Compilerbau

P. Köhler

Sommersemester 2018

Inhaltsverzeichnis

Kapitel 1

Vorwort

Compiler und Interpreter bilden einen wesentlichen Bestandteil eines jeden Computer-Systems; ohne sie würden wir – immer noch – in Assembler oder gar Maschinensprache programmieren! Aus diesem Grunde ist der Compilerbau eine wichtige, praxisorientierte Disziplin der Informatik, die aber andererseits auch von ihren theoretischen Grundlagen der Mathematik sehr nahesteht.

Ziel der Vorlesung kann es – schon aus Zeitgründen – nicht sein, eine der Standard-Informatik-Vorlesungen über Compilerbau zu ersetzen; es ist auch nicht beabsichtigt, einen erschöpfenden Überblick über den aktuellen Forschungsstand zu geben.

Vielmehr will ich versuchen, zum einen einen Abriss der mathematischen Grundlagen zu liefern; zum anderen soll aber – so praxisorientiert wie möglich – ein Einblick in die Arbeitsweise eines Compilers gegeben werden.

Vorausgesetzt werden zum einen Kenntnisse in einer höheren Programmiersprache wie C, C# oder PASCAL, und zu einem geringen Teil auch Kenntnisse einer Assemblersprache. Mathematische Grundlagen aus der Automatentheorie sind hilfreich; für das Verständnis dieser Vorlesung nötige Begriffe und Sätze aus der Automatentheorie werden hier jedoch wiederholt.

Kapitel 2

Einleitung

Ein *Übersetzer* ist ein Programm, das ein *Quellprogramm* in ein äquivalentes *Objektprogramm* übersetzt. Das Quellprogramm ist in der *Quellsprache* geschrieben, das Objektprogramm in der *Objektsprache*. Die Ausführung der Übersetzung geschieht zur *Übersetzungszeit*.

Ist die Quellsprache eine höhere Programmiersprache wie z.B. C++ oder PASCAL, und die Objektsprache die Assemblersprache oder die Maschinensprache eines Computers, so nennt man den Übersetzer auch *Compiler*. Maschinensprache wird auch *Code* genannt; daher heißt mitunter das Objektprogramm auch *Objekt-Code*. Die Übersetzungszeit nennt man in diesem Fall auch *Compile-Zeit*; die wirkliche Ausführung des Objektprogramms geschieht zur *Laufzeit*.

Ein *Assembler* ist ein Programm, das ein in der Assemblersprache geschriebenes Programm in die Maschinensprache eines Computers übersetzt; hier sind jedoch diese beiden Sprachen sehr ähnlich, die meisten Assembler-Befehle sind symbolische Darstellungen von Maschinen-Befehlen; außerdem haben Assembler-Befehle meist ein festes Format, was ihre Übersetzung sehr erleichtert.

Ein *Interpreter* einer Quellsprache liest ein in dieser Sprache geschriebenes Programm und führt es aus; anders als bei einem Compiler entsteht kein für sich lauffähiges Objektprogramm. Man unterscheidet zwei Varianten eines Interpreters: Die reine Version analysiert einen Quellbefehl jedes Mal, wenn er zur Ausführung gelangen soll. Dies war die Vorgehensweise der meisten BASIC-Interpreter der 90'er Jahre, die in den damals gängigen Mikrocomputern installiert waren. Allerdings ist diese Methode ineffizient und produziert ggf. sehr lange Laufzeiten. Besser ist es, zunächst das vollständige Quellprogramm zu analysieren und in eine interne Darstellung zu übersetzen. In einem zweiten Schritt wird dann diese interne Darstellung interpretiert, die dann so angelegt

sein sollte, dass die Zeit zum Analysieren eines Befehls minimiert wird. Auf diese Weise arbeiteten schon früher eine Reihe von PASCAL-Compilern, die einen sog. P-Code erzeugten, der beim eigentlichen Lauf vom Laufzeitsystem interpretiert wurde. In heutiger Zeit ist dieses Vorgehen in der Sprache JAVA – mit dem dort erzeugten Bytecode – und in der .NET-Umgebung mit der aus unterschiedlichen Quellsprachen erzeugten MSIL (Microsoft's Intermediate Language) wieder zu Ehren gekommen. In diesen beiden Umgebungen wird die Effizienz bei der Ausführung durch eine zweite Übersetzungsphase, das sog. *Just in time compiling*, nämlich die Übersetzung des Byte- bzw. Intermediate-Codes in die Maschinensprache des jeweiligen Zielsystems stark verbessert.

Ein Compiler führt immer zunächst eine *Analyse* des Quellprogramms durch und dann eine *Synthese* des Objektprogramms. Im Verlauf der Analyse baut der Compiler eine Reihe von Tabellen auf, die sowohl während der Analyse als auch während der Synthese benutzt werden. Abbildung 2.1 auf Seite 6 zeigt den zugehörigen Datenfluss etwas detaillierter. Dabei wird deutlich, dass sich die Analyse und Synthese weiter in logische Bestandteile zerlegen lassen, die wir im folgenden kurz beschreiben wollen.

Tabellen:

Während der Analyse gewinnt der Compiler Informationen aus Deklarationen, Prozedurköpfen usw., die er sich für einen späteren Zugriff speichern muss. Beispielsweise ist es nötig, bei jeder Variablenansprache zu wissen, wie die betreffende Variable deklariert bzw. an anderer Stelle verwendet wurde. Die genaue Struktur der zugehörigen Information hängt natürlich stark von der Quellsprache und der Objektsprache ab; jeder Compiler aber benutzt eine *Symboltabelle*, in der die benutzten Variablen mit ihren Attributen enthalten sind. Diese Attribute bestehen mindestens aus dem Typ der Variablen, ihrer Objektprogrammadresse und zusätzlichen Daten, die zur Codeerzeugung nötig sind. Weitere Tabellen sind die Konstantentabellen und Tabellen von Schleifen, die die Schachtelungsstruktur und Schleifenvariablen enthalten usw..

Lexikalische Analyse:

Der Teil des Compilers, der die lexikalische Analyse erledigt, heißt *Scanner*. Er ist der einfachste Teil eines jeden Compilers und liest das Quellprogramm zeichenweise ein; aus den Einzelzeichen baut er die aktuellen *Symbole* des Programms auf – Zahlen, Namen, Schlüsselwörter, Delimiter usw.. Die eigentliche Analyse setzt dann auf diesen Symbolen auf. Üblicherweise werden Kommentare vom Scanner überlesen, und der Scanner besorgt auch den – normalen – Teil des Listings. Die Symbole werden meistens vom Scanner an den Analyzer in einer internen Form hochgereicht, etwa als Integer-Zahl; Namen können etwa durch ein Paar von Zahlen beschrieben werden, von denen die erste die Namenskennung ist und die zweite ein Pointer auf den wirklichen Namensstring, der in einer geeigneten Tabelle abgelegt ist. Ein solches Vorgehen ermöglicht dem Rest

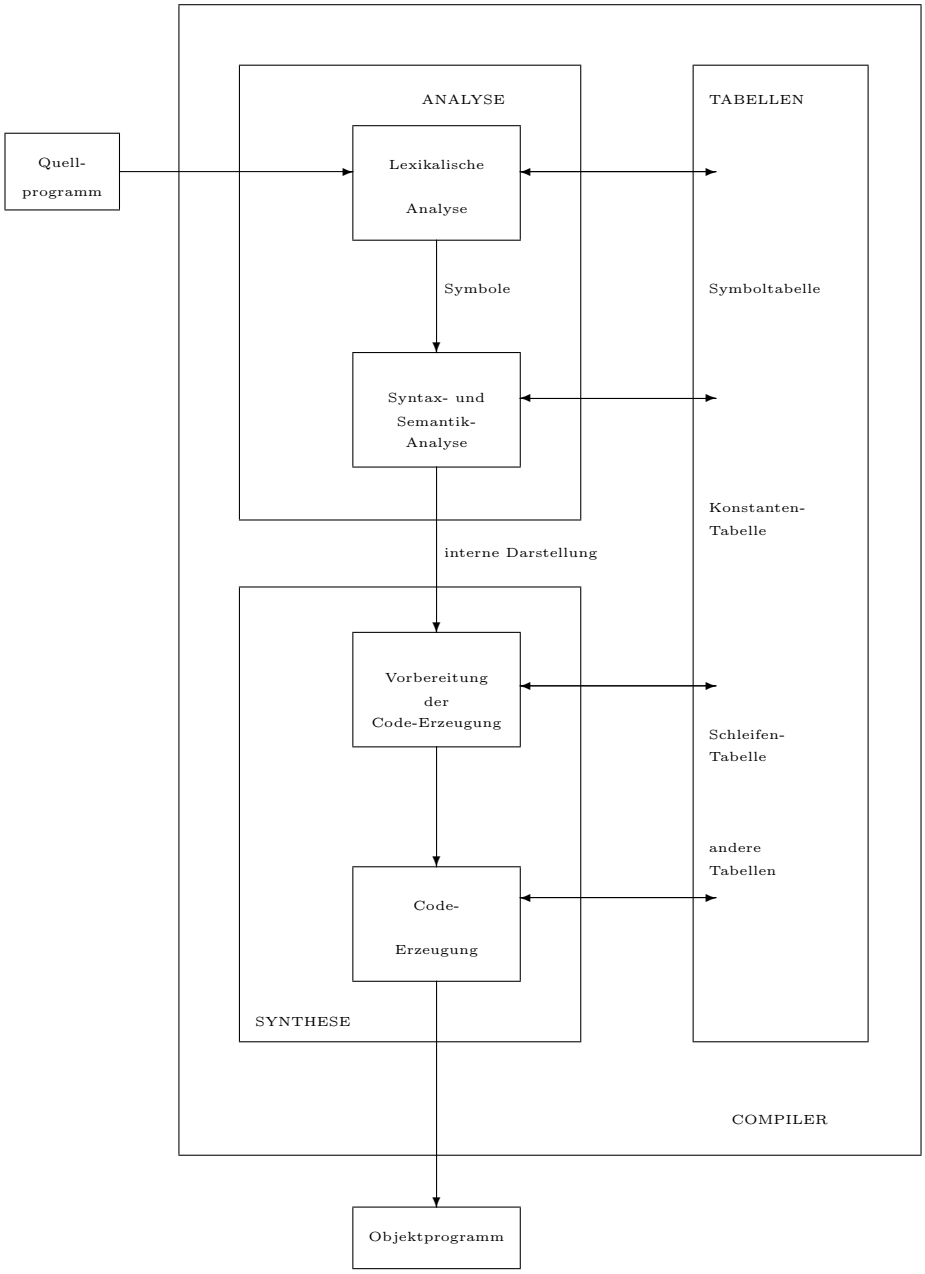


Abbildung 2.1: Schematischer Aufbau eines Compilers

des Compilers, mit festen Formaten statt mit unterschiedlich langen Strings zu arbeiten.

Syntaktische und semantische Analyse:

Hier wird ein vollständiger Syntax- und Semantik-Check des Quellprogramms durchgeführt, die interne Form des Programms erzeugt und die Symboltabelle und sonstige Tabellen aufgefüllt. Die meisten der heute verwendeten Analyzer werden von der Syntax des Programms kontrolliert; oft wird versucht, die syntaktische und semantische Analyse so weit wie möglich zu trennen. Wenn der *Parser*, so heißt der Syntax-Analyzer, ein Syntaxelement erkennt, ruft er eine sogenannte *semantische Routine* auf, die das vorliegende Syntaxelement auf semantische Korrektheit überprüft und dann notwendige Information in der Symboltabelle oder der internen Darstellung des Programms ablegt. Wird beispielsweise eine Variablendeklaration erkannt, so wird die entsprechende semantische Routine zunächst überprüfen, ob die deklarierte Variable schon einmal deklariert wurde und dann die Variable zusammen mit den zugehörigen Attributen in die Symboltabelle schreiben. Oder wenn eine Zuweisung der Form

```
variable := expression
```

erkannt wird, wird die semantische Routine die entsprechenden Inhalte auf Typverträglichkeit hin überprüfen und dann die Zuweisung in die interne Darstellung einfügen.

Interne Darstellung:

Die interne Darstellung hängt weitgehend davon ab, wie sie später weiterverarbeitet werden soll. Sie könnte etwa der Syntax-Baum des Quellprogramms sein, oder das Quellprogramm in der sog. polnischen Notation. Von vielen Compilern wird eine Liste von Quadrupeln (Operator, Operand, Operand, Ergebnis) in der auszuführenden Reihenfolge angelegt. So würde etwa die Zuweisung 'A := B + C $*$ D' in folgender Form erscheinen:

```
*,  C,  D,  T1
+,  B,  T1, T2
=,  T2,     A
```

hierbei sind T1 und T2 temporäre Variable, die vom Compiler erzeugt werden. Die Operanden sind natürlich nicht die symbolischen Namen selbst sondern Pointer auf die entsprechenden Einträge in der Symboltabelle.

