86] und [Wir96]. 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 die Produktion b → ß" 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. 3.12).

**Beispiel 3.12 [Wirth]** Aus [Wir86], und [Wir96] Gegeben sei folgende Grammatik:

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

Wir versuchen, den Satz "x" "x" ß" (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, ß" 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 ß" 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 ß" 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.

### first-Mengen

Im Beispiel 3.12 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 first($w$)

**Definition 3.10 [First]** *Es sei $w$ ein String aus $V$ . Dann ist 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 first($w$) . D.h. falls $w \Rightarrow \epsilon$ : first($w$) = $\{A \in T; w \Rightarrow Av\} \cup \{\epsilon\}$ sonst: first($w$) = $\{A \in T; w \Rightarrow Av\}$*

Die Grammatik wird wie folgt eingeschränkt:

**Gesetz 3.1 [first]** Für jede Produktion $a \rightarrow w_1|w_2|...|w_n$ wird verlangt, dass die Initialsymbolmengen aller Terme $w_i$ disjunkt sind, d.h. der Durchschnitt first($w_i$) ∩ first($w_j$) ist leer für verschiedene $i$ und $j$ .

Die first Mengen können mit folgendem Algorithmus berechnet werden.

**Algorithmus 3.1 [First]** Bestimmung von first($x$)

1. Falls $x \in T$ , dann first($x$) = $\{x\}$
2. Falls $x \rightarrow \epsilon$ eine Produktion ist, dann füge $\epsilon$ zu first($x$) hinzu.

3. Falls $x$ nicht Terminalsymbol ist und $x \rightarrow y_1 y_2 ... y_n$ eine Produktion ist, dann nimm $A$ zu first($x$) hinzu, falls $A$ für irgendein $i$ in first($y_i$) und $\epsilon$ in allen first($y_1$) , first($y_2$) ... first($y_{i-1}$) enthalten ist. Wenn $\epsilon$ in allen first($y_i$) enthalten ist, nimm $\epsilon$ zu first($x$) hinzu.

In der Syntax vom Beispiel 3.12 ist `"x"` sowohl in `first(a)` als auch in `first(b)` enthalten. Die Regel 3.1 ist also durch die Produktion `a → a | b` verletzt.

**Beispiel 3.13 [Wirth]** Aus [Wir86]und [Wir96]. Wir gehen nun die Probleme aus Beispiel 3.12 wie folgt um, indem eine äquivalente Syntax definiert wird, welche die Regel 3.1 beachtet:

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

Der Satz `"x" "x" ß"` 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"
""         ""
```

**follow-Mengen**

Leider genügt Regel 3.1 noch nicht, um alle Schwierigkeiten auszuschliessen:

**Beispiel 3.14 [Wirth]** Aus [Wir86] und [Wir96]. 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 3.14 kann nur entstehen, wenn ein Symbol die leere Folge erzeugen kann. Das Symbol `"x"` gehört zwar zu `first(a)` , `"x"` kann aber auch `a` unmittelbar folgen!

Die Menge aller Symbole, die einer aus $a$ hergeleiteten Folge unmittelbar nachfolgen können, wird mit follow($a$) bezeichnet:

**Definition 3.11 [follow]** *Es sei $a \in N$ . follow($a$) ist die Menge aller Terminale $A$ , die in einer Satzform direkt rechts neben $a$ stehen können.*

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

**Gesetz 3.2 [follow]** Für jedes Nichtterminalsymbol $a$ , aus welchen die leere Folge hergeleitet werden kann, muss die Menge first($a$) seiner Initialsymbole disjunkt von der menge follow($a$) der Folgesymbole sein:

Aus $a \Rightarrow \epsilon$ folgt dass first($a$) $\cap$ follow($a$) leer ist.

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

**Algorithmus 3.2 [follow]** Bestimmung von follow($a$) :

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

**Beispiel 3.15 [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> }
```

#### LL(1) Grammatiken

**Definition 3.12 [LL(1) Grammatik]** *Eine **LL(1) Grammatik** ist eine Grammatik, die die Regeln 3.1 und 3.2, 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 3.13 [LL(1) Grammatik]** *Eine LL(1)-Grammatik ist eine Grammatik, deren Parse-Tabelle keine Mehrfacheinträge besitzt (siehe Abschnitt 3.2.5).*

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

#### Semantik

**Bemerkung 3.3 [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 3.1 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.

### 3.2.3 Probleme in der Praxis

Regeln 3.1 und 3.2 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.

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

**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 3.16 [Taschenrechner]** Die Produktionen $e \rightarrow e"+"t|e"-"t|t$ besitzen das Nichtterminal $e$ als gemeinsames Präfix. Durch Einführung eines neuen Nichtterminals $e'$ erhalten wir äquivalente Produktionen $e \rightarrow ee'|t$ und $e' \rightarrow "+"t|"-"t$

**Algorithmus 3.3 [Links-Faktorisieren]** Solange es Produktionen $a \rightarrow vw_1$ und $a \rightarrow vw_2$ gibt, mit $v \neq \epsilon$

1. Streiche $a \rightarrow vw_1$ und $a \rightarrow vw_2$ aus $P$
2. Füge folgende Produktionen $a \rightarrow va'$ und $a' \rightarrow w_1|w_2$ hinzu, wobei $a'$ ein neues Nichtterminal ist.

**Links-Rekursion**

**Definition 3.14 [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 $a \Rightarrow aw$ gibt, mit $a \in N$ und $w \in V$*

Insbesondere sind Produktionen der Form $a \rightarrow aA|...$ oder $c \rightarrow dC|...$ und $d \rightarrow cD|...$ links-rekursiv.

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

**Beispiel 3.17 [Elimination der Linksrekursion]** Es sei $G = (T, N, P, s)$ eine kontextfreie Grammatik. Die Linksrekursion $a \rightarrow av|w_1|w_2|...|w_n$ lässt sich in $a \rightarrow w_1a'|w_2a'|...|w_na'$ und $a' \rightarrow va'|\epsilon$ umwandeln.

### 3.2.4 Rekursiver Abstieg

Wir wollen nun annehmen, dass die Gramatik die Bedingungen aus Regel 3.1 und Regel 3.2 erfüllt, d.h, dass die Grammatik die $LL(1)$ -Eigenschaft (siehe Abschnitt 3.2.2). 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.

[4] Es können aber dabei semantische Probleme auftreten

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

**Definition 3.15 [predict]** *Falls* $\epsilon \in first(w)$ *dann ist* $predict(a \to w) = (first(w) - \epsilon) \cup follow(w)$ *sonst ist* $predict(a \to w) = first(w)$ .

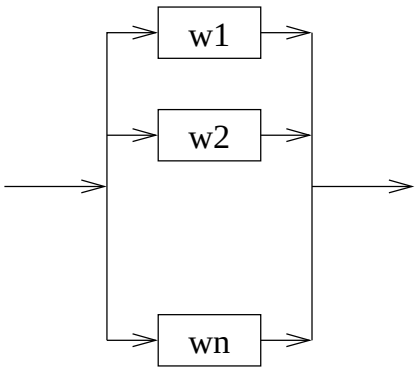
**Definition 3.16 [Rekursiver Abstieg]** *Ein Parser mit rekursivem Abstieg (recursive descent parser) enthält für jedes Nichtterminal a eine Methode* `a()` *und arbeitet wie folgt:*

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.*

### Rekursiver Abstieg: Kodierung

Aus [Wir86] und [Wir96]. Wir bezeichnen das programm, das aus der Übersetzung eines graphens $S$ hervorgeht, mit $p(S)$ .

- Jede Struktur mit der Form



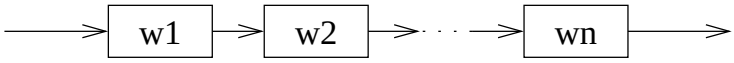

wird in eine bedingte oder selektive Anweisung übersetzt:

```
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($w_i$) .

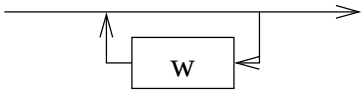
- Jede Struktur mit der Form




wird in eine Sequenz übersetzt:

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

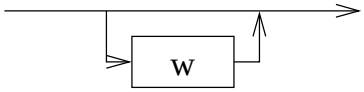
- Jede Struktur mit der Form




wird in eine Wiederholung übersetzt:

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

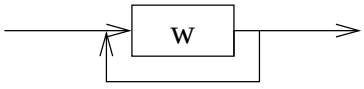
- Jede Struktur mit der Form




wird in eine bedingte Anweisung übersetzt:

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

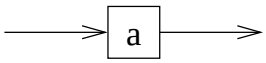
- Jede Struktur mit der Form




wird in eine Wiederholung übersetzt:

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

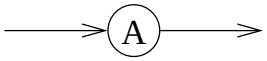
- Jede Struktur mit der Form




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 3.18 [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 ß" 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) = {ß"}` enthalten sein.

**Beispiel 3.19 [Tischrechner]**

```
program ::=   statl
          ;
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
          ;
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`

```
1  PARSER_BEGIN(Scanner)
2
3  public class Scanner { }
4
5  PARSER_END(Scanner)
6
7  TOKEN :
8  {
9    < LPAREN: "(" >
10 | < RPAREN: ")" >
11 | < SEMICOLON: ";" >
12 | < ASSIGN: "=" >
13 | < PLUS: "+" >
14 | < MINUS: "-" >
15 | < STAR: "*" >
16 | < SLASH: "/" >
17 | < PRINT: "print" >
18 | < IDENTIFIER:  ["a"-"z","A"-"Z"]
19                  ( ["a"-"z","A"-"Z","0"-"9"] )* >
20 | < NUMBER: ( ["0"-"9"] )+ >
21 }
22
```

```
23  SKIP :
24  {
25    " "
26  | "\t"
27  | "\n"
28  | "\r"
29  | "\f"
30  }
31
32
```

JAVA-Klasse Parser

```
 1  class Parser implements ScannerConstants {
 2
 3    static Token lookahead;
 4    static Scanner scanner;
 5
 6    public static void main (String args[]) {
 7      scanner = new Scanner(System.in);
 8      lookahead = scanner.getNextToken();
 9      program();
10      System.out.println("> parsing succesfull");
11    }
12
13    static void program() {
14      statl();
15      match(EOF);
16    }
17
18    static void statl() {
19      while (lookahead.kind != EOF)
20        stat();
21    }
22
23    static void stat() {
24      switch(lookahead.kind) {
25      case IDENTIFIER:
26        match(IDENTIFIER);
27        match(ASSIGN);
28        addExpr();
29        match(SEMICOLON);
30        break;
31      case PRINT :
32        match(PRINT);
33        match(LPAREN);
34        addExpr();
35        match(RPAREN);
36        match(SEMICOLON);
37        break;
```

```
38      default:
39        unexpectedToken(lookahead);
40      }
41    }
42
43    static void addExpr() {
44      mulExpr();
45      switch(lookahead.kind) {
46      case PLUS:
47        match(PLUS);
48        addExpr();
49        break;
50      case MINUS:
51        match(MINUS);
52        addExpr();
53        break;
54      }
55    }
56
57    static void mulExpr() {
58      unaExpr();
59      switch(lookahead.kind) {
60      case STAR:
61        match(STAR);
62        mulExpr();
63        break;
64      case SLASH:
65        match(SLASH);
66        mulExpr();
67        break;
68      }
69    }
70
71    static void unaExpr() {
72      if (lookahead.kind == MINUS) {
73        match(MINUS);
74      }
75      priExpr();
76    }
77
78    static void priExpr() {
79      switch (lookahead.kind) {
80      case LPAREN:
81        match(LPAREN);
82        addExpr();
83        match(RPAREN);
84        break;
85      case IDENTIFIER:
86        match(IDENTIFIER);
87        break;
```

```
88        case NUMBER:
89          match(NUMBER);
90          break;
91        default:
92          unexpectedToken(lookahead);
93        }
94      }
95
96      static void unexpectedToken(Token lookahead) {
97        System.out.println("> at line " + lookahead.beginLine +
98                   ", column " + lookahead.beginColumn);
99        System.out.println("> unexpected token : " +
100                   tokenImage[lookahead.kind]);
101       System.exit(1);
102     }
103
104     static void match(int tokenKind) {
105       if (lookahead.kind == tokenKind)
106         lookahead = scanner.getNextToken();
107       else
108         unexpectedToken(lookahead);
109     }
110   }
```

**Beispiel 3.20 [Reguläre Ausdrücke]**

Mit Hilfe der Grammatik aus Beispiel 3.6 kann folgendes Programgerüst entworfen werden:

```
public RegularExpression choice() {
  RegularExpression s = sequence();
  while (lookahead == '|') {
    match('|');
    s = new ChoiceExpression(s, choice());
  }
  return s;
}

public RegularExpression sequence() {
  RegularExpression r = iterate();
  while (lookahead != -1 && lookahead != '|' && lookahead != ')') {
    r = new SequenceExpression(r, sequence());
  }
  return r;
}

public RegularExpression iterate() {
  RegularExpression f = factor();
  if (lookahead == '*') {
    match('*');
    f = new IterateExpression(f);
```

```
    }
    return f;
  }

  public RegularExpression factor() {
    RegularExpression re = null;
    if (lookahead == '(') {
      match('(');
      re = choice();
      match(')');
    } else if (lookahead >= 0 && lookahead <= 127 && lookahead != '|'
        && lookahead != ')' && lookahead != '(' && lookahead != '*') {
      int c = lookahead;
      match(c);
      re = new CharacterExpression((char) c);
    } else {
      unexpectedChar();
    }
    return re;
  }
```

### 3.2.5 Prädiktive Parser

Ein **prädiktiver Parser** ist eine nicht rekursive Version eines Parsers mit rekursivem Abstieg. Dabei wird ein Stack explizit verwaltet.

#### 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$[a, A]$ , wobei $a$ , ein Nichtterminal und $A$ ein Terminal oder $ ist (siehe Abb. 3-5).

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 $[x, A] = x \rightarrow uvw$ so setzt der Parser $u$ , $v$ und $w$ auf den Stack ($u$ oben). Als Ausgabe erfolgt z.B. die angewendete Produktion.

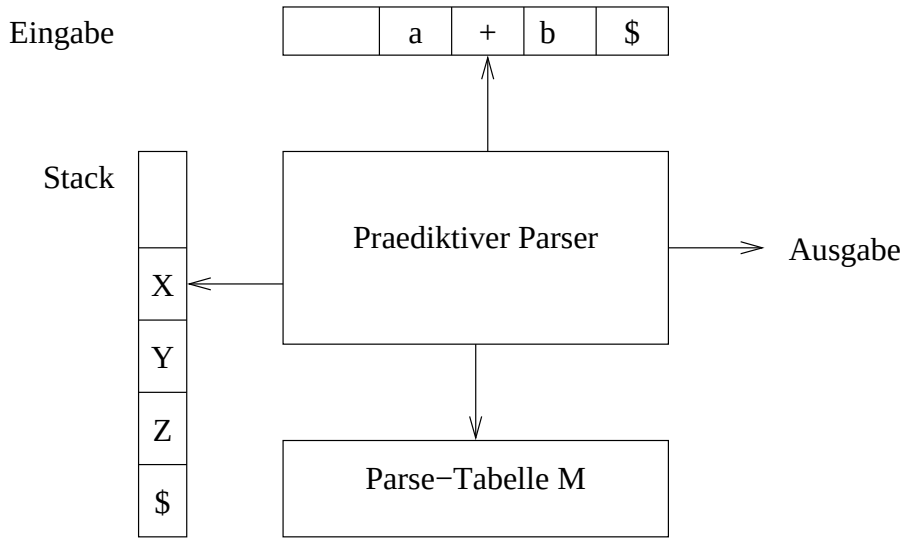



Abbildung 3-5: Prädiktiver Parser

**Algorithmus 3.4 [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** $M[x, A] = x \rightarrow y_1 y_2 ... y_k$ **then begin**
pop()
push($y_i$), $i = k, ..., 1$
gib dir Produktion $x \rightarrow y_1 y_2 ... y_k$ aus
**end**
**else** error()
**until** $x$ = \$

**Beispiel 3.21 [Parse Tabelle]** Die Grammatik aus Bsp. 3.15 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 `first` und `follow` verwendet. Zusätzlich sind die Elemente von `follow` sehr nützlich, als synchronisierende Symbole bei Fehler-Recovery.

### 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 $a \rightarrow w \in P$ und $A \in$

first($w$). Dann expandiert der Parser $a$ zu $w$ , wenn $A$ aktuelles Eingabesymbol ist. Zu Komplikationen kann es kommen, wenn $w = \epsilon$ oder $w \Rightarrow \epsilon$ gilt. In diesem Fall muss $a$ erneut zu $w$ expandiert werden, wenn das aktuelle Eingabesymbol in follow($a$) ist oder wenn in der Eingabe die Endmarkierung \$ erreicht wurde und \$ in follow($a$) enthalten ist.

**Algorithmus 3.5 [Parse-Tabelle]** Konstruktion einer prädiktiven Parse-Tabelle

**Eingabe** Grammatik $G$ .

**Ausgabe** Parse-Tabelle $M$ .

**Methode**
1. Führe für jede Produktion $a \rightarrow w$ die Schritte 2 und 3 durch.
2. Trage für jedes Terminal $A$ aus first($w$) die Produktion $a \rightarrow w$ in M[$a, A$] ein.
3. Wenn $\epsilon \in$ first($w$) enthalten ist, trage $a \rightarrow w$ für jedes Terminal $B$ aus follow($a$) an der Stelle M[$a, B$] ein. Ist $\epsilon \in$ first($w$) und \$ in follow($a$) , so trage $a \rightarrow w$ in M[$a$,\$] ein.
4. Trage in jedem undefinierten Eintrag `error()` ein.

**Beispiel 3.22 [Prädiktiver Parser]** Für die Grammatik $G$ aus Beispiel 3.15: Weil `first(t e') = first(t) = { "(" , <ID> }` ist, wird aufgrund der Produktion `e → t e'` an den Stellen `M[e,"("]` und `M[e,<ID>]` jeweils `e → t e'` eingetragen.

Produktion `e → "+" t e'` bewirkt, dass `e' → "+" t e'` in `M[e',"+"]` eingetragen wird. Produktion `e' → ϵ` bewirkt, dass Produktion `e' → ϵ` sowohl in `M[e',")"]` als auch in `M[e',"$"]` eingetragen wird, da `follow(e') = { ")", "$" }` ist.

**Beispiel 3.23 [Prädiktiver Parser]** Für die Grammatik $G$ aus Beispiel 3.15 sieht ein Programmgerüst wie folgt aus:

JAVA-Program `Prädictiver Parser`

```
1  interface ParserConstants {
2
3    final public static int  ID        = 257;
4    final public static int  ADDOP     = 258;
5    final public static int  MULOP     = 259;
6    final public static int  LEFTPAR   = 260;
7    final public static int  RIGHTPAR  = 261;
8    final public static int  END       = 262;
9    final public static int  ENDSIGN   = 263;
10
11   final public static int  EXPR      =   1;
12   final public static int  EXPR1     =   2;
13   final public static int  TERM      =   3;
14   final public static int  TERM1     =   4;
15   final public static int  FACTOR    =   5;
16
17   final public static int  ERR       =   0;
18   final public static int  P1        =   1;
19   final public static int  P2        =   2;
20   final public static int  P3        =   3;
21   final public static int  P4        =   4;
```

```
22    final public static int  P5        =   5;
23    final public static int  P6        =   6;
24    final public static int  P7        =   7;
25    final public static int  P8        =   8;
26  }
27
28  class PredictiveParser implements ParserConstants {
29
30    int production[][] = {
31      {       0,      0,         0}, /* ERR */
32      {    TERM,  EXPR1,         0}, /*  P1 */
33      {   ADDOP,   TERM,     EXPR1}, /*  P2 */
34      {       0,      0,         0}, /*  P3 */
35      {  FACTOR,  TERM1,         0}, /*  P4 */
36      {   MULOP, FACTOR,     TERM1}, /*  P5 */
37      {       0,      0,         0}, /*  P6 */
38      { LEFTPAR,   EXPR, RIGHTPAR}, /*  P7 */
39      {      ID,      0,         0}  /*  P8 */
40    };
41
42    int parseTable[][] = {
43      {  P1, ERR, ERR,  P1, ERR, ERR}, /*   EXPR */
44      { ERR,  P2, ERR, ERR,  P3,  P3}, /*  EXPR1 */
45      {  P4, ERR, ERR,  P4, ERR, ERR}, /*   TERM */
46      { ERR,  P6,  P5, ERR,  P6,  P6}, /*  TERM1 */
47      {  P8, ERR, ERR,  P7, ERR, ERR}  /* FACTOR */
48    };
49
50    private int lookahead;
51    private int stackPointer = -1;
52    private int stack[1000];
53
54    public static void main (String args[]) {
55      push(END);
56      push(EXPR);
57      lookahead = scanner.getNextToken();
58      parse();
59    }
60
61    void parse () {
62      int topOfStack;
63      boolean symbolIsTerminal;
64      int actualProduction;
65      int i;
66      do {
67        topOfStack = stack[stackPointer];
68        if ((topOfStack >= ID) && (topOfStack <= END))
69          symbolIsTerminal = true;
70        else symbolIsTerminal = false;
71        if (symbolIsTerminal) {
```

```
72          if (lookahead == topOfStack) {
73            pop();
74            if (lookahead != END)
75              lookahead = scanner.getNextToken();
76          }
77          else error ();
78        }
79        else {
80          actualProduction =
81            parseTable[topOfStack-1][lookahead-257];
82          if (actualProduction == ERR)
83            error ();
84          pop();
85          for (i=2;i>=0;i--) {
86            if (production[actualProduction][i] != 0)
87              push(production[actualProduction][i]);
88          }
89        }
90      }
91      while (topOfStack != END);
92    }
93  }
```

### 3.2.6 Aufgaben

**Aufgabe 3.10 [Linksrekursion]** *Es sei $G$ eine Grammatik mit Startsymbol $c$ . Eliminieren Sie die indirekte Linksrekursion in den Produktionen*
*$c \rightarrow dD|E$ und*
*$d \rightarrow cC|F$*

**Aufgabe 3.11 [if]** *Analysiere die `if` Anweisung der Programmiersprache Java unter Berücksichtigung der der Mehrdeutigkeit.*

**Aufgabe 3.12 [XML]**

*Gegeben sei die Grammatik aus Beispiel 3.19.*

*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:*

```
<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"/>
        <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=""/>
```

**Aufgabe 3.13 [Modulo]**

*Erweitern Sie die Grammatik aus Beispiel 3.19 mit dem Modulo Operator. Erweitern Sie den zugehörigen Parser dementsprechend.*

**Aufgabe 3.14 [Taschenrechner]**

*In einem echten Taschenrechner erfolgt die Ausgabe automatisch und nicht mit Hilfe einer `print` Funktion wie es im Beispiel 3.19. 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>
           ;
```

*Erklären Sie, welche Probleme mit dieser Grammatik auftreten*

*Wie können Sie diese Probleme lösen?*

**Aufgabe 3.15 [Modulo]** *Erweitern Sie die Grammatik aus Beispiel 3.19 mit dem Modulo Operator, Vergleichsoperatoren und boolsche Operatoren. Erweitern Sie dementsprechend den Parser (oder zugehöriger Besucher) aus Beispiel 3.43. Schreiben Sie schliesslich einen Codegenerator für diese erweiterte Grammatik.*

**Aufgabe 3.16 [Parser Generator]** *Schreiben Sie einen Parser, der aus einer Menge von Produktionen in BNF Schreibweise eine* JAVACC*-Spezifikation zur überprüfung der zugehörigen Grammatik erzeugt.*

**Aufgabe 3.17 [Reguläre Ausdrücke]** *Vervolständigen Sie die Programmskizze aus Beispiel 3.20 und fügen Sie zusätzliche Parsemethoden für Zeichenklassen gemäss folgender Spezifikation:*

| | |
|---|---|
| `[x-yz]` | *Gegebene Intervalle / Einzelzeichen* |
| `[^x-yz]` | *Alle Zeichen ausser angegebene Intervalle / Einzelzeichen* |

## 3.3 JAVACC

### 3.3.1 Parser Generatoren

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.

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.

#### Bison

Der wohl bekannteste Parser Generator ist YACC (yet another compiler compiler) ein Befehl, der auf dem UNIX-System verfügbar ist [LMB92]. 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 [LMB92] nachgelesen werden. Die komplette Beschreibung von BISON ist in [DS92] zu finden.

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

#### JavaCC

JAVACC ist ein Compilerbau-Werkzeug ähnlich FLEX und BISON. Im Unterschied zu FLEX und BISON erzeugt JAVACC JAVA 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.

JAVACC ist 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).

#### CUP

CUP[5] (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 besitzt im Wesentlichen dieselbe Funktionalität als BISON.

### 3.3.2 JAVACC

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

**Beispiel 3.24 [Wirth]**

Gemäss Beispiel 3.12, sollte JAVACC eine Fehlermeldung produzieren.

JAVACC-Spezifikation

```
1  PARSER_BEGIN(Wirth01)
2
3  public class Wirth01 {
4    public static void main (String args []) {
5      Wirth01 parser = new Wirth01(System.in);
6      try { parser.s(); }
7      catch (Exception e) { }
8    }
9  }
10
11 PARSER_END(Wirth01)
12
```

[5] CUP ist frei erhältlich unter `http://www.cs.princeton.edu/\char126\relaxappel/modern/java/CUP`

```
13  void s() : {}
14  {
15    a() | b()
16  }
17
18  void a() : {}
19  {
20    "x" a() | "y"
21  }
22
23  void b() : {}
24  {
25    "x" b() | "z"
26  }
```

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 3.25 [Wirth]**

JAVACC Spezifikation (mit Linksrekursion)

```
1   PARSER_BEGIN(Wirth02)
2
3   public class Wirth02 {
4     public static void main (String args []) {
5       Wirth02 parser = new Wirth02(System.in);
6       try { parser.s(); }
7       catch (Exception e) { }
8     }
9   }
10
11  PARSER_END(Wirth02)
12
13  void s() : {}
14  {
15    a() "x"
```

```
16  }
17
18  void a() : {}
19  {
20    s() "y" | "z"
21  }
```

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 3.26 [Wirth]**

Wird jetzt Beispiel 3.14 verwendet.

JAVACC Spezifikation

```
1   PARSER_BEGIN(Wirth03)
2
3   public class Wirth03 {
4     public static void main (String args []) {
5       Wirth03 parser = new Wirth03(System.in);
6       try { parser.s(); }
7       catch (Exception e) { }
8     }
9   }
10
11  PARSER_END(Wirth03)
12
13  void s() : {}
14  {
15    a() "x"
16  }
17
18  void a() : {}
19  {
20    ( "x" )?
21  }
```

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 3.27 [JavaCC]** Für die Grammatik aus Beispiel 3.19 hat eine JAVACC Spezifikation für die top-down Syntaxanalyse folgende Gestalt.

JAVACC-Spezifikation `Parser`

```
1   PARSER_BEGIN(Parser)
2
3   public class Parser {
4     public static void main (String args []) {
5       Parser parser = new Parser(System.in);
6       try {
7         parser.program();
8         System.out.println("programm is correct");
9       }
10      catch (Exception e) {
11        System.out.println("parse error");
12      }
13    }
14  }
15
16  PARSER_END(Parser)
17
18  TOKEN :
19  {
20    < LPAREN: "(" >
21  | < RPAREN: ")" >
22  | < SEMICOLON: ";" >
23  | < ASSIGN: "=" >
24  | < PLUS: "+" >
25  | < MINUS: "-" >
26  | < STAR: "*" >
27  | < SLASH: "/" >
28  | < PRINT: "print" >
29  | < IDENTIFIER:  ["a"-"z","A"-"Z"]
30                   ( ["a"-"z","A"-"Z","0"-"9"] )* >
```

```
31  | < NUMBER: ( ["0"-"9"] )+ >
32  }
33
34  SKIP :
35  {
36    " " | "\t" | "\n" | "\r" | "\f"
37  }
38
39  void program() : {}
40  {
41    statl()  <EOF>
42  }
43
44  void statl() : {}
45  {
46    ( stat() )*
47  }
48
49  void stat() : {}
50  {
51    <IDENTIFIER> <ASSIGN> addExpr() <SEMICOLON>
52  | <PRINT> <LPAREN> addExpr() <RPAREN> <SEMICOLON>
53  }
54
55  void addExpr() : {}
56  {
57    mulExpr() (( <PLUS> | <MINUS> ) addExpr() )?
58  }
59
60  void mulExpr() : {}
61  {
62    unaExpr() (( <STAR> | <SLASH> ) mulExpr() ) ?
63  }
64
65  void unaExpr() : {}
66  {
67    ( <MINUS> )? priExpr()
68  }
69
70  void priExpr() : {}
71  {
72    <LPAREN> addExpr() <RPAREN>
73  | <IDENTIFIER>
74  | <NUMBER>
75  }
```

**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>
            ;
```

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

JAVACC-Spezifikation `Parser`

```
1   PARSER_BEGIN(Parser)
2
3   public class Parser {
4     public static void main (String args []) {
5       Parser parser = new Parser(System.in);
6       try {
7         parser.program();
8       }
9       catch (Exception e) {
10        System.out.println("parse error");
11      }
12    }
13  }
14
15  PARSER_END(Parser)
16
17  TOKEN :
18  {
19    < LPAREN: "(" >
20  | < RPAREN: ")" >
21  | < SEMICOLON: ";" >
22  | < PLUS: "+" >
23  | < MINUS: "-" >
24  | < STAR: "*" >
25  | < SLASH: "/" >
26  | < NUMBER: ( ["0"-"9"] )+ >
27  }
28
29  SKIP :
30  {
31    " "
32  | "\t"
33  | "\n"
```

```
34  | "\r"
35  | "\f"
36  }
37
38  void program():
39  {}
40  {
41    ( statement() )* <EOF>
42    { System.out.println("parsing successful"); }
43  }
44
45  void statement():
46  {}
47  {
48    addExpr() <SEMICOLON>
49  }
50
51  void addExpr():
52  {}
53  {
54    mulExpr() ( ( <PLUS> | <MINUS> ) addExpr() )?
55  }
56
57  void mulExpr():
58  {}
59  {
60    unaExpr() ( ( <STAR> | <SLASH> ) mulExpr() )?
61  }
62
63
64  void unaExpr():
65  {}
66  {
67    ( <MINUS> )? priExpr()
68  }
69
70
71  void priExpr():
72  {}
73  {
74    <LPAREN> addExpr() <RPAREN>
75  | <NUMBER>
76  }
```