Vorbereitung der Code-Erzeugung:

Bevor das eigentliche Objektprogramm generiert werden kann, ist normalerweise nötig, die interne Darstellung geeignet zu modifizieren, etwa im Hinblick auf

eine Optimierung der Laufzeit des Objektprogramms. Es müssen den Variablen Laufzeitspeicherplätze zugewiesen werden.

Code-Erzeugung:

Hier wird schließlich das interne Quellprogramm in Assembler- oder Maschinensprache übersetzt. Wenn etwa das interne Programm aus Quadrupeln besteht, so wird für jedes Quadrupel in der abgelegten Reihenfolge Code erzeugt; für die drei oben angeführten Quadrupel könnte dies etwa zu folgendem Code führen:

```
lda c           ; lade c in den Akku
mul d           ; multipliziere Akku mit d
add b           ; addiere b zum Akku
sto a           ; speichere Akku nach a
```

Dies wäre schon eine optimierte Form, in der die temporären Variablen T1 und T2 nur noch in Form des Akkus auftauchen.

Mit dieser knappen Übersicht soll nun aber nicht gesagt werden, dass alle Compiler sich genau an diese Struktur halten. Ein sogenannter 1-Pass-Compiler, der das Objektprogramm in einem einzigen Durchlauf erzeugt, kann auf die interne Darstellung verzichten; die Code-Erzeugung geschieht innerhalb der semantischen Routinen. Nicht alle Quellsprachen aber lassen sich von 1-Pass-Compilern übersetzen.

Bild 1 gibt auch nicht unbedingt den zeitlichen Ablauf des Vorgehens wieder: Man könnte sich vorstellen, dass die vier Teile sequentiell in eben dieser Reihenfolge ablaufen, aber auch dass sie gewissermassen ineinander verzahnt sind. Abhängig ist dies sicher vom zur Verfügung stehenden Speicherplatz aber auch von Anforderungen an die Geschwindigkeit des Compilers bzw. des Objektprogramms oder die Fehlerbehandlung. Ein weiterer Punkt ist die Anzahl der am Bau des Compilers beteiligten Personen, von denen jede ggf. für einen separaten Pass verantwortlich sein kann.

Kapitel 3

Sprachen und Grammatiken

3.1 Ein Beispiel zum Einstieg

Um die nachfolgenden Überlegungen zu motivieren und auch jeweils an einem konkreten Beispiel illustrieren zu können, stellen wir uns die Aufgabe, ganz naiv ein (C#-) Programm zu schreiben, das arithmetische Rechenausdrücke verarbeiten und ihr Ergebnis bestimmen kann. Der Einfachheit halber beschränken wir uns dabei zunächst auf nichtnegative ganze Zahlen und lassen insbesondere keine symbolischen Ausdrücke zu; zulässige Rechenoperationen seien die Addition und die Multiplikation, wobei wie üblich die Multiplikation vor der Addition Vorrang haben soll, wenn dies nicht durch Klammern anders ausgedrückt wird. Um die Aufgabenstellung weiter zu präzisieren: Das Programm soll den Ausdruck zeichenweise einlesen, auf Korrektheit überprüfen und auf ein Gleichheitszeichen hin das Ergebnis ausdrucken, so dass ein solcher Dialog etwa wie folgt aussehen könnte:

```
(1+3)*(4+5)=
36
```

Unbeschadet ihrer Einfachheit ist die Aufgabe alles andere als trivial; das erste und entscheidende Problem besteht darin, aus der Unzahl von möglichen Zeichenfolgen die 'korrekten' auszusondern – das ist gerade die in der Einleitung schon erwähnte syntaktische Analyse. Voraussetzung dafür ist eine Präzisierung des intuitiv klaren Begriffs 'arithmetischer Ausdruck' – wieso ist obiger Ausdruck korrekt, nicht hingegen aber $(1+)*4$? Ein erster Definitionsversuch wäre

der folgende:

(3.1) *Eine Zahl ist ein arithmetischer Ausdruck.*

(3.2) *Sind x, y arithmetische Ausdrücke,*
*so auch (x), x + y und x * y.*

(3.3) *Arithmetische Ausdrücke sind nur solche,*
die aufgrund von 3.1 und 3.2 entstehen.

Diese (rekursive) Definition erlaubt es nun tatsächlich, für eine vorgelegte Zeichenkette in endlich vielen Schritten zu entscheiden, ob es sich um einen arithmetischen Ausdruck handelt. Im Fall $(1+)*4$ etwa schließen wir wie folgt: Wäre $(1+)*4$ ein Ausdruck, so müßte wegen 3.2 auch $(1+)$ ein Ausdruck sein, wiederum wegen 3.2 auch $1+$; $1+$ aber läßt sich weder nach 3.1 noch nach 3.2 als arithmetischer Ausdruck identifizieren.

So einfach diese Definition auch ist, sie läßt sich nicht dazu benutzen, arithmetische Ausdrücke auch auszuwerten: Der Ausdruck $1 + 2 * 3$ könnte nach 3.2 einerseits aus den beiden Ausdrücken 1 und $2 * 3$ entstanden sein, andererseits aber auch aus den Ausdrücken $1+2$ und 3. Die Definition trägt nicht der vereinbarten Priorität der Multiplikation vor der Addition Rechnung. Um dies zu gewährleisten, müssen wir die Definition etwas komplizierter gestalten, dabei wollen wir gleich die in der Grammatiktheorie übliche Kurzschreibweise (Backus-Naur-Form) einführen:

(3.4) $expr ::= term \mid term \ + \ expr$

(3.5) $term ::= factor \mid factor \ * \ term$

(3.6) $factor ::= number \mid (\ expr\ )$

In Worten ausgedrückt besagt 3.4: Ein arithmetischer Ausdruck ist entweder ein Term oder die Summe aus einem Term und einem arithmetischen Ausdruck; analoges gilt für 3.5 und 3.6.

Nach dieser Definition läßt sich dann der Ausdruck $1 + 2 * 3$ nur noch auf eine Art zerlegen, nämlich nach 3.4 in die Summe von 1 und $2 * 3$; eine Anwendung von 3.5 scheitert, da zwar 3 ein Term ist, aber 1+2 eben kein Faktor.

Durch die Hinzunahme zweier weiterer Definitionen

(3.7) $$stmt ::= expr \ =$$

(3.8) $$number ::= digit \mid digit\ number$$

läßt sich dann auch schon für unsere Aufgabe eine Formalisierung erreichen, die eine unmittelbare Umsetzung in ein Programm ermöglicht:

```
using System;
using System.Text;

namespace AdamRiese
{
  class Program
  {
    static int nextchar;
    static bool isdigit(int c)
    {
      return c >= '0' && c <= '9';
    }

    static void error()
    {
      Console.WriteLine("Syntax Error");
      // throw new System.ApplicationException();
    }

    static int number()
    {
      int value = nextchar - '0';
      nextchar = Console.Read();
      while (isdigit(nextchar))
      {
        value *= 10;
        value += nextchar - '0';
        nextchar = Console.Read();
      }
      return value;
    }

    static int factor()
```

```
{
  int value = 0;
  if (nextchar == '(')
  {
    nextchar = Console.Read();
    if (nextchar == '(' || isdigit(nextchar))
    {
      value = expr();
      if (nextchar != ')') error();
      nextchar = Console.Read();
    }
    else error();
  }
  else // if (isdigit(nextchar))
  {
    value = number();
  }
  // else error();
  return value;
}

static int term()
{
  int value = factor();
  if (nextchar == '*')
  {
    nextchar = Console.Read();
    if (nextchar == '(' || isdigit(nextchar))
    {
      value *= term();
    }
    else error();
  }
  return value;
}

static int expr()
{
  int value = term();
  if (nextchar == '+')
  {
    nextchar = Console.Read();
    if (nextchar == '(' || isdigit(nextchar))
    {
      value += expr();
    }
    else error();
  }
  return value;
}
```

```
    static void stmt()
    {
      int value = expr();
      if (nextchar != '=') error();
      else
      {
        nextchar = Console.Read();
        Console.WriteLine(value);
      }
    }

    static void Main(string[] args)
    {
      nextchar = Console.Read();
      if (nextchar == '(' || isdigit(nextchar))
      {
        stmt();
      }
      else error();
    }
  }
}
```

Die den Definitionen 3.4 bis 3.8 entsprechenden Prozeduren sind z.T. direkt und z.T. indirekt rekursiv, darauf ist ggf. bei einer eventuellen Umschreibung in eine andere Sprache zu achten.

Die Auswertung des eingegebenen Ausdrucks erfolgt über Funktionen, die den berechneten Wert zurückgeben.

Auf eine spezifische Fehlerbehandlung sowie eine komfortablere Eingabe, die etwa Leerzeichen überliest, ist bewusst verzichtet worden; wir werden darauf später noch gezielt eingehen.

Der Vollständigeit halber wollen wir – im Vorgriff auf später noch ausführlicher zu Behandelndes – eine einfache Lösung der Aufgabenstellung in der Programmiersprache *C* mit Hilfe der GNU-Compilerbau-Werkzeuge *flex* und *bison* angeben:

Die Flex-Quelle (für die lexikalische Analyse) kann dabei wie folgt formuliert werden:

```
%{
#include "adamry.tab.h"
extern int yylval;
%}
```

```
%%
[0-9]+  { yylval = atoi(yytext); return NUMBER; }
[\t\n ] ;
.       return yytext[0];
%%
```

Neben den formalen Anteilen, die dem Vorgehen der Werkzeuge im Hinblick auf die Erzeugung von C-Code entspechen, besagen die drei wesentlichen Zeilen, dass Ziffernfolgen als Zahlen (`NUMBER`) interpretiert werden, dass Whitespace-Charakter (Tab, Return, Leerzeichen) überlesen und dass alle übrigen Zeichen als für sich sinntragenden Einheiten (Token) aufgefasst werden sollen

Der Bison-Anteil (für die syntaktische und semantische Behandlung) geht in seiner Syntax-Formulierung sogar auf die erste (zweideutige) Form der Grammatik zurück, löst die Zweideutigkeiten dort aber durch Präzedenzregeln auf, die zudem mit Assoziationsregeln für die Auswertung von Ausdrücken mit dem gleichen Operator gekoppelt sind.

```
%{
#include "stdio.h"
%}

%token NUMBER

%left '+'
%left '*'

%%
statement: expression '=' { printf("%d\n", $1); }
  ;

expression: expression '+' expression { $$ = $1 + $3; }
  |         expression '*' expression { $$ = $1 * $3; }
  |         '(' expression ')'        { $$ = $2; }
  |         NUMBER                   { $$ = $1; }
  ;

%%

void yyerror( char * err )
{
  fprintf( stderr, "%s\n", err );
}

int main ( int argc, char ** argv )
{
  yyparse();
```

```
  return 0;
}
```

Die Anweisungen in geschweiften Klammern auf der rechten Seite der syntaktischen Regeln beziehen sich auf den von Bison immer mitgeführten Parse-Stack: Dabei stehen `$$` für den bei der Abarbeitung der Regel auf dem Stack zurückgelassenen Wert und `$1` bis `$3` für die Werte der entsprechenden Stack-Positionen.

3.2 Definitionen

Ein *Alphabet* ist eine endliche Menge, deren Elemente wir auch *Symbole* nennen. *Worte* über einem Alphabet A sind endliche Folgen von Symbolen aus A. Die Menge A* der Worte über A ist mit der Operation der *Juxtaposition*, des 'Anfügens', eine Halbgruppe mit einem Einselement – dem leeren Wort 0 – , in der Halbgruppentheorie kennzeichnet man diese Halbgruppe auch als das 'freie Monoid über A'. Die *Länge* eines Wortes ist die Anzahl der in ihm vorkommenden Symbole. Sind x, y, z Worte, so nennen wir x einen *Anfang* und z ein *Ende* von xyz.

Eine *Produktion* ist ein Paar (U,u), wobei U ein Symbol und u ein (ggf. leeres) Wort ist. Die übliche Schreibweise für eine Produktion ist

```
U ::= u
```

mitunter gebraucht man auch den Ausdruck *Ableitungsregel* anstelle von Produktion.

Eine *Grammatik* G besteht aus einer endlichen nichtleeren Menge von Produktionen und einem Symbol, das auf der linken Seite mindestens einer Produktion vorkommen muss; dieses ausgezeichnete Symbol heißt auch das Startsymbol von G. Die Symbole, die in den linken und rechten Seiten von G vorkommen, nennt man auch das *Vokabular* von G; die Symbole, die in den linken Seiten vorkommen heißen auch *Nichtterminale*, die übrigen *Terminale*.

Im Sinne dieser Definition bilden die Regeln 3.4 - 3.6 – mit der Interpretation aller Regeln als Paare von Produktionen – eine Grammatik mit dem Startsymbol expr, dem Vokabular $\{expr, term, factor, number, +, *, (,)\}$ und der Terminalmenge $\{number, +, *, (,)\}$.