### 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.

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

JAVACC-Spezifikation `Parser`

```
1  PARSER_BEGIN(Parser)
2
3  public class Parser {
4    public static void main (String args []) {
5      Parser parser = new Parser(System.in);
6      try {
7        parser.program();
8        System.out.println("parsing successful");
9      }
10     catch (Exception e) {
11       System.out.println("parse error");
12     }
13   }
14 }
15
16 PARSER_END(Parser)
17
18 TOKEN :
19 {
20   < LPAREN: "(" >
21 | < RPAREN: ")" >
22 | < SEMICOLON: ";" >
23 | < PLUS: "+" >
24 | < MINUS: "-" >
25 | < STAR: "*" >
26 | < SLASH: "/" >
27 | < NUMBER: ( ["0"-"9"] )+ >
28 }
29
30 SKIP :
31 {
32   " "
33 | "\t"
34 | "\n"
35 | "\r"
36 | "\f"
37 }
38
39 void program():
40 {
41   int a = 0;
42 }
43 {
44   ( a = addExpr()
45    <SEMICOLON>
46    {System.out.println("> " + a);}
47   ) *
```

```
48    <EOF>
49  }
50
51  int addExpr():
52  {
53    int a =0;
54    int m = 0;
55    Token tok = null;
56  }
57  {
58    m = mulExpr() ( ( tok=<PLUS> | tok=<MINUS> ) a = addExpr() ) ?
59    {
60      if (tok == null) return (m);
61      else if (tok.kind == PLUS) return (m + a);
62      else return (m - a);
63    }
64  }
65
66  int mulExpr():
67  {
68    int u = 0;
69    int m = 0;
70    Token tok = null;
71  }
72  {
73    u = unaExpr() ( ( tok=<STAR> | tok=<SLASH> ) m = mulExpr() ) ?
74    {
75      if (tok == null) return (u);
76      else if (tok.kind == STAR) return (u * m);
77      else return (u / m);
78    }
79  }
80
81  int unaExpr():
82  {
83    int p = 0;
84    Token tok = null;
85  }
86  {
87    ( tok=<MINUS> )? p = priExpr()
88    {
89      if (tok == null) return (p);
90      else return (-p);
91    }
92  }
93
94  int priExpr():
95  {
96    int e = 0;
97  }
```

```
 98  {
 99    <LPAREN> e = addExpr() <RPAREN> {return (e); }
100  | <NUMBER>
101    {
102      e=Integer.parseInt(token.image);
103      return (e);
104    }
105  }
```

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

### Tischrechner 3

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

JAVACC-Spezifikation `Parser`

```
 1  PARSER_BEGIN(Parser)
 2
 3  public class Parser {
 4    public static void main (String args []) {
 5      Parser parser = new Parser(System.in);
 6      try {
 7        parser.program();
 8        System.out.println("parsing successful");
 9      }
10      catch (Exception e) {
11        System.out.println("parse error");
12      }
13    }
14  }
15
16  PARSER_END(Parser)
17
18  TOKEN :
19  {
20    < LPAREN: "(" >
21  | < RPAREN: ")" >
22  | < SEMICOLON: ";" >
23  | < PLUS: "+" >
24  | < MINUS: "-" >
25  | < STAR: "*" >
26  | < SLASH: "/" >
27  | < NUMBER: ( ["0"-"9"] )+ >
28  }
29
30  SKIP :
31  {
```

```
32    " "
33  | "\t"
34  | "\n"
35  | "\r"
36  | "\f"
37  }
38
39  void program():
40  {
41    int a = 0;
42  }
43  {
44    ( a = addExpr()
45     <SEMICOLON>
46     {System.out.println("> " + a);}
47    ) *
48    <EOF>
49  }
50
51  int addExpr():
52  {
53    int m1=0, m2 = 0;
54  }
55  {
56    m1 = mulExpr() ( <PLUS> m2 = mulExpr()  {m1 += m2;}
57                   | <MINUS> m2 = mulExpr() {m1 -= m2;}
58                   ) * {return (m1);}
59  }
60
61  int mulExpr():
62  {
63    int u1 = 0, u2 = 0;
64  }
65  {
66    u1 = unaExpr() ( <STAR> u2 = unaExpr() {u1 *= u2;}
67                   | <SLASH> u2 = unaExpr() {u1 /= u2;}
68                   ) ? {return (u1);}
69  }
70
71  int unaExpr():
72  {
73    int p = 0;
74    Token tok = null;
75  }
76  {
77    ( tok=<MINUS> )? p = priExpr()
78    {
79      if (tok == null) return (p);
80      else return (-p);
81    }
```

```
82  }
83
84  int priExpr():
85  {
86    int e = 0;
87  }
88  {
89    <LPAREN> e = addExpr() <RPAREN> {return (e); }
90  | <NUMBER> {return (Integer.parseInt(token.image)); }
91  }
```

### 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.

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 `Parse`

```
1   // options { DEBUG_PARSER=true;}
2
3   PARSER_BEGIN(Parser)
4
5   public class Parser {
6
7     public static void main (String args []) throws ParseException {
8       Parser parser = new Parser(System.in);
9       parser.program();
10    }
11  }
12
13  PARSER_END(Parser)
14
15  JAVACODE
16  void skipTo(int kind) {
```

```
17    System.out.println("> error (skipping to next " + tokenImage[kind] + ")")
18    Token t;
19    do {
20      t = getNextToken();
21    } while (t.kind != kind && t.kind != EOF);
22  }
23
24  TOKEN :
25  {
26    < LPAREN: "(" >
27  | < RPAREN: ")" >
28  | < SEMICOLON: ";" >
29  | < PLUS: "+" >
30  | < MINUS: "-" >
31  | < STAR: "*" >
32  | < SLASH: "/" >
33  | < NUMBER: ( ["0"-"9"] )+ >
34  }
35
36  SKIP :
37  {
38    " "
39  | "\t"
40  | "\n"
41  | "\r"
42  | "\f"
43  }
44
45  void program():
46  {}
47  {
48    (
49      try {
50        statement()
51      }
52      catch (ParseException e) {
53        skipTo(SEMICOLON);
54      }
55    )* <EOF>
56  }
57
58  void statement():
59  {
60    int a;
61  }
62  {
63    a = addExpr()
64    <SEMICOLON>
65    { System.out.println("> " + a); }
66  }
```

```
67
68  int addExpr():
69  {
70    int m1 = 0 , m2 = 0;
71  }
72  {
73    m1 = mulExpr()
74    (
75      <PLUS>  m2 = mulExpr() {m1 += m2;}
76    |
77      <MINUS> m2 = mulExpr() {m1 -= m2;}
78    ) *
79    {return (m1);}
80
81  }
82
83  int mulExpr():
84  {
85    int u1 = 0, u2 = 0;
86  }
87  {
88    u1 = unaExpr()
89    (
90      <STAR>  u2 = unaExpr() {u1 *= u2;}
91    |
92      <SLASH> u2 = unaExpr(){u1 /= u2;}
93    ) *
94    {return (u1);}
95  }
96
97  int unaExpr():
98  {
99    int p = 0;
100 }
101 {
102   <MINUS> p = priExpr() {return (-p); }
103 |
104   p = priExpr() {return (p); }
105 }
106
107 int priExpr():
108 {
109   int e;
110 }
111 {
112   <LPAREN> e = addExpr() <RPAREN> {return (e); }
113 |
114   <NUMBER> {return (Integer.parseInt(token.image)); }
115 }
```

### 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.

JAVACC-Spezifikation `Parse`

```
 1  options {
 2    MULTI = true;
 3    VISITOR = true;
 4  }
 5
 6  PARSER_BEGIN(Parser)
 7    public class Parser {
 8      public static void main (String args []) {
 9        Parser parser = new Parser(System.in);
10          try {
11          program();
12          ((SimpleNode) jjtree.rootNode()).dump(" ");
13          } catch (Exception e) {
14            e.printStackTrace(); System.exit(1);
15          }
16      }
17    }
18  PARSER_END(Parser)
```

```
19
20  TOKEN :
21  {
22    < LPAREN: "(" >
23  | < RPAREN: ")" >
24  | < SEMICOLON: ";" >
25  | < PLUS: "+" >
26  | < MINUS: "-" >
27  | < STAR: "*" >
28  | < SLASH: "/" >
29  | < NUMBER: ( ["0"-"9"] )+ >
30  }
31
32  SKIP :
33  {
34    " "
35  | "\t"
36  | "\n"
37  | "\r"
38  | "\f"
39  }
40
41  void program() #Program :
42  {}
43  {
44    (statement())*
45  }
46
47  void statement() #Statement :
48  {}
49  {
50    addExpr() <SEMICOLON>
51  }
52
53  void addExpr() #void :
54  {}
55  {
56    ( mulExpr() (( <PLUS> | <MINUS> ) mulExpr())* ) #Add(>1)
57  }
58
59  void mulExpr() #void :
60  {}
61  {
62    ( unaExpr() (( <STAR> | <SLASH> ) unaExpr() )* ) #Mult(>1)
63  }
64
65  void unaExpr() #void :
66  {}
67  {
68    <MINUS> priExpr() #Uminus
```

```
69  |
70    priExpr()
71  }
72
73
74  void priExpr() #void :
75  {
76    Token t;
77  }
78  {
79    <LPAREN> addExpr() <RPAREN>
80  |
81    t=<NUMBER> { jjtThis.setName(t.image); } #Number
82  }
```

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

### 3.3.3 Aufgaben

**Aufgabe 3.18 [EBNF]** *Schreiben Sie eine* JAVACC*-Spezifikation für die EBNF-Notation aus Abschnitt 3.1.6.*

**Aufgabe 3.19 [JavaCC]** *Überlegen Sie sich, wie die Taschenrechner-Programme aus Abschnitt 3.5 mittels* JAVACC *spezifiziert werden können.*

## 3.4 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 3.28 [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.

### 3.4.1 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 3.29 arbeiten.

**Beispiel 3.29 [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 3.30 [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 3.30 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.

### 3.4.2 LR-Syntaxanalysealgorithmus

Im Abschnitt 3.4.1 wurde die Syntax mittels dem sogenannten LR-Syntaxanalysealgorithmus analysiert. Wir wollen nun diesen Algorithmus vorstellen, ohne auf die Konstruktion des zugehörigen Automaten einzugehen.

**Definition 3.17 [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*

$$x_1 x_2 ... x_m A_i A_{i+1} ... A_n\#$$

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$a \rightarrow w$ 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 3.6 [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] = shifts'    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   a → w    then begin
```

```
            hole 2|w| Symbole vom Stack;
            sei s' der Zustand, der nun als oberstes Element
                im Stack steht;
            lege zuerst a , dann goto[s', a] auf den Stack;
            gebe die Produktion a → w aus;
        end
        else if   action[s, A] = accept   then return
        else   error()
    end
```

**Bemerkung 3.4 [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.

### 3.4.3 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 3.18 [Handle]** *Ein **Handle** einer rechtsabgeleiteten Satzform $w$ ist eine Produktion $a \to w_2$ und eine Position in $w$ . An dieser Position wird der String $w_2$ gefunden, d.h. $w = w_1 w_2 w_3$ . Weiter kann $w_2$ in $w$ durch $a$ ersetzt werden, um die in einer Rechtsableitung von $w$ unmittelbar vorherige rechtsabgeleitete Satzform zu erzeugen. Das bedeutet, dass unter der Voraussetzung $S \Rightarrow w_1 a w_3 \Rightarrow w_1 w_2 w_3$ die Produktion $a \to w_2$ an der $w_1$ folgenden Position ein Handle von $w_1 w_2 w_3$ ist. Der String $w_3$ rechts vom Handle enthält nur Terminalsymbole.*

**Beispiel 3.31 [bu.exa.handle]** Betrachte die Grammatik aus Beispiel 3.29 und die Rechtsableitung

```
e  ::=  e "+" e
   ::=  e "+" e "*" e
   ::=  e "+" e "*" <ID3>
   ::=  e "+" <ID2> "*" <ID3>
   ::=  <ID1> "+" <ID2> "*" <ID3>
```

$ID_1$ ist ein Handle der rechtsabgeleiteten Satzform $ID_1 + ID_2 * ID_3$ weil $ID$ die rechte Seite der Produktion $e \to ID$ ist und das ersetzen von $ID_1$ durch $e$ die unmittelbar vorher abgeleitete Satzform $e + ID_2 * ID_3$ erzeugt.

Ferner gilt

### 3.4.4 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 3.19 [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 3.32 [bu.exa.items]** Menge der Items zur Produktion $e \to t + f$

| | |
|---|---|
| $e \to .t + f$ | Die Produktion $e \to t + f$ kann potentiell angewendet werden. |
| $e \to t. + f$ | Ein Teilbaum für $t$ wurde bereits aufgebaut. |
| $e \to t + .f$ | Ein Teilbaum für $t$ wurde bereits aufgebaut und das Terminal + wurde gelesen. |
| $e \to t + f.$ | 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 $z_a$ , der durch das Item $[a \to w_1.xw_2]$ beschrieben ist.

- Ist $x$ ein Terminalsymbol und ist das Lookahead auch $x$ , so wird eine `shift`-Operation durchgeführt und $[a \to w_1x.w_2]$ 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 $a \to w_1xw_2$ durchgeführt. Sei $s_p$ der Vorgängerzustand von $s_a$ , so wird im Anschluss an die Reduktion ein Zustandsübergang nach $s_p$ durchgeführt. Dieser Zustandsübergang wird als `goto`-Operation oder Jumpoperation bezeichnet.

### 3.4.5 Konstruktion des Automaten

Um einen Bottom-Up-Parser zu konstruieren, muss die Parsetabelle konstruiert werden. Dazu werden zwei Funktionen `Hülle` und `goto` benötigt.

### 3.4.6 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 3.29 gäbe es $\epsilon$ -Zustandsübergänge vom Zustand $[e' \to .e]$ zu $[e \to .e + t]$ oder zu $[e \to .t]$ , etc. Will man einen deterministischen Automaten konstruieren, so ist eine Hüllenoperation notwendig.

**Definition 3.20 [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* $[a \rightarrow w_1.bw_2] \in Hülle(I)$ *und* $b \rightarrow w_3 \in P$ *folgt* $[b \rightarrow .w_3] \in Hülle(I)$

**Beispiel 3.33 [bu.exa.huelle]** Es sei $G'$ die Grammatik aus Beispiel 3.29. Ferner sei $I = \{[e' \rightarrow .e]\}$ . 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>]
```

### 3.4.7 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 3.21 [Goto]** *Es sei I eine Menge von Items und x# ein Symbol der Grammatik (Terminal oder Nichtterminal). Dann ist*
$goto(I) = Hülle(\{[a \rightarrow w_1 x > x.w_2]; [a \rightarrow w_1.xw_2] \in I\}$

**Beispiel 3.34 [Goto]** Es sei $G'$ die Grammatik aus Beispiel 3.29. Ferner sei $I = \}[e' \rightarrow e.][e \rightarrow e. + t]\}$ . Dann enthält `goto` $(I, +)$ die Elemente:

```
[e ::= e "+" . t]
[t ::= . t "*" f]
[t ::= . t "/" f]
[t ::= . f]
[f ::= . "(" E ")"]
[f ::= . <ID>]
```

### 3.4.8 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` $([s_{new} \rightarrow .s_{old}])$ und die Zustände werden wie folgt bestimmt:

**Algorithmus 3.7 [LR(0)-Mengen]** Es sei $Z$ die Menge der Zustände.

Setze $Z = \{I_0 =$ `Hülle` $([s_{new} \rightarrow .s_{old}])\}$
**repeat**
  Es sei $Z = \{I_0, ..., I_n\}$
  **for** $x \in V$ **do**

**for** $i = 1$ **to** $n$ **do**
**if** goto$(I_i, x) \neq \emptyset$ 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 3.35 [bu.exa.lr0]** Es sei $G'$ die Grammatik aus Beispiel 3.29.

```
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 $I_k$ .

### 3.4.9 SLR Syntaxanalysetabellen

Es gibt verschiedene Methoden, Syntaxanalysetabellen aufzubauen. Wir zeigen hier die **SLR**-Methode (Simpel Left Right):

**Algorithmus 3.8 [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 $\text{goto}(I_i, A) = I_j$ , 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 $follow(a)$ auf reduce$a \to w_1$ . 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 $\text{goto}[I_i, a] = I_j$ setze $\text{goto}[i, a] = j$ .
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 3.36 [SLRtabellen]** Es sei $G'$ die Grammatik aus Beispiel 3.29.

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 $follow$ 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 3.5 [Tabellenaufbau]** Beim Tabellenaufbau im Beispiel 3.36 sind keine Konflikte aufgetreten. Werden die `shift`- und die `reduce`-Tabelle zusammengefügt, so gibt es höchstens einen Eintrag pro Stelle in der Tabelle.

**Beispiel 3.37 [bu.exa.slrimpl]** Es sei $G'$ die Grammatik aus Beispiel 3.29. 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);
    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();
  }
}
```

### 3.4.10 Bemerkungen

**Definition 3.22 [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 3.8 erfolgt.*

**Bemerkung 3.6 [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 3.7 [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 [LMB92]. 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.

### 3.4.11 Aufgaben

**Aufgabe 3.20 [Shift/Reduce-Tabellen]** *Erweitern Sie die Grammatik aus Beispiel 3.29 mit dem Modulooperator und berechnen Sie die `shift`- und `reduce`-Tabellen.*

**Aufgabe 3.21 [SLR-Tabellen]** *Betrachte folgende Grammatik G*

```
s ::= a s | "B" ;
a ::= s a | "A" ;
```

*Konstruieren Sie die SLR-Syntaxanalysetabelle von G .*

## 3.5 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 [WM92]:

- 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 [WM92]) 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 3.4 [Attribute]** Sei $x \in V$ ein Grammatiksymbol, so bezeichnen wir die zu $x$ **Attribute** $\alpha$ , $\beta$ , etc. mit $x.\alpha$ , $x.\beta$ , etc.

**Definition 3.23 [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* $a \to xy$ *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* $a \to xy$ *und Attrribute* $a.\alpha$ , $x.\alpha$ , $y.\alpha$ *gilt* $x.\alpha = g(a.\alpha)$ , $y.\alpha = g(a.\alpha)$

**Notation 3.5 [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 $a \to ba \quad D$ als Funktion der Attribute der linken Seite berechnet werden.

**Beispiel 3.38 [Deklarierheitseigenschaft]** Wir wollen nun die Grammatik aus Beispiel 3.19 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`

```
1  class Entry {
2      String lexem;
3      int  address;
4      Entry( String lexem, int  address) {
5        this.lexem = lexem;
6        this.address = address;
7     }
8  }
```

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

JAVA-Klasse `Parser`

```
1  import java.util.Hashtable;
2
3  class Parser implements ScannerConstants {
4
5    static Token  lookahead;
6    static Scanner scanner;
7    static Hashtable symbolTable = new Hashtable();
8    static int adress = 0;
9
10   public static void main (String args[]) {
```

```
11      scanner = new Scanner(System.in);
12      lookahead = scanner.getNextToken();
13      program();
14      System.out.println("> parsing succesfull");
15    }
16
17    static void program() {
18      statl();
19      match(EOF);
20    }
21
22    static void statl() {
23      while (lookahead.kind != EOF)
24        stat();
25    }
26
27    static void stat() {
28      switch(lookahead.kind) {
29      case IDENTIFIER:
30        if (symbolTable.get(lookahead.image) == null)
31          symbolTable.put(lookahead.image,
32                  new Entry(lookahead.image,adress++));
33        match(IDENTIFIER);
34        match(ASSIGN);
35        addExpr();
36        match(SEMICOLON);
37        break;
38      case PRINT :
39        match(PRINT);
40        match(LPAREN);
41        addExpr();
42        match(RPAREN);
43        match(SEMICOLON);
44        break;
45      default:
46        unexpectedToken(lookahead);
47      }
48    }
49
50    static void addExpr() {
51      mulExpr();
52      switch(lookahead.kind) {
53      case PLUS:
54        match(PLUS);
55        addExpr();
56        break;
57      case MINUS:
58        match(MINUS);
59        addExpr();
60        break;
```

```
61      }
62    }
63
64    static void mulExpr() {
65      unaExpr();
66      switch(lookahead.kind) {
67      case STAR:
68        match(STAR);
69        mulExpr();
70        break;
71      case SLASH:
72        match(SLASH);
73        mulExpr();
74        break;
75      }
76    }
77
78    static void unaExpr() {
79      if (lookahead.kind == MINUS) {
80        match(MINUS);
81      }
82      priExpr();
83    }
84
85    static void priExpr() {
86      switch (lookahead.kind) {
87      case LPAREN:
88        match(LPAREN);
89        addExpr();
90        match(RPAREN);
91        break;
92      case IDENTIFIER:
93        if (symbolTable.get(lookahead.image) == null) {
94          System.out.println("> at line " +
95                             lookahead.beginLine +
96                             ", column " +
97                             lookahead.beginColumn);
98          System.out.println("> identifier \"" +
99                             lookahead.image +
100                            "\" not declared");
101       }
102       match(IDENTIFIER);
103       break;
104     case NUMBER:
105       match(NUMBER);
106       break;
107     default:
108       unexpectedToken(lookahead);
109     }
110   }
```

```
111
112     static void unexpectedToken(Token lookahead) {
113       System.out.println("> at line " +
114                          lookahead.beginLine +
115                          ", column " +
116                          lookahead.beginColumn);
117       System.out.println("> unexpected token : " +
118                  tokenImage[lookahead.kind]);
119       System.exit(1);
120     }
121
122     static void match(int tokenKind) {
123       if (lookahead.kind == tokenKind)
124         lookahead = scanner.getNextToken();
125       else
126         unexpectedToken(lookahead);
127     }
128   }
```

### 3.5.1 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 3.39 [Anzahl Variablen]** Wir wollen nun in der Grammatik aus Beispiel 3.19 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); }
           ;
addExpr: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`

```
1   class Parser implements ScannerConstants {
2
3     static Token  lookahead;
4     static Scanner scanner;
5
6     public static void main (String args[]) {
7       scanner = new Scanner(System.in);
8       lookahead = scanner.getNextToken();
9       program();
10      System.out.println("> parsing succesfull");
11    }
12
13    static void program() {
14      statl();
15      match(EOF);
16    }
17
18    static void statl() {
19      while (lookahead.kind != EOF)
20      stat();
21    }
22
23    static void stat() {
24      int n = 0;
25      switch(lookahead.kind) {
26      case IDENTIFIER:
27        match(IDENTIFIER);
28        match(ASSIGN);
29        n = 1 + addExpr();
30        System.out.println("> statement contains "
31                           + n + " variables");
32        match(SEMICOLON);
33        break;
34      case PRINT :
35        match(PRINT);
36        match(LPAREN);
37        n = addExpr();
38        System.out.println("> statement contains "
39                           + n + " variables");
40        match(RPAREN);
```

```
41        match(SEMICOLON);
42        break;
43      default:
44        unexpectedToken(lookahead);
45      }
46    }
47
48    static int addExpr() {
49      int n =     mulExpr();
50      switch(lookahead.kind) {
51      case PLUS:
52        match(PLUS);
53        n = n + addExpr();
54        break;
55      case MINUS:
56        match(MINUS);
57        n = n + addExpr();
58        break;
59      }
60      return (n);
61    }
62
63    static int mulExpr() {
64      int n = unaExpr();
65      switch(lookahead.kind) {
66      case STAR:
67        match(STAR);
68        n = n + mulExpr();
69        break;
70      case SLASH:
71        match(SLASH);
72        n = n + mulExpr();
73        break;
74      }
75      return (n);
76    }
77
78    static int unaExpr() {
79      int n = 0;
80      if (lookahead.kind == MINUS) {
81        match(MINUS);
82        n = priExpr();
83      }
84      else
85        n = priExpr();
86      return (n);
87    }
88
89    static int priExpr() {
90      int n = 0;
```

```
91        switch (lookahead.kind) {
92        case LPAREN:
93          match(LPAREN);
94          n = addExpr();
95          match(RPAREN);
96          break;
97        case IDENTIFIER:
98          match(IDENTIFIER);
99          n = 1;
100         break;
101       case NUMBER:
102         match(NUMBER);
103         n = 0;
104         break;
105       default:
106         unexpectedToken(lookahead);
107       }
108       return (n);
109     }
110
111     static void match(int tokenKind) {
112       if (lookahead.kind == tokenKind)
113         lookahead = scanner.getNextToken();
114       else
115         unexpectedToken(lookahead);
116     }
117
118     static void unexpectedToken(Token lookahead) {
119       System.out.println("> at line " + lookahead.beginLine +
120                          ", column " + lookahead.beginColumn);
121       System.out.println("> unexpected token : " +
122                          tokenImage[lookahead.kind]);
123       System.exit(1);
124     }
125   }
```

### 3.5.2 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 3.40 [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`

```
 1  import java.util.Hashtable;
 2
 3  class Entry {
 4    String lexem;
 5    int  type;
 6    Entry( String lexem, int  type) {
 7      this.lexem = lexem;
 8      this.type = type;
 9     }
10  }
11
12  class Parser implements ScannerConstants {
13
14    static Token  lookahead;
15    static Scanner scanner;
16    static Hashtable symbolTable = new Hashtable();
17
18    public static void main (String args[]) {
19      scanner = new Scanner(System.in);
20      lookahead = scanner.getNextToken();
21      declaration();
22      System.out.println("> parsing succesfull");
23    }
24
25    static void declaration() {
26      int t = type();
27      while (lookahead.kind != EOF) {
28        idlist(t);
29      }
```

```
30    }
31
32    static int type() {
33      int t = -1;
34      switch(lookahead.kind) {
35      case INT:
36        match(INT);
37        t = 0;
38        break;
39      case REAL:
40        match(REAL);
41        t = 1;
42        break;
43      default:
44        unexpectedToken(lookahead);
45      }
46      return(t);
47    }
48
49    static void idlist(int in) {
50      message(lookahead,in);
51      match(IDENTIFIER);
52      while (lookahead.kind == COMMA) {
53        match(COMMA);
54        message(lookahead,in);
55        match(IDENTIFIER);
56      }
57      match(SEMICOLON);
58    }
59
60    static void message (Token lookahead, int in) {
61      if (lookahead.kind == IDENTIFIER) {
62        System.out.print("> new ");
63        switch (in) {
64        case 0:
65          System.out.print("int");
66          break;
67        case 1:
68          System.out.print("real");
69          break;
70        default:
71          System.out.print("unknown");
72          break;
73        }
74        System.out.println(" variable " +
75                           lookahead.image);
76        symbolTable.put(lookahead.image,
77                        new Entry(lookahead.image,in));
78      }
79    }
```

```
80
81      static void unexpectedToken(Token lookahead) {
82        System.out.println("> at line " + lookahead.beginLine +
83                           ", column " + lookahead.beginColumn);
84        System.out.println("> unexpected token : " +
85                           tokenImage[lookahead.kind]);
86        System.exit(1);
87      }
88
89      static void match(int tokenKind) {
90        if (lookahead.kind == tokenKind)
91          lookahead = scanner.getNextToken();
92        else
93          unexpectedToken(lookahead);
94      }
95    }
```

Typisch ist auch die Tiefe der Knoten des Ableitungsbaumes. Siehe z.B. die Aufgabe 3.12.

### 3.5.3 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 [App98]).

**Definition 3.24 [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 3.25 [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 3.41 [Abstrakter Ableitungsbaum]** Für die Grammatik aus Beispiel 3.19, hat die zugehörige abstrakte Syntax folgende Gestalt:

```
statl      ::=   ( stat )*
               ;
```

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

Das Kompositum [GHJV95], [GHJV96] 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`

```
1  abstract class Node {
2    abstract Object accept (Visitor v, Object o);
3  }
```

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`

```
1  import java.util.Vector;
2
3  class Statl extends Node {
4
5    Vector v;
6
7    Statl () {
8      v = new Vector();
9    }
10
11   public void addElement (Node stat) {
12     v.addElement(stat);
13   }
14
15   Object accept (Visitor v, Object o) {
16     return (v.visitStatl (this,o));
17   }
18 }
```

JAVA-Klasse `Assign`

```
1   class Assign extends Node {
2
3     Node id, addExpr;
4
5     Assign (Node id, Node addExpr) {
6       this.id = id;
7       this.addExpr = addExpr;
8     }
9
10    Object accept (Visitor v, Object o) {
11      return (v.visitAssign (this,o));
12    }
13  }
```

JAVA-Klasse Print

```
1   class Print extends Node {
2
3     Node addExpr;
4
5     Print (Node addExpr) {
6       this.addExpr = addExpr;
7     }
8
9     Object accept (Visitor v, Object o) {
10      return (v.visitPrint (this,o));
11    }
12  }
```

JAVA-Klasse Plus

```
1   class Plus extends Node {
2
3     Node mulExpr, addExpr;
4
5     Plus (Node mulExpr, Node addExpr) {
6       this.mulExpr = mulExpr;
7       this.addExpr = addExpr;
8     }
9
10    Object accept (Visitor v, Object o) {
11      return (v.visitPlus (this,o));
12    }
13  }
```

JAVA-Klasse Minus

```
1   class Minus extends Node {
2
3     Node mulExpr, addExpr;
```

```
 4
 5    Minus (Node mulExpr, Node addExpr) {
 6      this.mulExpr = mulExpr;
 7      this.addExpr = addExpr;
 8    }
 9
10    Object accept (Visitor v, Object o) {
11      return (v.visitMinus (this,o));
12    }
13  }
```

JAVA-Klasse Times

```
 1  class Times extends Node {
 2
 3    Node unaExpr, mulExpr;
 4
 5    Times (Node unaExpr, Node mulExpr) {
 6      this.unaExpr = unaExpr;
 7      this.mulExpr = mulExpr;
 8    }
 9
10    Object accept (Visitor v, Object o) {
11      return (v.visitTimes (this,o));
12    }
13  }
```

JAVA-Klasse Div

```
 1  class Div extends Node {
 2
 3    Node unaExpr, mulExpr;
 4
 5    Div (Node unaExpr, Node mulExpr) {
 6      this.unaExpr = unaExpr;
 7      this.mulExpr = mulExpr;
 8    }
 9
10    Object accept (Visitor v, Object o) {
11      return (v.visitDiv (this,o));
12    }
13  }
```

JAVA-Klasse Uminus

```
 1  class Uminus extends Node {
 2
 3    Node priExpr;
 4
 5    Uminus (Node priExpr) {
```

```
 6       this.priExpr = priExpr;
 7     }
 8
 9     Object accept (Visitor v, Object o) {
10       return (v.visitUminus (this,o));
11     }
12   }
```

JAVA-Klasse Identifier

```
 1   class Identifier extends Node {
 2
 3     String lexem;
 4
 5     Identifier (String lexem) {
 6       this.lexem = lexem;
 7     }
 8
 9     Object accept (Visitor v, Object o) {
10       return (v.visitIdentifier (this,o));
11     }
12   }
```

JAVA-Klasse Number

```
 1   class Number extends Node {
 2
 3     String n;
 4
 5     Number (String n) {
 6       this.n = n;
 7     }
 8
 9     Object accept (Visitor v, Object o) {
10       return (v.visitNumber (this,o));
11     }
12   }
```

JAVA-Klasse Parser

```
 1   import java.io.*;
 2
 3   class Parser implements ScannerConstants {
 4
 5     static Token  lookahead;
 6     static Scanner scanner;
 7
 8     public static void main (String args[])
 9         throws java.io.IOException{
10       scanner = new Scanner(System.in);
```

```
11      lookahead = scanner.getNextToken();
12      Node node = program();
13      System.out.println("> parsing succesfull");
14      node.accept(new PrintVisitor(),"");
15      node.accept(new EvalVisitor(),null);
16      /*
17        node.accept(new PostfixVisitor(),null);
18        PrintStream out =
19          new PrintStream(new FileOutputStream(new File("a.j")));
20        node.accept(new GenVisitor(out),null);
21      */
22    }
23
24    static Node program() {
25      Node op = statl();
26      match(EOF);
27      return (op);
28    }
29
30    static Node statl() {
31      Statl n = new Statl();
32      while (lookahead.kind != EOF)
33        n.addElement(stat());
34      return (n);
35    }
36
37    static Node stat() {
38      Node id, a;
39      switch(lookahead.kind) {
40      case IDENTIFIER:
41        id = new Identifier(lookahead.image);
42        match(IDENTIFIER);
43        match(ASSIGN);
44        a = addExpr();
45        match(SEMICOLON);
46        return (new Assign(id, a));
47      case PRINT :
48        match(PRINT);
49        match(LPAREN);
50        a = addExpr();
51        match(RPAREN);
52        match(SEMICOLON);
53        return (new Print(a));
54      default:
55        unexpectedToken(lookahead);
56        return(null);
57      }
58    }
59
60    static Node addExpr() {
```

```
61      Node m, a;
62      m = mulExpr();
63      switch(lookahead.kind) {
64      case PLUS:
65        match(PLUS);
66        a = addExpr();
67        return new Plus(m, a);;
68      case MINUS:
69        match(MINUS);
70        a = addExpr();
71        return (new Minus(m, a));
72      }
73      return (m);
74    }
75
76    static Node mulExpr() {
77      Node u, m;
78      u = unaExpr();
79      switch(lookahead.kind) {
80      case STAR:
81        match(STAR);
82        m = mulExpr();
83        return (new Times(u, m));
84      case SLASH:
85        match(SLASH);
86        m = mulExpr();
87        return (new Div(u, m));
88      }
89      return (u);
90    }
91
92    static Node unaExpr() {
93      if (lookahead.kind == MINUS) {
94        match(MINUS);
95        return (new Uminus(priExpr()));
96      }
97      else
98        return (priExpr());
99    }
100
101   static Node priExpr() {
102     Node n;
103     switch (lookahead.kind) {
104     case LPAREN:
105       match(LPAREN);
106       n = addExpr();
107       match(RPAREN);
108       break;
109     case IDENTIFIER:
110       n = new Identifier(lookahead.image);
```

```
111         match(IDENTIFIER);
112         break;
113       case NUMBER:
114         n = new Number(lookahead.image);
115         match(NUMBER);
116         break;
117       default:
118         n = null;
119         unexpectedToken(lookahead);
120       }
121       return (n);
122     }
123
124     static void match(int tokenKind) {
125       if (lookahead.kind == tokenKind)
126         lookahead = scanner.getNextToken();
127       else
128         unexpectedToken(lookahead);
129     }
130
131     static void unexpectedToken(Token lookahead) {
132       System.out.println("> at line " +
133                          lookahead.beginLine +
134                          ", column " +
135                          lookahead.beginColumn);
136       System.out.println("> unexpected token : " +
137                          tokenImage[lookahead.kind]);
138       System.exit(1);
139     }
140   }
```

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

JAVA-Klasse `Visitor`

```
1   abstract class Visitor {
2     abstract Object visitStatl(Statl statl, Object o);
3     abstract Object visitAssign(Assign assign, Object o);
4     abstract Object visitPrint(Print print, Object o);
5     abstract Object visitPlus(Plus plus, Object o);
6     abstract Object visitMinus(Minus minus, Object o);
7     abstract Object visitTimes(Times times, Object o);
8     abstract Object visitDiv(Div div, Object o);
9     abstract Object visitUminus(Uminus uminus, Object o);
10    abstract Object visitIdentifier(Identifier identifier,
11                                    Object o);
12    abstract Object visitNumber(Number number, Object o);
13  }
```

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 Pre-

order 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`

```
 1  import java.util.Vector;
 2
 3  class PrintVisitor extends Visitor {
 4
 5    Object visitStatl(Statl statl, Object o) {
 6      String indent = (String) o);
 7      System.out.println(indent + "Statl");
 8      for (int j = 0; j < statl.v.size(); j++)
 9        ((Node)
10         (statl.v.elementAt(j))).accept(this,indent + " ");
11      return(null);
12    }
13
14    Object visitAssign(Assign assign, Object o) {
15      String indent = (String) o);
16      System.out.println(indent + "Assign");
17      assign.id.accept(this,indent + "  ");
18      assign.addExpr.accept(this,indent + "  ");
19      return(null);
20    }
21
22    Object visitPrint(Print print, Object o) {
23      String indent = (String) o);
24      System.out.println(indent + "Print");
25      print.addExpr.accept(this,indent + "  ");
26      return(null);
27    }
28
29    Object visitPlus(Plus plus, Object o) {
30      String indent = (String) o);
31      System.out.println(indent + "Plus");
32      plus.mulExpr.accept(this,indent + "  ");
33      plus.addExpr.accept(this,indent + "  ");
34      return(null);
35    }
36
37    Object visitMinus(Minus minus, Object o) {
38      String indent = (String) o);
39      System.out.println(indent + "Minus");
40      minus.mulExpr.accept(this,indent + "  ");
41      minus.addExpr.accept(this,indent + "  ");
42      return(null);
43    }
```

```
44
45    Object visitTimes(Times times, Object o) {
46      String indent = (String) o);
47      System.out.println(indent + "Times");
48      times.unaExpr.accept(this,indent + "  ");
49      times.mulExpr.accept(this,indent + "  ");
50      return(null);
51    }
52
53    Object visitDiv(Div div, Object o) {
54      String indent = (String) o);
55      System.out.println(indent + "Div");
56      div.unaExpr.accept(this,indent + "  ");
57      div.mulExpr.accept(this,indent + "  ");
58      return(null);
59    }
60
61    Object visitUminus(Uminus uminus, Object o) {
62      String indent = (String) o);
63      System.out.println(indent + "Uminus");
64      uminus.priExpr.accept(this,indent + "  ");
65      return(null);
66    }
67
68    Object visitIdentifier(Identifier id, Object o) {
69      String indent = (String) o);
70      System.out.println(indent + "Identifier " + id.lexem);
71      return(null);
72    }
73
74    Object visitNumber(Number number, Object o) {
75      String indent = (String) o);
76      System.out.println(indent + "Number " + number.n);
77      return(null);
78    }
79  }
```

### 3.5.4 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

```
1  import java.util.Hashtable;
```

```
 2
 3  class EvalVisitor extends Visitor {
 4
 5    private Hashtable symbolTable = new Hashtable();
 6
 7    Object visitStatl(Statl statl, Object o) {
 8      for (int j = 0; j < statl.v.size(); j++)
 9        ((Node) (statl.v.elementAt(j))).accept(this,null);
10      return (null);
11    }
12
13    Object visitAssign(Assign assign, Object o) {
14      Identifier id = (Identifier) assign.id;
15      Integer value = (Integer) assign.addExpr.accept(this,null);
16      if (symbolTable.get(id.lexem) == null)
17        symbolTable.put(id.lexem,
18                        new Entry(id.lexem,value.intValue()));
19      else {
20        Entry e = (Entry) symbolTable.get(id.lexem);
21        e.value =  value.intValue();
22      }
23      return (null);
24    }
25
26    Object visitPrint(Print print, Object o) {
27      Integer value = (Integer) print.addExpr.accept(this,null);
28      System.out.println(" > " + value.toString());
29      return (null);
30    }
31
32    Object visitPlus(Plus plus, Object o) {
33      int i1 =
34        ((Integer) plus.mulExpr.accept(this,null)).intValue();
35      int i2 =
36        ((Integer) plus.addExpr.accept(this,null)).intValue();
37      return (new Integer(i1 + i2));
38    }
39
40    Object visitMinus(Minus minus, Object o) {
41      int i1 =
42        ((Integer) minus.mulExpr.accept(this,null)).intValue();
43      int i2 =
44        ((Integer) minus.addExpr.accept(this,null)).intValue();
45      return (new Integer(i1 - i2));
46    }
47
48    Object visitTimes(Times times, Object o) {
49      int i1 =
50        ((Integer) times.unaExpr.accept(this,null)).intValue();
51      int i2 =
```

```
52        ((Integer) times.mulExpr.accept(this,null)).intValue();
53      return (new Integer(i1 * i2));
54    }
55
56    Object visitDiv(Div div, Object o) {
57      int i1 =
58        ((Integer) div.unaExpr.accept(this,null)).intValue();
59      int i2 =
60        ((Integer) div.mulExpr.accept(this,null)).intValue();
61      return (new Integer(i1 / i2));
62    }
63
64    Object visitUminus(Uminus uminus, Object o) {
65      int i =
66        ((Integer) uminus.priExpr.accept(this,null)).intValue();
67      return (new Integer(-i));
68    }
69
70    Object visitIdentifier(Identifier id, Object o) {
71      int value = 0;
72      if (symbolTable.get(id.lexem) == null) {
73        System.out.println(" > identifier \"" + id.lexem +
74                           "\" not declared");
75        System.exit(1);
76      }
77      else {
78        Entry e = (Entry) symbolTable.get(id.lexem);
79        value = e.value;
80      }
81      return (new Integer(value));
82    }
83
84    Object visitNumber(Number number, Object o) {
85      return (new Integer(number.n));
86    }
87  }
```

### 3.5.5 Aufgaben

**Aufgabe 3.22 [Tracing]** *Erweitern Sie die Grammatik aus Beispiel 3.19 mit einem Attribut* `indent`*. Mit Hilfe von* `indent` *soll eine formatierte Ausgabe des Ableitungabaumes möglich sein.*

**Aufgabe 3.23 [Taschenrechner 1]** *Vereinfachen Sie die Grammatik aus Beispiel 3.19 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 entsprechen-*

*des* JAVA *programm.*

**Aufgabe 3.24 [Taschenrechner 2]** *Der Taschenrechner aus Aufgabe 3.23 liefert* falsche *Resultate. Warum? Was kann man dagegen unternehmen?*

**Aufgabe 3.25 [Taschenrechner 3]** *Erweitern Sie den Taschenrechner aus Aufgaben 3.23 und 3.23 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.*

**Aufgabe 3.26 [Taschenrechner 4]** *Erweitern Sie den Taschenrechner aus Aufgabe 3.25 so, dass die Applikation bei einer Division durch `0` das Resultat `0` ergibt.*

**Aufgabe 3.27 [Taschenrechner 5]** *Erweitern Sie den Taschenrechner aus Aufgabe 3.26 mit der Exponential- und der Logarithmus-Funktion (Dies bedeutet eine entsprechende Erweiterung der Grammatik).*

**Aufgabe 3.28 [Taschenrechner 6]** *Der Taschenrechner aus Beispiel 3.41 rechnet von rechts nach links. Modifizieren sie die* JAVA*-Klasse `Parser` so, dass der Taschenrechner wie üblich von links nach rechts rechnet.*

**Aufgabe 3.29 [Taschenrechner 7]** *Erweitern Sie den Taschenrechner aus Beispiel 3.41 mit einem Postfix-Visitor. Dabei soll jede Anweisung (`stat`) in Postfix-Notation ausgegeben werden.*

**Aufgabe 3.30 [XML]** *In der Aufgabe 3.12 wurde mit Hilfe eines Parsers den Ableitungsbaum für den Taschenrechner aus Beispiel 3.19 in* XML *Format herausgegeben. In dieser Aufgabe geht es nun darum, die erzeugte* XML *Datei ins ursprüngliche Eingabeformat des Taschenrechners zurückzutransformieren, und zu überprüfen, ob die* XML *Datei wohlgeformt ist. Sie finden eine Kurze Einführung in* XML *am Anfang des zweiten Kapitels des* XML *Skripts (`https://staff.hti.bfh.ch/blj2/sc/xml.html`).*

***Bemerkung 3.8 [Fixpunkt]*** *In dem Sie eine Taschenrechner Eingabe nach* XML *und anschliessend zurück nach ASCII transformieren, können Sie sich von der richtigkeit ihrer Lösung überzeugen: Die Rücktransformierte Datei muss mit der ursprünglichen Datei bis auf Leerschläge übereinstimmen. Man spricht hier von einem **Fixpunkt**. Ihre Applikation sollte auch einen Fixpunkt in die andere Richtung (von* XML *nach ASCII nach* XML*) besitzen.*

## 3.6 Code generierung (Vorschau)

In diesem Abschnitt wollen wir noch kurz zeigen, wie Code erzeugt werden kann. Das bearbeitete Beispiel basiert auf dem Parser von Beispiel 3.19.

Aus Portabilitätsgründen, wird JAVA-JASMIN-Assembler [MD97] Code erzeugt [6]

[6] JASMIN kann an der Internetdresse `http://jasmin.sourceforge.net` geladen werden.

### 3.6.1 JAVA Bytecode

Der Erzeugte Code hat viel ähnlichkeiten mit der Postfix Notation [vL90]. 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 3.42 [Code Generierung]** Code Generierung für das Programm

```
x = 2;
y = 3;
z = x * y + 2;
print(z*z);
print(1);
```

Jeder einfachen Zuweisung (e.g. `x = 2;`) entspricht einer Push- und eine Storeoperation.

JASMIN-Programm `T0`

```
1  bipush 2
2  istore_0                    ; x = 2;
3  bipush 3
4  istore_1                    ; y = 3;
```

```
 5  iload_0
 6  iload_1
 7  imul
 8  bipush 2
 9  iadd
10  istore_2                        ; z = x * y + 2;
11  iload_2
12  iload_2
13  imul
14  invokestatic MCCLib/puti(I)V
15  sipush 10
16  invokestatic MCCLib/putc(I)V    ; print (z * z);
17  bipush 1
18  invokestatic MCCLib/puti(I)V
19  sipush 10
20  invokestatic MCCLib/putc(I)V    ; print (1);
21  return
```

### 3.6.2 Semantische Analyse

Für die Grammatik aus Beispiel 3.19 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 3.43 [Code Generierung]** Für die Grammatik aus Beispiel 3.19 hat ein Programmgerüst (Codeerzeugungsbesucher) folgende Gestalt:

JAVA-Klasse `GenVisitor`

```
 1  import java.util.Hashtable;
 2  import java.io.*;
 3
 4  class GenVisitor extends Visitor {
 5
 6    private Hashtable symbolTable = new Hashtable();
 7
 8    private static int maxStack  = 0;
 9    private static int actStack  = 0;
10    private static int address = 0;
11
12    private PrintStream out;
13
14    GenVisitor (PrintStream out) {
15      this.out = out;
16    }
17
18    private void emit (String s) {
19        out.print(s + "\n");
20    }
21
```

```
22    private static void adjustStack (int i) {
23      actStack += i;
24      if (actStack > maxStack) maxStack = actStack;
25    }
26
27    Object visitStatl(Statl statl, Object o) {
28
29      /* output jasmin header */
30      emit(".class public a");
31      emit(".super A");
32      emit(".method public <init>()V");
33      emit("        aload_0");
34      emit("        invokespecial A/<init>()V");
35      emit("        return");
36      emit(".end method\n");
37      emit(".method public exec()V");
38
39      for (int j = 0; j < statl.v.size(); j++)
40        ((Node) (statl.v.elementAt(j))).accept(this,null);
41
42      /* output jasmin footer */
43      emit("      return");
44      emit("   .limit locals " + String.valueOf(address+1));
45      emit("   .limit stack " + String.valueOf(maxStack));
46      emit(".end method");
47
48      /* translate and execute code */
49      Exec e = new Exec();
50      e.exec();
51
52      return (null);
53    }
54
55    Object visitAssign(Assign assign, Object o) {
56      Identifier id = (Identifier) assign.id;
57      int a;
58      if (symbolTable.get(id.lexem) == null) {
59        a = address++;
60        symbolTable.put(id.lexem,
61                        new Entry(id.lexem,a));
62      }
63      else {
64        Entry e = (Entry) symbolTable.get(id.lexem);
65        a = e.value;
66      }
67      assign.addExpr.accept(this,null);
68      emit("      istore " + a + " ; " + id.lexem);
69      adjustStack(-1);
70      return (null);
71    }
```

```
72
73    Object visitPrint(Print print, Object o) {
74      print.addExpr.accept(this,null);
75      emit("      invokestatic MCCLib/puti(I)V");
76      emit("      sipush 10");
77      emit("      invokestatic MCCLib/putc(I)V");
78      adjustStack(-1);
79      return (null);
80    }
81
82    Object visitPlus(Plus plus, Object o) {
83      plus.mulExpr.accept(this,null);
84      plus.addExpr.accept(this,null);
85      emit("      iadd");
86      adjustStack(-1);
87      return (null);
88    }
89
90    Object visitMinus(Minus minus, Object o) {
91      minus.mulExpr.accept(this,null);
92      minus.addExpr.accept(this,null);
93      emit("      isub");
94      adjustStack(-1);
95      return (null);
96    }
97
98    Object visitTimes(Times times, Object o) {
99      times.unaExpr.accept(this,null);
100     times.mulExpr.accept(this,null);
101     emit("      imul");
102     adjustStack(-1);
103     return (null);
104   }
105
106   Object visitDiv(Div div, Object o) {
107     div.unaExpr.accept(this,null);
108     div.mulExpr.accept(this,null);
109     emit("      idiv");
110     adjustStack(-1);
111     return (null);
112   }
113
114   Object visitUminus(Uminus uminus, Object o) {
115     uminus.priExpr.accept(this,null);
116     emit("      ineg");
117     adjustStack(-1);
118     return (null);
119   }
120
121   Object visitIdentifier(Identifier identifier, Object o) {
```

```
122       int a = 0;
123       if (symbolTable.get(identifier.lexem) == null) {
124         System.out.println(" > identifier \"" +
125                            identifier.lexem +
126                            "\" not declared");
127         System.exit(1);
128       }
129       else {
130         Entry e = (Entry) symbolTable.get(identifier.lexem);
131         a = e.value;
132           emit("      iload " + a + " ; " + identifier.lexem);
133           adjustStack(1);
134       }
135       return (null);
136     }
137
138     Object visitNumber(Number number, Object o) {
139       emit("      sipush " + number.n);
140       adjustStack(1);
141       return (null);
142     }
143   }
```

**Aufgabe 3.31 [Codeerzeugung]** *Betten Sie den Codegenerator aus Beispiel 3.43 direkt in einem Top-Down Parser mit rekursiven Abstieg ein.*

## 3.7 Projekt

### 3.7.1 Scanner Generator

Im Kapitel 2 haben wir kennengelernt, wie ein `RegularExpression` Objekt in einen NEA übersetzt werden kann. Im Beispiel 3.20 wurde eine Parser-Skizze zur übersetzung eines Strings in einen regulären Ausdruck vorgestellt. Wir sind also nun in der Lage, einen kleinen Scanner Generator à la JAVACC (ohne lexikalische Aktionen) zu implementieren.

Im wesentlichen müssen wir dabei zwei Probleme Lösen:

1. Erkennen verschiedener Tokens
2. Code-Generierung

#### Erkennen verschiedener Tokens

Nehmen wir an, dass wir $n$ reguläre Ausdrücke $r_1, .., r_m$ erkennen wollen. Jedem regulären Ausdruck $r_i$ entspricht einem Token. Um alle reguläre Ausdrücke zu erkennen könnten wir einen neuen regulären Ausdruck $r = r_1|..|r_m$ bilden und anschliessend mit den vorhandenen Programme in einen NEA umwandeln. Dieses Vorgehen funktioniert leider nicht, da wir am Schluss nur wissen dass eine gegebene Eingabe akzeptiert wird. Das genügt nicht weil wir wissen müssen, zu welchem regulären Ausdruck die Eingabe passt.

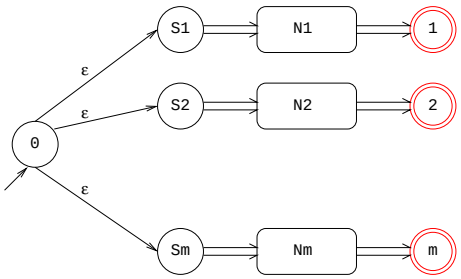



Abbildung 3-6: Erweiterung der Thompson Konstruktion zur Erkennung verschiedener regulären Ausdrücke

Wir erweitern also die Thompson Konstruktion mit dem Konstrukt aus Abbildung 3-6

Somit erreichen wir dass es für jeden regulären Ausdruck $r_i$ genau einen akzeptierenden NEA-Zustand $i$ gibt. Werden mehrere reguläre Ausdrücke akzeptiert, so wählen wir daraus den kleinsten Zustand und dieser entspricht genau einem regulären Ausdruck. So können wir die Arbeitsweise Regeln aus Abschnitt 2.4.2 erfüllen. Wir dürfen diesen Vorgang bei der Transformation des NEA's in einen DEA auch nicht vergessen.

Technisch werden wir unsere Struktur für reguläre Ausdrücke mit einer entsprechenden `ListExpression` Klasse erweitern. Wir müssen leider auch die Besucherklassen ad hoc modifizieren. Die Parser Klasse muss so erweitert werden, dass sie als Resultat ein `RegularExpression` Objekt zurückliefert.

#### Code Generierung

Die Code Erzeugung ist eine recht einfache angelegenheit. Wir können als Raster die DEA Klasse aus Abschnitt 2.3.5 übernehmen. Wir werden daraus nur die Methode `execute()` behalten (die `init()` Methode ist überflüssig). Die DEA-Tabelle `tabe`, der Array `accepting` können direkt aus unserem konstruierten DEA als Array-Konstanten eintragen.

Falls Sie schwierigkeiten haben geht es technisch am am einfachsten, zuerst eine Beispiel Klasse (von Hand) zu schreiben. Sie merken sich anschliessend, welche Teile dieser Klasse Problemunabhängig sind. Diese Teile können direkt generiert werden, oder aus einem Template kopiert werden. Die anderen Teile werden mit Hilfe des konstruierten DEA's erzeugt.

### 3.7.2 Parser Generator

Die Implementierung eines Parser-Generators ist eine interessante Aufgabe:

- Sie beinhaltet eine Repetition der ganzen Theorie, insbesondere ein gutes Verständnis der BNF-Notation.
- Die Code-Generierung ist relativ einfach weil die Zielsprache eine höhere Programmiersprache ist.
- Ist mal ein Parser-Generator vorhanden, dann gibt es viele interessante Erweiterungsmöglichkeiten wie Bootstraping, EBNF-Erweiterung, Scanner-Optimierung, Generierung eines abstrakten Syntax-Baumes, etc.

Wir wollen nun unseren Parser-Generator MPG (Mini Parser Generator) nennen. Die Notation von MPG soll auf der BNF-Notation des Skripts basieren. Wir wollen auch wie bei JAVACC einen Scanner und einen Parser spezifizieren können.

### MPG Spezifikation

Folgende Datei ist eine spezifikation der MPG Grammatik in MPG:

```
skip: "([ \t\n\r\f])*"
token: MARK "%%%%"
token: SKI "skip:"
token: TOK "token:"
token: SC ";"
token: IS "::="
token: OR "\|"
token: EMPTY "\"\""
token: NUM "[0-9]+"
token: TS "[A-Z][A-Z0-9]*"
token: NTS "[a-z][a-zA-Z0-9]*"
token: PATTERN "\"(\\.|[^\\\"])*\""

s

%%%

s              ::= mpg EOF
                 ;
mpg            ::= regExprList start MARK productionList
                 ;
regExprList    ::= regExpr regExprList | ""
                 ;
regExpr        ::= TOK TS PATTERN | SKI PATTERN
                 ;
start          ::= NTS
                 ;
productionList ::= production productionList | ""
                 ;
production     ::= NTS IS choice SC
                 ;
choice         ::= sequence or
                 ;
or             ::= OR sequence or | ""
                 ;
sequence       ::= symbol sequence | ""
                 ;
symbol         ::= NTS | TS | EMPTY
```

Die Spezifikation der Tokens erfolgt wie bei unserem Scanner-Generator. Anschliessend folgen die Angabe des Startsymbols, ein Marker und die Produktionen in BNF-Notation.

Terminalsymbole bestehen aus Grossbuchstaben gefolgt von Grossbuchstaben oder Ziffern. Nonterminalsymbole fangen mit einem Kleinbuchstaben, gefolgt von Buschstaben (gross oder klein) oder Ziffern.

### Semantik

Die Semantik von MPG basiert im Wesentlichen auf derjenigen derjenigen von JAVACC. Die Unterschiede zu JAVACC sehen wie folgt aus:

- MPG kennt nur BNF
- MPG kann nur LL(1)-Grammatiken verarbeiten
- Der generierte Code kann nur die Syntax überprüfen.
- Semantische Aktionen werden nicht unterstützt. Sie können aber ohne Problem im generierten Parser eingebaut werden.
- Für die Tokendefinitionen wird die Syntax von unserem Scanner-Generator verwendet.

### Aufbau

Um MPG zu entwerfen, müssen wir zuerst einen Parser für die Spezifikationsdatei entwerfen. Dies Aufgabe ist nicht sehr schwierig, da die Lösung im Wesentlichen in Abschnitt 3.7.2 angegeben ist (es bleibt nur noch die Bestimmung der First- und Follow-Mengen). Ist mal der Parser vorhanden, so müssen semantische Aktionen für das Erkennen der Regulären Ausdrücke, sowie für den Aufbau der Grammatik eingebaut werden. Für die regulären Ausdrücke können wir unsere Klasse `RegularExpressionParser` verwenden. Die Konstrultion der Produktionen ist einfach. Schwieriger wird das Testen der LL(1)-Eigenschaft der Grammatik.

Wenn diese beiden Probleme gelöst sind, können wir den Scanner mit unserer `TokenManager`-Klasse generieren. Für die Code-Generierung des Parsers können wir im wesentlichen den Code des Parsers für die MPG-Spezifikation als Beispiel-Targetcode verwenden. Der Aufbau der Parse Methoden ist im Abschnitt 3.2.4 angegeben. Da unser Parser nur die BNF-Notation verwendet ist der Vorgang der Codegenerierung für jede Produktion einfach. Gegeben sei z.B. folgende Produktion:

```
sequence ::= symbol sequence | "" ;
```

Der generierte Code sieht wir folgt aus:

```
void sequence() throws ParseException{
  switch (lookahead) {
    // first(sequence ::= symbol sequence;)
    case NTS:
    case TS:
    case EMPTY:
      symbol();
```

```
        sequence();
        break;
      // follow(sequence)
      case OR:
      case SC:
        break;
      default:
        throw new ParseException("unexpected token " +
          names[lookahead] + " was expecting {NTS,TS,EMPTY,OR,SC}");
    }
  }
```

### 3.7.3 Aufgaben

**Aufgabe 3.32 [Scannergenerator]** *Implementieren Sie ein Scannergenerator gemäss Angaben in Abschnitt 3.7.1. Verwenden Sie dabei folgendes Eingabeformat.*

```
token: AA "(a(a|b)*a|a)c"
skip: "x yz"
token: IF "if"
token: ID "(i|f)*"
```

*Sie finden unter* `prog/pr/SpecificationFileParser.java` *einen einfachen Parser für solche Eingabedateien. Ein solcher Parser ist einfach zu schreiben, es braucht dafür fast keine Kenntnisste der Compilerbau-Methoden.*

*Unter* `prog/pr/skeleton` *finden Sie ein Template für die Generierung der Scannr-Klasse. Die Markierung* `$$$$$$$$` *ist ein Platzhalter für die variablen Teile der generierten Scannerklasse.*

*Unter* `prog/pr/TokenManagerGenerator.java` *finden Sie schliesslich die Codegenerierung-Klasse.*

**Aufgabe 3.33 [Parsergenerator]** *Implementieren Sie ein Scannergenerator gemäss Angaben in Abschnitt 3.7.2.*

*Unter* `prog/mp/grammar.zip` *finden Sie alle Klassen zum Aufbau und überprüfung der Grammatik (inklusive LL(1)-Test).*

*Der generierte Code für die* MPG*-Spezifikation finden Sie unter* `prog/mp/MPGParser.java`*.*

*Die Klasse* `MPGParser` *ist nur dann lauffähig, wenn ein entsprechender Scanner vorhanden ist. Diesen Scanner kann mit Hilfe der Lösung der Aufgabe 3.32 erzeugt werden. Als Eingabe wird das erste Teil der* MPG*-Spezifikation verwendet.*

*Wird diese Spezifikation als Testdatei für den Parsergenerator verwendet, so wird derselbe Scanner nochmals erzeugt!*

# Kapitel 4

# Code Generierung

## 4.1 Zwischencode

### 4.1.1 Zwischensprachen

Zwischensprachen sollten unabhängig von möglichen Zielmaschinen sein und in der Komplexität zwischen Quell- und Zielsprache liegen. Häufigverwendet werden **Postfix-Code**, **Syntaxbäume** sowie **3-Adress-Code** .

#### Postfix-Code

Beispiel `a+b*c` → `abc*+`, `if a then b else c` → `abc?` (`?` steht für die ternäre if-then-else Operation). Die Auswertung von Postfix-Code kann bei Wertzuweisungen einfach mittels eines Stacks erfolgen. Bei der Codeerzeugung müssen also nur `push` und `pop` Befehle erzeugt werden. Die Codegenerierung für Kontrollanweisungen (`if`, `goto`) ist komplizierter.

#### Syntaxbäume

Ein **Syntaxbaum** ist ähnlich einem Ableitungsbaum, enthält aber keine Nonterminals. Die Codegenerierung basiert auf einer Traversierung des Syntaxbaumes.

#### 3-Adress-Code

Der **3-Adresscode** ist eine Folge von Anweisungen der allgemeinen Form:

```
[OP, arg1, arg2, res]
```

wobei `arg1`, `arg2` und `res` Namen, Konstanten oder vom Compiler generierte temporäre Werte sind; `OP` steht für irgendeinen Operator. 3-Adresscode Befehle sind einem Prozessor zugeschnitten, der über beliebig viele Register verfügt.

Typen von Operationen:

- Wertzuweisungen mit binärem Operator: `[OP,arg1,arg2,res]`, wobei `OP` eine arithmetische oder boolsche Operation, `arg1`, `arg2` und `res` Identifier des Quellprogramms oder Names von Hilfsvariables, die vom Compiler erzeugt wurden.
- Wertzuweisungen mit unärem Operator: `[OP,arg1,null,res]`, wobei `OP` unäres `"-"`, `"!"`, Shift um feste Anzahl Stellen, Typenkonversion, Identität. Die zweite Adresse wird nicht verwendet.
- Unbedingte Sprünge: `[GOTO,null,null,l]`, wobei `l` die Zeile `l` in der Liste der 3-Adress-Code Befehle.
- Bedingte Sprünge: `[IF<RELOP>,arg1,arg2,l]`, wobei `<RELOP>` ein Vergleichsoperator.
- Prozeduraufrufe: `[PARAM,arg1,null,null]`, `[CALL,arg1,arg2,null]` und `[RETURN,null,nul`
- Andere Operationen, z.B. Operationen auf Arrays, Pointerbefehle.