Die durch eine Grammatik G definierte *Sprache* ist die Menge aller nur aus Terminalen bestehenden Worte, die aus dem Startsymbol abgeleitet werden können.

Um den Ableitungsbegriff zu präzisieren, definieren wir zunächst die Relation der *direkten Ableitbarkeit* durch

$$v \;\delta\; w \iff \textit{Es gibt } x, y \textit{ und eine Regel } U ::= u \textit{ mit } v = xUy \textit{ und } w = xuy. \tag{3.9}$$

Die *Ableitbarkeit* Δ ist dann die transitive Hülle der direkten Ableitbarkeit.

Die in obigem Beispiel durch die Grammatik definierte Sprache besteht demzufolge gerade aus der Menge aller – korrekt gebildeten – formalen arithmetischen Ausdrücke.

Ein Element der durch eine Grammatik definierten Sprache heißt auch ein *Satz*; hingegen nennen wir ein Wort, das aus dem Startsymbol ableitbar ist, aber nicht notwendig nur aus Terminalen besteht, auch eine *Satzform*.

Wie wir in dem einführenden Beispiel gesehen haben, kann es für ein- und dieselbe Sprache durchaus unterschiedliche Grammatiken geben, die sie beschreiben. Die Wahl einer Grammatik für eine gegebene Sprache wird im allgemeinen nach praktischen Gesichtspunkten erfolgen.

In den meisten Anwendungsfällen wird man eine Grammatik suchen, die eine vorgegebene unendliche Sprache beschreibt. Man kann sich leicht überlegen, dass das prinzipiell nur durch rekursive Regeln möglich ist. Wir nennen eine Grammatik *rekursiv* in einem Nichtterminal U, wenn es Worte x, y gibt mit U Δ xUy; sind hierbei x bzw. y leer, so nennt man die Grammatik auch *linksrekursiv* bzw. *rechtsrekursiv* in U.

Ein wesentliches Hilfsmittel zur Veranschaulichung von Ableitungen von Sätzen – und damit ihrer 'Struktur' – sind die sog. *Syntaxbäume*. Abbildung 3.1 auf Seite 17 zeigt einen Syntaxbaum für die Ableitung des Satzes $number * (number + number)$ aus dem Startsymbol expr unserer Beispielsgrammatik.

Betrachten wir dagegen die Grammatik für arithmetische Ausdrücke, die dem Definitionsversuch 3.1-3.2 entspricht,

$$expr ::= number \tag{3.10}$$

$$expr ::= expr \;+\; expr \tag{3.11}$$

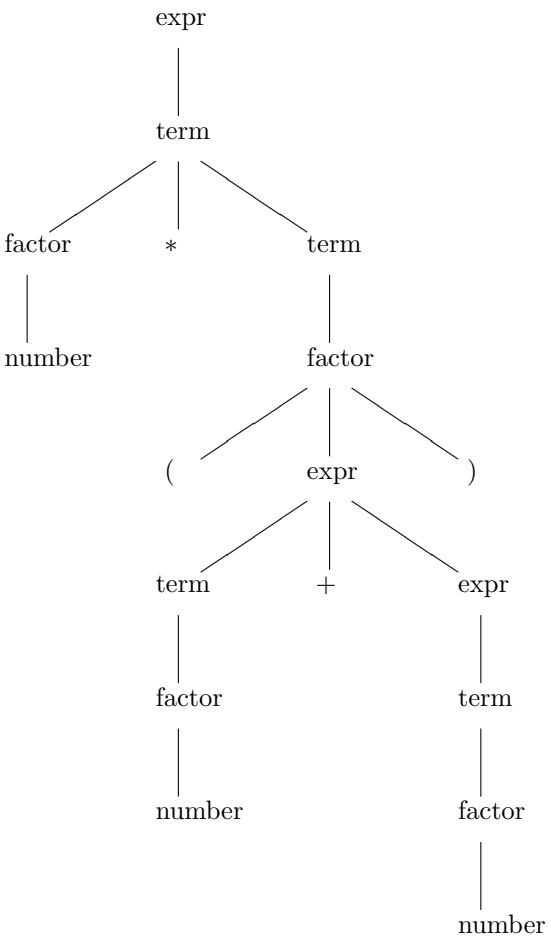


Abbildung 3.1: Syntaxbaum für $number * (number + number)$

(3.12) $expr ::= expr \ * \ expr$

(3.13) $expr ::= (\ expr\ )$

so finden wir zwei wesentlich verschiedene Syntaxbäume für den formalen arithmetischen Ausdruck $number + number * number$.

Im allgemeinen wollen wir einen Satz einer Sprache *zweideutig* (in Bezug auf eine zugrundeliegende Grammatik) nennen, wenn er zwei verschiedene Syntaxbäume besitzt, und eine Grammatik *zweideutig*, wenn die durch sie definierte Sprache einen zweideutigen Satz enthält. Es ist gezeigt worden, dass das Zweideutigkeitsproblem für Grammatiken unentscheidbar ist, d.h. es gibt keinen Algorithmus, der für eine beliebig vorgegebene Grammatik in endlicher Zeit entscheidet, ob sie zweideutig ist oder nicht. Man wird sich daher mit geeigneten hinreichenden Kriterien für die Unzweideutigkeit zufrieden geben müssen.

Die Aufgabe der Syntax-Analyse besteht darin, zu einer vorgegebenen Satzform eine Ableitung aus dem Startsymbol zu finden – oder den zugehörigen Syntaxbaum zu konstruieren. In der Praxis findet man nur Analysatoren, die ein Wort von links nach rechts analysieren, das entspricht auch unserem normalen Sprachempfinden; man unterscheidet jedoch Parser nach der Art, wie sie den Syntax-Baum konstruieren, in Top-Down und Bottom-Up-Parser: Der Top-Down-Parser geht vom Startsymbol aus und versucht, auf Grund des vorliegenden Wortes geeignete Produktionen zu finden, die zu diesem Wort führen. Das zu Beginn des Kapitels vorgestellte Programm zur Auswertung von arithmetischen Ausdrücken war von dieser Art, so dass wir an dieser Stelle auf eine eingehendere Betrachtung verzichten können.

Im Gegensatz dazu versucht der Bottom-Up-Parser die vorliegende Satzform symbolweise aufgrund der Ableitungsregeln solange zu *reduzieren*, bis das Startsymbol auftritt. Wir werden später noch ausführlich auf die beiden unterschiedlichen Analysemethoden zu sprechen kommen.

Wir wollen dieses Kapitel abschließen mit erweiterten Syntaxnotationen und einer knappen Einordnung unseres Grammatikbegriffs in allgemeinere Begriffsbildungen.

Unsere bisherige Notation von Produktionen entspricht im wesentlichen der sog. Backus-Naur-Form (BNF). In der erweiterten Backus-Naur-Form (EBNF) kennt man zwei zusätzliche Konstrukte, die Iteration und ein optionales Syntaxelement, die in der Form

```
U ::= { u }
```

bzw.

```
U ::=  u [ v ] w
```

dargestellt werden und jeweils Kurzformen für U ::= λ (leer), U ::= uU (d.h. für kein oder mehrmaliges Auftreten von u) bzw. U ::= uw, U ::= uvw (d.h. für ein optionales Auftreten von v) sind. Mit dem Element der Iteration kann man allerdings bei der Analyse tatsächlich eine Rekursion sparen und diese durch eine while- oder ggf. do-Schleife ersetzen. Damit ließe sich dann unsere Beispielsgrammatik kürzer durch

(3.14) $$expr ::= term\{ + term \}$$

(3.15) $$term ::= factor\{ * factor \}$$

(3.16) $$factor ::= number \mid (expr)$$

beschreiben. (Im Sinne obiger Bemerkung wäre auch das zugehörige Programm leicht zu modifizieren.)

In der allgemeinen Sprachtheorie definiert man nach Chomsky (1956) eine Grammatik als ein Quadrupel (V,T,P,Z), wobei V ein Alphabet, T eine nichtleere Teilmenge von V (die Menge der Terminale), P eine endliche Menge von Produktionen und Z (das Startsymbol) ein Element von V \ T ist. Man unterscheidet je nach Gestalt der Produktionen vier Typen von Grammatiken:

Eine Grammatik vom *Typ 0* (Phrasen-Struktur-Grammatik) hat Produktionen der Form u ::= v, wobei u ein nichtleeres Wort und v ein beliebiges Wort ist. Entsprechend der Allgemeinheit der Definition gibt es wenig konkrete Ergebnisse zu solchen Grammatiken.

Eine Grammatik vom *Typ 1* (Kontext-Sensitive Grammatik) hat Produktionen der Form xUy ::= xuy, wobei x,y beliebige Worte, u ein nichtleeres Wort und U ein Nichtterminal ist; diese Produktionen sollen ausdrücken, dass u aus U nur in dem Kontext xUy ableitbar ist. Kontext-sensitive Grammatiken sind ein Versuch, natürliche Sprachen zu beschreiben.

Im Gegensatz dazu stehen die Grammatiken vom *Typ 2* (Kontextfreie Grammatiken), die sich durch Produktionen der Form U ::= u auszeichnen, wobei U ein Nichtterminal und u ein beliebiges Wort ist. Bis auf die Forderung, dass die rechte Seite einer Produktion ein nichtleeres Wort sein soll, sind die kontextfreien Grammatiken genau die, die wir unserer Betrachtung zugrundegelegt haben.

Eine wichtige Unterklasse der kontextfreien Grammatiken bilden die Grammatiken vom *Typ 3* (reguläre Grammatiken), bei denen die Produktionen auf die Form U ::= u und U ::= Vu eingeschränkt sind; dabei müssen u ein Terminal und U sowie V Nichtterminale sein. Die regulären Grammatiken stehen in einem engen Zusammenhang mit den endlichen Automaten (daher auch der Name) und erlauben sehr effiziente Analysetechniken; wir werden im folgenden Kapitel im Zusammmenhang mit dem Scanner speziell auf reguläre Grammatiken eingehen.

Übungsaufgaben

1. Nennen Sie kurz den wesentlichen Unterschied zwischen einem Compiler und einem Interpreter. Welche zwei Varianten eines Interpreters gibt es?

2. Nennen Sie die drei Hauptkomponenten eines Compilers und ihre Aufgaben.

3. Erweitern Sie das Beispielprogramm der Vorlesung so, dass es auch die Speicherung von Zwischenergebnissen unter symbolischen Namen und die Verwendung der Zwischenergebnisse in Ausdrücken zulässt. (Der Einfachheit halber können Sie annehmen, dass die symbolischen Namen nur aus einem Zeichen zwischen 'a' und 'z' bestehen.)

   Erweitern Sie dazu die Grammatik wie folgt:

   ```
   stmt_list ::= stmt | stmt stmt_list

   stmt      ::= ident "=" expr  | "?" expr

   expr      ::= term | term "+" expr

   term      ::= factor | factor "*" term

   factor    ::= number | ident | ( expr )
   ```

4. Welche Art von Problemen bringt die (gewohnte) Syntaxdefinition von arithmetischen Ausdücken gemäß

   ```
   expr  ::= number |
             "(" expr ")" |
             expr "+" expr |
             expr "*" expr
   ```

   mit sich?

5. Gegeben sei das Alphabet A mit den 4 Symbolen a, b, c, d.

   - Wieviele verschiedene Worte der Länge 4 gibt es über diesem Alphabet?
   - Die Grammatik G über A sei durch die Produktionen

     ```
     a ::=  bc | db

     c ::=  da | bd
     ```

     definiert, das Startsymbol von G sei a. Welches sind die Terminalsymbole und welches die Nichtterminalsymbole von G?

- Geben Sie einen Satz der Länge 8 über G an.
- Wie lassen sich die Sätze über G auch anders charakterisieren?
- Ist die Grammatik eindeutig?

6. Gegeben sei das Alphabet A mit den 4 Symbolen a, b, c, d. Die Grammatik G sei durch die EBNF-Regeln

```
a ::=  b { cb } d

c ::=  d [ b ] d
```

   definiert. Geben Sie BNF-Regeln an, die die gleiche Sprache festlegen. (Ggf. müssen Sie dazu das Alphabet erweitern.)

7. In der Dokumentation zu JSON (Java Script Object Notation) findet sich folgende Grammatik:

```
object
    {}
    { members }
members
    pair
    pair , members
pair
    string : value
array
    []
    [ elements ]
elements
    value
    value , elements
value
    string
    number
    object
    array
    true
    false
    null
```

   Stellen Sie eine äquivalente EBNF-Formulierung für JSON auf, die mit vier Regeln für *object*, *pair*. *array*, *value* auskommt.

Kapitel 4

Der Scanner

Wie wir schon in der Einleitung angemerkt haben, besteht die Aufgabe des Scanners darin, die lexikalische Analyse des Eingabetextes durchzuführen. Darüberhinaus wird man dem Scanner alle solchen Aufgaben überlassen, die sich nicht auf die Syntaxanalyse auswirken, wie z.B. das Schreiben eines Listings oder das Überlesen von Kommentaren.