Die Implementierung von Programmen in 3-Adress-Code kann einfach mithilfe einer **Quadrupeltabelle** erfolgen. Eine solche Tabelle besitzt eine Zeile pro Befehl und vier Spalten `OP`, `arg1`, `arg2` und `res` zur Angabe des Operators, der Beiden Argumente und des Resultats. Der Inhalt von `arg1`, `arg2` und `res` ist i.a. ein Zeiger in die Symboltabelle.

**Beispiel 4.1 [3-Adress Code]**

Die übersetzung von `a := -b * (c + d)` führt zu:

```
[INEG,  b,  null, t1]
[IADD,  c,  d,    t2]
[IMUL,  t1, t2,   t3]
[ICOPY, t3, null,  a]
```

Bzw. mit Optimierung:

```
[INEG, b,  null, t1]
[IADD, c,  d,    t2]
[IMUL, t1, t2,    a]
```

### Zwischencodeerzeugung bei Wertzuweisungen

Wir definieren hier die möglichen semantischen Aktionen

**Attribute**:

`e.place` ist der Name einer Hilfsvariable zum Abspeichern des Zwischenergebnisses, das vorliegt, nachden der zu `e` gehörige Zwischencode ausgeführt worden ist.

`<ID>.place` Name der Speicherzelle, in welcher sich der Wert des Identifiers `<ID>` befindet. In den folgen Beispielen ist `<ID>.place` immer identisch mit `<ID>`.

**Prozeduren**:

`newtemp()` Generiert neue Namen für Hilfsvariablen.

`emit(op,arg1,arg2,res)` trägt den Quadrupel, welche `[op,arg1,arg2,res]` repräsentieren, in die Quadrupeltabelle ein.

Bei jedem Reduktionsschritt des Parsers werden die zur Produktion gehörigen semantischen Aktionen ausgeführt.

**Grammatik**:

```
a ::= <ID> "=" e ;
e ::= e "+" e ;
e ::= e "*" e ;
e ::= "-" e ;
e ::= "(" e ")" ;
e ::= <ID> ;
```

**Attributgrammatik**:

```
a   ::= <ID> "=" e:e
        {
           emit(ICOPY, e.place, null, <ID>);
        }
     ;
e.e ::= e:e1 "+" e:e2
        {
           t = newtemp();
           e.place = t;
           emit(IADD, e1.place, e2.place, t);
        }
     ;
e.e ::= e:e1 "*" e:e2
        {
           t = newtemp();
           e.place = t;
           emit(IMUL, e1.place, e2.place, t);
        }
     ;
e.e ::= "-" e:e1
        {
           t = newtemp();
           e.place = t;
           emit(INEG, e1.place, null, t);
        }
     ;
e.e ::= "(" e:e1 ")"
        {
           e.place = e1.place;
        }
     ;
e.e ::= <ID>
        {
           e.place = <ID>.place;
        }
     ;
```

Bei realen Programmiersprachen sind im Zusammenhang mit Wertzuweisungen auch Zwischencodebefehle zur **Typenkonversion** zu erzeugen. Wir führen ein neues Attribute `e.mode` ein, welches den Typ des durch `e` repräsentierten Zwischenergebnisses angibt. Semantische Aktionen aus vorherigen Beispiels werden wie folgt erweitert:

**Attributgrammatik**:

```
e.e ::= e:e1 <OP> e:e2
        {
           if (e1.mode == INT && e2.mode == INT) {
             emit(I<OP>, e1.place, e2.place, t);
             e.mode = INT
           }
           else if (e1.mode == FLOAT && e2.mode == FLOAT) {
             emit(f<OP>, e1.place, e2.place, t);
             e.mode = FLOAT
           }
           else if (e1.mode == INT) {
             u = newtemp();
             emit(I2F, e1.place, null, u);
             emit(F<OP>, u, e2.place, t);
             e.mode = FLOAT
           }
           else
             u = newtemp();
             emit(I2F, e2.place, null, u);
             emit(F<OP>, e1.place, u, t);
             e.mode = FLOAT
           }
        }
     ;
```

### Zwischencodeerzeugung bei boolschen Ausdrücken (Variante 1)

Die Zwischencodegenerierung für logische Operationen ist völlig analog zu arithmetischen Operatoren. Z.B. `A || B && C` wird in `T1 := B && C; T2 := A || T1;` übersetzt.

Bei Relationaloperatoren werden jeweils Schablonen von mehreren Zwischencodeanweisungen generiert. Z.B. `A < B` wird in

```
(i)   [IF<,   A,    B,    i+3]
(i+1) [ICOPY, 0,    null, t  ]
(i+2) [GOTO,  null, null, i+4]
(i+3) [ICOPY, 1,    null, t  ]
(i+4) /* nächste Zwischencodeanweisung */
```

**Prozedur**:

`nextquad()` liefert die Nummer der nächsten freien Zeile in der Quadrupeltabelle.

**Grammatik**:

```
e ::= e <BOOLOP> e ;
e ::= e <RELOP> e ;
```

**Attributgrammatik**:

```
            {
              t = newtemp();
              e.place = t;
              emit(<BOOLOP>,e1.place,e2.place,t);
            }
          ;
e:e   ::= e:e1 <RELOP> e:e2
            {
              t = newtemp();
              e.place = t;
              emit(IF<RELOP>, e1.place, e2.place, nextquad() + 3);
              emit(ICOPY, 0, null, t);
              emit(GOTO, null, null nextquad() + 2);
              emit(ICOPY, 1, null, t);
            }
          ;
```

Es ist möglich, zusätzliche semantische Aktionen für eine Typenkonversion anzugeben, falls Operanden gemischten Typs verwendet werden dürfen. Bei Vergleichsoperationen dürfen beide Operanden sowohl vom Typ `boolean` als auch vom Typ `int` sein.

### Zwischencodeerzeugung bei boolschen Ausdrücken (Variante 2)

Die explizite Abspeicherung des Zwischenergebnisses bei der Auswertung eines boolschen (Teil-) Ausdrucks ist redundant , da der Wert implizit bereits durch die Nummer des Zwischencodebefehls, an welche der kontrollfluss gelangt, gegeben ist.

**Prozeduren**:

`makelist(i)` generiert die Menge `{i}`, wobei `i` die Nummer einer Zwischencodeanweisung ist. Die generierte Menge wird als Attribut `e.false` oder `e.true` verwendet.

`merge(set1,set2)` bildet die Mengentheoretische Vereinigung `set1` $\cup$ `set2`. Sowohl `set1` als auch `set2` enthält als Elemente Nummern von Zwischencodeanweisungen.

`backpatch(set,j)` Bei dieser Variante müssen oft Vorwärtssprünge mit noch unbekannten Sprungzielen generiert werden. Diese Sprungziele bleiben bei der Erzeugung der Sprungbefehle vorläufig offen. Die Prozedur `backpatch` trägt später offene Sprungziele in `goto`'s nach. Konkret: `set` ist eine Menge von Adressen von `goto`'s mit offenen Sprungzielen. `backpatch(set,j)` trägt die Sprungadresse `j` bei allen diesen `goto`'s ein.

**Terminals**:

Zusätzliches Dummy Nonterminal `m`

**Attribute**:

`m.quad`: Mithilfe des Attributs `m.quad` des Nonterminals `m` wird die Nummer des nächsten Zwischencodebefehls gemerkt, bevor die übersetzung des nächsten Teilausdrucks

beginnt. `m.quad` gibt also die Adresse an, wo der Zwischencode für diesen Teilausdruck beginnt. Das Nonterminal `m` hat keinen Einfluss auf die Syntax der Quellsprache, da es mittels m::=""; gelöscht wird.

`e.true` Eine Menge von Adressen von `goto`'s mit noch offenen Sprungziele. Eines dieser `goto`'s muss ausgeführt werden, wenn sich für den Teilausdruck, der durch `e` repräsentiert wird, `true` ergibt.

`e.false` Analog zu `e.true` für den fall, dass sich `false` für `e` ergibt.

**Grammatik**:

```
e ::= e "||" e ;
e ::= e "&&" e ;
e ::= "!" e ;
e ::= <ID> ;
e ::= e <RELOP> e ;
```

**Attributgrammatik**:

```
e:e ::= e:e1 "||" m:m e:e2
        {
           backpatch(e1.false,m.quad);
           e.true = merge(e1.true,e2.true);
           e.false = e2.false;
        }
     ;
e:e ::= e:e1 "&&" m:m e:e2
        {
           backpatch(e1.true,m.quad);
           e.true = e2.true;
           e.false = merge(e1.false,e2.false);
        }
     ;
e:e ::= "!" e:e1
        {
           e.true = e1.false;
           e.false = e1.true;
        }
     ;
e:e ::= <ID>
        {
           e.true = makelist(nextquad());
        }
     ;
e:e ::= <ID>
        {
           e.true = makelist(nextquad());
           e.false = makelist(nextquad()+1);
           emit(IF, <ID>, null, _);
           emit(GOTO, null, null, _);
        }
```

```
        ;
e:e ::= e:e1 <RELOP> e:e2
        {
           e.true = makelist(nextquad());
           e.false = makelist(nextquad()+1);
           emit(IF<RELOP>, e1.place, e2.place, _);
           emit(GOTO, null, null, _);
        }
        ;
m:m ::= ""
        {
           m.quad = nextquad();
        }
        ;
```

Erläuterungen:

**Produktion (1)** Ergibt sich für `e1` der Wert `false`, so muss der Code für die Ausführung von `e2` angesprungen werden. `m.quad` bezeichent die Adresse, wo dieser Code beginnt.

Die Menge der `goto`'s die für den Fall `e1 || e2 == true` ausgeführt werden ist die Vereinigung der beiden Mengen von `goto`'s, die im Fall `e1 == true`, bzw. `e2 == true` ausgeführt werden.

Die Menge der `goto`'s die für den Fall `e1 || e2 == false` ausgeführt werden, ist identisch mit der Menge der `goto`'s, die im Fall `e2 == false` ausgeführt werden (Denn `e1 == false` führt an den beginn von `e2`.)

**Produktion (2)** Analog zu (1)

**Produktion (3)** Die Menge der `goto`'s, die im Fall `e1 == false` ausgeführt werden, ist identisch mit der Menge der `goto`'s die im Fall `!e1 == true` ausgeführt werden.

**Produktion (4)** Identische übertragung der Attribute von `e1` auf `e`.

**Produktion (5)** Es werden zwei Sprünge generiert. Der erste in der Zeile `nextquad()` wird in dem Fall ausgeführt, dass `<ID>== true` ist. Sonst wird der zweite Sprung in der Zeile `nextquad() + 1` ausgeführt.

**Produktion (6)** Analog zu (5)

**Produktion (7)** Mit Hilfe des Attributs `m.quad` wird die Adresse des nächsten zu generierenden Befehls gemerkt. Dieser Befehl entspricht dem Anfang von `e2` in den Produktionen (1) und (2).

**Beispiel 4.2 [Backpatching]** Syntaxbaum:

```
(1) Node(e,<ID>,<,<ID>)  (p < q)       e.t={100}     e.f={101}
(2) Node(m)                            m.q=102
(3) Node(e,<ID>,<,<ID>)  (r < s)       e.t={102}     e.f={103}
(4) Node(m)                            m.q=104
(5) Node(e,<ID>,<,<ID>)  (t < u)       e.t={104}     e.f={105}
(6) Node(e,e,&&,m,e)     ((3) && (5))  e.t={104}     e.f={103,105}
(7) Node(e,e,||,m,e)     ((1) || (6))  e.t={100,104} e.f={103,105}
```

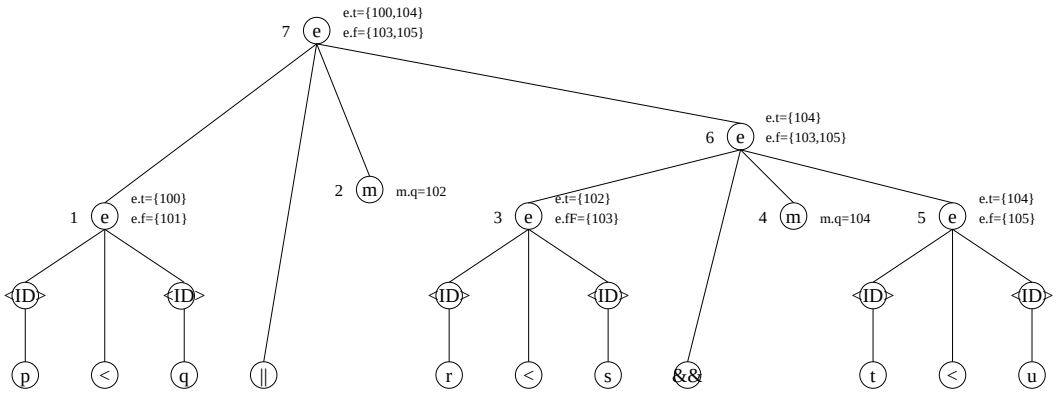



Abbildung 4-1: Syntaxbaum für `p <LT> q <OR> r <LT> s <AND> t <LT> u`

Zwischencode nach Generierung von Knoten (1)

```
(100) [IF<,    p,     q,     _  ]
(101) [GOTO,   null,  null,  _  ]
```

Zwischencode nach Generierung von Knoten (3)

```
(100) [IF<,    p,     q,     _  ]
(101) [GOTO,   null,  null,  _  ]
(102) [IF<,    r,     s,     _  ]
(103) [GOTO,   null,  null,  _  ]
```

Zwischencode nach Generierung von Knoten (5)

```
(100) [IF<,    p,     q,     _  ]
(101) [GOTO,   null,  null,  _  ]
(102) [IF<,    r,     s,     _  ]
(103) [GOTO,   null,  null,  _  ]
(104) [IF<,    t,     u,     _  ]
(105) [GOTO,   null,  null,  _  ]
```

Zwischencode nach Generierung von Knoten (6)

```
(100) [IF<,    p,     q,     _  ]
(101) [GOTO,   null,  null,  _  ]
(102) [IF<,    r,     s,     104]
(103) [GOTO,   null,  null,  _  ]
(104) [IF<,    t,     u,     _  ]
(105) [GOTO,   null,  null,  _  ]
```

Zwischencode nach Generierung von Knoten (7)

```
(100) [IF<,    p,     q,     _  ]
(101) [GOTO,   null,  null,  102]
(102) [IF<,    r,     s,     104]
(103) [GOTO,   null,  null,  _  ]
(104) [IF<,    t,     u,     _  ]
(105) [GOTO,   null,  null,  _  ]
```

### Zwischencodeerzeugung bei boolschen Ausdrücken (Variante 3)

Backpatching entspricht eigentlich der Verwendung von **Labels**. Das Ersetzen der Labels durch die entsprechende Zeilennummer nach der Erstellung der vollständigen Quadrupeltabelle ist nichts anderes als Backpatching

In diesem Fall müssen vererbte Attribute verwendet werden.

**Attribute**:

`e.true` Ist ein Label. Dorthin muss gesprungen werden, falls die Auswertung von `e` den Wert `true` ergibt.

`e.false` analog.

**Prozeduren**:

`newLabel()` generiert ein neues Label.

`setLabel()` setzt ein neues Label für das nächste Quadrupel.

**Attributgrammatik**:

```
e:e ::= {
           l = newLabel();
           e1.false = l;
           e1.true = e.true;
         }
         e:e1 "||"
         {
           setLabel(l);
           e2.false = e.false;
           e2.true = e.true;
         }
         e:e2
      ;
e:e ::= {
           l = newLabel();
           e1.false = e.false;
           e1.true = l;
         }
         e:e1 "&&"
         {
           setLabel(l);
           e2.false = e.false;
           e2.true = e.true;
           e2.true = e.true;
```

```
            }
            e:e2
         ;
    e:e ::= "!"
            {
               e1.true = e.false;
               e1.false = e.true;
            }
            e:e1
         ;
    e:e ::= <ID>
            {
               emit(IF, <ID>, null, e.true);
               emit(GOTO, null, null, e.false);
            }
         ;
    e:e ::= e:e1 <RELOP> e:e2
            {
               emit(IF<RELOP>, e1.place, e2.place, e.true);
               emit(GOTO, null, null, e.false);
            }
         ;
```

Dabei wird die Grammatik der Variante 2 verwendet.

**Zwischencodeerzeugung bei Kontrollstrukturen (Variante 1)**

`if a & b then s1 else s2` sollte etwa folgendes produzieren:

```
    (1)   [IF<,    a,     b,     3  ]
    (2)   [GOTO,   null,  null,  n+1]
    (3)   // code von s1


    (n-1) // code von s1
    (n)   [GOTO,   null,  null,  m  ]
    (n+1) // code von s2

    (m-1) //code von s2
    (m)   // code von der naechsten Anweisung
```

`e.true = {3}, e.false = {2}`

**Nonterminals:**

Zusätzliches Dummy Nonterminal `n`

Dieses Nonterminal wird im Zusammenhang mit Anweisungen `"if" e "then" s1 "else" s2` benötigt. nach der übersetzung von `s1` muss ein zusätzliches `goto` erzeugt werden zu dem Befehl, der statisch auf `s2` folgt. Die Erzeugung dieses `goto` wird von `n` bewirkt.

**Attribute:**

`l.next` Menge von Adressen von `goto`'s mit offenem Sprungziel. Der Sprung führt zur Anweisung, die Dynamisch auf `l` folgt. `l` ist eine Liste von Anweisungen.

`s.next` entspricht wie `l.next` einer einzelnen Anweisung (möglicherweise ein Block).

**Grammatik**:

```
s ::= "if" e "then" s ;
s ::= "if" e "then" s "else" s ;
s ::= "while" e "do" s;
s ::= "begin" l "end"
s ::= a ;
l ::= l ";" s ;
l ::= s ;
```

`s`: Statement, `l`: Statement List, `a`: Assignment Statement `e`: Boolean Expression.

**Attributgrammatik**:

```
s:s ::= "if" e:e "then" m:m s:s1
        {
           backpatch(e.true,m.quad);
           s.next = merge(e.false,s1.next);
        }
     ;
m:m ::= ""
        {
           m.quad = nextquad();
        }
     ;
s:s ::= "if" e:e "then" m:m1 s:s1 n:n "else" m:m2 s:s2
        {
           backpatch(e.true,m1.quad);
           backpatch(e.false,m2.quad);
           s.next = merge(s1.next,n.next,s2.next);
        }
     ;
n:n ::= ""
        {
           n.next = makelist(nextquad());
           emit(GOTO, null, null, _);
        }
     ;
s:s ::= "while" m:m1 e:e "do" m:m2 s:s1
        {
           backpatch(s1.next,m1.quad);
           backpatch(e.true,m2.quad);
           s.next = e.false;
           emit(GOTO, null, null, m1.quad);
        }
     ;
s:s ::= "begin" l:l "end"
```

```
        {
           s.next = l.next;
        }
     ;
s:s ::= a
        {
           s.next = makelist();
        }
     ;
l:l ::= l:l1 ";" m:m s:s
        {
           backpatch(l1.next,m.quad);
           l.next = s.next;
        }
     ;
l:l ::= s:s
        {
           l.next = s.next;
        }
     ;
```

**Produktion (3)** Die Nummer des ersten Quadrupels, welches zu `s` gehört, wird als Ziel in die offene Sprünge für den Fall `e == true` nachgetragen.

`s.next` enthält die Nummern aller Quadrupel, die einen offenen Sprung auf die dynamisch der `if`-Anweisung folgende Anweisung enthalten. Dies sind Sprünge aus `s1` und `s2` heraus sowie das durch `n` generierte `goto`.

**Produktion (5)** Erstes Quadrupel des Codes für `e` wird dynamischer Nachfolger von `s1`.

Erstes Quadrupel des Codes für `s1` wird dynamischer Nachfolger von `e` falls `e == false`.

Dynamischer Nachfolger der `while`-Anweisung wird dynamischer nachfolger von `e` falls `e == false`.

An das Ende des Codes für `s1` wird Sprung zum ersten Quadrupel von `e` angeführt.

### Zwischencodeerzeugung bei Kontrollstrukturen (Variante 2)

**Grammatik**:

```
s ::= "if" e "then" s ;
s ::= "if" e "then" s "else" s ;
s ::= "while" e "do" s;
s ::= "begin" l "end"
s ::= a ;
l ::= l ";" s ;
l ::= s ;
```

**Attributgrammatik**:

```
s:s ::= "if"
        {
          lTrue = newLabel();
          lFalse = newLabel();
          e.true = lTrue;
          e.false = lFalse;
        }
        e:e "then"
        {
           setLabel(lTrue);
        }
        s:s1
        {
           setLabel(lFalse);
        }
     ;
s:s ::= "if"
        {
          lTrue = newLabel();
          lFalse = newLabel();
          e.true = lTrue;
          e.false = lFalse;
        }
        e:e "then"
        {
           setLabel(lTrue);
        }
        s:s1 "else"
        {
           emit(GOTO, null, null, l);
           setLabel(lFalse);
        }
        s:s2
        {
           setLabel(l);
        }
     ;
s:s ::= "while"
        {
          lStart = newLabel();
          lTrue = newLabel();
          lFalse = newLabel();
          e.true = lTrue;
          e.false = lFalse;
          setLabel(lStart);
        }
        }
        e:e "do"
        {
           setLabel(lTrue);
```

```
        }
        s:s1
        {
           emit(GOTO, null, null, lStart);
           setLabel(lFalse);
        }
     ;
s:s ::= "begin" l:l "end"
     ;
s:s ::= a
     ;
l:l ::= l:l1 ";" m:m s:s
     ;
l:l ::= s:s
     ;
```

### Zwischencodeerzeugung bei Deklarationen

Bei Deklarationen von einfachen Typen wie `int`, `float` und `boolean` ist kein Zwischencode zu erzeugen, sondern lediglich ein Eintrag in der Symboltabelle vorzunehmen.

**Attribut**:

`d.attr` gibt den Typ an.

**Prozedur**:

`enter(i,a)` legt einen Eintrag für den Identifier `i` in der Symboltabelle an und trägt `a` als Typ von `i` ein.

**Grammatik**:

```
d ::= "int" <ID> ;
d ::= "float" <ID> ;
d ::= d "," <ID> ;
```

**Attributgrammatik**:

```
d:d ::= "int" <ID>
        {
           enter(<ID>,I);
           d.attr = I;
        }
     ;
d:d ::= "float" <ID>
        {
           enter(<ID>,F);
           d.attr = F;
        }
     ;
d:d ::= d:d1 "," <ID>
        {
```

```
            enter(<ID>,d1.attr);
            d.attr = d1.attr;
         }
      ;
```

**Zwischencodeerzeugung bei Prozeduraufrufe**

**Grammatik**:

```
s     ::= "call" <ID> "(" elist ")" ;
elist ::= elist  "," e ;
elist ::= e ;
```

Wenn ein Prozeduraufruf geschieht, muss Speicherplatz für das Aktivierungssegment der gerufenen Prozedur zugewiesen werden. Die Argumente de Prozedur müssen ausgewertet werden und ihr an einem bekannten Platz verfügbar gemacht werden. Umgebungszeiger müssen gesetzt werden, um der Prozedur den Zugriff auf Daten in umgebenden Böcken zu ermöglichen. Der Zustand der Prozedur muss gerettet werden, so dass sie die Ausführung nach dem Aufruf aufnehmen kann. Ebenso wird Platz für die Rückkehradresse reserviert. Anschliessend muss ein Sprung zum Anfang des Codes der Prozedur generiert werden.

**Attributgrammatik**:

```
s     ::= "call" <ID> "(" elist ")"
          {
             int n = 0;
             for p in queue {
               emit(PARAM, null, null, p);
               n++;
             }
             emit(CALL, <ID>, n, null);
          }
       ;
elist ::= elist  "," e:e
          {
             insert(e.place,queue)
          }
       ;
elist ::= e:e
          {
             init(queue);
             insert(e.place,queue)
          }
       ;
```

### 4.1.2 Zwischencode Optimierung

Eine Zwischencodeerzeugungsstrategie, die einen Befehl nach dem anderen abarbeitet, erzeugt oft Zwischencode mit überflüssigen Anweisungen und nicht-optimalen Konstruk-

ten. Die Qualität dieses Codes kann durch Anwendung von optimierenden Transformationen auf das Zwischencode verbessert werden. Da diese Transformationen einen lokalen Charakter besitzen ist also gar nicht garantiert, dass das transformierte optimal ist. Trotzdem können viele einfache transformationen die Laufzeit oder den Speicherbedarf des Zwischencodes deutlich verbessern

Eine einfache aber effektive Technik für lokale Verbesserung im Zwischencode ist die **Guckloch Optimierung** (**peephole optimization**). Diese Methode versucht, durch Betrachten von kutzen Ausschnitten des Codes Leistungsverbesserungen zu erzielen. Typische Verbesserungen sind:

**Branches to Fall Thrus** Ist die Sprungadresse eines `GOTO`, `IFEQ`, ... Quadrupels `Q` das unmittelbar folgende Quadruppel, so kann `Q` gelöscht werden. Z.B. kann

```
       [IFEQ, a,  b,  L]
   L:  [IADD, c,  d,  t]
```

durch

```
   L:  [IADD, c,  d,  t]
```

ersetzt werden.

**Branches to `GOTO`** Ist `Q` ein `GOTO`, `IFEQ`, ... Quadrupel dessen Sprungadresse `L` ein `GOTO` Quadruppel ist, dann ersetze `L` in `Q` durch die Sprungadresse von `L`. Z.B. kann

```
       [GOTO, null, null,    L]
       ......................
   L:  [GOTO, null, null,    M]
```

durch

```
       [GOTO, null, null,    M]
       ......................
   L:  [GOTO, null, null,    M]
```

ersetzt werden.

**Unnecessary Labels** Nicht mehr gebrauchte Labels können gelöscht werden.

**Unreachable Code** Eliminiere alle Quadrupel ohne Label, die unmittelbar nach einem `GOTO` Quadrupel vorkommen. Z.B. kann

```
       [GOTO, null, null,   L1]
       [IMUL,    a ,   b,    c]
       [IMUL,    b ,   c,    c]
   L2: [IADD,    a,    b,    c]
```

durch

```
       [GOTO, null, null,   L1]
   L2: [IADD,    a,    b,    c]
```

ersetzt werden.

**Unnecessary `ICOPY`** Falls ein `ICOPY` Quadrupel eine Hilfsvariable `t` als Argument besitzt, die das Resultat des vorherigen Quadrupels ist, ersetze `t` im vorherigen Quadrupel durch das Resultat von `ICOPY` und eliminiere das `ICOPY` Quadrupel. Z.B. kann

```
[ IMUL,    a,    b,    t]
[ICOPY,    t, null,    c]
```

durch

```
 IMUL,    a,    b,    c]
```

ersetzt werden.

**Bemerkung 4.1 [Labels]** Es dürfen nur Quadrupel ohne Label gelöscht werden!

**Bemerkung 4.2 [Weitere Optimierungen]** Es gibt auch eine unbegrenzte Anzahl von algebraischen Vereinfachungen, die durch Guckloch Optimierung realisiert werden können. So können die Quadrupel `[IADD, x, 0, x][IMUL, x, 1, x]` entfernt werden. Algebraische Ausdrücke mit konstanten Argumenten können direkt ausgewertet (interpretiert) werden.

### 4.1.3 Aufgaben

**Aufgabe 4.1 [Backpatching]** *Interpretieren Sie die semantische Aktionen `quad`, `merge` und `backpatch` mit Hilfe von labels.*

**Aufgabe 4.2 [Java Compiler]** *Untersuchen Sie, ob der standard* JAVA *Compiler von* SUN *Backpatching verwendet. Begründen Sie ihre Antwort. Verwenden Sie für Ihre Untersuchung die Versionen 1.3 und 1.4 des Compilers.*

*Schreiben Sie zu diesem Zweck ein kleines* JAVA *Programm und untersuchen Sie das erzeugte Bytecode.*

*Sie können* JAVA *Klassen mittels `javap` dekompilieren.*

**Aufgabe 4.3 [Peephole Optimierung]** *Implementieren Sie ein Programm zur Optimierung einer Quadrupel Tabelle. Wiederholen Sie dabei die 5 Transformationen aus Abschnitt 4.1.2 bis keine mehr Änderung der Quadrupel Tabelle möglich ist.*

## 4.2 Codeerzeugung

### 4.2.1 Allgemeines

Es gibt verschiedene mögliche Typen von Zielcode:

- Maschinencode mit festen Adressen.
- Maschinencode mit relativen Adressen. Hier müssen vor der Ausführung noch Binder und Lader in Aktion treten. Ein Vorteil dieses Typs von Zielcode ist die Möglichkeit, Unterprogramme getrennt zu übersetzen.
- Assembler.

Fragen bei der Codeerzeugung:

- Welche Maschinenbefehle (Assemblerbefehle) sollen erzeugt werden?

  Man könnte zum Beispiel die Zwischencodeanweisung

  `[IADD, a, 1, a]`

  entweder in eine Zielcodeanweisung:

  `INC a`

  oder in die Folge

  ```
  MOV a,R0
  ADD #1,R0
  MOV R0,a
  ```

  übersetzen, wobei die Existenz der entsprechenden Befehle im Zielcode vorausgesetzt werden soll. Welche Möglichkeit besser ist, lässt sich i.a. nur in einem grösseren Kontext entscheiden.

- Welche Speicherplätze und Register sollen verwendet werden? Befehle mit Operanden, die in Registern stehen, sind i.a. kürzer und schneller als solche mit Speicheroperanden. Deshalb sollte man die Register der Zielmaschine möglichst effizient nutzen. Die optimale Zuweisung von Register an Variablen ist aber kompliziert (NP-vollständig).

- In welcher Reihenfolge sollen die Befehle generiert werden? I.a. sind für eine Folge von Zwischencodeanweisungen mehrere verschiedene Reihenfolgen von Zielcodeanweisungen möglich, welche hinsichtlich des Ergebnisses, nicht aber hinsichtlich der Ausführungszeit äquivalent sind. Das Auffinden der optimalen Reihenfolge ist ebenfalls kompliziert (NP-vollständig).

Das Ziel bei der Codeerzeugung ist die Generierung von möglichst gutem Code bezüglich Ausführungszeit und Speicherbedarf. Es werden i.a. keine optimalen, sondern heuristischen Methoden zur Minimierung von Laufzeit und Speicherbedarf des erzeugten Codes angewendet.