Prinzipiell ist es sicher möglich, auch die lexikalische Analyse als Teil der syntaktischen Analyse anzusehen und damit den Parser zu beaufschlagen; es gibt aber eine Reihe von Gründen, die für eine Separation sprechen.

An erster Stelle sind hier Effizienzüberlegungen zu nennen: Der Compiler benötigt viel Zeit dazu, Einzelzeichen zu lesen. Eine separate Behandlung dieses Teils ermöglicht es, ihn entsprechend zu optimieren, was wegen der einfacheren Regeln, die der lexikalischen Analyse zugrundeliegen, auch relativ problemlos ist. Diese einfachen Regeln erlauben es darüberhinaus auch, effizientere Analysetechniken zu verwenden. Hierher gehört auch die Überlegung, dass der Scanner dem Parser jeweils ein vollständiges Symbol übergibt und dieser damit mehr Information darüber besitzt, was im nächsten Schritt zu tun ist.

An zweiter Stelle stehen Portabilitätsüberlegungen: Eine Trennung von Scanner und Parser ermöglicht es, einerseits den gleichen Parser mit verschiedenen Scannern, die etwa unterschiedliche Hardwareanforderungen respektieren, zu kombinieren, andererseits wird für ähnliche Sprachen oft der gleiche Scanner mit ggf. geringfügigen Modifikationen einsetzbar sein.

Wie schon erwähnt könnte ein Scanner so arbeiten, dass er in einem ersten Pass eine vollständige Analyse des Quellenprogramms durchführt und der Parser dann in einem zweiten die Syntaxanalyse. Im allgemeinen ist es aber besser, den

Scanner als Unterprogramm des Parsers anzusehen, das immer dann gerufen wird, wenn der Parser ein neues Symbol benötigt; wir wollen diese Sichtweise für den Rest der Vorlesung einnehmen.

Die meisten Symbole üblicher Programmiersprachen lassen sich in folgende Kategorien einteilen: Namen, Schlüsselwörter (das sind besondere Namen), Zahlen, Operatoren und Delimiter wie +, -, ; usw. In manchen Sprachen gibt es darüberhinaus auch Delimiter, die aus zwei und mehr Einzelzeichen bestehen, wie etwa die Wertzuweisung := in PASCAL oder in C die Kommentareinleitung /∗, sowie etwa die Operatoren += bzw. ||. All diese Symbole lassen sich – wie hier vereinfacht angedeutet – durch folgende Regeln beschreiben:

```
identifier ::= letter | identifier letter
               | identifier digit
number     ::= digit | number digit
delimiter  ::= "(" | ")" | "*" | ";" |
                plus "=" | plus "+" | bar "="
                bar "|" | ...
plus       ::= "+"
bar        ::= "|"
```

(Wir haben hier, um deutlich zwischen dem Metasymbol | und dem Zeichen | zu unterscheiden, letzteres – und auch alle übrigen Einzelzeichensymbole – in Anführungszeichen gesetzt.) Die separate Behandlung von +, / und | hat ihren Grund darin, dass die Grammatik auf diese Weise regulär im Sinne von Kapitel 3 ist, d.h. jede Regel eine der beiden Formen

```
U ::= t
U ::= Vt
```

wobei U, V Nichtterminale und t ein Terminal sind. Wir wollen an dieser Stelle auf den schon erwähnten Zusammenhang mit den endlichen Automaten eingehen.

Exkurs: Endliche Automaten

Ein (deterministischer) *endlicher Automat* ist ein 5-Tupel (I, S, d, a, E), das folgenden Bedingungen genügt:

(4.1) *I ist die endliche Menge der Eingabesymbole.*

(4.2) *S ist die endliche Menge der Zustände.*

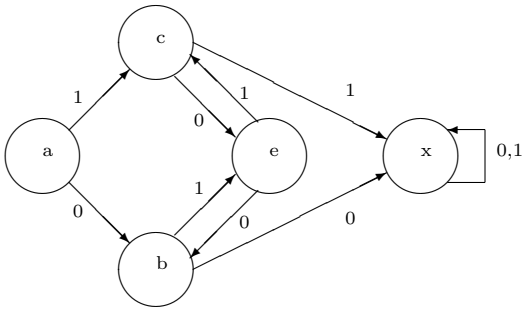


Abbildung 4.1: Einfacher Zustandsautomat

(4.3) *d ist die Übergangsabbildung* $d : I \times S \longrightarrow S,.$

(4.4) $a \in S$ *ist der Anfangszustand.*

(4.5) $E \subseteq S$ *ist die Menge der Endzustände.*

Endliche Automaten veranschaulicht man sich oft durch ein sog. Zustandsdiagramm. Dies ist ein gerichteter Graph mit S als Eckenmenge; zwei Zustände werden durch einen Pfeil verbunden, wenn es ein Eingabesymbol gibt, das den einen Zustand in den anderen überführt; das entsprechende Eingabesymbol wird an diesem Pfeil notiert. Abbildung 4.1 auf Seite 25 zeigt ein solches Diagramm für einen nicht sehr komplizierten Automaten.

Die Vorstellung, die man mit einem endlichen Automaten verbindet, ist die, dass er dazu dient, gewisse Folgen von Eingabesymbolen in dem Sinne zu erkennen, dass diese Folgen den Automaten vom Anfangszustand in einen der Endzustände überführen. Um diese Vorstellung zu präzisieren, erweitern wir die Übergangsabbildung d zu einer Abbildung $d^* : I^* \times S \longrightarrow S$ durch

(4.6) $$d^*(\lambda, s) = s \qquad (s \in S)$$

(4.7) $$d^*(xw, s) = d(w, d^*(x, s)) \qquad (x \in I^*, w \in I, s \in S)$$

Der Automat A *akzeptiert* ein Wort $x \in I^*$, wenn $d^*(x, a) \in E$.

Wie man sich leicht überlegt, akzeptiert der Automat von Abbildung 4.1 auf Seite 25 mit a als Startzustand und { e } als Endzustandsmenge genau die Folgen von Nullen und Einsen, die sich aus den beiden Blöcken 01 und 10 zusammensetzen lassen.

Einer der Hauptsätze der Automatentheorie charakterisiert die Mengen von Worten, die von einem endlichen Automaten akzeptiert werden können, als die regulären Teilmengen von I*. Dabei ist die Menge der regulären Mengen die kleinste Teilmenge der Potenzmenge von I*, die die endlichen Mengen umfasst und abgeschlossen ist unter Vereinigung, Durchschnitt, Komplement, sowie Komplexprodukt und Erzeugnisbildung bez. der Multiplikation in I*.

Der schon angedeutete Zusammenhang mit den regulären Grammatiken findet sich in folgendem Satz:

Satz 4.1 *Es sei G eine reguläre Grammatik, deren Produktionen eindeutige rechte Seiten besitzen. Dann gibt es einen endlichen Automaten, der die Sprache von G akzeptiert.*

Beweis: Wir beschränken uns hier darauf, die Konstruktion des gesuchten Automaten anzugeben. Es sei dazu N die Menge der Nicht-Terminale von G, o.B.d.A. $a, f \notin V(G)$. Dann setzen wir $S = N \cup \{a, f\}$, $I = V(G) \setminus N$ sowie $E = \{Z\}$. Die Übergangsabbildung d definieren wir durch:

$$d(t, a) = \begin{cases} n & \text{falls es eine Regel gibt mit } n ::= t \\ f & \text{sonst} \end{cases}$$

$$d(t, f) = f$$

$$d(t, s) = \begin{cases} n & \text{falls es eine Regel gibt mit } n ::= st \\ f & \text{sonst} \end{cases}$$

für alle $t \in I$, $s \in S$. Die vorausgesetzte Rechtseindeutigkeit der Produktionen garantiert, dass d wirklich eine Abbildung ist.

Der hiermit konstruierte Automat ist gewissermaßen ein Bottom-Up-Parser für die Grammatik G. Will (oder muss) man auf die Rechtseindeutigkeit der Pro-

duktionen verzichten, so handelt es sich bei d – wie oben konstruiert – nicht mehr um eine Abbildung, sondern um eine Relation. Wenn man die Definition des Automaten entsprechend modifiziert, erhält man den Begriff eines nicht-deterministischen Automaten. Man zeigt in der Automatentheorie, dass es zu jedem nicht-deterministischen endlichen Automaten einen deterministischen endlichen Automaten (mit ggf. sehr viel mehr Zuständen) gibt, der die gleiche Akzeptanzmenge hat. Damit ergeben sich zumindest theoretisch keine zusätzlichen Schwierigkeiten.

Umgekehrt kann man die vorgestellte Beweisidee auch dazu verwenden, um für einen endlichen Automaten eine Grammatik zu finden, deren Sprache gerade die Akzeptanzmenge des Automaten ist. Man überlegt sich leicht, dass die Produktionen

$$E ::= C\ 0 \mid B\ 1 \tag{4.8}$$

$$C ::= 1 \mid E\ 1 \tag{4.9}$$

$$B ::= 0 \mid E\ 0 \tag{4.10}$$

mit dem Startsymbol E genau die Sprache definieren, die der Automat aus Abbildung 4.1 akzeptiert.

Wir wollen nun nach diesem Exkurs konkret darauf eingehen, wie man einen Scanner sinnvoll programmiert. Dabei wollen wir uns an die oben schon angeführten Regeln halten, d.h. die Symbole der von uns angedachten Sprache bestehen aus Namen, (vorzeichenlosen ganzen) Zahlen, Delimitern; hinzu kommen noch Schlüsselwörter. Weitere Vereinbarungen sind, dass Blanks (Leerzeichen) zwar Symbole trennen, aber sonst keine weitere Bedeutung haben (also sind z.B. die Zeichenfolgen + = und + = einmal ein und das andere Mal zwei Delimiter); Kommentare werden durch // eingeleitet und durch das Zeilenende beendet.

Der Scanner übergibt dem Parser eine interne Darstellung jedes von ihm erkannten Symbols. Steht eine Sprache mit Aufzähltypen (wie etwa PASCAL oder C) zur Verfügung, so ist es sinnvoll, einen Datentyp *SymbolType* zu vereinbaren, der jedes vorkommende Sprachsymbol genau einmal enthält. Vom Scanner erkannte Namen (Identifier) und auch Zahlen werden jeweils als ein Symbol betrachtet; die Information, um welchen Namen bzw. um welche Zahl es sich handelt, wird separat übergeben.

Um einen Namen oder eine Zahl zu erkennen, muss der Scanner das auf den Namen (bzw. die Zahl) folgende Zeichen bereits gelesen haben, bei einem Ein-Zeichen-Delimiter hingegen nicht. Aus Gründen der Zweckmäßigkeit fordern wir,

dass der Scanner auch in diesen Fällen schon das nächste Zeichen gelesen hat, bevor er das Symbol an den Parser übergibt.

Zur weiteren Implementation orientieren wir uns zunächst an dem Zustandsdiagramm, das den o.a. Regeln entspricht; dieses Diagramm findet sich in Abbildung 4.2 auf Seite 29. Hierbei ist unter jedem nicht gelabelten Pfeil die Alternative zu den vom gleichen Zustand ausgehenden gelabelten Pfeilen zu verstehen. *start* ist der Anfangs- und *ziel* der (einzige) Endzustand des Scanners; die Verarbeitung von Kommentaren ist noch nicht eingebaut. Wenn wir dies noch integrieren und der Sprache exemplarisch einige Schlüsselwörter stiften, kommen wir zu folgendem Gerüst eines Scanners:

```
public SymbolType GetNextSymbol()
{
  SymbolType symbol = SymbolType.SymNull;
  bool inComment;
  do
  {
    inComment = false;
    // Whitespaces überlesen
    while (isspace (nextchar) ) getNextChar();
    if (isdigit(nextchar))
    {
      number();
      symbol = SymbolType.SymNumber;
    }
    else if (isalpha(nextchar))
    {
      identifier();
      symbol = checkForKeyword();
    }
    else
    {
      switch (nextchar)
      {
        case '+':
          getNextChar();
          if (nextchar == '+')
          {
            getNextChar();
            symbol = SymbolType.SymPlusPlus;
          }
          else if (nextchar == '=')
          {
            getNextChar();
            symbol = SymbolType.SymPlusEqual;
          }
          else symbol = SymbolType.SymPlus;
          break;
```

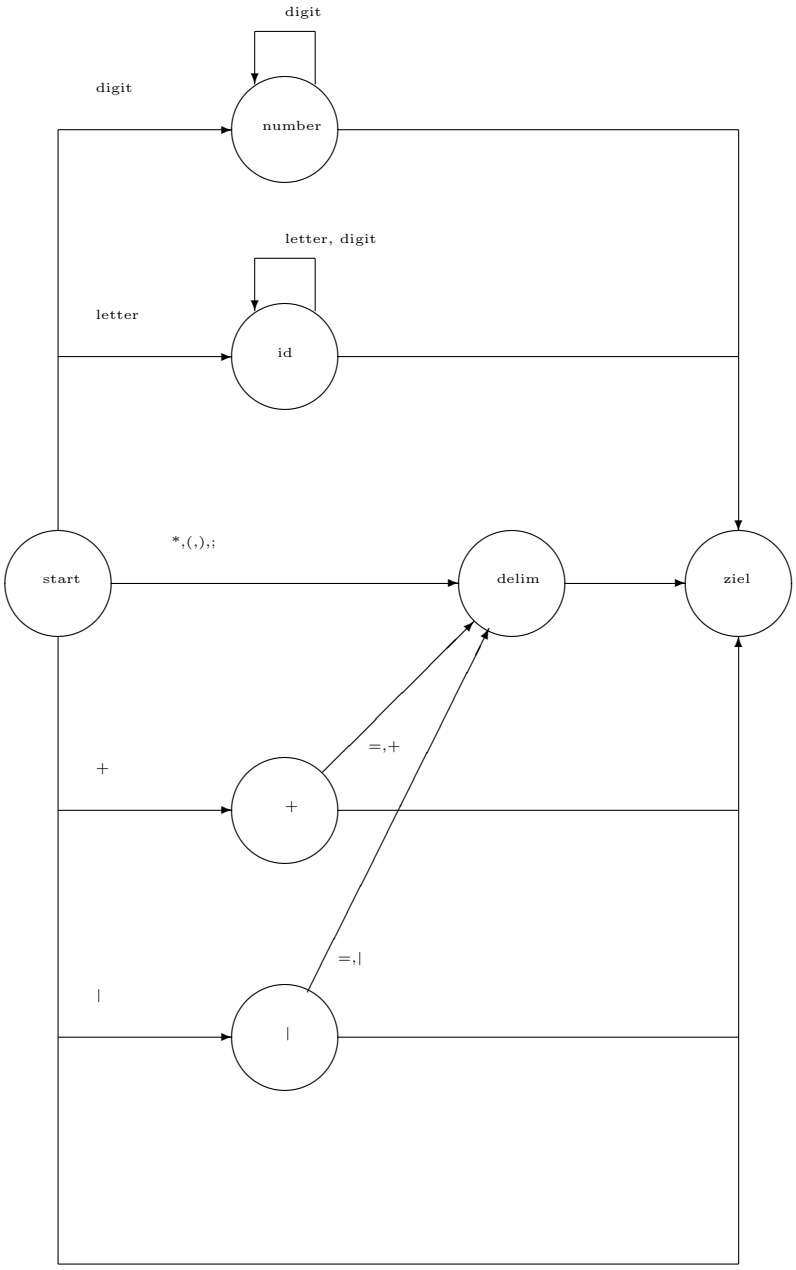


Abbildung 4.2: Zustandsübergangsdiagramm für den Scanner

```
        case '|':
          getNextChar();
          if (nextchar == '|')
          {
            getNextChar();
            symbol = SymbolType.SymOrOr;
          }
          else symbol = SymbolType.SymOr;
          break;
        case '/':
          getNextChar();
          if (nextchar == '/')
          {
            comment();
            inComment = true;
          }
          else symbol = SymbolType.SymDiv;
          break;
        case '*':
          getNextChar();
          symbol = SymbolType.SymTimes;
          break;
        case '(':
          getNextChar();
          symbol = SymbolType.SymLeftParen;
          break;
        case ')':
          getNextChar();
          symbol = SymbolType.SymRightParen;
          break;
        case '=':
          getNextChar();
          symbol = SymbolType.SymEqual;
          break;
        case ';':
          getNextChar();
          symbol = SymbolType.SymSemicolon;
          break;
        default:
          getNextChar();
          // Console.WriteLine("Illegal Character {0} - ignored", nextchar);
          break;
      }
    }
  } while (inComment);
  return symbol;
}
```

Die in diesem Gerüst verwendeten Variablen und Funktionen haben im einzelnen

folgende Bedeutung:

Der Rückgabewert *symbol* ist vom Aufzähltyp SymbolType; weitere Zusatzinformationen werden in den globalen Variablen *numberValue*, *identValue*, *stringValue* hinterlegt und die Variable *nextchar* enthält das jeweils aktuell gelesene Zeichen.

```
int nextchar;
int numberValue;
String identValue;

public enum SymbolType { SymNull,
  SymNumber, SymIdentifier,
  SymIf, SymElse, SymWhile, SymPrint,
  SymPlus, SymTimes, SymDiv,
  SymPlusPlus, SymPlusEqual, SymOr, SymOrOr,
  SymLeftParen, SymRightParen, SymEqual, SymSemicolon,
};
```

getNextChar ist eine Methode, die das jeweils nächste Zeichen aus dem Eingabestream holt, im Bedarfsfall sorgt *getNextChar* auch für das Schreiben des Listings und das zeilenweise Einlesen.

Die Methode *number* besorgt das Einlesen und Zusammensetzen einer Zahl, der Wert der Zahl wird in einer globalen Variablen *numberValue* übergeben. Eine einfache Variante, die Dezimalzahlen (ohne Vorzeichen) analysiert, könnte folgendermaßen aussehen:

```
void number()
{
  numberValue = nextchar - '0';
  getNextChar();
  while (isdigit(nextchar))
  {
    numberValue *= 10;
    numberValue += nextchar - '0';
    getNextChar();
  }
}
```

identifier ist eine Methode, die das Einlesen und Zusammensetzen eines Namens übernimmt. Der Name wird in der String-Variablen *identValue* abgelegt.

```
void identifier()
{
  identValue = "";
```

```
  identValue += (Char)nextchar;
  getNextChar();
  while (isalnum(nextchar))
  {
    identValue += (Char)nextchar;
    getNextChar();
  }
}
```

checkForKeyword ist eine Funktion, die überprüft, ob die – in der Methode *identifier* ermittelte – Zeichenkette ein Schlüsselwort ist. Sie gibt ggf. das entsprechende Symbol zurück, anderenfalls das Symbol SymIdentifier.

```
SymbolType checkForKeyword()
{
  SymbolType symbol = SymbolType.SymIdentifier;
  switch (identValue)
  {
    case "if":
      symbol = SymbolType.SymIf;
      break;
    case "print":
      symbol = SymbolType.SymPrint;
      break;
    case "else":
      symbol = SymbolType.SymElse;
      break;
    case "while":
      symbol = SymbolType.SymWhile;
      break;
    // no default necessary;
  }
  return symbol;
}
```

Die Methode *comment* überliest die restlichen Zeichen der aktuellen Zeile einschließlich des Zeilenendes.

```
void comment()
{
  while ( nextchar != '\n' )
  {
    getNextChar();
  }
  getNextChar();
}
```

Nach einem Kommentar sorgt das Flag *inComment* dafür, dass mit dem Einlesen der nächsten Zeile fortgesetzt wird.

Zum Abschluss des Abschnittes soll hier – ohne auf Details der Implementierung einzugehen – erwähnt werden, dass das Unix-Werkzeug *lex* bzw. dessen Open-Source-Variante *flex* die in diesem Kapitel vorgestellten Strategien umsetzen. Eine Formulierung der lexikalischen Struktur einer fiktiven Sprache, die über die Rudimente des Beispiels von Kapitel 3 hinausgeht, könnte folgendermaßen aussehen:

```
%{
#include "string.h"
#include "parser.tab.h"
static int lineno = 1;
%}

NUMBER    [0-9]+|0x[0-9a-fA-F]+
ID        [a-zA-Z][a-zA-Z0-9]*

%%

[ \t]+    /* whitespace */
\n        ++ lineno;
\/\/.*    /* comment    */

{NUMBER}  { yylval.val = strtol(yytext,0,0); return NUM; }

if        return IF;
while     return WHILE;
else      return ELSE;
print     return PRINT;

{ID}      { yylval.id = strdup(yytext); return IDENT; }

+=        return PLUSEQUALS;
++        return PLUSPLUS;
\|\|      return LOR;

.         return yytext[0];

%%

void yyerror( const char * msg )
{
  fprintf( stderr, "Line %d near %s: %s\n", lineno, yytext, msg );
}
```

Eine Lex-Spezifikation besteht im allgemeinen aus drei durch %% getrenn-

ten Abschnitten, nämlich den *Definitionen*, den *Regeln* und den *Benutzer-Funktionen*. In unserem Beispiel beginnt der Definitionsabschnitt mit einem in %{ und }% geklammerten Block, der so in den von Lex generierten C-Code kopiert wird, und hier benötigte Include-Files und Initialisierungen enthält. Es folgt die Definition von Zahlen (inkl. Hex-Darstellungen) und Identifiern mit Hilfe von geeigneten regulären Ausdrücken.

Der Regel-Abschnitt beschreibt ebenfalls mit Hilfe regulärer Ausdrücke die Struktur von Trennern und Kommentaren, wobei hier Kommentare im C^{++}-Stil gewählt worden sind; die separate Regel für das Zeilenende dient lediglich der Zeilenzählung für Fehlermeldungen.

Die Werte von Symbolen, die hier im von yacc erzeugten Header-File parser.tab.h definiert sind, werden mit return-Statements im Code-Anteil jeder Regel zurückgegeben. Für das NUM-Token wird zusätzlich der Zahlenwert in der Komponente *val* der globalen Variablen *yylval* zur weiteren Verwendung durch den Parser notiert. Die Definition von Schlüsselworten, wie *if*, *while* etc. muss vor der Definition von Identifiern erfolgen. Bei Identifiern ist hier dafür Sorge getragen, dass der Identifier-String in die Komponente *id* von *yylval* kopiert wird. Der Parser muss zur Vermeidung von Speicherlecks darauf achten, diese Strings ggf. auch wieder freizugeben. Nach den Doppelzeichen-Symbolen wie ==, !=, && und || folgt dann noch eine Catchall-Regel, die alle übrigen Zeichen als ein lexikalisches Symbol betrachtet, das mit seinem ASCII-Wert codiert wird.

Der Abschnitt für die Benutzer-Funktionen enthält hier nur die Definition der vom Scanner (und Parser) benötigten Funktion *yyerror*, die eine Fehlermeldung mit Zusatzinformationen auf den Standard-Error-Kanal ausgibt.

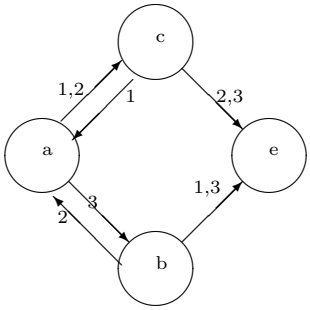
Für eine weitergehende Einführung in Lex (und Yacc) verweisen wir auf [5].

Übungsaufgaben

1. Nennen Sie zwei Gründe für die Trennung der Analyse eines Compilers in einen lexikalischen und einen syntaktischen Anteil.

2. Gibt es einen endlichen Automaten, dessen Akzeptanzmenge gerade die durch die Grammatik von Aufgabe 5, Blatt 1 definierte Sprache ist? Zeichnen Sie ggf. das Zustandsübergangsdiagramm dieses Automaten auf.

3. Gegeben sei der Automat A mit dem Zustandsübergangsdiagramm



   Versuchen Sie eine Grammatik anzugeben, die die Akzeptanzmenge von A (mit dem Startzustand a und dem Endezustand e) definiert.

4. Die Definition von Strings in der Sprache C ist wie folgt:

   Strings werden durch doppelte Hochkommas " begrenzt, die beliebige ASCII-Zeichen einschließen können. Kommt in einem String das doppelte Hochkomma selbst als Zeichen vor, so ist es durch ein vorangestelltes Backslash-Zeichen \ zu kennzeichen. Der Backslash selbst muss ebenfalls durch einen vorangestellten Backslash gekennzeichnet werden.

- Definieren Sie mit EBNF-Regeln einen String
- Schreiben Sie - unter Verwendung der Funktion *getnextchar* - eine Funktion *getstring*, die die lexikalische Analyse eines Strings erledigt und den String selbst in der globalen Variablen $Vstring$ ablegt. $Vstring$ kann der Einfachheit halber als Charakter-Array der maximalen Länge 128 angenommen werden, längere Strings dürfen abgeschnitten werden.)

5. Die Syntax einer zu analysierenden Sprache kenne die Schlüsselwörter 'if', 'then', 'else'. Identifier der Sprache beginnen mit einen Alpha-Zeichen, folgen dürfen beliebige alphanumerische Zeichen.

   - Definieren Sie mit EBNF-Regeln einen Identifier
   - Schreiben Sie - unter Verwendung der Funktionen *getnextchar*, *isalpha*, *isalnum* eine Funktion $identifier$, die die Analyse eines Identifiers besorgt und am Ende einen Vergleich mit den drei Schlüsselwörtern durchführt, wobei zwischen Groß- und Kleinschreibung nicht unterschieden werden soll. Die Funktion soll ihr Analyseergebnis in der globalen Variablen Vsymbol in Form der Konstanten symIdent, symIf, symThen, symElse ablegen - und im Falle eines Identifiers den Inhalt in der Variablen Vname (ggf. auf 64 Zeichen) gekürzt.