### 4.2.2 Eine einfache Zielmaschine

Es seien verfügbar

- `n` allgemeine Register `R0, R1, R2,... ,Rn`
- 2-Adress-Befehle der Form

  `OP Quelle,Ziel`

  Zum Beispiel

  ```
  ADD Quelle,Ziel   (Addition; Inhalt von Ziel geht verloren)
  SUB Quelle,Ziel   (Subtraktion
  MOV Quelle,Ziel   (Kopieren)
  ```

Wir nehmen an, dass Quelle und Ziel nicht gross genug sind, um volle Speicheradressen aufzunehmen. Deshalb steht in Quelle und Ziel jeweils nur ein spezielles Bitmuster, welches die Art der Adressierung festlegt. Wird eine oder mehrere Speicheradressen benötigt, so stehen diese im nächsten bzw. den nächsten Worten, die auf das Befehlswort folgen.

Unsere Zielmaschine möge über folgende Adressierungsarten verfügen, wobei inhalt(a) den Inhalt der Registers oder der Speicheradresse a ist:

1. **Direkte Adressierung**
   Schreibweise: `M`
   `M` ist die Adresse des Operanden. `M` steht im Wort, welches direkt auf das befehl folgt

2. **Adressierung durch Register**
   Schreibweise: `R`
   Der Operand steht im Register `R`.

3. **Indizierte Adressierung**
   Schreibweise: `c(R)`
   Die Adresse des Operanden ist`c + inhalt(R)`, wobei `R` ein Register ist. Der Wert `c` steht im Wort, welches auf das Befehlswort folgt.

4. **Indirekte Adressierung durch Register**
   Schreibweise: `*R`
   Die Adresse des Operanden ist `inhalt(R)`

5. **Indirekte indizierte Adressierung**
   Schreibweise: `*c(R)`
   Die Adresse des Operanden ist `inhalt(c + inhalt(R))`. Der Wert `c` steht wieder im auf das Befehlswort folgenden Wort.

6. **Literale Adressierung**
   Schreibweise: `#c`
   Der Operand ist die Konstante `c`. Sie steht im Wort, welches auf das Befehlswort folgt.

**Beispiel 4.3 [Adressierung]**

1. `MOV R0,R1`
   Kopiert den Inhalt der Registers `R0` in das Register `R1`.

2. `MOV R5,M`
   Kopiert den Inhalt des Registers `R5` in den Speicherplatz mit der Adresse `M`.

3. `ADD #1,R3`
   Addiert die Konstante `1` zum Inhalt des Registers `R3`. Der ursprüngliche Wert von `R3` geht verloren.

4. `SUB 4(R0),*12(R1)`
   Subtrahiert `inhalt(4 + inhalt(R0))` von `inhalt(12 + inhalt(R1))` und speichert das Ergebnis unter der Adresse `inhalt(12 + inhalt(R1))`.

Zur Beurteilung der Güte des erzeugten Codes werden für jeden befehl **Kosten** definiert. Die Kosten eines Befehls werden im folgenden als *Anzahl der Worte*, welche der Befehl insgesamt belegt, angenommen. Dies ist ein stark vereinfachtes, aber dennoch sinnvolles Modell, da i.a. das Holen eines Befehls aus dem Speicher länger dauert als die eigentliche Ausführung des Befehls, also die Ausführungszeit eines Befehls wesentlich von seiner Länge abbhängt.

**Beispiel 4.4 [Kosten]** Die Befehle von Beispiel 4.3 besitzen folgende Kosten:

1. Kosten `1`, dan neben dem Befehlswort selbst kein weiterer Speicherplatz benötigt wird.
2. Kosten `2`, da ein Speicherwort für das Befehlswort selbst und ein weiteres für `M` benötigt wird.
3. Kosten `2` da ein Speicherwort für das Befehlswort selbst und ein weiteres für die Konstante `1` benötigt wird.
4. Kosten `3`, da die Konstanten `4` und `12` in den beiden Worten, die auf das Befehlswort folgen, gespeichert werden.

Mit Hilfe der oben definierten kosten können wir nun mögliche Varianten, die bei der übersetzung eines Zwischencodebefehls existieren, beurteilen.

**Beispiel 4.5 [Übersetzung]** Bei der übersetzung des Zwischencodebefehls

```
[IADD, b, c, a]
```

existieren folgende Möglichkeiten:

1. ```
   MOV b,R0
   ADD c,R0
   MOV R0,a
   ```
   Kosten `6`
2. ```
   MOV b,a
   ADD c,a
   ```
   Kosten `6`
3. ```
   MOV *R1,*R0
   ADD *R2,*R0
   ```
   Kosten `2`

   Wir nehmen hier an, dass `R0`, `R1` und `R2` die Adressen von `a`, `b` und `c` enthalten.
4. ```
   ADD R2,R1
   MOV R1,a
   ```
   Kosten `3`

   Hier wird angenommen, dass `R1` und `R2` die Werte von `b` und `c` enthalten und dass der Wert von `b` nach der Zuweisung nicht mehr benötigt wird.

Anhand obigen Beispiels erkennt man, dass es sehr wichtig ist, ob die Operanden oder ihre Adressen in Registern oder im Speicher stehen. Operationen mit Registern sind i.a. effizienter. Deshalb sollte man bestrebt sein, Operanden oder Ergebnisse, die in naher Zukunft (nochmals) gebraucht werden, in Registern zu halten. Indirekte oder indizierte Adressierung spielt vor allem bei der übersetzung von zusammengesetzten Datenstrukturen und Zeigern eine Rolle.

### 4.2.3 Grundlegende Konzepte

Ein **Basisblock** ist eine Folge fortlaufender Anweisungen des Zwischencodes, die in der Reihenfolge ihrer Niederschrift ohne einen Halt, eine Verzweigung oder einen Sprung ausgeführt werden. Lediglich die letzte Anweisung eines blocks kann ein Halt, eine Verzweigung oder ein Sprung sein. Das Betreten eines Basisblocks kann nur über seine erste, das Verlassen nur über seine letzte Anweisung erfolgen.

**Beispiel 4.6 [Basisblock]** `[IMUL, 2, b, t1]`
`[ISUB, t1, 5, a]`
`[IF<, a, b, 5]`

Um effizienten Code zu erzeugen, ist es wichtig, informationen über die nächste Verwendung oder Benutzung von Namen zu sammeln. Der Wert eines Namens braucht nur dann im Speicher gehalten zu werden, wenn nachfolgend eine Verwendung erfolgt. Wir sagen dass in der Wertzuweisung `x := y + z` der Name `x`**definiert** und die Namen `y` und `z`**verwendet** werden. Liegt das Programmstück

```
...
(i) [ICOPY, .., null, x]
...
(j) [.., .., x, ..]
...
```

vor, wobei die Kontrolle von (`i`) nach (`j`) gelangen kann, ohne dass in der Zwischenzeit eine Wertzuweisung an `x` erfolgt, so sagt man, dass der Befehl (`j`) den vom Befehl (`i`) berechneten Wert verwendet.

#### Bestimmung der nächsten Verwendung innerhalb eines Basisblocks

Durchlaufe schrittweise alle Befehle von hinten nach vorne. Bei Erreichen der Befehls
`(i) x := y op z`
werden folgende Aktionen durchgeführt:

1. Füge der gegenwärtig in der Statustabelle vorhandene Information über `x`, `y` und `z` den Befehl `(i)` hinzu (Die Quadrupeltabelle soll dementsprechend erweitert werden).
2. Setze den Status von `x` auf **Keine nächste Verwendung**
3. Setze den Status von von `y` und `z` auf **nächste Verwendung = `(i)`** (Beachte: Da `x` und `y` oder `z` gleich sein können, z.B. bei `x := x + 1`, darf die Reihenfolge von (2) und (3) nicht vertauscht werden).

Für Sprungbefehle oder Befehle mit unären Operatoren wird analog verfahren.

Die **Statustabelle** gibt für jeden Namen die nächste Vervendung an, z.B.:

| Name | Verwendung |
|---|---|
| x | – |
| y | j |
| z | j |

Bei Initialisieren dieser tabelle wird der Status eines jeden Namens auf **nächste Verwendung = nächster Basisblock** gesetzt.

Bei der Codeerzeugung ist es wichtig, Buch darüber zu führen, welche Register mit welchen Werten belegt sind und wo die Werte der einzelnen Datenobjekte abgelegt sind. Ein **Register-Deskriptor** gibt den Inhalt eines Registers zu einem bestimmten Zeitpunkt an. Man beachte, dass Register gleichzeitig Werte von mehr als einem Namen enthalten kann, falls die Werte identisch sind. Ein **Adress-Deskriptor** gibt die Adresse an, unter welcher der Aktuelle Wert eines Namens zu finden ist. Die Adresse kann z.B. ein Register oder ein Hauptspeicherplatz sein. Der Wert eines namens kann unter verschiedenen Adressen abgelegt sein.

**Beispiel 4.7 [Speicherbelegung]** Es seien `R0`,... `R3` Register, `W`, `X`, `Y`, `$` Hauptspeicheradressen uns `a`, `b`, `c`, `d`, `e` die Werte der Variablen `A`, `B`, `C`, `D`, `E`. Die Speicherbelegung

```
R0 : a = b  W : a
R1 : c      X : b
R2 : e      Y : c
R3 : e      Z : d
```

entspricht einer Register-Deskriptor-Tabelle

```
R0  a, b
R1  c
R2  e
R3  e
```

und einer Adress-Deskriptor-Tabelle

```
A  R0, W
B  R0, X
C  R1, Y
D  Z
E  R2, R3
```

**Algorithmus 4.1 [Basisblockzerlegung]**

**Eingabe:** Eine Folge von Drei-Adress-Befehlen

**Ausgabe:** Eine Menge von Basisblöcken, wobei jeder Drei-Adress-Befehl in genau einem Basisblock enthalten ist.

**Algorithmus:**

1. Bestimmung des Anfangs von Basisblöcken:
   (a) Der erste Befehl ist Basisblockanfang
   (b) Jeder Befehl, der Ziel eines bedingten oder unbedingten Sprung ist ist Basisblockanfang.
   (c) Jeder Befehl, der direkt auf einen bedingten oder unbedingten Sprung folgt ist Basisblockanfang.
2. Bestimmung des zu einem Anfang gehörigen Basisblocks:
   Ein Basisblock enthält den Basisblockanfang und alle folgenden Befehle bis zu, jedoch ausschliesslich, dem nächsten Basisblockanfang oder bis zum Programmende.

### Transformationen in Basisblöcken

Ein Basisblock berechnet eine Menge von Ausdrücken. Diese Ausdrücke sind die Werte der am Ausgang des Blocks aktiven Namen. Zwei Basisblöcke heissen äquivalent, wenn sie die gleiche Menge von Ausdrücken berechnen.

Eine gewisse Anzahl von Transformationen kann auf Basisblöcke angewendet werden, ohne dass sich die Menge der von dem Block berechneten Ausdrücke dadurch ändert. Viele dieser Transformationen dienen zur Verbesserung des Codes, der letztendlich aus den Basisblöcken erzeugt wird.

### Strukturerhaltende Transformationen

Die wesentlichen strukturerhaltenden Transformationen auf Basisblöcken sind:

1. Elimininierung gemeinsamer Teilausdrücke
2. Entfernung passiven Codes
3. Umbenennung von Hilfsvariablen
4. Vertauschung zweier unabhängiger, benachbarter Befehle

Diese Transformationen wollen wir nun etwas ausführlicher betrachten. Dazu nehmen wir an, dass die Basisblöcke keine Felder, Zeiger oder Prozeduraufrufe enthalten.

1. Eliminierung gemeinsamer Teilausdrücke. Nehmen wir den Basisblock

   ```
   [IADD, b, c, a]
   [ISUB, a, d, b]
   [IADD, b, c, c]
   [ISUB, a, d, d]
   ```

   Die zweite und vierte Anweisung berechnen den gleichen Ausdruck, nämlich `b + c - d`. Damit kann der Basisblock in den folgenden äquivalenten Block transformiert werden:

   ```
   [IADD, b, c, a]
   [ISUB, a, d, b]
   [IADD, b, c, c]
   [ICOPY, b, null, d]
   ```

   Zu bemerken ist, dass, obwohl die erste und dritte Anweisung scheinbar den gleichen Ausdruck auf der rechten Seite besitzen, `b` von der zweiten Anweisung neu berechnet wird. Dadurch besitzt `b` in der dritten Anweisung einen anderen Wert als in der ersten, und deshalb berechnen die erste und dritte Anweisung nicht den gleichen Ausdruck.

2. Entfernung passiven Codes. Nehmen wir an, dass `x` passiv ist, d.h. dass es nach einer Anweisung `x := y + z` nirgends mehr verwendet wird. Dann kann diese Anweisung entfernt werden, ohne dass sich der Wert des Basisblocks ändert.

3. Umbenennung von Hilfsvariablen. Nehmen wir an, es existiert eine Anweisung `t := b + c`, wobei `t` eine Hilfsvariable ist. Wenn wir diese Anweisung in `u := b + c` umwandeln, wobei `u` eine neue Hilfsvariable ist, und an allen Stellen, wo der Wert von `t` benutzt wird, `u` eintragen, so ändert sich der Wert des Basisblocks nicht. Daher können wir auch jeden Basisblock in einen äquivalenten Basisblock umwandeln, in dem jede Anweisung, die eine Hilfsvariable definiert, jeweils eine neue Hilfsvariable definiert. Ein solcher Basisblock heisst Block in Normalform.

4. Vertauschung von Anweisungen. Nehmen wir an, einen Block mit den beiden benachbarten Anweisungen

   ```
   [IADD, b, c, t1]
   [IADD, x, y, t2]
   ```

   Wir können die beiden Anweisungen genau dann vertauschen, ohne dass sich der Wert des Blocks ändert, wenn weder `x` noch `y` gleich `t1` und weder `b` noch `c` gleich `t2` Sind. Für einen Basisblock in Normalform gilt, dass alle denkbaren Vertauschungen auch möglich sind.

### Algebraische Transformationen

Unzählige algebraische Transformationen können angewendet werden, um die Menge der von einem Basisblock berechneten Ausdrücke in eine algebraisch äquivalente Menge umzuwandeln. Davon sind solche nützlich, welche Aucdrücke vereinfachen oder kostspielige Operationen durch effizientere ersetzen. Zum Beispiel können Anweisungen wie

```
[IADD, x, 0, x]
```

oder

```
[IMUL, x, 1, x]
```

aus Basisblöcken entfernt werden, ohne sich dass die Menge der berechneten Ausdrücke ändert. Der Exponential-Operator in der Anweisung

```
[IEXP, y, 2, x]
```

wird meistens durch einen Funktionsaufruf realisiert. Mit einer algebraischen Transformation kann diese Anweisung durch eine billigere, aber äquivalente Anweisung

```
[IMUL, y, y, x]
```

ersetzt werden.

### Flussgraphen

Durch Hinzufügen von Kontrollfluss-Informationen zu der Menge der Basisblöcke eines Programms erhalten wir einen gerichteten Graphen, den sogenannten Flussgraphen. Die Knoten des Flussgraphen sind die Basisblöcke. Ein Knoten wird als Startknoten gekennzeichnet; es ist derjenige Block, dessen Anfang der erste Befehl ist. Es gibt eine gerichtete Kante vom Block `B1` zum Block `B2`, falls `B2` in einer Ausführungssequenz direkt nach `B1` folgen kann, d.h. falls

1. es einen bedingten oder unbedingten Sprung von der letzten Anweisung in `B1` zur ersten Anweisung in `B2` gibt, oder

2. B2 im Programm direkt nach B1 folgt und B1 am Ende keinen unbedingten Sprung enthält.

In diesem Fall heisst B1**Vorgänger** von B2, und B2**Nachfolger** von B1.

**Beispiel 4.8 [Flussgraph]** Programms, 3-Adress-Code und zugehöriger Flussgraph. B1 ist der Startknoten. Der in der letzten Anweisung enthaltene Sprung zur Anweisung (3) wurde durch einen äquivalenten Sprung zum Anfang des Blocks B2 ersetzt.

```
begin
  prod := 0;
  i := 1;
  do begin
    prod := prod + a[i]*b[i];
    i := i + 1
  end;
  while i <= 20
end
```

```
01  [ICOPY,    0, null, prod]
01  [ICOPY,    1, null,    i]
03  [IMUL,     4,    i,   t1]
04  [IREFA,    a,   t1,   t2]
05  [IMUL;     4,    i,   t3]
06  [IREFA,    b,   t3,   t4]
07  [IMUL     t2,   t4,   t5]
08  [IMUL,  prod,   t5,   t6]
09  [ICOPY    t6, null, prod]
10  [IADD,     i,    1,   t7]
11  [ICOPY,   t7, null,    i]
12  [IF<=,     i,   20,    3]

01  [ICOPY,    0, null, prod] B1
01  [ICOPY,    1, null,    i]

03  [IMUL,     4,    i,   t1] B2
04  [IREFA,    a,   t1,   t2]
05  [IMUL;     4,    i,   t3]
06  [IREFA,    b,   t3,   t4]
07  [IMUL     t2,   t4,   t5]
08  [IMUL,  prod,   t5,   t6]
09  [ICOPY    t6, null, prod]
10  [IADD,     i,    1,   t7]
11  [ICOPY,   t7, null,    i]
12  [IF<=,     i,   20,    3]
```

### Darstellung der Basisblöcke

Basisblöcke können durch eine Vielzahl von Datenstrukturen dargestellt werden. Zum Beispiel kann, nach der Aufteilung der Drei-Adress-Befehle mit Hilfe des Basisblock-Algorithmus, jeder Basisblock durch einen Record dargestellt werden, der die Anzahl

von Quadrupeln im Block, gefolgt von einem Zeiger auf den Blockanfang (erstes Quadrupel) und einer Liste von Nachfolgern und Vorgängern des Blocks enthält. Als Alternative könnte man für jeden Block eine verkettete Liste von Quadrupeln erzeugen. Die explizite Angabe von Quadrupelnummern in Sprunganweisungen am Ende eines Basisblocks kann Probleme verursachen, wenn während der Code-Optimierung Quadrupel bewegt worden sind. Zum Beispiel müsste die (3) in `[IF<=, i, 20, 3]` geändert werden, falls der Block `B2`, der die Anweisungen (3) bis (12) des Zwischencodes enthält, an eine andere Stelle im Quadrupelfeld gesetzt worden oder geschrumpft wäre. Daher ist es besser, wenn Sprünge auf Blöcke statt auf Quadrupel zeigen.

Es sollte erwähnt werden, dass eine Kante in einem Flussgraph von Block `B` zu Block `B'` nicht die Bedingung angibt, unter der die Kontrolle von `B` nach `B'` übergeht. Dies bedeutet, dass die Kante selbst nichts darüber aussagt, ob der bedingte Sprung am Ende von `B` (falls dort ein bedingter Sprung vorkommt) zum Anfang von `B'` führt, falls die Bedingung erfüllt oder nicht erfüllt ist. Diese Information kann man, falls sie benötigt wird, aus dem Sprungbefehl in `B` erhalten.

### Schleifen

Was ist eine Schleife in einem Flussgraph und wie findet man alle Schleifen? Meistens ist es leicht, diese Fragen zu beantworten. In obigem Beispiel gibt es eine Schleife, die aus dem Block `B2` besteht. Eine generelle Antwort auf diese Fragen erfordert eine gewisse Technik. Für den Augenblick genügt die Feststellung, dass eine Schleife eine Ansammlung von Knoten in einem Flussgraph ist, für die gilt:

1. Alle Knoten in der Ansammlung sind streng verbunden, d.h. von einem Knoten in der Schleife zu irgendeinem anderen gibt es einen Pfad der Länge eins oder mehr, der ganz in der Schleife liegt, und
2. Die Ansammlung von Knoten hat einen eindeutigen Eingang, d.h. einen Knoten in der Schleife, für den gilt, dass jeder Weg von einem Knoten ausserhalb zu einem Knoten in der Schleife durch diesen Eingang führt.

Eine Schleife, die keine andere Schleife enthält, wird innere Schleife genannt.

### Informationen über die nächste Verwendung

In diesem Abschnitt sammeln wir Informationen über die nächste Verwendung von Namen in Basisblöcken. Falls der Name in einem Register nicht mehr benötigt wird, kann das Register einem anderen Namen zugewiesen werden. Die Idee, dass Namen nur dann im Speicher gehalten werden, wenn sie nachfolgend benutzt werden, kann in vielen Bereichen eingesetzt werden. Der einfache Code-Generator, der im nächsten Abschnitt beschrieben wird, wendet diese Methode bei der Registerzuweisung an. Als letzte Anwendungsmöglichkeit soll die Zuweisung von Speicherplatz für Hilfsvariablen erwähnt werden.

### Berechnung der nächsten Verwendung

Die Verwendung eines Namens in einem Drei-Adress-Befehl ist folgendermassen definiert. Nehmen wir an, der Drei-Adress-Befehl `i` weist `x` einen Wert zu. Wenn `x` ein Operand des

Befehls `j` ist und die Kontrolle vom Befehl `i` auf einem Pfad zum Befehl `j` übergehen kann, auf dem `x` keine neue Zuweisung erhält, so sagen wir: Befehl `j` verwendet den bei `i` berechneten Wert von `x`.

Wir wollen für jeden Drei-Adress-Befehl `[IOP, y, z, x]` die nächsten Verwendungen von `x`, `y` und `z` bestimmen. Zunächst befassen wir uns nicht mit Verwendungen, die ausserhalb des Basisblocks liegen, der diesen Befehl enthält.

Unser Algorithmus zur Bestimmung der nächsten Verwendung durchläuft jeden Basisblock rückwärts. Wir können beim Durchlaufen einer Folge von Drei-Adress-Befehlen die Grenzen der Basisblöcke leicht finden. Da Prozeduren beliebige Seiteneffekte haben können, nehmen wir zur Vereinfachung an, dass jeder Prozeduraufruf mit einem neuen Basisblock beginnt.

Haben wir das Ende eines Basisblocks gefunden, so laufen wir rückwärts bis zum Anfang, wobei für jeden Namen `x` (in der Symboltabelle) eingetragen wird, ob `x` eine nächste Verwendung in diesem Block besitzt. Gibt es keine, so wird eingetragen, ob x beim Verlassen des Blocks aktiv ist. Wurde die Datenflussanalyse durchgeführt, so wissen wir für jeden Block, welche Variablen am Ausgang aktiv sind. Wenn wir keine derartige Analyse zur Feststellung der Aktivierung der Variablen vorgenommen haben, sollten wir vorsichtshalber annehmen, dass am Ausgang alle nicht temporären Variablen aktiv sind. Falls die Algorithmen zur Erzeugung von Zwischencode oder zur Code-Optimierung es erlauben, dass gewisse Hilfsvariablen in unterschiedlichen Blöcken benutzt werden, so müssen auch diese als aktiv angesehen werden. Solche Hilfsvariablen sollten dann markiert werden, damit nicht alle Hilfsvariablen als aktiv angesehen werden müssen.

Nehmen wir an, wir erreichen beim Rückwärtsdurchlauf durch den Basisblock den Drei-Adress-Befehl `i: [IOP, y, z, x]`. Dann führen wir die folgenden Aktionen durch:

1. An den Befehl `i` werden die Informationen gebunden, die wir momentan in der Symboltabelle über die nächste Verwendung von `x`, `y` und `z` finden [1].
2. `x` wird in der Symboltabelle auf *nicht aktiv* und *eine nächste Verwendung* gesetzt.
3. `y` und `z` werden in der Symboltabelle auf *aktiv* und die nächste Verwendung von `y` und `z` auf `i` gesetzt. Dabei darf die Reihenfolge der Schritte (2) und (3) darf nicht vertauscht werden, weil `x` gleich `y` oder `z` sein kann.

Falls der Drei-Adress-Befehl `i` die Form `[ICOPY, y, null, x]` oder `[IOP, y, z, x]` besitzt, so bleiben die Schritte wie oben, wobei allerdings `z` nicht berücksichtigt wird.

### Speicher für Hilfsvariablen

Obgleich es für einen optimierenden Compiler nützlich sein kann, unterschiedliche Namen zu benutzen, wenn eine Hilfsvariable benötigt wird, muss Speicherplatz zur Aufnahme der Werte dieser Hilfsvariablen zugewiesen werden. Die Grösse des Bereichs für Hilfsvariablen wächst mit der Anzahl der Hilfsvariablen.

Im allgemeinen können wir zwei Hilfsvariablen im gleichen Speicherplatz halten *packen*, wenn sie nicht gleichzeitig aktiv sind. Da Hilfsvariablen meistens in einem Basisblock definiert und benutzt werden, können wir Informationen über die nächste Verwendung

[1] Ist `x` nicht aktiv, so kann diese Anweisung gelöscht werden.

zum Packen von Hilfsvariablen heranziehen. Für Hilfsvariablen, die über Blockgrenzen hinaus verwendet werden, muss eine Datenflussanalyse zur Bestimmung der Aktivierung ausgeführt werden.

Speicherplätze für Hilfsvariablen können wir dadurch vergeben, dass wir die Hilfsvariable dem ersten Speicherplatz im Bereich für Hilfsvariablen zuweisen, der keine aktive Hilfsvariable enthält. Falls eine Hilfsvariable nicht an einen bereits zuvor erzeugten Platz gebunden werden konnte, wird ein neuer Speicherplatz zum Datenbereich der aktuellen Prozedur hinzugefügt. In vielen Fällen können Hilfsvariablen eher in Registern als in Speicherplätzen gehalten werden, wie es im nächsten Abschnitt der Fall ist.

Zum Beispiel können die sechs Hilfsvariablen folgendes Basisblocks in zwei Speicherplätze gepackt werden.

```
[IMUL,    a,    a,   t1]
[IMUL,    a,    b,   t2]
[IMUL,    2,   t2,   t3]
[IADD,   t1,   t3,   t4]
[IMUL,    b,    b,   t5]
[IADD,   t4,   t5,   t6]
```

Diese Plätze bezeichnen wir mit `t1`, und `t2` in:

```
[IMUL,    a,    a,   t1]
[IMUL,    a,    b,   t2]
[IMUL,    2,   t2,   t2]
[IADD,   t1,   t2,   t1]
[IMUL,    b,    b,   t2]
[IADD,   t1,   t2,   t1]
```

### 4.2.4 Ein einfacher Code-Generator

Die Code-Erzeugungsstrategie dieses Abschnitts erzeugt Zielcode für eine Folge von Drei-Adress-Befehlen. Sie untersucht für jeden Befehl, ob sich Operanden des Befehls gerade in Registern befinden und nutzt dies nach Möglichkeit aus. Zur Vereinfachung nehmen wir an, dass es für jeden Operator in einem Befehl einen entsprechenden Operator in der Zielsprache gibt. Ausserdem nehmen wir an, dass berechnete Ergebnisse so lange wie möglich in einem Register gehalten werden können. Sie werden nur abgespeichert, wenn (a) das Register für eine andere Berechnung benötigt wird oder (b) direkt vor einem Prozeduraufruf, Sprung oder markierten Befehl [2].

Die Bedingung (b) bewirkt, dass vor dem Ende eines Basisblocks alles abgespeichert werden muss [3]. Der Grund dafür ist, dass wir nach dem Verlassen eines Basisblocks zu unterschiedlichen Blöcken gehen können, oder wir können zu einem bestimmten Block gehen,

[2] Um jedoch einen symbolischen Dump zu erzeugen, der die Werte von Speicherplätzen und Registern bezogen auf die Namen des Quellprogramms zur Verfügung stellt, wäre es nützlicher, wenn selbstdefinierte Variablen (aber nicht unbedingt die vom Compiler erzeugten Hilfsvariablen) unmittelbar nach einer Berechnung abgespeichert werden, falls ein Programmfehler eine plötzliche Unterbrechung oder sogar einen Abbruch verursacht.

[3] Wir nehmen nicht an, dass die Quadrupel vom Compiler tatsächlich in Basisblöcke aufgeteilt werden; die Bezeichnung Basisblock ist von der Bedeutung her auf jeden Fall sinnvoll.

der auch noch von anderen Blöcken aus erreicht werden kann. Auf jeden Fall können wir nicht ohne zusätzlichen Aufwand annehmen, dass ein von einem Block benutzter Wert sich in einem festen Register befindet, unabhängig davon, von wo aus die Kontrolle diesen Block erreicht. Um daher einen möglichen Fehler zu vermeiden, speichert unser einfacher Code-Erzeugungsalgorithmus alles ab, unabhängig davon, ob sich die Kontrolle über Blockgrenzen bewegt oder ob ein Prozeduraufruf erfolgt. Später geben wir Möglichkeiten an, einige Werte über Blockgrenzen hinweg in Registern zu lassen.

Wir können für den Drei-Adress-Befehl `[IADD, b, c, a]` vernünftigen Code erzeugen, falls wir die einfache Anweisung `ADD Rj,Ri` - die die Kosten eins besitzt - erzeugen, wobei das Ergebnis `a` in Register `Ri` steht. Diese Sequenz ist nur möglich, falls `b` in Register `Ri` und `c` in Register `Rj` enthalten ist und `b` darüber hinaus nach der Anweisung nicht aktiv ist, d.h. `b` wird nach der Anweisung nicht mehr verwendet.

Falls `b` in `Ri` enthalten ist, `c` sich jedoch in einem Speicherplatz (zur Vereinfachung `c` genannt) befindet, so können wir die folgende Sequenz erzeugen:

`ADD c,Ri` (Kosten = 2)

oder

`MOV c,R` (Kosten = 3)
`ADD Rj,Ri`

Voraussetzung ist, dass `b` danach nicht aktiv ist. Die zweite Sequenz wird interessant, wenn der Wert von `c` nachfolgend erneut benutzt wird. In diesem Fall können wir den Wert aus Register `Rj` nehmen. Man könnte noch viele andere Fälle betrachten, die davon abhängen, ob sich `b` und `c` im Speicher befinden und ob der Wert von `b` nachfolgend wieder benutzt wird. Ausserdem müssen wir die Fälle betrachten, in denen `b` oder `c` oder beide Konstanten sind. Die Anzahl der Fälle kann sich noch weiter erhöhen, wenn wir annehmen, dass der Operator + kommutativ ist. Daran erkennen wir, dass die CodeErzeugung eine grosse Anzahl von Fällen berücksichtigen muss. Die Auswahl des anzuwendenden Falls hängt vom Kontext ab, in dem ein Drei-Adress-Befehl steht.

### Register- und Adress-Deskriptoren

Der Code-Erzeugungsalgorithmus benutzt Deskriptoren zur Beobachtung von Registerinhalten und Adressen für Namen.

1. Ein Register-Deskriptor beobachtet, was sich momentan in jedem Register befindet. Er wird jedesmal benutzt, wenn ein neues Register benötigt wird. Wir nehmen an, dass der Register-Deskriptor zu Beginn anzeigt, dass jedes Register leer ist. (Falls Register über Blockgrenzen hinweg zugewiesen werden, ist dies nicht der Fall.) Beim Voranschreiten der Code-Erzeugung für den Block enthält jedes Register zu einem bestimmten Zeitpunkt den Wert von beliebig vieler Namen.

2. Ein Adress-Deskriptor beobachtet den Platz (oder die Plätze), wo der aktuelle Wert eines Namens zur Laufzeit zu finden ist. Dies kann ein Register, ein Eintrag im Stack, eine Speicheradresse oder eine Menge dieser Möglichkeiten sein, da ein Wert, nachdem er kopiert wurde, auch dort enthalten bleibt, wo er war. Diese Information kann in der Symboltabelle gespeichert werden und wird dazu benötigt, die Zugriffsmethoden für Namen zu bestimmen.

**Algorithmus 4.2 [Codeerzeugung]** Der Code-Erzeugungsalgorithmus erhält als Eingabe eine Folge von Drei-Adress-Befehlen, die zusammen einen Basisblock bilden. Für jeden Drei-Adress-Befehl `[IOP, y, z, x]` werden die folgenden Aktionen durchgeführt:

1. Rufe eine Funktion **getreg** auf, um den Ort `L` zu bestimmen, wo das Ergebnis der Berechnung `y op z` abgespeichert werden soll. Normalerweise ist `L` ein Register, es kann aber auch ein Speicherplatz sein. In Kürze werden wir die Details von **getreg** beschreiben.

2. Betrachte den Adress-Deskriptor für `y`, um `y'`, einen der aktuellen Plätze von `y`, zu bestimmen. Ziehe für `y'` ein Register vor, falls sich der Wert von `y` momentan sowohl im Speicher als auch in einem Register befindet. Falls sich der Wert von `y` noch nicht in `L` befindet, so erzeuge die Anweisung `MOV y',L`, um in `L` eine Kopie von `y` zu erhalten.

3. Generiere die Anweisung `OP z',L`, wobei `z'` der momentane Ort von `z` ist. Dabei wird wieder ein Register einem Speicherplatz vorgezogen, falls `z` sich in beiden befindet. Aktualisiere den Adress-Deskriptor für `x`, um anzuzeigen, dass sich `x` an dem Ort `L` befindet. Falls `L` ein Register ist, so aktualisiere seinen Deskriptor, um anzuzeigen, dass es den Wert von `x` enthält.

4. Falls die aktuellen Werte von `y` und/oder `z` keine nächste Verwendung besitzen, am Ende des Blocks nicht aktiv sind und sich in Registern befinden, so verändere den Register-Deskriptor so, dass diese Register nach der Ausführung von `x := y op z` nicht mehr die Werte von `y` und/oder `z` enthalten.

Falls der aktuelle Drei-Adress-Befehl einen unären Operator enthält, so sind die Schritte analog zu den oben angegebenen. Wir übergehen an dieser Stelle die Einzelheiten. Ein wichtiger Spezialfall stellt der Drei-Adress-Befehl `[ICOPY, y, null, x]` dar. Falls `y` sich in einem Register befindet, so wird nur der Register- und Adress-Deskriptor geändert, um anzuzeigen, dass sich der Wert von `x` nun ausschliesslich in dem Register befindet, dass den Wert von `y` besitzt. Falls `y` keine nächste Verwendung besitzt und am Ende des Blocks nicht aktiv ist, so hält das Register nicht länger den Wert von `y`.

Falls `y` nur im Speicher enthalten ist, so können wir im Prinzip eintragen, dass sich der Wert von `x` im Ort von `y` befindet. Diese Möglichkeit würde allerdings unseren Algorithmus komplizieren, weil wir dann den Wert von `y` nicht ändern könnten, ohne den Wert von `x` zuvor zu retten. Falls sich `y` im Speicher befindet, benutzen wir daher `getreg`, um ein Register zu finden, in das `y` geladen werden kann. Ausserdem tragen wir das Register als Ort für `x` ein.

Als Alternative können wir eine Anweisung `MOV y,x` erzeugen, die vorzuziehen ist, falls der Wert von `x` keine nächste Verwendung im Block besitzt. Es ist erwähnenswert, dass die meisten, wenn auch nicht alle, Kopieranweisungen entfernt werden, wenn wir Blockverbesserungs- und Kopienweitergabe-Algorithmumen benutzen.

Sobald wir alle Drei-Adress-Befehle in einem Basisblock bearbeitet haben, speichern wir durch einen `MOV`-Befehl die Namen ab, die am Ende aktiv sind und sich nicht in ihren Speicherplätzen befinden. Dafür benutzen wir den Register- Deskriptor, um zu bestimmen, welche Namen sich noch in Registern befinden, den Adress-Deskriptor, um festzustellen, ob sich der gleiche Name noch nicht an seinem Speicherplatz befindet, und die Informationen über aktive Variablen, um zu bestimmen, ob der Wert gespeichert werden muss. Falls keine Datenflussanalyse zur Ermittlung von Informationen über aktive Variablen

zwischen den Blöcken durchgeführt wurde, müssen wir annehmen, dass alle vom Benutzer definierten Variablen am Ende eines Blocks aktiv sind.

### Die Funktion getreg

Die Funktion `getreg` liefert für die Anweisung `x := y op z` den Ort `L` zurück, der den Wert von `x` enthält. Bei der Implementierung dieser Funktion kann man grosse Anstrengungen unternehmen, damit eine günstige Auswahl für `L` getroffen wird. In diesem Abschnitt beschreiben wir ein einfaches, leicht zu implementierendes Verfahren, basierend auf Informationen über nächste Verwendungen, die im letzten Abschnitt ermittelt wurden.

1. Falls sich der Name `y` in einem Register befindet, das nicht den Wert eines anderen Namens enthält (zur Erinnerung: ein Kopier-Befehl wie `[ICOPY, y, null, x]` kann bewirken, dass ein Register den Wert von zwei und mehr Variablen gleichzeitig enthält), und ist `y` nach der Ausführung von `[IOP, y, z, x]` nicht aktiv und hat keine nächste Verwendung, dann gebe das Register von `y` als `L` zurück. Aktualisiere den Adress-Deskriptor für `y`, um anzuzeigen, dass `y` nicht länger in `L` enthalten ist.
2. Schlägt (1) fehl, so gebe für `L` ein leeres Register an, falls eines existiert.
3. Schlägt (2) fehl, und besitzt `x` eine nächste Verwendung im Block oder ist op ein Operator wie z.B. Indizierung, welcher ein Register benötigt, so muss ein belegtes Register `R` gesucht werden. Speichere den Wert von `R` in einen Speicherplatz (durch `MOV R,M`), falls er sich noch nicht am richtigen Speicherplatz `M` befindet, aktualisiere den Adress-Deskriptor für `M` und gebe `R` zurück. Falls `R` den Wert mehrerer Variablen enthält, so muss für jede zu speichernde Variable ein `MOV`-Befehl erzeugt werden. Unter den besetzten Registern sind dabei vor allem jene geeignet, deren Wert als letzter in der Zukunft benutzt wird oder deren Wert sich schon im Speicher befindet. Wir lassen die genaue Wahl offen, da es keine bewiesenermassen beste Art der Auswahl gibt.
4. Falls `x` im jeweiligen Block nicht benutzt wird oder kein geeignetes besetztes Register gefunden werden kann, so wähle den Speicherplatz von `x` als `L`.

Eine anspruchsvollere `getreg`-Funktion würde zur Bestimmung des Registers, das den Wert von `x` aufnehmen soll, ausserdem die nachfolgenden Verwendungen von `x` und die Kommutativität des Operators `op` benutzen. Wir stellen diese Erweiterungen von `getreg` als übungsaufgabe.

**Beispiel 4.9 [cg.exa.getreg]** Die Zuweisung `d := (a - b) + (a - c) + (a - c)` kann in die folgende Sequenz von Drei-Adress-Befehlen übersetzt werden:

```
[ISUB, a, b, t]
[ISUB, a, c, u]
[IADD, t, u, v]
[IADD, v, u, d]
```

Dabei ist `d` am Ende aktiv. Der obige Code-Erzeugungsalgorithmus würde für diese Folge von Drei-Adress-Befehlen die in der nächsten Abbildung dargestellte Code-Sequenz erzeugen. Beim Voranschreiten der Code-Erzeugung werden auch die Werte der Register- und Adress-Deskriptoren dargestellt. Im Adress-Deskriptor wird jedoch nicht angezeigt,

dass sich a, b und c immer im Speicher befinden. Ausserdem nehmen wir an, dass t, u und v, die Hilfsvariablen sind, sich nicht im Speicher befinden, bevor sie nicht explizit durch einen MOV-Befehl gespeichert wurden.

Der erste Aufruf von getreg liefert R0 als Ort zur Berechnung von t zurück. Da sich a nicht in R0 befindet, erzeugen wir die Anweisungen MOV a,R0} und SUB b,R0. Dann aktualisieren wir den Register-Deskriptor, um anzuzeigen, dass t in R0 enthalten ist.

Die Code-Erzeugung fährt auf diese Art fort, bis die letzte Anweisung [IADD, v, u, d] bearbeitet wurde. Zu beachten ist, dass R1 leer wird, da u keine nächste Verwendung besitzt. Danach erzeugen wir MOV R0,d, um die aktive Variable d am Ende des Blocks zu speichern.

Die Kosten des in der nächsten Abbildung erzeugten Codes sind 12. Durch Erzeugung von MOV R0,R1 direkt nach der ersten Anweisung und durch Entfernen der Anweisung MOV a,R1 können wir die Kosten auf 11 reduzieren. Um dies zu erreichen, wäre jedoch ein besserer Code-Erzeugungsalgorithmus nötig. Der Grund für die Einsparung liegt darin, dass es billiger ist, R1 aus R0 statt aus dem Speicher zu laden.

| Anweisungen | erzeugter Code | Register-Deskriptor | Adress-Deskriptor |
|---|---|---|---|
| | | Register sind leer | |
| t := a-b | MOV a,R0<br>SUB b,R0 | R0 enthältt | t in R0 |
| u := a-c | MOV a,R1<br>SUB c,R1 | R0 enthält t<br>R1 enthält u | t in R0<br>u in R1 |
| v := t+u | ADD R1,R0 | R0 enthält v<br>>R1 enthält u | u in R1<br>v in R0 |
| d := v+u | ADD R1,R0 | R0 enthält d | d in R0 |
| | MOV R0,d | | d in R0 und im Speicher |

**Bedingte Anweisungen**

Bedingte Sprünge können in einer Maschine auf zwei Arten implementiert werden. Eine Möglichkeit ist, zu verzweigen, wenn der Wert eines bestimmten Registers eine der folgenden sechs Bedingungen erfüllt: negativ, null, positiv, nicht negativ, nicht null, nicht positiv. Auf einer solchen Maschine kann ein Drei-Adress-Befehl der Form [IF<, x, y, z] dadurch implementiert werden, dass y von x im Register R subtrahiert wird und anschliessend nach z gesprungen wird, falls der Wert von Register R negativ ist.

Eine zweite Möglichkeit, die in vielen Maschinen eingesetzt wird, benutzt eine Menge von Bedingungscodes, um anzuzeigen, ob der zuletzt berechnete oder in ein Register geladene Wert negativ, null oder positiv ist. Oft hat eine Vergleichsanweisung (CMP1 in unserer Maschine) die wünschenswerte Eigenschaft, dass sie den Bedingungscode setzt, ohne aktuell einen Wert zu berechnen. Dies bedeutet, dass CMP x,y den Bedingungscode auf positiv setzt, falls x > y gilt usw. Ein Maschinen-Befehl für einen bedingten Sprung führt den Sprung aus, falls eine bestimmte Bedingung <, =, >, <=, >= oder != erfüllt

ist. Dabei benutzen wir die Anweisung `CJ<= z`, welche die Bedeutung hat: Sprung nach `z`, falls der Bedingungscode negativ oder null ist. Zum Beispiel kann `[IF<, x, y, z]` durch

```
CMP x, y
CJ< z
```

implementiert werden.

Wenn wir Code für eine Maschine mit Bedingungscodes generieren, so ist es nützlich, einen Bedingungscode-Deskriptor bei der Code-Erzeugung zu benutzen. Dieser Deskriptor gibt den Namen an, der als letzter den Bedingungscode gesetzt hat, oder ein Paar verglichener Namen, falls der Bedingungscode zuletzt durch einen Vergleich gesetzt wurde. So können wir

```
[IADD, y, x, x]
[IF<, x, 0, z]
```

durch

```
MOV y,R0
ADD z,R0
MOV R0,x
CJ< z
```

implementieren, falls wir wissen, dass der Bedingungscode durch `x` nach der Anweisung `ADD z,R0` bestimmt wurde.

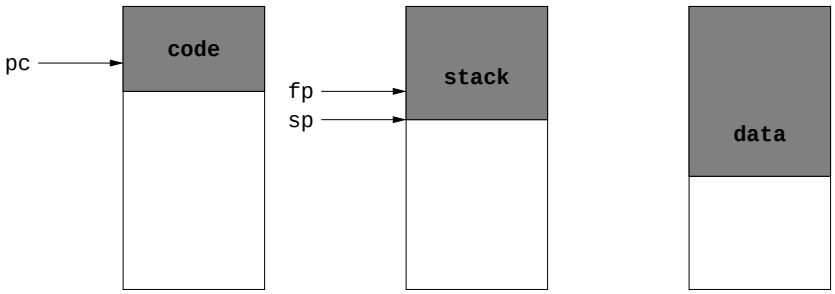
# Anhang A

# JAVA Byte Code

## A.1 Stack-Maschinenmodell

Eine Stack-Maschine besteht aus einem:

- Codebereich
- Stack
- Datenbereich
- pc (Program Counter)
- sp (Stack Pointer)
- fp (Frame Pointer)




Typische Anweisungen für Stack-Maschinen sind

| | |
|---|---|
| `push $n` | schiebe die Zahlkonstante $n$ auf den Stack |
| `push si` | schiebe den Inhalt von Stackadresse `fp+i` auf den Stack |
| `pop ri` | entferne das oberste Element vom Stack und schreibe es in `register[i]` |
| `pop $i` | entferne das oberste Element vom Stack und schreibe es in Stackadresse `fp+i` |
| `+,-,*,/` | arithmetische Operationen auf dem Stack. Diese betreffen jeweils die beiden obersten Elemente. Diese werden entfernt und das Ergebnis wird auf dem Stack abgelegt |
| `cmpX` | vergleicht die obersten beiden Werte auf dem Stack, entfernt diese und legt 1 auf den Stack, falls der Vergleich gültig ist, sonst 0 . `X` steht hier für die Vergleichsoperatoren <, <=, >, >=, =, !=, etc. |
| `ifeq L` | conditional jump. Springt zum Label `L` falls die Konstante 1 auf dem Stack liegt. |
| `goto L` | unconditional jump. Springt zum Label `L`. |
| `move x, y` | lädt den Inhalt von `x` nach `y`. Dabei bezeichnen `x` und `y` Register, Speicherkonstanten oder Speicheradressen |

Je nach Sprachdefinition und Maschinendefinitionen gehören weitere Adressierungsarten und Operationen zum Befehlssatz.

## A.2 Einfache JAVA Bytecode-Instruktionen

Ziel dieses Abschnittes ist es nicht, die ganze JAVA Virtuelle Maschine zu beschreiben (siehe [GJS96]). Es werden nur gerade genügend Instruktionen für den Bau eines kleineren Compilers vorgestellt. Siehe etwa [LY96], [GJS96] für weitere Erläuterungen.

JAVA arbeitet mit einem Stack. Alle locale Variablen werden in lokalen Stackregistern gespeichert. Diese *Register* sind von 0 bis $n$ (Anzahl Variablen) numeriert. Lokale variablen werden mit der entsprechenden `load` Operation auf den Stack gelegt. Gespeichert werden Sie mit der entsprechenden Instruktion `store`.

`i`, `j` und `k` sind ganze Zahlen oder Adressen. `x`, `y`, `z` sind reelwertig. `a`, `b`, `c` sind Arrayreferenzen. `L` ist ein Label.

### A.2.1 Ganzzahlige arithmetische Operationen

| Operation | Stack | Bedeutung |
|---|---|---|
| `iadd` | `...,i,j ⇒ ...,k` | `k = i + j` |
| `isub` | `...,i,j ⇒ ...,k` | `k = i - j` |
| `imul` | `...,i,j ⇒ ...,k` | `k = i * j` |
| `idiv` | `...,i,j ⇒ ...,k` | `k = i / j` |
| `irem` | `...,i,j ⇒ ...,k` | `k = i - (i/j) * j` |
| `ineg` | `...,i ⇒ ...,k` | `k = -i` |

### A.2.2 Reelwertige artithmtische Operationen

| Operation | Stack | Bedeutung |
|---|---|---|
| fadd | ...,y,z ⇒ ...,x | x = y + z |
| fsub | ...,y,z ⇒ ...,x | x = y - z |
| fmul | ...,y,z ⇒ ...,x | x = y * z |
| fdiv | ...,y,z ⇒ ...,x | x = y / z |
| fneg | ...,y ⇒ ...,x | x = -y |

### A.2.3 Typ Umwandlung

| Operation | Stack | Bedeutung |
|---|---|---|
| i2f | ...,i ⇒ ...,x | x = (float) i |
| f2i | ...,y ⇒ ...,k | k = (int) y |

### A.2.4 Vergleichsoperationen

| Operation | Stack | Bedeutung |
|---|---|---|
| fcmpg | ...,y,z ⇒ ...,k | k = 1 falls x > y, k = 0 falls x == y, k = -1 falls x < y |
| fcmpl | ...,y,z ⇒ ...,k | k = 1 falls x < y, k = 0 falls x == y, k = -1 falls x > y |

### A.2.5 Bitmanipulationen

| Operation | Stack | Bedeutung |
|---|---|---|
| iand | ...,i,j ⇒ ...,k | k = i & j |
| ior | ...,i,j ⇒ ...,k | k = i \| j |
| ixor | ...,i,j ⇒ ...,k | k = i ^ j |
| ishr | ...,i,j ⇒ ...,k | k = i » j |
| ishl | ...,i,j ⇒ ...,k | k = i « j |

### A.2.6 Sprünge

| Operation | Stack | Bedeutung |
|---|---|---|
| goto L | ... ⇒ ... | Sprung zu L |
| ifeq L | ...,i ⇒ ... | falls i == 0 Sprung zu L |
| iflt L | ...,i ⇒ ... | falls i < 0 Sprung zu L |
| ifle L | ...,i ⇒ ... | falls i <= 0 Sprung zu L |
| ifgt L | ...,i ⇒ ... | falls i > 0 Sprung zu L |
| ifge L | ...,i ⇒ ... | falls i >= 0 Sprung zu L |
| ifne L | ...,i ⇒ ... | falls i != 0 Sprung zu L |
| L: | ... ⇒ ... | die nächste Anweisung ist mit L markiert |

### A.2.7 Zuweisunge

| Operation | Stack | Bedeutung |
|---|---|---|
| `iload n` | `... ⇒ ...,j` | `j` wird vom Stackregister `n` geladen |
| `istore n` | `..., i ⇒ ...` | `i` wird im Stackregister `n` gespeichert |
| `fload n` | `... ⇒ ...,x` | `y` wird vom Stackregister `n` geladen |
| `fstore n` | `...,x ⇒ ...` | `x` wird im Stackregister `n` gespeichert |
| `aload n` | `... ⇒ ...,a` | Die Arrayreferenz `a` wird vom Stackregister `n` geladen |
| `astore n` | `...,a ⇒ ...` | Die Arrayreferenz `a` wird im Stackregister `n` gespeichert |
| `iaload n` | `...,a,i ⇒ ...,j` | `j` wird vom `a[i]` geladen |
| `iastore n` | `...,a,i,j ⇒ ...` | `j` wird an der Stelle `a[j]` gespeichert |
| `faload n` | `...,a,i ⇒ ...,x` | `x` wird vom `a[i]` geladen |
| `fastore n` | `...,a,i,x ⇒ ...` | `x` wird an der Stelle `a[j]` gespeichert |
| `sipush i` | `... ⇒ ...,i` | Die Konstante `i` wird geladen |

### A.2.8 Methode

| Operation | Stack | Bedeutung |
|---|---|---|
| `ireturn` | `...,i ⇒ ...` | `return(i)` |
| `freturn` | `...,x ⇒ ...` | `return(x)` |
| `return` | `... ⇒ ...` | `return()` |

Bei `ireturn` und `freturn` wird das oberste Stackelement von aktuellen Stack entfernt (siehe obige Tabelle) und auf dem aufrufenden Stack gelegt (in der obigen Tabelle nicht abgebildet).

## A.3 Einfaches Beispiel

Gegeben sei folgendes Programm:

```
{
  print(1);
  x = 1;
  x = x * 2;
  y = 3;
  z = x * y + 2;
  t = 3;
  print(z*z);
  print(1+(x--y)*z);
}
```

Eine mögliche Codierung des obigen Programms in JAVA Assembler:

```
.class public aOut
.super java/lang/Object

.method public <init>()V
aload_0
invokenonvirtual java/lang/Object/<init>()V
return
.end method

.method public static main([Ljava/lang/String;)V
sipush 1
invokestatic MCCLib/puti(I)V
sipush 10
invokestatic MCCLib/putc(I)V
sipush 1
istore 1
iload 1
sipush 2
imul
istore 1
sipush 3
istore 2
iload 1
iload 2
imul
sipush 2
iadd
istore 3
sipush 3
istore 4
iload 3
iload 3
imul
invokestatic MCCLib/puti(I)V
sipush 10
invokestatic MCCLib/putc(I)V
sipush 1
iload 1
iload 2
ineg
isub
iload 3
imul
iadd
invokestatic MCCLib/puti(I)V
sipush 10
invokestatic MCCLib/putc(I)V
sipush 1
```

```
iload 1
iload 2
ineg
isub
iload 3
imul
iadd
invokestatic MCCLib/puti(I)V
sipush 10
invokestatic MCCLib/putc(I)V
return
.limit locals 5
.limit stack 3
.end method
```

## A.4 Unterprogramme und globale Variablen

Will mann Unterprogramme verwenden, so müssen Sie in JASMIN-Assembler als statische Methoden deklariert werden. Notwendig bei der Prototypdeklaration ist die Angabe der Signatur (Die Signatur ist die Spezifikation der Ein- und Ausgabe Parameter einer Funktion (Methode)).

Eine ganze Zahl hat die Signatur `I`, ein ganzahliger Array hat die Signatur `[I`.

Eine ganzzahlige Funktion `f` mit einem ganzzahligen Parameter, einem ganzzahligen Array Parameter und einem String Parameter hat die Signatur `(I[ILjava/lang/String;)I`. Die JASMIN Deklaration dieser Funktion hat die Form:

```
.method static f(I[ILjava/lang/String;)I
```

Der Aufruf erfolgt mit:

```
invokestatic T7/f(I[ILjava/lang/String;)I
```

Dabei ist `T7` der Name der Klasse, wo `f` definiert wurde.

### A.4.1 Signaturen

Folgende Grammatik beschreibt den Aufbau von Signaturen:

```
methodDescriptor    ::= "(" parameterDescriptor* ")"
                        returnDescriptor
                     ;
parameterDescriptor ::= fieldType
                     ;
returnDescriptor    ::= fieldType | "V"
                     ;
fieldType           ::= baseType | objectType | arrayType
```

```
                    ;
baseType           ::= "B" | "C" | "D" | "F" |
                       "I" | "J" | "S" | "Z"
                    ;
objectType         ::= "L" className ";"
                    ;
arrayType          ::= "[" fieldType
                    ;
```

Dabei haben die sogenannte `FieldType`'s folgende Bedeutung:

| Operation | JAVA | Bedeutung |
|---|---|---|
| B | byte | signed byte [8] |
| C | char | character [8] |
| D | double | double-precision IEEE 754 float [64] |
| F | float | single-precision IEEE 754 float [32] |
| I | int | integer [32] |
| J | long | long integer [64] |
| LClassName; | ... | an instance of the class |
| S | short | signed short [16] |
| Z | boolean | true or false |

**Beispiel A.1 [Unterprogramm]]**

Aufruf eines Unterprogramms:

```
main ()
{
  int x;
  puti(f(3)); putc(10);
  x = getc();
}

int f (int n)
{
  if (n <= 1)
    return (1);
  else
    return(n*f(n-1));
}
```

Eine mögliche Codierung des obigen Programms in JAVA Assembler [1].

```
.source T7.mcc
.class public T7
.super java/lang/Object

.method public <init>()V
```

[1] Wir verwenden hier die JASMIN Notation [MD97].

```
    aload_0
    invokenonvirtual java/lang/Object/<init>()V
    return
.end method

.method public static main([Ljava/lang/String;)V
    sipush 3
    invokestatic T7/f(I)I
    invokestatic MCCLib/puti(I)V
    sipush 10
    invokestatic MCCLib/putc(I)V
    invokestatic MCCLib/getc()I
    istore 1 ; x
    return
.limit locals 2
.limit stack 1
.end method

.method static f(I)I
    iload 0 ; n
    sipush 1
    isub
    ifle L2
    bipush 0
    goto L3
  L2:
    bipush 1
  L3:
    ifeq L0
    sipush 1
    ireturn
    goto L1
  L0:
    iload 0 ; n
    iload 0 ; n
    sipush 1
    isub
    invokestatic T7/f(I)I
    imul
    ireturn
  L1:
    bipush 0
    goto L3
  L2:
    bipush 1
  L3:
    ifeq L0
    sipush 1
    ireturn
    goto L1
```

```
  L0:
    iload 0 ; n
    iload 0 ; n
    sipush 1
    isub
    invokestatic T7/f(I)I
    imul
    ireturn
  L1:
    bipush 0
    ireturn
.limit locals 1
.limit stack 3
.end method
```

### A.4.2 Globale Variablen

Will mann globale Variablen verwenden, so müssen diese Variablen als statische Felder deklariert werden.

**Beispiel A.2 [Ganzzahlige Variablen]** Deklaration einer globalen ganzzahligen Variable `i` mit Initialwert `22`:

```
.field static i I = 22
```

Lesen:

```
getstatic ClassName/i I
```

Schreiben;

```
putstatic ClassName/i I
```

**Beispiel A.3 [Ganzzahliges Array]**

Deklaration einer globalen ganzzahligen Array Variable `a[5]`:

```
.field static a [I
```

Initialisieren:

```
sipush 5
newarray int
putstatic ClassName/a [I
```

Lesen von `a[3]`:

```
getstatic ClassName/a [I
sipush 3
iaload
```

Schreiben von 77 auf a[3]:

```
sipush 3
sipush 77
iastore
```

## A.5 JAVA und Bytecode

Folgende Programme wurden in der Programmiersprache JAVA geschrieben. Sie wurden anschliessend kompiliert und mit javap disassembliert.

### A.5.1 Schlaufe

JAVA

```
class loop {

  public static void main (String[] args) {
    int i = 100;
    int j = 0;
    while (i > 0) {
      j = j + i;
      i = i - 1;
    }
  }
}
```

JAVA-Bytecode

```
Compiled from loop.java
synchronized class loop extends java.lang.Object
    /* ACC_SUPER bit set */
{
    public static void main(java.lang.String[]);
        /* Stack=2, Locals=3, Args_size=1 */
    loop();
        /* Stack=1, Locals=1, Args_size=1 */
}

Method void main(java.lang.String[])
   0 bipush 100
   2 istore_1
   3 iconst_0
   4 istore_2
   5 goto 16
   8 iload_2
   9 iload_1
```

```
  10 iadd
  11 istore_2
  12 iload_1
  13 iconst_1
  14 isub
  15 istore_1
  16 iload_1
  17 ifgt 8
  20 return

Method loop()
   0 aload_0
   1 invokespecial #3 <Method java.lang.Object()>
   4 return
```

### A.5.2 Auswahl

JAVA

```
class ifthenelse {

  public static void main (String[] args) {
    int i = 100;
    int j = 0;
    if (i > 0 && j <= 50) {
      j = j + i;
    }
    else {
      j = j + i;
    }
  }
}
```

JAVA-Bytecode

```
Compiled from ifthenelse.java
synchronized class ifthenelse extends java.lang.Object
    /* ACC_SUPER bit set */
{
    public static void main(java.lang.String[]);
        /* Stack=2, Locals=3, Args_size=1 */
    ifthenelse();
        /* Stack=1, Locals=1, Args_size=1 */
}

Method void main(java.lang.String[])
   0 bipush 100
   2 istore_1
   3 iconst_0
```

```
   4 istore_2
   5 iload_1
   6 ifle 20
   9 iload_2
  10 bipush 50
  12 if_icmpgt 20
  15 iload_2
  16 iload_1
  17 iadd
  18 istore_2
  19 return
  20 iload_2
  21 iload_1
  22 iadd
  23 istore_2
  24 return

Method ifthenelse()
   0 aload_0
   1 invokespecial #3 <Method java.lang.Object()>
   4 return
```

### A.5.3 Globale Variablen

JAVA

```
class global {

  static int i;
  static int a[] = new int[10];

  public static void main (String[] args) {
    i = 50;
    a[40] = 30;
  }
}
```

JAVA-Bytecode

```
Compiled from global.java
synchronized class global extends java.lang.Object
    /* ACC_SUPER bit set */
{
    static int i;
    static int a[];
    public static void main(java.lang.String[]);
        /* Stack=3, Locals=1, Args_size=1 */
    global();
        /* Stack=1, Locals=1, Args_size=1 */
```

```
    static static {};
        /* Stack=1, Locals=0, Args_size=0 */
}

Method void main(java.lang.String[])
   0 bipush 50
   2 putstatic #5 <Field int i>
   5 getstatic #4 <Field int a[]>
   8 bipush 40
  10 bipush 30
  12 iastore
  13 return

Method global()
   0 aload_0
   1 invokespecial #3 <Method java.lang.Object()>
   4 return

Method static {}
   0 bipush 10
   2 newarray int
   4 putstatic #4 <Field int a[]>
   7 return
```

### A.5.4 Lokale Variablen

JAVA

```
class local {

  public static void main (String[] args) {
    int i;
    int a[] = new int[10];
    i = 50;
    a[40] = 30;
  }
}
```

JAVA-Bytecode

```
Compiled from local.java
synchronized class local extends java.lang.Object
    /* ACC_SUPER bit set */
{
    public static void main(java.lang.String[]);
        /* Stack=3, Locals=3, Args_size=1 */
    local();
        /* Stack=1, Locals=1, Args_size=1 */
}
```

```
Method void main(java.lang.String[])
   0 bipush 10
   2 newarray int
   4 astore_2
   5 bipush 50
   7 istore_1
   8 aload_2
   9 bipush 40
  11 bipush 30
  13 iastore
  14 return

Method local()
   0 aload_0
   1 invokespecial #3 <Method java.lang.Object()>
   4 return
```

## A.6 Aufgaben

**Aufgabe A.1 [Desassemblieren]** *Schreiben Sie kleinere* JAVA *Programme und desassemblieren Sie den Code mit* `javap`*. Analiesieren Sie anschliessend den Assembler-Code.*

**Aufgabe A.2 [Schlaufe]** *Analysieren Sie die Assembler-Umsetzung der* `while`*-Instruction in* JAVA*.*

# Anhang B

# Die Programmiersprache MCC

Wir wollen nun zeigen, wie ein Compiler für MCC, eine Teilmenge der Programmiersprache C implementiert werden kann. Wir werden dabei JAVA Byte Code generieren.