Kapitel 5

Der Parser

Aufgabe des Parsers ist es, die syntaktische Analyse des Eingabetextes – aufsetzend auf den Ergebnissen der durch den Scanner besorgten lexikalischen Analyse – durchzuführen. Wie schon in Kapitel 3 erwähnt unterscheidet man nach der Art der Abarbeitung der Syntax-Bäume Top-Down- und Bottom-Up-Parser, auf die wir im folgenden gesondert eingehen wollen.

5.1 Top-Down-Parser

Die Vorgehensweise des Top-Down-Parsers besteht im Prinzip darin, aus dem Startsymbol mit Hilfe der Produktionsregeln die (vorliegende) Folge der Symbole des Eingabetextes abzuleiten.

Praktische Bedeutung unter den Top-Down-Parsern haben eigentlich nur diejenigen, die nach der Methode des *rekursiven Abstiegs* (recursive descent) vorgehen; wir wollen uns daher auch auf solche beschränken. Bei einem solchen Parser gibt es für jedes Nicht-Terminal-Symbol eine spezielle Prozedur – meist durch einen entsprechenden Namen angedeutet – deren Aufgabe es ist, die zu diesem Nicht-Terminal gehörende(n) Produktionsregel(n) am Eingabetext nachzuvollziehen, d.h. insbesondere die Eingabesymbole in der Reihenfolge, wie sie der Scanner liefert, entgegenzunehmen und zu verarbeiten; ein Rückgriff auf ein früheres Eingabesymbol (Backup) ist nicht möglich.

Ist im einfachsten Fall die rechte Seite einer Produktion eine Sequenz etwa der Form

```
U ::= A b C
```

so besteht die zugehörige syntaktische Routine letztlich nur aus den Aufrufen der zu A, b und C gehörigen Prozeduren; hinzu kommt allerdings noch jeweils die Überprüfung, dass das gerade vorliegende Symbol eines der zulässigen Startsymbole des entsprechenden Nicht-Terminals ist bzw. mit dem gerade erwarteten Terminal übereinstimmt. Sind also etwa A und C Nicht-Terminale und b ein Terminalsymbol, so könnte der Code der syntaktischen Routine folgendermaßen aussehen:

```
void U()
{
  A();
  mustBe( sym_b );
  nextSymbol = scanner.GetNextSymbol();
  mustBeIn ( head_C );
  C();
}
```

Dabei sind *mustBe* und *mustBeIn* Methoden, die die o.g. Überprüfung auf Zulässigkeit des aktuellen Symbols – und ggf. auch die Ausgabe von Fehlermeldungen im Sinne von 'b erwartet' und einer geeigneten Fehlerreaktion – vornehmen.

mustbe erwartet ein Symbol, *mustBeIn* eine Menge von Symbolen. head_C steht hier für die Menge der Startsymbole von C. Es ist zu beachten, dass die Überprüfung zu Beginn der Routine U deshalb entfallen kann, weil davon auszugehen ist, dass diese Überprüfung bereits in der Ebene der aufrufenden Funktion stattgefunden hat.

```
private void mustBe(Scanner.SymbolType sym)
{
  if (nextSymbol != sym) syntaxError(Scanner.GetText(sym) + " expected");
}

private void mustBeIn(HashSet<Scanner.SymbolType> syms)
{
  if (! syms.Contains(nextSymbol) )
  {
    String message = "";
    foreach(Scanner.SymbolType sym in syms) message += Scanner.GetText(sym) + " ";
    message += "expected";
    syntaxError(message);
  }
}
```

Den weiteren Grundkonstrukten der EBNF wie Alternativen, Iterationen, Optionen und Klammern entsprechen in dieser Reihenfolge switch-, while-, if-Anweisungen und Blöcke.

D.h. der Produktion

```
U ::= A | b | C
```

entspricht die syntaktische Routine

```
void U()
{
  switch ( symbol )
  {
    case sym_a1 :
    case sym_a2 :
      A();
      break;
    case sym_b  :
      nextSymbol = scanner.GetNextSymbol();
      break;
    case sym_c1 :
      C();
      break;
  }
}
```

Hierbei sind sym_a1, sym_a2 die Startsymbole von A, sym_c1 das einzige Startsymbol von C.

Der Produktion

```
U ::= A { b C }
```

entspricht die syntaktische Routine

```
void U()
{
  A();
  while ( nextSymbol == sym_b ) ) {
    nextSymbol = scanner.GetNextSymbol();
    mustBeIn(head_C);
    C();
  }
}
```

Wenn die Iteration anstelle eines Terminalsymbols mit einem Nicht-Terminal beginnt, das mehr als ein Anfangssymbol besitzt, kann statt des direkten Vergleichs eine Funktion *isIn* – in Analogie zu *mustBeIn* – verwendet werden, die als Ergebnis zurückgibt, ob das aktuelle Symbol in der entsprechenden Menge enthalten ist.

```
private bool isIn(HashSet<Scanner.SymbolType> syms)
{
  return syms.Contains(nextSymbol);
}
```

Schließlich entspricht der Option – exemplarisch in der Produktion

```
  U ::= A [ b C ]
```

die syntaktische Routine

```
void U()
{
  A();
  if ( symbol == sym_b ) {
    nextSymbol = scanner.GetNextSymbol();
    mustBeIn(head_C);
    C();
  }
}
```

entspricht.

Für die Anwendung gibt es allerdings zwei in der Regel nicht zu vernachlässigende Probleme:

Zum einen setzt das Verfahren voraus, dass die Start-Symbol-Mengen von Alternativen disjunkt sind, denn von der Philosophie des rekursiven Abstiegs her (kein Backup) kann die Entscheidung, welche der Alternativen verfolgt werden soll, nur aus der Kenntnis des aktuellen Symbols des Eingabestreams erfolgen. Dies ist eine Einschränkung für die der gewünschten Sprache zugrundeliegenden Grammatik; Grammatiken, die diese Eigenschaft besitzen, nennt man auch LL(1)-Grammatiken. Häufig jedoch kann man durch formale Korrekturen eine vorliegende Grammatik so modifizieren, dass sie dieser Bedingung genügt:

Offensichtlich sind die Produktionen

```
  U ::= A b | A C
```

und

```
U ::= A V
V ::= b | C
```

äquivalent, wobei die beiden letzteren nach oben Gesagtem durch die syntaktische Routinen

```
void U()
{
  A();
  mustbeIn ( head_V );
  V();
}

void V()
{
  switch ( symbol ) {
  case sym_b :
    nextSymbol = scanner.GetNextSymbol();
    break;
  case sym_c :
    C();
  }
}
```

realisiert werden.

Alternativ könnte man auch durch Verwendung von Klammern die Einführung des Nichtterminals V einsparen:

```
U ::= A ( b | C )
```

Dies würde zu dem folgendem Code für die syntaktische Routine führen:

```
void U()
{
  A();
  mustbeIn ( head_b_C );
  switch ( symbol ) {
  case sym_b :
    nextSymbol = scanner.GetNextSymbol();
    break;
  case sym_c :
```

```
    C();
  }
}
```

Eine solche Problemlösung wird aber nicht in allen Fällen auf einfache Weise möglich sein; ein prominentes Beispiel ist der Konflikt in der Sprache C der durch Typedef-Namen und normale Identifier entsteht. Die naheliegende Lösung für einen Top-Down-Parser ist in diesem Fall, auch noch das nächste Symbol ('2 symbol lookahead') zur Entscheidung heranzuziehen. Eine interessante Alternative, die einen 'dynamischen Scanner' verwendet, findet sich in [7].

Das zweite Problem ist das der nicht-eindeutigen Optionen, das dann auftritt, wenn ein Start-Symbol eines optionalen Konstrukts gleichzeitig Start-Symbol eines Folgekonstrukts ist. Dieses Problem taucht in den Sprachen PASCAL und C als 'dangling if problem' auf, nämlich in dem Sinn, dass bei zwei ineinanderverschachtelten if-Statements von der Grammatik her nicht eindeutig zu entscheiden ist, zu welchem if ggf. der else-Zweig gehört. Die pragmatische Lösung ist hier, die Option jeweils vorzuziehen – was auch von o.a. Codierung der Option geleistet wird – d.h. im konkreten Fall wird der else-Zweig dem inneren if-Statement zugeschlagen.

Mit diesen Einschränkungen kann die Konstruktion eines Top-Down-Parsers weitgehend automatisch erfolgen; ein solcher Parser-Generator sollte aus einer formalisierten Sprachbeschreibung in EBNF den Code für den Parser generieren können. Hier bietet es sich an, auch die zugehörige Semantik – mit entsprechenden Vereinbarungen über die Form – als C-Code in die Grammatikformulierung einzubetten, damit der Generator bei der Code-Generierung gleich die nötigen semantischen Aktionen mit hinzufügen kann, so dass eine manuelle Nachbearbeitung des Codes unterbleiben kann.

Die entsprechende Formalisierung von EBNF könnte dabei die folgende Form haben:

```
syntax ::= { production } .

production ::= ident "::=" expression "." .

expression ::= term { "|" term } .

term ::= factor { factor } .

factor ::= ident | string | "(" expression ")"
           | "{" expression "}" | "[" expression "]" .
```

Dabei haben wir EBNF noch um Ausdrücke mit normalen Klammern erweitert -

diese können wie oben gezeigt dazu dienen, um in einfachen Fällen Produktionen einzusparen.

Einschübe für die Semantik wären allerdings noch entsprechend zu ergänzen.

5.2 Bottom-up Parser

Ein Bottom-up Parser geht prinzipiell so vor, dass er solange Eingabesymbole aufsammelt, bis er eine Regel erkennt, mit Hilfe derer er die jeweils letzten Symbole zu einem Nicht-Terminal reduzieren kann. Er hat sein Ziel erreicht, wenn er die gesamte Eingabe zum Startsymbol reduziert hat.

Praktische Bedeutung haben hier nur die sog. *Shift-Reduce-Parser*. Diese benutzen einen Parse-Stack, der zu Beginn der Analyse leer ist und während der Analyse Terminale und bereits reduzierte Nicht-Terminale enthält. Immer dann, wenn der Scanner ein neues Symbol liefert, wird dieses zunächst auf dem Parse-Stack abgelegt. Der Parser versucht dann, die obersten Symbole auf dem Stack sinnvoll zu reduzieren – als Entscheidungshilfe wird ggf. auch noch das nächste Symbol herangezogen. In der Reduktion werden die zu reduzierenden Symbole vom Stack entfernt und durch die linke Seite der entsprechenden Produktionsregel ersetzt. Sobald der Stack zum Zielsymbol reduziert ist – und außer End-Of-File kein weiteres Symbol ansteht – ist die Analyse erfolgreich beendet.

Ohne auf Implementierungsdetails einzugehen, soll am Beispiel der Analyse des Ausdruck $3 + 4 * 5$ und der folgenden – für die Bottom-Up-Analyse besser geeigneten – Grammatik

(5.1) $expr ::= term \mid expr \ + \ term$

(5.2) $term ::= factor \mid term \ * \ factor$

(5.3) $factor ::= number \mid (\ expr\ )$

das Vorgehen des Bottom-Up-Parsers erläutert werden

Stack	Input	Aktion
	3+4*5	shift
number	+4*5	reduce
factor	+4*5	reduce

term	+4∗5	reduce	
expr	+4∗5	shift	
expr +	4∗5	shift	
expr + number	∗5	reduce	
expr + factor	∗5	reduce	
expr + term	∗5	shift	(!)
expr + term ∗	5	shift	
expr + term ∗ number		reduce	
expr + term ∗ factor		reduce	
expr + term		reduce	
expr			

Die Entscheidung über die jeweils notwendigen Aktionen entnimmt der Parser aus von der Grammatik abgeleiteten Tabellen, in die dann auch semantische Aktionen – wie z.B. die Auswertung der Ausdrücke auf dem erweiterten Parse-Stack – integriert werden können.

Im konkreten Fall ist die einzig problematische Stelle die durch (!) markierte Zeile. Hier wäre prinzipiell auf dem Parse-Stack auch eine Reduktion von *expr* + *term* zu *expr* möglich, was in der Folge aber zu einer nicht weiter reduzierbaren Symbolfolge auf dem Stack führen würden. Um diesen Irrweg zu vermeiden, können aber die Entscheidungstabellen zumindest den Kontext des nächsten Input-Symbols benutzen, um zu erkennen, dass in dieser Situation nur eine Shift-Aktion in Frage kommt.

Übungsaufgaben

1. Welche zwei Hauptarten von Parsern gibt es und wie unterscheiden sie sich?

   Welche der beiden Varianten ist meist codegesteuert und welche meist tabellengesteuert?

2. Die Grammatik einer Sprache sei durch folgende Produktionsregeln

   ```
   syntax ::= { production } .
   production ::= ident "::=" expression "."
   expression ::= term { "|" term }
   term ::= factor { factor }
   factor ::= ident | string | "(" expression ")"
              | "{" expression "}" | "[" expression "]"
   ```

   gegeben.

   (a) Bestimmen Sie die Startsymbolmengen aller Nichtterminale

   (b) Sind diese für alle in der Grammatik vorkommenden Alternativen disjunkt?

   (c) Sind diese auch für alle optionalen (und iterativen !) Elemente disjunkt zu denen ihrer Nachfolger?

   (d) Läßt sich diese Sprache mit einem 1-Symbol-Lookahead-Top-Down-Recursive-Descent-Parser analysieren?

3. Geben Sie ein prominentes Beispiel einer Programmiersprache, in der das Prinzip der "disjunkten Optionen" verletzt ist.

4. Ein Ausschnitt einer Grammatik für einfache Variablen, Strukturen und Arrays möge wie folgt aussehen:

   ```
   variable ::= ident | record | array
   record   ::= ident "." variable
   array    ::= ident indexlist [ "." variable ]
   indexlist ::= "[" number "]" [ indexlist ]
   ```

   (a) Warum ist diese Definition für einen Top-Down-Recursive-Descent-Parser problematisch?

   (b) Geben Sie eine alternative Grammatikdefinition, die diese Probleme vermeidet.

   (c) Wenn die obige Definition aus semantischen Gründen vorzuziehen ist, würde eine 2-Symbol-Lookahead-Strategie die Probleme beheben können?

(d) Skizzieren Sie den Code die Syntaxprozedur *variable*, die der entsprechenden Produktion entspricht (wahlweise für (b) oder (c)).

5. Die Sprache *C* erlaubt Mehrfachzuweisungen der Form

   a = b = c = ... = x

   wobei die zugewiesenen Elemente Variablen (*variable*) und der zugewiesene Wert ein Rechenausdruck (*expression*) sein müssen.

   Geben Sie eine Ableitungsregel für *assignment* an, die die Nichtterminale *variable*, *expression* und das Terminal "=" verwendet.

6. Formulieren Sie die Original-Grammatik aus Aufgabe 4 in BNF um und beschreiben Sie die Schritte, die ein Bottom-Up-Parser durchführt, um den Satz

   a . b [5] .c

   zum Symbol *variable* zu reduzieren.

Kapitel 6

Symboltabellen

Einer der Vorzüge von höheren Programmiersprachen liegt in der Verwendbarkeit von symbolischen Namen für relevante Objekte – dies trifft in gewissem Umfang auch schon für symbolische Assembler zu, wo allerdings Namen ausschließlich für (Daten- bzw. Instruktions-) Adressen stehen, wenn man einmal von Makronamen absieht. In allen entsprechenden Übersetzern findet man daher eine sog. Symboltabelle, die es dem Compiler bzw. Assembler ermöglicht, die zugehörigen Objektspezifikationen für die Namen zu substituieren.

In erster Annäherung sind Symboltabellen von der Form

	Argument	Wert
Eintrag 1		
Eintrag 2		
...		
Eintrag n		

- sie lassen sich also in der einfachsten Form Liste von Einträgen implementieren, wobei ein einzelner Eintrag durch eine Instanz einer Klasse *SymbolEntry* repräsentiert sein kann.

```
class SymbolEntry
{
  String arg;
  int    val;
}
```

beschrieben sein kann.

Im allgemeinen wird man aber komplexere Wertbeschreibungen zulassen müssen, so etwa wenn die Sprache verschiedenartige Variablentypen zuläßt. Erste Implementierungen realisierten das, indem als Wert lediglich einen Zeiger auf einen sog. 'Descriptor-Block' in die Symboltabelle eingetragen wurde, in dem dann alle relevanten Informationen festzuhalten werden. Dies war insbesondere dann sinnvoll, wenn die Implementationssprache des Compilers eine dynamische Speicherverwaltung zuläßt: Die fest dimensionierte Symboltabelle nimmt vergleichsweise wenig Platz in Anspruch, und die Descriptor-Blöcke können dynamisch erzeugt werden; ggf. kann sogar der gleiche Descriptor-Block verschiedenen Namen zugeordnet werden, wie dies etwa für Sprachen vom PL/1-Typ mit ihren 'LITERALLY'-Anweisungen möglich ist.

Mit den aktuellen objektorientierten Sprachen bietet es sich dagegen an, für die Symboltabelleneinträge eine (abstrakte) Basisklasse anzulegen

```
class SymbolEntry
{
  String    name;
  EntryType type;
}
```

Dabei fungiert das Attribute *type* mit dem Aufzähltyp

```
enum EntryType
{ TypeVar, TypeFunc, TypePar, TypeConst, TypeLabel }
```

als Selektor für den Typ des Eintrags, dessen typspezifische Inhalte in geeigneten abgeleiteten Klassen abgelegt werden:

```
class VarEntry : SymbolEntry
{
  Type  varType;
  int  dim;
       ...
}

class FuncEntry : SymbolEntry
{
  Type returnType;
  List<ParEntry> parameters;
       ...
}
```

Dabei ist *Type* ein Aufzählltyp über die möglichen Variablen-Typen der Sprache; die weitere Struktur über den Namen hinaus, die hier nur in der Form

von Dimensionsanzahl bzw. Returntyp und Parameterliste angedeutet ist, hängt sehr stark von der Semantik der Sprache ab; wir werden später noch hierauf zurückkommen.

An dieser Stelle soll aber noch bemerkt werden, dass auch früher ggf. auch Ablageoptimierungen im Zusammenhang mit den Namenseinträgen sinnvoll sein konnten: Etwa bei Sprachen, die beliebig lange Namen zulassen, wurde statt des fest dimensionierten Namensstrings selbst ein Zeiger auf den Namen - der seinerseits in einem eigenen Speicherbereich abgelegt wurde, in die Symboltabelle eingetragen.

Diese Methode war insbesondere dann selbst bei festen Namenslängen empfehlenswert, wenn die Symboltabelle in alphabetischer Reihenfolge angelegt werden sollte, da beim Sortieren der Einträge lediglich Zeiger verschoben werden müssen und nicht ganze Strings. Eine solche Sortierung wird man i.a. dann anstreben, wenn man auf Symbole häufig zugreift; der Sortieraufwand beim Anlegen der Tabelle ist gegenüber dem Zeitgewinn z.B. durch 'binary search' zu vernachlässigen.

Ist die Sprache (wie C oder PASCAL) so strukturiert, dass alle Namen vor ihrer Verwendung deklariert sein müssen, so kann man so vorgehen, dass man die Symboltabelleneinträge zunächst in Form eines binären Baumes ablegt und ggf. nach dem Ende des Deklarationsteils durch Traversieren des Baums eine geordnete Liste herstellt.

In vielen Compilern wird aber eine andere Methode benutzt, um einen schnellen Zugriff zu erreichen, das sog. 'Hashing'. Es besteht im Prinzip darin, die Tabelle so zu organisieren, dass der Tabellenindex sich auf einfache Weise aus dem Namen berechnen läßt; genauer gesagt, man definiert eine 'Hash-Funktion' von der Menge der (möglichen) Namen in die Menge der natürlichen Zahlen. Solange sich je zwei Namen durch die Werte unter dieser Funktion unterscheiden lassen, kostet die Suche nach einem Namen genauso viel Zeit wie die Berechnung des Funktionswertes. 'Hashen' jedoch zwei verschiedene Namen nach dem gleichen Index, so muss man entweder mit einer anderen Hash-Funktion weitersuchen und dies ggf. iterieren – wobei die Reihenfolge natürlich die gleiche sein muss wie beim Eintrag in die Tabelle – oder aber es sind dann andere Suchkriterien anzuwenden. Die Kunst liegt hier darin, die Hash-Funktion so zu wählen, dass solche 'Kollisionen' möglichst selten auftreten, ohne dass die Tabelle übermäßig groß dimensioniert ist. Ein wesentlicher Nachteil der Namenssuche nach dem 'Hash-Verfahren' besteht darin, dass sie sich nicht in einfacher Weise mit einer dynamischen Speicherallokation verbinden läßt, wie das etwa für den 'binären Baum' möglich ist; vorteilhaft ist die gerade bei großen Tabellen höhere Suchgeschwindigkeit.

Aktuelle Entwicklungssprachen stellen mit passenden *Dictionary*- und *HashSet*-Containertypen geeignete Instrumente hierfür zur Verfügung.

Unabhängig davon, welche Suchmethode benutzt wird, sollte man den eigentlichen Zugriff auf die Symboltabelle in speziellen Prozeduren konzentrieren und von dem Rest des Compilers 'abschotten'; wir wollen uns für das folgende damit begnügen, dass die Symboltabelle Methoden *enter* und *lookup* zur Verfügung stellt, mit denen sich Symbole in die Tabelle eintragen und auslesen lassen.

```
bool enter(SymbolEntry entry);

SymbolEntry lookup(String name);
```

wobei beim Neueintrag zurückgegeben wird, ob der Eintrag erfolgreich war oder nicht, und beim Auslesen das gesuchte Objekt bzw. *null* zurückgegeben wird.

Bei blockstrukturierten Sprachen wie PASCAL und C tauchen zusätzliche Probleme dadurch auf, dass Blöcke bzw. Prozeduren lokale Variable besitzen können, deren Namen mit denen von globalen Variablen übereinstimmen. Dies bedingt, dass die Symboltabelle des Gesamtprogramms eine Art 'pulsierende Liste' von Symboltabellen der einzelnen Blöcke ist, in der Namen in absteigender Blocktiefe gesucht werden; nach dem Abarbeiten eines Blockes muss - zumindest bei 1-Pass-Compilern – auch der zum Block gehörige Teil der Symboltabelle wieder gelöscht bzw. ungültig gemacht werden.

Eine weitere Komplizierung der Namenssuche besteht darin, dass es – wiederum bei PASCAL und C – jeweils eigene Namensräume für die Komponenten von Strukturen gibt. Will man hier auf einen separaten Suchalgorithmus verzichten, so bietet sich die Methode des 'binären Baumes' an, die es ermöglicht, beliebige Teilbäume mit eigenen 'Zugangswurzeln' in den gesamten Namensbaum zu integrieren. Dabei sind diese Wurzeln nur im entsprechenden Typ-Eintrag sichtbar, so dass Namenskollisionen von Strukturkomponenten mit globalen Variablen ausgeschlossen sind. Insbesondere läßt sich so auf einfache Weise die Wirkung des 'WITH'-Statements in PASCAL nachvollziehen: Die 'Zugangswurzeln' zu den Namensbäumen der einzelnen Definitionshierarchien werden vom Compiler zur Übersetzungszeit auf einem stackartigen Display verwaltet, der Einstieg in einen Block bewirkt das 'Pushen' der zugehörigen Wurzel auf das Display, das Verlassen des Blockes wieder das 'Poppen'.

Die weiteren Inhalte der Symboltabelleneinträge haben naturgemäß für verschiedene Quellsprachen unterschiedliche Ausprägung. Wir wollen hier exemplarisch am Beispiel der Sprache C andeuten, welche Informationen für die semantische Analyse und die Code-Erzeugung von Nutzen sind:

Variable:

- Typ (einfache Variable, Array, Struktur)
- Sichtbarkeit (extern, global, lokal)

- Flag, ob schon eine Wertzuweisung erfolgt war
- Laufzeitadresse

Funktionen und Prozeduren:

- Rückgabetyp (leer für Prozedur)
- Sichtbarkeit (extern, global, lokal)
- Flag, ob bereits definiert
- Zeiger auf ersten formalen Parameter
- Einsprungadresse
- Stackbedarf für (Parameter und) lokale Variable
- Flag, ob (direkt) rekursiv oder nicht

Enumerationskonstanten:

- Wert
- Typ
- Zeiger auf Nachfolger

Formale Parameter von Funktionen:

- Typ
- Adresse (relativ zum Basiszeiger)
- Zeiger auf Nachfolger

Strukturkomponenten:

- Typ
- Zeiger auf Nachfolger
- Offset zum Beginn der Struktur

Strukturen :

- Zeiger auf erste Komponente
- Flag, ob Überlagerung
- Speicherbedarf

Enumerationstypen:

- Zeiger auf erste Konstante

Label:

- Laufzeitadresse

Typen:

- Speicherbedarf
- Typkennung für Feld, Struktur, Zeiger, const, volatile
- Folgetyp, falls Feld, Struktur, Zeiger, const, volatile
- Dimension, falls Feld

Kapitel 7

Code-Erzeugung

Wenn auch die Code-Erzeugung stark vom jeweiligen Zielprozessor bzw. der Zielsprache (Maschinencode, Assemblerquellcode) abhängig ist, so gibt es doch eine Reihe von allgemeinen Prinzipien, auf die wir in diesem Abschnitt eingehen wollen.