Die Implementierung des Parsers basiert zum teil auf [App98]

**Bemerkung B.1 [Vereinfachung]** Programmausschnitte sind sind zum Teil stark vereinfacht

## B.1 MCC

### B.1.1 Token-Deklarationen

```
LETTER     ::= ["a"-"z","A"-"Z","_"]
    ;
DIGIT      ::= [ "0"-"9"]
            ;

INTEGER    ::= <DIGIT> (<DIGIT>)*
            ;

STRING     ::= "\""
               (
                 (~["\"","\\","\n","\r"])
               |
                 (
                   "\\"
                   (
                     ["n","t","b","r","f","\\","'","\""]
                   |
                     ["0"-"7"] ( ["0"-"7"] )?
                   |
                     ["0"-"3"] ["0"-"7"] ["0"-"7"]
                   )
                 )
               )*
               "\""
```

```
            ;

IDENTIFIER ::= <LETTER> (<LETTER>|<DIGIT>)*
               ( "." <LETTER> (<LETTER>|<DIGIT>)* )*
            ;
```

### B.1.2 Grammatik

```
mccProg     ::= (varDec)*
                (funcProtDec)*
                (funcDec)*
             ;

varDec      ::= simpleTy() <ID>
                ("[" <INT> "]")? ";"
             ;

simpleTy    ::= "int" | "void" | "text"
             ;

funcProtDec ::= ("import")? simpleTy <ID> "(" typeList ")" ";"
             ;

typeList    ::= simpleTy ("[" "]")? ("," simpleTy ("[" "]")?)*
             ;

funcDec     ::= simpleTy <ID> "(" funcParam ")" block
             ;

funcParam   ::= simpleTy <ID> ("[" (<INT>)? "]")?
                ("," simpleTy <ID> ("[" (<INT>)? "]")?)*
             ;

block       ::= "{" (varDec)* (stat)* "}"
             ;

stat        ::=   <ID> (funArgs | "[" addExp "]")? "=" addExp ";"
                |
                  retStat
                |
                  whileStat
                |
                  ifStat
                |
                  block
             ;

whileStat   ::= "while" "(" orExp ")" stat
             ;
```

```
ifStat      ::= "if" "(" orExp ")" stat ("else" stat)?
                ;

retStat     ::= "return" (addExp)? ";"
                ;

orExp       ::= andExp ("||" andExp)*
                ;

andExp      ::= relExp ("&&" relExp)*
                ;

relExp      ::= addExpr
                (">" | ">=" | "==" | "!=" | "<" | "<=")
                addexpr
                ;

addExp      ::= mulExpr (("+" | "-") mulExpr)*
                ;

mulExp      ::= negExpr (("*" | "/") negExpr)*
                ;

negExp      ::= ("-" | "!")? priExp
                ;

priExp      ::=   literal
                |
                      "(" orExp ")"
                |
                  <ID> ("(" (funArgs)? ")" | "[" addExp "]")?
                ;

literal     ::= <INT> | <TEXT>
                ;

funArgs     ::= addExp ("," addExp)*
                ;
```

### B.1.3 Semantik

Die Semantik von MCC basiert im Wesentlichen auf derjenigen der Programmiersprache C (Siehe [KR90]). Die Unterschiede zu C sehen wie folgt aus:

- Jede Funktion muss zuerst als Prototyp deklariert werden. Dies ist keine Einschränkung, vereichfacht aber den Aufbau des Compilers.
- Sofern überhaupt JAVA Bytecode generiert wird, dürfen `static`JAVA Funktionen importiert werden.

- Jeder Ausstieg aus einer `void` Funktion muss mittels `return` stattfinden.
- Arrays sind eindimensional.

### B.1.4 Programm Beispiel

Folgendes MCC Programm ist eine Implementierung des Quicksort Algorithmus:

```
int r;
int i;
int x[50];

void main(text[]);
int quick(int[], int, int);
int part(int[], int, int);

import void MCCLib.puti(int);
import void MCCLib.putc(int);
import void MCCLib.putm(text);

void main(text args[])
{
  x[0] = 33; x[1] = 25; x[2] = 7; x[3] = 99; x[4] = -5;
  x[5] = 34; x[6] = 26; x[7] = 8; x[8] = 100; x[9] = -4;
  x[10] = 35; x[11] = 27; x[12] = 9; x[13] = 101; x[14] = -3;
  MCCLib.putm("before quicksort :");
  MCCLib.putc(10);
  i = 0;
  while (i < 15) {
    MCCLib.puti(x[i]); MCCLib.putc(10);
    i = i + 1;
  }
  MCCLib.putm("after quicksort :");
  MCCLib.putc(10);
  r = quick(x, 0, 14);
  i = 0;
  while (i < 15) {
    MCCLib.puti(x[i]); MCCLib.putc(10);
    i = i + 1;
  }
  return;
}

int quick(int a[], int left, int right)
{
  int mid;
  int r;
  int i;

  if (left < right) {
```

```
    mid = part(a, left, right);
    r = quick(a, left, mid - 1);
    r = quick(a, mid + 1, right);
  }
  return (0);
}

int part(int a[], int left, int right)
{
  int temp;
  int i;
  int j;
  int aktuell;

  i = (left + right) / 2;
  if (i != right) {
    temp = a[i];
    a[i] = a[right];
    a[right] = temp;
  }
  aktuell = a[right];
  i = left - 1;
  j = right;
  while(j > i)
  {
    int t;
    i = i + 1;
    while(a[i] < aktuell) {
      i = i + 1;
   }
    j = j - 1;
    t = 1;
    while (t == 1) {
      if (j >= 0) {
        if (a[j] > aktuell) j = j - 1;
        else t = 0;
      }
      else t = 0;
    }
    if (i < j) {
      temp = a[i];
      a[i] = a[j];
      a[j] = temp;
    }
  }
  if (i < right)
  {
    temp = a[i];
    a[i] = a[right];
    a[right] = temp;
```

```
  }
  return (i);
}
```

Man beachte die `import` Anweisungen, die es ermöglichen, `static`JAVA Methoden anzubinden.

## B.2 Aufbau des Compilers

In einer ersten phase wird die Syntax mit JAVACC analysiert. Ist die Syntax korrekt, so wird dabei eiene abstrakten (AST) Syntaxbaum aufgebaut.

In einer zweiten Phase wird die semantische Analyse (Typüberprüfung und Verifizierung der Deklarierheitseigenschaft) durchgeführt und gleichzeitig (falls keine semantischen Fehler) JASMIN Assembler Code erzeugt. Diese Aufgabe wird von der Besucherklasse Direct Code Generation wahrgenommen.

### B.2.1 AST

Die abstrakte Syntax von `mcc` sieht wie folgt aus:

```
mccProg          ::= (varDec)*
                     (funcProtDec)*
                     (funcDec)*
                   ;

varDec           ::= ty() var
                   ;

ty               ::= simpleTy | arrayTy
                   ;

simpleTy         ::= "int" | "void" | "text"
                   ;

arrayTy          ::= simpleTy <INT>
                   ;

var              ::= simpleVar | subscriptVar
                   ;

funcProtDec      ::= ("import")? ty <ID> (ty)*
                   ;

funcDec          ::= simpleTy <ID> (varDec)* block
                   ;

block            ::= (varDec)* (stat)*
```

```
                ;

stat            ::=   assignStat
                    |
                      retStat
                    |
                      whileStat
                    |
                      ifStat
                    |
                      block
                    |
                      expStat
                ;

assignStat      ::= var exp
                ;

retStat         ::= (exp)?
                ;

whileStat       ::= exp stat
                ;

ifStat          ::= exp stat (stat)?
                ;

expStat         ::= exp
                ;

exp             ::=   exp
                      (
                        "||"
                      |
                        "&&"
                      |
                        ">"
                      |
                        ">="
                      |
                        "=="
                      |
                        "!="
                      |
                        "<"
                      |
                        "<="
                      |
                        "+"
                      |
```

```
                        "-"
                      |
                        "*"
                      |
                        "/"
                      )
                      exp
                    |
                      "-" exp
                    |
                      "!" exp
                    |
                      <INT>
                    |
                      <TEXT>
                    |
                      callExp
                    |
                      simpleVarExp
                    |
                      subscriptVarExp
                  ;

callExp          ::= <ID> (exp)*
                  ;
```

### B.2.2 Typen

```
public class INT extends Type {

  public String signature(){}
}

public class VOID extends Type {

  public String signature() {}
}

public class ARRAY extends Type {

  public Type element;
  public int size;

  public String signature(){}
}
```

### B.2.3 Die Symboltabelle

Die Symboltabelle wird im wesentlichen verwendet, um alle benutzerdefinierte Namen zu verwalten. Sie ist in Form einer Hashtabelle implementiert. Als Schlüssel werden `Symbol`

Objekte verwendet. Gespeichert werden Variablen (VAR Objekte) oder Funktionen (FUNC Objekte). Siehe [App98] für mehr Details.

### Die Klasse Table

```
public class Table {

  public Object get(Symbol key){}

  public void put(Symbol key, Object value){}

  public void beginScope(){}

  public void endScope(){}

}
```

Die Symboltabelle kann auch die vom Compiler erzeugte Hilfsvariablen und Labels verwalten. Konstanten werden auch in der Symboltabelle eingetragen.

### Die Klassen VAR und FUNC

Einträge in der Symboltabelle sind Entry Objekte. Entry Objekte sind entweder Variablen oder Funktionen.

Eine Variable kennt ihren Namen, ihren Typ, ihre Adresse und weiss in welcher Funktion sie deklariert wurde:

```
public class VAR extends Entry {
public boolean global = false;

public Symbol symbol;

public Type type;

public FUNC func;

public int address;

}
```

Eine Funktion kennt ihre Parameter, ihre Ausgabetyp sowie alle in ihr deklarierten Variablen:

```
public class FUNC extends {

  public Symbol symbol;

  public VARList paramList;
```

```
  public Type result;

  public VARList varList;

  public TEMPList tempList;

  public TypeList formals;

}
```

## B.3 Direkte Codegenerierung

Die direkte Codegenerierung erfolgt mit Hilfe einer Besucherklasse für den abstrakten Ableitungsbsaum. Falls nötig, erhalten die `visit` Methoden des Besuchers den Typ des Unterbaumes mittels Rückgabewert. Sie teilen dem Unterbaum mit, wo gesprungen werden muss mittels Übergabeparameter.

Es folgen nun exemplarisch einige `visit` methoden.

### B.3.1 Deklarationen

```
public Object visitVarDec(VarDec a, Object o) {
  Type type = (Type) a.type.accept(this, o);
  VAR var = new VAR(a.name, type, address++,
    func, global, param);
  table.put(a.name, var);
  if (global) {
    em(".field static " + a.name.toString() +
       " " + type.signature());
  }
  return type;
}
```

Hier geht es vor allem um die Ermittlung des Typs sowie um den Eintrag einer Variable in die Symboltabelle. Ist die Variable global, so wird sie auch dementsprechend deklariert.

### B.3.2 Arithmetische Operationen

```
public Object visitAddExp(AddExp a, Object o) {
  TrueFalse tf = (o != null ? (TrueFalse) o : null);
  Type it1 = (Type) a.arg1.accept(this, tf);
  Type it2 = (Type) a.arg2.accept(this, tf);
  Type.checkSignature(new INT(), it1, a.arg1.pos,
      it2, a.arg2.pos);
  emt("iadd");
  as(-1);
  return new INT();
}
```

Beim Aufruf der `visit` Methoden der beiden Unterbäume wird erstens ihr Code generiert und zweitens ihr Typ zurückgegeben. Die Typkompatibilität wird überprüft und anschliessend wird Code fur die Addition generiert. Diese Methode gibt den Typ der Addition als Resultat zurück.

### B.3.3 Boolsche Operationen

```
public Object visitAndExp(AndExp a, Object o) {
  TrueFalse tf = (o != null ? (TrueFalse) o : null);
  LABEL l = new LABEL();
  Type it1 = (Type) a.arg1.accept(this,
  new TrueFalse(l, tf.f));
  eml(l.name);
  Type it2 = (Type) a.arg2.accept(this, tf);
  Type.checkSignature(new BOOLEAN(), it1, a.arg1.pos,
      it2, a.arg2.pos);
  return new BOOLEAN();
}
```

Bei einem Boolschen Ausdruck wird eigentlich kein Code generiert. Es werden nur Labels generiert und gesetzt. Den beiden Unterbäumen wird mitgeteilt, wohin gesprungen werden muss.

### B.3.4 Auswahl

```
public Object visitIfStat(IfStat a, Object o) {
  LABEL ltc = new LABEL();
  LABEL lec = new LABEL();
  LABEL len = new LABEL();
  Type t = (Type) a.test.accept(this,
new TrueFalse(ltc, lec));
  eml(ltc.name);
  Type tc = (Type) a.thenclause.accept(this, o);
  emt("goto " + len.name);
  eml(lec.name);
  if (a.elseclause != null) {
    Type ec = (Type) a.elseclause.accept(this, o);
  }
  eml(len.name);
  return null;
}
```

Bei der Auswahl werden 3 Labels erzeugt. `then` und `else` Labels werden dem Testausdruck als Parameter übergeben. Anschliessend wird das `then` Label gesetzt und Code für das `then` Teil generiert. Falls nötig erfolgt dasselbe für das `else` Teil. Hier erfolgt keine Typüberprüfung.

### B.3.5 Zugriff auf Variable

```
public Object visitSimpleVarExp(SimpleVarExp a, Object o) {
  Object v = table.get(a.name);
  if (v == null) {
    error();
  } else {
    VAR var = (VAR) v;
    if (var.global) {
      emt("getstatic A/" + var.name + " " +
  var.type.signature());
    } else if (var.type instanceof ARRAY) {
      emt("aload " + var.address);
    } else {
      emt("iload " + var.address);
    }
    as(1);
    return var.type;
  }
}
```

Hier wird zuerst verifiziert, ob die Variable deklariert wurde. Anschliessend wird je nach Typ Code generiert. Das Typ der variable wird turückgegeben.

### B.3.6 Funktionsaufruf

```
public Object visitCallExp(CallExp a, Object o) {
  Type params = (Type) a.args.accept(this, o);
  Object fo = table.get(a.func);
  Type result;
  if (fo == null) {
    error();
  } else {
    FUNC f = (FUNC) fo;
    result = f.result;
    Type.checkSignature(params, f.formals, a.pos);
    emt("invokestatic " + f.name + f.signature());
    as(1 - f.formals.length());
    return result;
  }
  return result;
}
```

Hier wird zuerst verifiziert, ob die Funktion deklariert wurde. Anschliessend wird die Signatur überprüft. Da die virtuelle JAVA Maschine das grösste Teil der Methodenverwaltung übernimmt ist die Code Generierung besonders einfach.

## B.4 Zwischencode Generierung

Der MCC Compiler kann auch Quadrupel als Zwischencode Erzeugen. Die Zwischencode befehle sind stark an den Namen der JAVA Bytecode Befehle angelehnt:

```
Expressions

  AND OR NOT IADD ISUB IMUL IDIV INEG

Conditional jumps

  IFLE IFLT IFEQ IFNE IFGE IFGT

Jumps

  GOTO

Memory Operations

  ICOPY ISTOA IREFA

Function Calls

  PARAM CALL IRET RET

Utilities

  METH EMETH HALT
```

### B.4.1 Die Quadrupel

Ein Quadrupel besteht aus einem Operator, zwei Argumenten und ein Resultat

```
public abstract class Quad {

  public QuadEntry arg1 = null;

  public QuadEntry arg2 = null;

  public QuadEntry res = null;

  public String op = null;

  public LABELList labelList = null;

  public void setLabel(LABEL l) {}
}
```

`arg1`, `arg2` und `res` sind im Wesentlichen `VAR` oder `LABEL` Objekte.

### B.4.2 Zwischencode Generierung

Die Zwischencode Generierung ist der direkten Code Generierung seht ähnlich. Es müssen dabei aber keine Adressen für Variablen berechnet werden, da die Quadrupel nachträglich optimiert werden können.

Es folgt nun exemplarisch die `visit` Methode für arithmetische Ausdrücke.

```
public Object visitAddExp(AddExp a, Object o) {
  TrueFalse tf = (o != null ? (TrueFalse) o : null);
  QuadType it1 = (QuadType) a.arg1.accept(this, tf);
  QuadType it2 = (QuadType) a.arg2.accept(this, tf);
  Type.checkSignature(new INT(), it1.type, a.arg1.pos,
      it2.type, a.arg2.pos);
  ITMP res = new ITMP(table,func);
  quadTable.insertQuad(new IADD(it1.entry, it2.entry,
res));
  return new QuadType(res, new INT());
}
```

Der Wesentliche Unterschied liegt in der Verwendung von Hilfsvariablen (`TEMP`). Für das Resultat wird immer eine Hilfsvariable generiert. Die `accept` Methoden der Argumente liefern neu den Typ und die von ihnen erzeugten Hilfsvariablen (in `QuadType` Objekten verpackt) als Rückgabewert.

## B.5 Backend

Die Klasse `Generate` ist für die JAVA Bytecode Erzeugung verantwortlich. Die wichtigste Methode dieser Klasse ist `generate`. `generate` liest die Quadrupel Tabelle und generiert entsprechende Bytecode Instruktionen. Im Wesentlichen wird bei arithmetischen Operationen ein Quadrupel `[op,arg1,ar2,res]` in folgende Folge von Instruktionen übersetzt: `push arg1`, `push arg2`, `op` und `pop res`.

### B.5.1 push

```
public void push(QuadEntry e) {
  if (e instanceof ICST || e instanceof TCST) {
    emit("    ldc " + e.name);
    as(1);
  } else if (e instanceof VAR) {
    VAR var = (VAR) e;
    if (var.global) {
      emit("    getstatic A/" + var.name + " " +
   var.type.signature());
    } else {
      if (var.type instanceof ARRAY) {
emit("    aload " + var.address + "\t ;; " +
     var.name);
```

```
      } else {
emit("    iload " + var.address + "\t ;; " +
     var.name);
      }
    }
  }
}
```

### B.5.2 pop

```
public void pop(QuadEntry e) {
  VAR var = (VAR) e;
  if (var.global) {
    emit("    putstatic A/" + var.name + " " +
 var.type.signature());
  } else {
    emit("    istore " + var.address + "\t ;; "
 + var.name);
  }
  as(-1);
}
```

### B.5.3 generate

```
public void generate() {
  // generate the target class haeder
  // generate fields declarations
  // generate the init method
  // generate clinit for global arrays
  // generate java methods (functions)
  QuadList actual = table.head;
  while (actual != null) {
    Quad q = actual.quad;
    // set labels
    if (q instanceof METH) {
      // method
    } else if (q instanceof EMETH) {
      // end method
    } else if (q instanceof INEG || q instanceof IADD ||
       q instanceof ISUB || q instanceof IMUL ||
       q instanceof IDIV) {
      // code for arithmetic quadruple
      push(q.arg1);
      push(q.arg2);
      emit("    " + q.op);
      pop(q.res);
    } else if (q instanceof ICOPY) {
      push(q.arg1);
      pop(q.res);
```

```
    } else if (q instanceof GOTO) {
      emit("    goto " + q.res.name);
    } else if (q instanceof IFEQ || q instanceof IFNE ||
       q instanceof IFGT || q instanceof IFGE ||
       q instanceof IFLT || q instanceof IFLE) {
      push(q.arg1);
      push(q.arg2);
      emit("    isub");
      emit("    " + q.op + " " + q.res.name);
    } else if (q instanceof IREFA) {
      push(q.arg1);
      push(q.arg2);
      emit("    iaload");
      pop(q.res);
    } else if (q instanceof ISTOA) {
      push(q.arg1);
      push(q.arg2);
      push(q.res);
      emit("    iastore");
    } else if (q instanceof IRET) {
      push(q.arg1);
      emit("    ireturn");
    } else if (q instanceof RET) {
      emit("     return");
    } else if (q instanceof PARAM) {
      push(q.arg1);
    } else if (q instanceof CALL) {
      FUNC f = (FUNC) q.arg1;
      emit("    invokestatic " + f.name + f.signature());
      pop(q.res);
    } else if (q instanceof HALT) {
    }
    actual = actual.tail;
  }
}
```

## B.6 Aufgaben

**Aufgabe B.1 [mcc]** *Unter `prog/mc/mcc-script.jar` befindet sich ein Parser für die Programmiersprache* MCC.

*Die Grammatik soll nun um weitere Konstrukte der Programmiersprache* C *erweitert werden: `for`, `dowhile`, `break`. Ferner sollten auch mehrdimensionale Arrays unterstützt werden.*

*Dabei sollen der Parser sowie der Parse-Baum und die Visitoren entsprechend angepasst werden. Arbeiten Sie zuerst mit dem `IdentityVisitor`. Dies kann beim Testen sehr hilfreich sein: Übersetzen vom Programm X nach X', anschliessend übersetzen von X' nach X''. Jetzt müssen X' und X'' vollständig identisch sein sonst stimmt mit Sicherheit bei der*

*Übersetzung etwas nicht.*

**Aufgabe B.2 [Bytecode]** *Erzeugen Sie direkt* JAVA *Bytecode mit Parser aus Aufgabe B.1 (ohne Verwendung der Visitor Klassen).*

**Aufgabe B.3 [Erweiterungen]** *Implementieren Sie folgende Erweiterungen des Parsers aus Aufgabe B.1.*

1. *Erweiterung der Sprache um weitere Datentypen und Operationen darauf (z.B.* `float`, `boolean`, `set`*)*
2. *Implementation von* `computeBasicBlocks()`
3. *Implementation einer peephole optimization* `cleanQuads()`
4. *Berechnen der next use Information* `computeN()`
5. *Erweiterung der Sprache um weitere Elemente (*`switch`, `for`*-Schleife, Schleifenkontrollanweisungen* `break`/`continue`*, Bedingte Zuweisung =?:, Konkatenationsoperator + für Strings, Konstanten)*
6. *Implementation eines* MCC*-Interpreters.*
7. *Implementation der semantischen Analyse. Z.B.:*
   (a) *Sind die Variablen initialisert bevor sie benutzt werden?*
   (b) *Sind Konstanten im Zulässigen Bereich (Bei Array Indices)?*
   (c) *Warnung bei nicht benutzten Variablen / Funktionen.*
8. *Optimierung: Auswerten konstanter Ausdrücke beim Compilieren.*

# Anhang C

# Gruppenarbeit

## C.1 Aufgabenstellung

### C.1.1 Umfeld

Im Kapitel B wurde gezeigt, wie der Compiler MCC funktionieren und aufgebaut ist. In den Kapiteln 2 und 3 wurde ein Parser-Generator (in Form eines Projektes) entworfen. Wir wollen diesen Parser-Generator MPG (Mini Parser Generator) nennen.

Beide sind ziemlich rudimentär. Dies ergibt die Möglichkeit vieler interessanter Erweiterungen.

#### Aufgabe

In den Abschnitten C.2 und C.3 befinden sich je drei Aufgaben mit einem Muss- und einem Sollziel. Lösen Sie eine dieser Aufgaben nach freien Wahl.

Einige Aufgaben benötigen möglicherweise Informationen, die sich im Skript nicht befinden. Verwenden Sie zu diesem Zweck die Literaturliste am Schluss dieses Dokumentes. Die meisten Information finden Sie in den Büchern von Aho, Sethi und Ullmann [ASU92a], [ASU92b], [ASU07], [ALSU08] sowie von Appel [App98].

#### Dokumentation

Erstellen Sie für Ihre Aufgabe eine kleine Dokumentation (Grössenordnung 8-10 A4 Seiten). Die Dokumentation soll sich auf die Änderungen, bzw. Erweiterungen des Compilers oder Parser-Generators beschränken.

Code Änderungen, bzw. Erweiterungen müssen zusätzlich anständig mit JAVADOC dokumentiert sein.

#### Administratives

- Abgabetermin: Samstagtag 26. Juli 2008, 24:00 Uhr per Email an `mailto:pierre.fierz@bfh.ch`

- Für das Erreichen der Maximalnote müssen sowohl die Mussaufgabe als auch die Sollaufgabe gelöst werden. Wird nur die Mussaufgabe gelöst, so können höchstens 52 Punkte (aus 60) erreicht werden!
- Die Dokumentation wird in PDF-Format abgegeben.
- Ihre Software wird in Form eines ECLIPSE-Projektes abgegeben (JAR-Datei).
- Ihr ECLIPSE-Projekte muss ein Verzeichniss `test` enthalten, mit allen Testbeispielen ihrer Arbeit.
- Für die Softwareentwicklung dürfen nur ECLIPSE EUROPA, JAVA 1.6, JAVACC 4.0 und JASMIN 2.3 verwendet werden. Einzige Ausnahme ist die Assembler Aufgabe.

## C.2 MCC

### C.2.1 Umfeld

Im Kapitel B wurde gezeigt, wie MCC. ein Compiler für eine Teilmenge von C konstruiert werden kann.

Der implementierte Compiler MCC ist ziemlich rudimentär. Dies ergibt die Möglichkeit vieler interessanter Erweiterungen.

Unter `prog/mc/mcc-script.jar` befindet sich ein ECLIPSE Projekt im JAR-Format mit dem Quellcode von MCC. Im Verzeichnis TEST sind auch einige einfache MCC-testprogramme enthalten.

**Bemerkung C.1 [JRE]** MCC generiert JASMIN-Code. JASMIN ist in der Schule installiert auf `/hti/apps/java` und ist auch im ECLIPSE-Projekt vorhanden. Bei geeigneter Konfiguration, kann der vom MCC erzeugte Code (unter Verwendung von `java.lang.reflect` direkt in der ECLIPSE-Umgebung ausgeführt werden. Dies war bei mir aber nur möglich, wenn alle JAVA mit der Version 1.4.2 des JAVA-Compilers kompiliert wurden. Wird die Version 1.5 verwendet, so muss das erzeugte Code separat ausgeführt werden, sonst wirft JAVA eine Reflection Exception (dies kann auch umgegangen werden, indem die Version-Nummer der Klasse `MCCLib` manuell geändert wird).

Lösen Sie eine der folgenden Aufgaben:

### C.2.2 Aufgabe: Assembler

MCC erzeugt zur Zeit JASMIN Assembler (JAVA Assembler).

#### Mussziele

Schreiben die die Methode `Generate` im Package `cb.mcc.backend.java` so um, dass direkt Maschinencode generiert werden kann. Sie dürfen dabei die Assemblersprache selber wählen (z.B. INTEL Prozessor, SPARC Prozessor, MOTOROLLA Prozessor, LEGO MINDSTORM Prozessor, GNU Assembler).

Versuchen Sie dabei, die Register Ihres Prozessors möglichst optimal auszunutzen.

#### Sollziele

Wird eine Andere Zielsprache als JAVA Assembler erzeugt, so kann MCC nicht mehr ohne weiteres JAVA-Klassen importieren (und verwenden). Erweiterm Sie MCC so, dass in Ihrem Fall Native-Funktionen (z.B. C-Funktionen) importiert werden können.

### C.2.3 Aufgabe: Spracherweiterung

#### Mussziele

Erweitern Sie MCC mit folgenden Konstrukten:

- `for` Konstrukt
- `do while` Konstrukt
- `break` und `continue` Instruktion für beide obige Konstrukte
- Multidimensionale Arrays

Schreiben Sie für jede Erweiterung entsprechende Testbeispiele.

**Bemerkung C.2 [Vorgehen]** Der Parser, der Parse-Baum und die Visitoren sollten zuerst dementsprechend angepasst werden. Arbeiten Sie mit `IdentityVisitor`. Dies kann beim Testen sehr hilfreich sein: Übersetzen vom Programm `X` nach `X'`, anschliessend übersetzen von `X'` nach `X"`. Jetzt müssen `X'` und `X"` vollständig identisch sein sonst stimmt mit Sicherheit bei der Übersetzung etwas nicht.

**Bemerkung C.3 [Arrays]** Für die Verarbeitung von multidimensionalen Arrays brauchen Sie zusätzliche JAVA Bytecode Instruktionen (z.b. `aaload`). Informationen dazu finden Sie auf `http://java.sun.com` unter *The Java Virtual Machine Specification* (`http://java.sun.com/docs`

**Bemerkung C.4 [Tip]** Falls Sie Schwierigkeiten haben, können Sie zuerst ein kleines JAVA Programm mit diesen Instruktionen schreiben, kompilieren und anschliessend mit JAVAP desassemblieren.

#### Sollziele

Erweitern Sie MCC zusätzlich mit den Datentypen `float` und `boolean`.

### C.2.4 Aufgabe: Quadrupel

Bei der Erzeugung von Quadrupel könnte noch einiges optimiert werden, insbesondere sind viele Hilfsvariablen unnötig und viele Sprünge nicht optimal.

**Mussziele**

1. Implementation von `computeBasicBlocks()`
2. Implementation einer peephole optimization `cleanQuads()`
3. Berechnen der next use Information `computeN()`

**Bemerkung C.5 [Tests]** Schreiben Sie entsprechende Testbeispiele und vergleichen Sie ihre ausführung mit und ohne Peephole Optimierung.

**Sollziele**

Auch nach der Peephole-Optimierung kommt es vor, dass im generierten JASMIN-Code Hilfsvariablen (und sogar benutzerdefinierte Variablen) genau einmal geschrieben werden, unmittelbar nachher auf dem Evaluationsstack gelegt werden und sonst nie mehr verwendet werden. Solche Variablen können eliminiert werden (Folge `istore iload`).

Versuchen Sie, eine entsprechende Optimierung in der Methode `Generate` im Package `cb.mcc.backend.java` einzubauen.

## C.3 MPG

### C.3.1 Umfeld

In den Kapiteln 2 und 3 wurde gezeigt, wie MPG, ein einfacher Parser-Generator, konstruiert werden kann.

Die Implementierung eines Parser-Generators ist eine interessante Übung:

- Dies beinhaltet eine Repetition der ganzen Theorie, insbesondere ein gutes Verständnis der BNF-Notation und der Überprüfung der LL(1) eigenschaft
- Die Code-Generierung ist relativ einfach weil die Zielsprache eine höhere Programmiersprache (JAVA) ist.

er implementierte Parser-generator MPG ist ziemlich rudimentär. Dies ergibt die Möglichkeit vieler interessanter Erweiterungen.

Unter `prog/mp/mpg.jar` befindet sich ein ECLIPSE Projekt im JAR-Format mit dem Quellcode von MPG. Im Verzeichnis `test` sind auch einige einfache Grammatiken enthalten.

Lösen Sie eine der folgenden Aufgaben:

### C.3.2 Aufgabe: EBNF

MPG unterstützt nur LL(1) Grammatiken in BNF-Notation. Erweitern Sie MPG so, dass die EBNF-Notation auch unterstützt wird.

Zur Lösung dieser Aufgabe werden alle spezielle EBNF-Konstrukte ( `(..)*`, `(..)+`, `(..)?` und `(..)`, mit einem neuen Namen versehen und durch die entsprechende BNF-Notation ersetzt. Dabei gibt es grunsätzlich zwei Strategien:

1. Man kann zu diesem Zweck mit einem Preprozessor (analog zu JJTree) arbeiten
2. Die Übersetzung EBNF zu BNF erfolgt dynamisch beim Parsen der Eingabe.

In beiden Fällen soll bei einem Fehler in der Grammatik, z.B. beim Nichterfüllen der LL1(1)-Eigenschaft eine möglichst präzise Fehlermeldung (mit Zeilenangabe) erzeugt werden (z.B. wie bei JavaCC).

#### Mussziele

1. Erweitern Sie MPG für die EBNF-Notation.
2. Schreiben Sie ein Programm für die Transformation einer LL(1)-Grammatik in EBNF-Notation in eine äquivalente Grammatik in BNF-Notation.
3. Entwerfen Sie einige Grammatiken zum Testen ihres programms, insbesondere:
   - EBNF Grammatik
   - Eine Grammatik mit einem First/First Konflikt.
   - Eine Grammatik mit einem First/Follow Konflikt.
   - Eine Grammatik mit einer Linksrekursion.

   Vergleichen Sie dabei Ihre Fehlermeldungen mit derejenigen von JavaCC.

#### Sollziele

MPG verwendet eienen eigenen Parser (handgeschrieben). Es wäre schöner, wenn MPG zu diesem Zweck MPG verwenden würde. Schreiben Sie MPG dementsprechend um.

Zu diesen Zweck müssen Sie Ihre Aufgabe zuerst mit dem handeschriebenen Parser lösen, und anschliessend den generierten Parser mit semantischen Aktionen erweitern.

### C.3.3 Aufgabe: Scannerzustände

Im Vergleich zu JavaCC sind die Tokendeklarationen in MPG vereinfacht. Insbesondere unterstützt MPG keine lexikalische Zustände.

#### Mussziele

1. Erweitern Sie die Klasse `TokenManager` so, dass Scannerzustände (wie in JavaCC) unterstützt werden. Zustandswechsel sollten dabei auch vom Parser aus möglich sein (`switchTo()`).

#### Sollziele

MPG verwendet eienen eigenen Parser (handgeschrieben). Es wäre schöner, wenn MPG zu diesem Zweck MPG verwenden würde. Schreiben Sie MPG dementsprechend um.

Zu diesen Zweck müssen Sie Ihre Aufgabe zuerst mit dem handeschriebenen Parser lösen, und anschliessend den generierten Parser mit semantischen Aktionen erweitern.

### C.3.4 Aufgabe: AST

Wird MPG mit semantischen Aktionen erweitert, so wird die (reine) BNF Notation für die Produktionen unnötig verschmutzt. Will man trotzdem semantische Aktionen ausführen, so gibt es einen eleganten Ausweg mittels AST. Der generierte Parser soll so modifiziert werden, dass bei seiner Ausführung ein AST mit zugehöriger Visitor Klasse generiert wird. Semantische Aktionen können dann (sauber) in einem geeignetem Visitor implementiert werden.

#### Mussziele

Erweitern Sie MPG mit AST Generierung. Jede AST Instruktion soll wie folgt aussehen: `#Node(3)` bedeutet, dass ein Knoten mit Namen `Node` und `3` Unterknoten erzeugt werden muss. `#Node` bedeutet dass ein Blattknoten mit Inhalt Wert des vorherigen Tokens erzeugt werden.

Es müssen alle für AST Klassen, eine Visitor Schnittstelle sowie eine `IdentityVisitor` Methode Erzeugt werden. Der `IdentityVisitor` transformiert die Eingabe in eine bis auf Whitespaces äquivalente Darstellung.

#### Sollziele

MPG verwendet eienen eigenen Parser (handgeschrieben). Es wäre schöner, wenn MPG zu diesem Zweck MPG verwenden würde. Schreiben Sie MPG dementsprechend um.

Zu diesen Zweck müssen Sie Ihre Aufgabe zuerst mit dem handeschriebenen Parser lösen, und anschliessend den generierten Parser mit semantischen Aktionen erweitern.

# Anhang D

# Lösungen der Aufgaben

In diesem Kapitel befinden sich die Lösungen ausgewählter Aufgaben

## D.1 Einführung

### D.1.1 Bootstrapping

**Aufgabe 1.1** Es sei $U$ eine *universelle* Zwischensprache. Dann genügt es $m$ Frontends von $l_i$ nach $U$ ($i = 1, \ldots, m$) und $n$ Backends von $U$ nach $M_j$ ($j = 1, \ldots, n$) zu schreiben. Es müssen also nur $m + n$ Übersetzer geschrieben werden.

## D.2 Lexikalische Analyse

### D.2.1 Spezifikation von Symbolen

**Aufgabe 2.1**

1. `a(a|b)*a|a`
2. `(ba)?(a*|bb+)*b?` oder auch `b?(a|bb+)*b?`
3. Unmöglich, reguläre Ausdrücke können nicht zählen.

**Aufgabe 2.2**

1. `(ba)? (a* | b(b|c)+ | c(c|b)*)* b?` oder auch `b? ( a | b(b|c)+ | cb* )* b?`
2. `a*b*c*`
3. `a?b?c? | a?c?b? | b?a?c? | b?c?a? | c?a?b? | c?b?a?`

**Aufgabe 2.3**

```
v ::= ("+"|"-")?
d ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
e ::= ("E"|"e") v d+
r ::= v ( d+ "." d* | "." d+) e?
```

**Aufgabe 2.4**

1. Enthält a an der drittletzten Stelle.
2. Alle Strings in {a,b}.
3. Genau ein a.

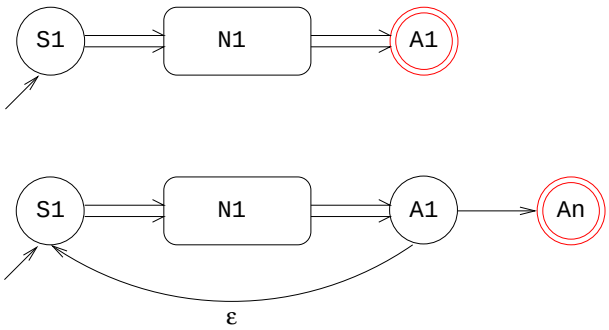
**Aufgabe 2.5**

```
"\""
(
  (~["\"","\\","\n","\r"])
|
  ("\\" ["n","t","r","\\","\""]
)*
```

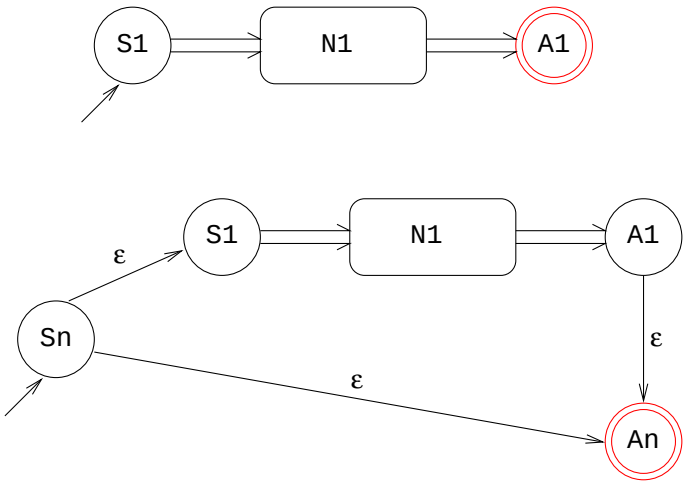
### D.2.2 Erkennen von Symbolen

**Aufgabe 2.9**

\+ Operator




? Operator




### D.2.3 Scanner Generatoren

**Aufgabe 2.10**

Skizze:

JAVACC-Spezifikation `Braces`

```
1   PARSER_BEGIN(Braces)
2
3   public class Braces {
4
5     public static void main(String args[])
6       throws ParseException {
7       Braces lexer = new Braces(System.in);
8       Token t = lexer.getNextToken();
9       while (t.kind != BracesConstants.EOF) {
10        t = lexer.getNextToken();
11      }
12    }
13  }
14
15  PARSER_END(Braces)
16
17  TOKEN_MGR_DECLS :
18  {
19    static String line = "";
20    static int parens = 0;
21    static int braces = 0;
22    static int comments = 0;
23    static int lineNr = 1;
24
25    static void output(String image) {
26      line += image;
27      switch (String.valueOf(lineNr).length()) {
28      case 1: System.out.print("00"); break;
```

```
29      case 2: System.out.print("0"); break;
30      }
31      System.out.print(lineNr  + " /*" + comments + "*/ (" +
32                       " {" + braces + "} (" + parens + ") " + line);
33      line = "";
34      lineNr++;
35    }
36  }
37
38  /* COMMENTS */
39
40  TOKEN :
41  {
42    <SINGLE_LINE_COMMENT_BEGIN: "//" >
43    {
44      line += matchedToken.image;
45    } : SINGLE_LINE_COMMENT
46  }
47
48  <SINGLE_LINE_COMMENT> TOKEN :
49  {
50    <SINGLE_LINE_COMMENT_END: "\n" | "\r" | "\r\n" >
51    {
52      output(matchedToken.image);
53    } : DEFAULT
54  }
55
56  <SINGLE_LINE_COMMENT> TOKEN :
57  {
58    <SINGLE_LINE_COMMENT_CONTENT: ~[] >
59    {
60      line += matchedToken.image;
61    }
62  }
63
64  <DEFAULT,MULTI_LINE_COMMENT> TOKEN :
65  {
66    <MULTI_LINE_COMMENT_BEGIN: "/*" >
67    {
68      comments++;
69      line += matchedToken.image;
70    } : MULTI_LINE_COMMENT
71  }
72
73  <MULTI_LINE_COMMENT> TOKEN :
74  {
75    <MULTI_LINE_COMMENT_END: "*/" >
76    {
77      comments--;
78      line += matchedToken.image;
```

```
79      if (comments == 0) SwitchTo(DEFAULT);
80    }
81  }
82
83  <MULTI_LINE_COMMENT> TOKEN :
84  {
85    <MULTI_LINE_COMMENT_EOL: "\n" | "\r" | "\r\n" >
86    {
87      output(matchedToken.image);
88    }
89  }
90
91  <MULTI_LINE_COMMENT> TOKEN :
92  {
93    <MULTI_LINE_COMMENT_CONTENT: ~[] >
94    {
95      line += matchedToken.image;
96    }
97  }
98
99  TOKEN:
100 {
101   < CHARACTER_LITERAL:
102       "'"
103       (   (~["'","\\","\n","\r"])
104         | ("\\"
105             ( ["n","t","b","r","f","\\","'","\""]
106             | ["0"-"7"] ( ["0"-"7"] )?
107             | ["0"-"3"] ["0"-"7"] ["0"-"7"]
108             )
109           )
110       )
111       "'"
112   >
113   {
114     line += matchedToken.image;
115   }
116 }
117
118 TOKEN:
119 {
120   < STRING_LITERAL:
121       "\""
122       (   (~["\"","\\","\n","\r"])
123         | ("\\"
124             ( ["n","t","b","r","f","\\","'","\""]
125             | ["0"-"7"] ( ["0"-"7"] )?
126             | ["0"-"3"] ["0"-"7"] ["0"-"7"]
127             )
128           )
```

```
129         )*
130         "\""
131     >
132     {
133       line += matchedToken.image;
134     }
135   }
136
137   TOKEN :
138   {
139     < LPAREN: "(" >
140     {
141       parens++;
142       line += matchedToken.image;
143     }
144   }
145
146   TOKEN:
147   {
148     < RPAREN: ")" >
149     {
150       parens--;
151       line += matchedToken.image;
152     }
153   }
154
155   TOKEN:
156   {
157     < LBRACE: "{" >
158     {
159       braces++;
160       line += matchedToken.image;
161     }
162   }
163
164   TOKEN:
165   {
166     < RBRACE: "}" >
167     {
168       braces--;
169       line += matchedToken.image;
170     }
171   }
172
173   TOKEN:
174   {
175     < EOL: "\n" | "\r" | "\r\n" >
176     {
177       output(matchedToken.image);
178     }
```

```
179  }
180
181  TOKEN :
182  {
183    <REST: ~[]>
184    {
185      line += matchedToken.image;
186    }
187  }
```

**Aufgabe 2.11**

Wir zeigen hier eine Lösung für die Programmiersprache JAVA. Die Umsetzung für die Programmiersprache C ist trivial.

Das Programm besteht aus folgenden Dateien:

JAVACC-Spezifikation `Crossref`

```
1   options {
2     JAVA_UNICODE_ESCAPE = true;
3   }
4
5   PARSER_BEGIN(Crossref)
6
7   public class Crossref {
8
9   }
10
11  PARSER_END(Crossref)
12
13  TOKEN_MGR_DECLS :
14  {
15    static String source = "";
16    static int n = 1;
17    static String line =  new String(n + "\t");
18  }
19
20  SKIP : // skip single line remark
21  {
22    <"//">
23    { line += image; } : SingleLineRemark
24  }
25
26  <SingleLineRemark> SKIP : // skip newline in single line remark
27  {
28    <"\n">
29    {
30      line += image;
31      source = source.concat(line);
32      n++;
33      line = new String(n + "\t");
```

```
34    } : DEFAULT
35  }
36
37  <SingleLineRemark> SKIP : // skip single line remark
38  {
39    <~[]>
40    { line += image; }
41  }
42
43  SKIP : // skip multi line remark
44  {
45    <"/*">
46    {  line += image; } : MultilineRemark
47  }
48
49  <MultilineRemark> SKIP : // skip multi line remark
50  {
51    <"*/">
52    {  line += image; } : DEFAULT
53  }
54
55  <MultilineRemark> SKIP : // skip newline in multi line remark
56  {
57    <"\n">
58    {
59      line += image;
60      source = source.concat(line);
61      n++;
62      line = new String(n + "\t");
63    }
64  }
65
66  <MultilineRemark> SKIP : // skip multi line remark
67  {
68    <~[]>
69    { line += image; }
70  }
71
72  SKIP : // skip strings (they could be interpreted as identifiers)
73  {
74    <
75     "\""
76     (   (~["\"","\\","\n","\r"])
77       | ("\\"
78           ( ["n","t","b","r","f","\\","'","\""]
79           | ["0"-"7"] ( ["0"-"7"] )?
80           | ["0"-"3"] ["0"-"7"] ["0"-"7"]
81           )
82         )
83     )*
```

```
    "\""
  >
  { line += image; }
}

SKIP :  // skip char constants (they could be interpreted as identifiers)
{
  <
   "'"
   (   (~["'","\\","\n","\r"])
     | ("\\"
         ( ["n","t","b","r","f","\\","'","\""]
         | ["0"-"7"] ( ["0"-"7"] )?
         | ["0"-"3"] ["0"-"7"] ["0"-"7"]
         )
       )
   )
   "'"
  >
  { line += image; }
}

SKIP : // skip java keywords
{
  <
   "abstract" | "boolean" | "break" | "byte" | "case" | "catch" |
   "char" | "class" | "const" | "continue" | "default" | "do" | "double" |
   "else" | "extends" | "false" | "final" | "finally" | "float" | "for" |
   "goto" | "if" | "implements" | "import" | "instanceof" | "int" |
   "interface" | "long" | "native" | "new" | "null" | "package"| "private"
   "protected" | "public" | "return" | "short" | "static" | "super" |
   "switch" | "synchronized" | "this" | "throw" | "throws" | "transient" |
   "true" | "try" | "void" | "volatile" | "while"
  >
  { line += image; }
}

TOKEN : // pass tokens to the caller
{
  <IDENTIFIER:  <LETTER> (<LETTER>|<DIGIT>)* ("." (<LETTER>|<DIGIT>)*)* >
  { line += image; }
|
  < #LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
```

```
        "\u00f8"-"\u00ff",
        "\u0100"-"\u1fff",
        "\u3040"-"\u318f",
        "\u3300"-"\u337f",
        "\u3400"-"\u3d2d",
        "\u4e00"-"\u9fff",
        "\uf900"-"\ufaff"
       ]
  >
|
  < #DIGIT:
      [
       "\u0030"-"\u0039",
       "\u0660"-"\u0669",
       "\u06f0"-"\u06f9",
       "\u0966"-"\u096f",
       "\u09e6"-"\u09ef",
       "\u0a66"-"\u0a6f",
       "\u0ae6"-"\u0aef",
       "\u0b66"-"\u0b6f",
       "\u0be7"-"\u0bef",
       "\u0c66"-"\u0c6f",
       "\u0ce6"-"\u0cef",
       "\u0d66"-"\u0d6f",
       "\u0e50"-"\u0e59",
       "\u0ed0"-"\u0ed9",
       "\u1040"-"\u1049"
      ]
  >
}

SKIP : // skip newline
{
  <"\n">
  {
    line += image;
    source = source.concat(line);
    n++;
    line = new String(n + "\t");
  }
}

SKIP : // skip anything else
{
  <~[]>
  { line += image; }
}
```

JAVA-Klasse `Main`

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

```
 2
 3  class Main {
 4
 5    public static void main(String args[]) {
 6      if (args.length == 0 || args[0] == null) {
 7        System.out.println("usage: java Main file");
 8        System.exit(1);
 9      }
10      String name = args[0];
11      FileInputStream inFile = null;
12      try {
13        Table table = new Table();
14        inFile = new FileInputStream(new String(name + ".java"));
15        Crossref lexer = new Crossref(inFile);
16        // get tokens and put line information into table
17        Token t = lexer.getNextToken();
18        while (t.kind != CrossrefConstants.EOF) {
19          Table.put(t);
20          t = lexer.getNextToken();
21        }
22        // write the sorted cross reference table to file
23        String outString = new String(Crossref.token_source.source +
24                                      Table.resultString());
25        File f = new File(new String(name + ".crf"));
26        FileWriter fw = new FileWriter(f);
27        fw.write(outString);
28        fw.flush();
29        fw.close();
30        // view the sorted cross reference table
31      }
32      catch (IOException e) {
33        e.printStackTrace();
34        System.exit(1);
35      }
36    }
37  }
```

JAVA-Klasse Table

```
 1  import java.util.Hashtable;
 2  import java.util.Vector;
 3  import java.util.Enumeration;
 4
 5  class Table {
 6
 7    private static String fileName;
 8    private static Hashtable table;
 9    private static String keys [];
10
11    public static void put (Token t) {
```

```
12      if (table == null) table = new Hashtable();
13      String name = t.image;
14      int line = t.beginLine;
15      Vector v;
16      if (table.containsKey(name)) {
17        v = (Vector) table.get(name);
18        v.addElement(new Integer(line));
19      }
20      else {
21        v = new Vector();
22        v.addElement(new Integer(line));
23        table.put(name,v);
24      }
25    }
26
27    private static void build () {
28      // build a sorted string array with the keys
29      Enumeration e = table.keys();
30      int k = 0;
31      keys = new String[table.size()];
32      while (e.hasMoreElements()) {
33        String name = (String) e.nextElement();
34        keys[k++] = new String(name);
35      }
36      // sort the array in lexicographic order
37      Sort.sort(keys);
38    }
39
40    public static String resultString () {
41      build();
42      String text = "";
43      text += "\n\nCross reference table\n\n";
44      for (int k=0; k<keys.length; k++) {
45        text += keys[k] + ":\t";
46        Vector v = (Vector) table.get(keys[k]);
47        for (int i=0; i<v.size(); i++) {
48          int j = ((Integer) v.elementAt(i)).intValue();
49          text += j + " ";
50        }
51        text += "\n";
52      }
53      return(text);
54    }
55  }
```

JAVA-Klasse Sort

```
1   class Sort {
2
3     public static void sort (String keys[]) {
```

```
4       // we use a simple bubble sort here
5       for (int i=0; i<keys.length; i++) {
6         for (int j=keys.length-1; j>i; j--) {
7           String k1 = new String(keys[j]);
8           String k2 = new String(keys[j-1]);
9           k1 = k1.toLowerCase();
10          k2 = k2.toLowerCase();
11          if (k1.compareTo(k2) < 0) {
12            String s = new String(keys[j]);
13            keys[j] = keys[j-1];
14            keys[j-1] = s;
15          }
16          else {
17            if (k1.compareTo(k2) == 0 && keys[j].compareTo(keys[j-1]) <= 0) {
18              String s = new String(keys[j]);
19              keys[j] = keys[j-1];
20              keys[j-1] = s;
21            }
22          }
23        }
24      }
25    }
26  }
```

Die Ausgabe des Programms beim Aufruf `java Crossref Main` hat folgende Gestalt:

```
1 import java.io.*;
2
3 class Main {
4
5   public static void main(String args[]) {
6     if (args.length == 0 || args[0] == null) {
7       System.out.println("usage: java Main file");
8       System.exit(1);
9     }
10     String name = args[0];
11     FileInputStream inFile = null;
12     try {
13       Table table = new Table();
14       inFile = new FileInputStream(new String(name + ".java"));
15       Crossref lexer = new Crossref(inFile);
16       // get tokens and put line information into table
17       Token t = lexer.getNextToken();
18       while (t.kind != CrossrefConstants.EOF) {
19         Table.put(t);
20         t = lexer.getNextToken();
21       }
22       // write the sorted cross reference table to file
23       String outString = new String(Crossref.token_source.source +
24                                     Table.resultString());
```

```
25       File f = new File(new String(name + ".crf"));
26       FileWriter fw = new FileWriter(f);
27       fw.write(outString);
28       fw.flush();
29       fw.close();
30       // view the sorted cross reference table
31     }
32     catch (IOException e) {
33       e.printStackTrace();
34       System.exit(1);
35     }
36   }
37 }
```

```
Cross reference table

args: 5 6 10
args.length: 6
Crossref: 15 15
Crossref.token_source.source: 23
CrossrefConstants.EOF: 18
e: 32
e.printStackTrace: 33
f: 25 26
File: 25 25
FileInputStream: 11 14
FileWriter: 26 26
fw: 26
fw.close: 29
fw.flush: 28
fw.write: 27
inFile: 11 14 15
IOException: 32
java.io.: 1
lexer: 15
lexer.getNextToken: 17 20
Main: 3
main: 5
name: 10 14 25
outString: 23 27
String: 5 10 14 23 23 25
System.exit: 8 34
System.out.println: 7
t: 17 19 20
t.kind: 18
Table: 13 13
table: 13
Table.put: 19
Table.resultString: 24
```

```
Token: 17
```

**Aufgabe 2.12**

Ein Doublequote Zeichen als Character Konstante ist nicht der Anfang eines Strings. Siehe auch Lösung der Aufgabe 2.10

**Aufgabe 2.13**

JAVACC-Spezifikation `Rmtag`

```
1   PARSER_BEGIN(Rmtag)
2
3   public class Rmtag {
4
5     public static void main(String args[])
6       throws ParseException {
7       Rmtag lexer = new Rmtag(System.in);
8       Token t = lexer.getNextToken();
9       while (t.kind != RmtagConstants.EOF) {
10        t = lexer.getNextToken();
11      }
12    }
13
14  }
15
16  PARSER_END(Rmtag)
17
18  SKIP :
19  {
20    "<" : TAG
21  }
22
23  <TAG> SKIP :
24  {
25    <(~[">", "\"", "\'"])+>
26  }
27
28  <TAG> SKIP :
29  {
30    "\"" : ATTR1
31  }
32
33  <ATTR1> SKIP :
34  {
35    <(~["\""])+>
36  }
37
38  <ATTR1> SKIP :
39  {
40    "\"" : TAG
41  }
```

```
42
43  <TAG> SKIP :
44  {
45    "\'" : ATTR2
46  }
47
48  <ATTR2> SKIP :
49  {
50    <(~["\'"])+>
51  }
52
53  <ATTR2> SKIP :
54  {
55    "\'" : TAG
56  }
57
58  <TAG> SKIP :
59  {
60    ">" : DEFAULT
61  }
62
63  SKIP :
64  {
65    "<!--" : COMMENT
66  }
67
68  <COMMENT> SKIP :{
69    <(~["-"])+>
70  }
71
72  <COMMENT> SKIP :
73  {
74    "-"
75  }
76
77  <COMMENT> SKIP :
78  {
79    "-->" : DEFAULT
80  }
81
82  TOKEN :
83  {  <PCDATA: ~["<", "&"]>
84    { System.out.print(matchedToken.image); }
85  }
86
87  TOKEN :
88  {
89    <ENTGT: "&gt;">
90    { System.out.print(">"); }
91  }
```

```
92
93  TOKEN :
94  {
95    <ENTLT: "&lt;">
96    { System.out.print("<"); }
97  }
98
99  TOKEN :
100 {
101   <ENTAMP: "&amp;">
102   { System.out.print("&"); }
103 }
104
105 TOKEN :
106 {
107   <ENTAUML: "&auml;">
108   { System.out.print("ä"); }
109 }
110
111 TOKEN :
112 {
113   <ENTOUML: "&ouml;">
114   { System.out.print("ö"); }
115
116 TOKEN :
117 {
118   <ENTAMP: "&amp;">
119   { System.out.print("&"); }
120 }
121
122 TOKEN :
123 {
124   <ENTAUML: "&auml;">
125   { System.out.print("ä"); }
126 }
127
128 TOKEN :
129 {
130   <ENTOUML: "&ouml;">
131   { System.out.print("ö"); }
132 }
133
134 TOKEN :
135 {
136   <ENTUUML: "&uuml;">
137   { System.out.print("ü"); }
138 }
139
140 // usw...
141 TOKEN :
```

```
142   {
143     <AMP: "&">
144     { System.out.print(matchedToken.image); }
145   }
```

### D.2.4 Projekt

**Aufgabe 2.16**

Siehe `prog/la/la01.zip`

## D.3 Syntaktische Analyse

### D.3.1 Formale Sprachen

**Aufgabe 3.1**

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 3.2**

Zur Erinnerung: Sei $A$ ein Alphabet. Die regulären Ausdrüke $\alpha$ 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 müssen wir zeigen, dass mit einer Typ 3 Grammakik die Konstrukte Sequenz, Auswahl und Wiederholung konstruierbar sind.

Die Sequenz $ABC$ kann mit den Produktionen $seq \rightarrow A \quad seq1$, $seq1 \rightarrow B \quad seq2$, $seq2 \rightarrow C$ realisiert werden.

Die Auswahl $A|B$ kann mit den Produktionen $choice \rightarrow A$, $choice \rightarrow B$ realisiert werden.

Die Wiederholung $A*$ kann mit den produktionen $rep \rightarrow A \quad rep, rep \rightarrow \epsilon$ realisiert werden.

2. Sofern Produktionen von Typ $a \rightarrow A \quad b$ ; keine (indirekte) Rekursionen enthalten, werden ausschliesslich Sequenzen gebildet. Dies ist mit Hilfe von regulären Ausdrücken realisierbar.

   Rekursionen sind im wesentlichen Produktionen vom Typ $a \rightarrow A \quad a$ ;. Dies entspricht dem regulären Ausdruck $(A)*$.

**Aufgabe 3.3**

```
s ::= "(ß ") ;
```

**Aufgabe 3.4**

```
e ::= e "+ë | "(ë ")"x";
```

**Aufgabe 3.5**

1. Siehe Ableitungsbaum.

2.
$$
\begin{aligned}
s &\Rightarrow s\ s\ \text{"*"} \\
&\Rightarrow s\ s\ \text{"+"}\ s\ \text{"*"} \\
&\Rightarrow \text{ä"}\ s\ \text{"+"}\ s\ \text{"*"} \\
&\Rightarrow \text{ä"}\ \text{ä"}\ \text{"+"}\ s\ \text{"*"} \\
&\Rightarrow \text{ä"}\ \text{ä"}\ \text{"+"}\ \text{ä"}\ \text{"*"}
\end{aligned}
$$

3. Umgekehrte Polnische Notation (HP taschenrechner)

4. Eindeutig da für Taschenrechner verwendet.

**Aufgabe 3.6**

```
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)*
         ;
```

# Abbildungsverzeichnis

# Literaturverzeichnis


[ALSU08] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compiler*. Pearson Studium, München, 2008.

[App98] A.W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[ASU92a] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilerbau*, volume 1. Addison-Wesley, Bonn, 1992.

[ASU92b] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilerbau*, volume 2. Addison-Wesley, Bonn, 1992.

[ASU07] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Pearson Education, Boston, 2nd edition, 2007.

[BBB+57] J. W. Backus, R.J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, R. A. Hughes, and R. Nutt. The fortran automatic coding system. In *Proc. AFIPS 1957 Western Joint Computer Conference*, 1957.

[Dij60] E. W. Dijkstra. Algol 60 translation. Supplement ALGOL Bulletin 10, 1960.

[DS92] C. Donnelly and R. Stallman. *Bison, the YACC-compatible Parser Generator*. Free Software Foundation, December 1992. Bison Version 1.20.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, Reading, Massachusetts, 1995.

[GHJV96] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Entwurfsmuster, Elemente wiederverwendbarer objektorintierter Software*. Professional Computing. Addison-Wesley, Bonn, 1996.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, 1996.

[Gos96] J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, Massachusetts, 1996.

[HMU07] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Automata Theory, Languages and Computation*. Pearson Education, Boston, 3rd edition, 2007.

[KR90] B.W. Kernighan and D.M. Ritchie. *Programmieren in C: mit dem C-Reference Manual in deutscher Sprache*. PC professionel. Carl Hanser und Prentice-Hall International, München, 2nd edition, 1990. Deutsche Übersetzung.

[Les75] M.E. Lesk. Lex – a lexical analyser generator. Technical Report Computer Science Technical Report 39, AT & T Bell Laboratories, Murray Hill, N.J., 1975.

[LMB92] J.R. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly & Associates. Inc., Sebastopol, CA 95472, 2nd edition, 1992.

[LY96] T. Lindholm and F Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, 1996.

[MD97] J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly, 1997.

[vL90] J. van Leeuwen. *Handbook of Theoretical Computer Science, Formal Models and Semantics*, volume B. Elsevier, Amsterdam, 1990.

[Wir71] N. Wirth. The programing language pascal. *Acta Informatica*, 1(1):35–63, 1971.

[Wir86] N. Wirth. *Compilerbau*. Teubner, Stuttgart, 4th edition, 1986.

[Wir89] N. Wirth. *Programming in Modula-2*. Springer, 4th edition, 1989. ISBN 0387501509.

[Wir96] N. Wirth. *Grundlagen und Techniken des Compilerbaus*. Addison-Wesley, Bonn, 1996.

[WM92] R. Wilhelm and D. Maurer. *Übersetzerbau*. Springer, Berlin, 1992.

# Index

# Inhaltsverzeichnis