Um unsere Überlegungen einerseits nicht zu kompliziert werden zu lassen, sie andererseits aber auch nicht zu weit von realen Gegebenheiten zu entfernen, wählen wir als virtuellen Zielprozessor eine sog. „Register-Maschine“ mit einem reduzierten Befehlssatz (*RISC*) (mit gelegentlichen Verweisen auf die Unterschiede zur Code-Erzeugung für reale Prozessoren wie etwa die der Intel- oder Motorola-Reihen).

Die Register des Prozessors nutzen wir dabei zur Auswertung von Ausdrücken und zur Zuweisung von Speicheradressen in einer stackähnlichen Weise – dies erleichtert insbesondere dem Parser die 1-Pass-Codegerierung, da die die Übersetzung der Transformation von Ausdrücken in die (klammerfreie) UPN-Notation entspricht, wie wir sie schon in Kapitel 3 kennengelernt haben.

PASCAL ist von seiner Definition her eine Sprache, die auf eine reine Stack-Architektur zugeschnitten ist – nicht umsonst produzierte der PASCAL-Compiler von Niklaus Wirth sog. P-Code, der dann von einem speziellen P-Code-Interpreter ausgeführt wurde. (Interessanterweise wird jetzt im Zuge der sog. Binärkompatibilität von unterschiedlichen Hardware-Plattformen diese P-Code-Erzeugung wieder aktuell.)

Im Gegensatz dazu ist C eine stärker register-orientierte Sprache – wie das aus der Entstehungsgeschichte als Assemblerersatz für die Entwicklung von UNIX auch zu verstehen ist. Von daher bietet sich unser Vorgehen gut für C-ähnliche

Sprachen an.

Wir trennen die folgenden Überlegungen in drei Unterabschnitte auf, die jeweils die Behandlung von Kontrollstrukturen, Zuweisungen und Rechenausdrücken sowie von Funktionsaufrufen enthalten.

Eine genauere Beschreibung des Prozessors und seines Befehlssatzes findet sich im Anhang. Wir beschränken uns hier zunächst darauf, den erzeugten Zielcode in einer Assembler-Notation wiederzugeben.

7.1 Kontrollstrukturen

Die üblichen Kontrollstrukturen für if-, while-, do-while, for-, switch- und goto-Anweisungen lassen sich über bedingte und unbedingte Sprünge einfach realisieren. Unsere Maschine kennt hierzu die unbedingte Sprunganweisung *jmp* und die beiden bedingten Sprunganweisungen *jeq* und *jne*, die je nach dem Wert des Flag-Register *fl* einen Sprung bewirken oder die Ausführung mit der nächsten Anweisung fortsetzen. Das Flag-Register selbst kann mit einer der beiden Vergleichsanweisungen *cmp* und *cmpi* gefüllt werden.

So ergibt sich als Übersetzung für die if-Anweisung

```
if ( expression ) statement
```

der folgende Code:

```
        ...             ; Code fuer expression mit Wert in R1
        cmpi 1 0        ; Vergleich mit 0
        jeq label       ; Wenn gleich, Sprung zu label
        ...             ; Code fuer statement
label:                  ; Sprungmarke
```

Kommt ein else-Zweig hinzu,

```
if ( expression ) statement_1
else statement_2
```

so äußert sich das in zwei Sprungmarken :

```
        ...             ; Code fuer expression mit Wert in R1
        cmpi 1 0        ; Vergleich mit 0
        jeq label_1     ; Wenn gleich, Sprung zu label_1
        ...             ; Code fuer statement_1
        jmp  label_2    ; Sprung ueber den else-Zweig
label_1:                ; Sprungmarke fuer else-Zweig
        ...             ; Code fuer statement_2
label_2:                ; Sprungmarke fuer Folgenaweisung
```

Der für die 8086-Familie erzeugte Code sieht im Wesentlichen genau so aus:

```
        cmp   ax,0      ; Bedingung ist in ax-Register
        je    label     ; Wenn = 0, Sprung zu label
```

Die while-Schleife

```
        while ( expression ) statement
```

wird in analoger Weise durch

```
label_1:                ; Sprungmarke fuer Schleifenkopf
        ...             ; Code fuer expression mit Wert in R1
        cmpi 1 0        ; Vergleich mit 0
        jeq label_2     ; Wenn gleich, Sprung zu label_2
        ...             ; Code fuer statement
        jmp  label_1    ; Sprung zum Schleifenkopf
label_2:                ; Sprungmarke fuer Folgeanweisung
```

realisiert. Einfacher noch ist die “fußgesteuerte” do- oder repeat-Schleife

```
        do statement while ( expression )
```

deren Realisierung

```
label_1:                ; Sprungmarke fuer Schleifenkopf
        ...             ; Code fuer statement
        ...             ; Code fuer expression mit Wert in R1
        cmpi 1 0        ; Vergleich mit 0
        jne label_1     ; Wenn ungleich, Sprung zu label_1
```

mit einer Sprungmarke auskommt.

Die wiederum "kopfgesteuerte" for-Anweisung

```
for ( assign1 ; expr ; assign2 ) statement
```

entspricht der while-Anweisung

```
assign1 ;
while ( expr )
 {
  statement
  assign2 ;
 }
```

Während also bei der for-Anweisung in C die Abbruchbedingung (expr) in jedem Schleifendurchlauf ausgewertet wird, ist dies für die PASCAL-Entsprechung anders: Dort wird nur einmal zur Laufzeit die Anzahl der Wiederholungen bestimmt und dann in jedem Schleifendurchlauf in einer (vom Compiler angelegten) lokalen Variablen zu Null heruntergezählt.

Für einen Single-Pass-Parser wirft die in dieser Sequenz sichtbare Verschiebung der Ausführungsreihenfolge ein kleines Problem auf, das aber auf einfache Weise durch die Generierung von Sprungbefehlen zu lösen ist:

```
        ...             ; Code fuer assign1
label_1
        ...             ; Code fuer expr mit Wert in R1
        cmpi 1 0        ; Vergleich mit 0
        jeq label_2     ; Wenn gleich, Sprung zu label_2
        jmp  label_3    ; Sonst Sprung zu statement
label_4
        ...             ; Code fuer assign2
        jmp  label_1    ; Sprung zum Schleifenkopf
label_3
        ...             ; Code fuer statement
        jmp  label_4    ; Sprung zu label_4
label_2:                ; Sprungmarke fuer Folgeanweisung
```

Auch bei der switch-Anweisung und ihrem PASCAL-Gegenstück, der CASE-Anweisung zeigen sich Unterschiede. Während in PASCAL der Sprung hinter das CASE-Statement vom Compiler erzeugt wird, muss in C der Programmierer durch eine break-Anweisung selbst dafür Sorge tragen (hat andererseits aber

dadruch auch die Möglichkeit, mehrere Zweige hintereinander durchlaufen zu lassen. Üblicherweise generieren PASCAL-Compiler eine Sprungtabelle (und bestrafen damit große CASE-Bereiche), C-Compiler begnügen sich meist mit eine Kaskadierung von Vergleichen (und bestrafen damit viele CASE-Labels).

Für die exemplarische switch-Anweisung (die wir der Einfachheit halber mit PASCAL-Semantik versehen)

```
switch ( expression )
 {
  case 1 : statement_1
  case 2 : statement_2
  default: statement_3
 }
```

wird der Code für unsere Maschine so aussehen:

```
        ...             ; Code fuer expression mit Wert in R1
        cmpi 1 1        ; Test auf 1
        jne label_1     ; Wenn ungleich, Sprung zu label_1
        ...             ; Code fuer statement_1
        jmp label_0     ; Sprung ans Ende
label_1:
        cmpi 1 2        ; Test auf 2
        jne label_2     ; Wenn ungleich, Sprung zu label_2
        ...             ; Code fuer statement_2
        jmp label_0     ; Sprung ans Ende
label_2:
        ...             ; Code fuer statement_3
label_0:                ; Ende
```

Der für die 8086-Familie erzeugte Code sieht vom Prinzip her ähnlich aus:

```
        cmp   ax,1      ; Wert ist in ax-Register
        je    label1    ; Wenn gleich, Sprung zu label1
```

Allerdings trägt dort der bedingte Sprung bei Gleichheit der Tatsache Rechnung, dass C-Semantik vorliegt, die es gestattet, ohne die break-Anweisung von einem Switch-Zweig in den folgenden überzugehen.

Bei einer switch-Anweisung mit C-Semantik ist darüberhimaus Vorsorge für das break zu treffen; dies gilt in analoger Weise auch für Schleifen, in denen eine break-Anweisung den Aussprung aus der Schleife und eine continue-Anweisung den Sprung an das Ende des Schleifenkörpers bewirkt.

7.2 Zuweisungen und Ausdrücke

Der Zugriff auf eine Variable wird in unserer Register-Maschine durch die Anweisungen *lod* und *lodi* gewährleistet, die jeweils das erste Register mit dem Wert des Speichers füllen, dessen Adresse im zweiten Register steht, bzw. das erste Register mit einer Konstanten.

Die Zuweisung auf eine eines Variable wird in analoger Weise durch die Anweisungen *sto* und *stoi* ermöglicht, die den Wert im ersten Register an die im zweiten Register enthaltene Adresse bzw. die Adresse im Befehl kopiert.

In diesem Sinne könnte die Zuweisung

```
a = 10 ;
```

zur Codesequenz:

```
lodi 1 a        ; Adresse von a in R1
lodi 2 10       ; Konstante 10 in R2
sto 2 1         ; Speichern nach a
```

führen, wobei in analoger Weise auch statt a eine kompliziertere linke Seite auftreten kann (Struktur-Komponente, Feld-Element, dereferenzierter Pointer u.ä.).

Wenn der Compiler allerdings weiss, dass – wie im vorliegenden Fall – auf der linken Seite lediglich eine globale Variable angesprochen ist, könnte er den Code auch zu

```
lodi 1 10       ; Konstante 10 in R1
stoi 1 a        ; Speichern nach a
```

Nur unwesentlich komplizierter gestaltet sich eine kombinierte Operations-Zuweisung in der Form:

```
a += 10 ;
```

bei der zunächst noch der Wert von a ausgewertet werden muss:

```
lodi 1 a        ; Adresse von a in R1
lod 1 1         ; Auswerten
lodi 2 10       ; Konstante 10 in R2
add 1 2         ; Summe von R1 und R2
stoi 1 a        ; Speichern nach a
```

Auch hier könnte der Compiler eine Vereinfachung vornehmen, indem er statt des Register-Additions-Befehl *add* den Befehl *addi* erzeugt, der zum Inhalt eines Registers eine Konstante addiert.

```
lodi 1 a        ; Adresse von a in R1
lod 1 1         ; Auswerten
addi 1 10       ; 1o dazu addieren
stoi 1 a        ; Speichern nach a
```

Der Code für reale Prozessoren kann sogar noch einfacher ausfallen, wenn die Adresse zur Übersetzungszeit bekannt ist:

```
mov   _a,10     ; Speichern von 10 nach a
```

bzw.

```
add   _a,10     ; Erhoehen von a um 10
```

Die Berechnung von Ausdrücken geschieht – wie schon erwähnt – in unserer Registermaschine durch Stack-ähnliches Nutzen der Register. Ein unärer Operator verändert den Wert im Register, ein binärer Operator verknüpft zwei Register und legt das Verknüpfungsergebnis im ersten Register ab.

Der Einfachheit halber hat unsere Maschine bis auf die logischen Operatoren und den Fragezeichen-Operator die gleichen Operatoren wie C, so dass die Übersetzung hier relativ einfach ist, wobei – wie oben schon beim Additionsoperator gesehen – für jede Operation sowohl ein Register-Register- als auch ein Register-Konstanten-Befehl zur Verfügung steht. In den Ausnahmefällen garantiert C die sog. Kurzschluss-Auswertung, so dass der Compiler hier Code für die bedingte Auswertung von Ausdrücken generieren muss.

Für einen normalen arithmetischen Ausdruck wie

```
a + 10 * b
```

erhalten wir also:

```
        lodi 1 a        ; Adresse von a in R1
        lod 1 1         ; Auswerten
        lodi 2 10       ; Konstante 10 in R2
        lodi 3 b        ; Adresse von b in R3
        lod 3 3         ; Auswerten
        mul 2 3         ; Produkt in R2
        add 1 2         ; Summe in R1
```

Hier tun sich wiederum die realen Prozessoren leichter; allerdings ist die Aufgabe für den Compiler schwieriger, da er die Operanden nicht ohne weiteres in der Reihenfolge auswerten kann, wie sie in der C-Quelle erscheinen.

```
        mov  ax,10      ; Konstante 10 nach ax-Register
        imul _b         ; Produkt
        add ax,_a       ; Summe
```

Die Auswertung etwa des logischen Oder

```
        a || b
```

geschieht so:

```
        lodi 1 a        ; Adresse von a in R1
        lod 1 1         ; Auswerten
        cmpi 1 0        ; Vergleich mit 0
        jne label_1     ; Falls ungleich, Sprung nach label_1
        lodi 1 b        ; Adresse von b in R1
        lod 1 1         ; Auswerten
        cmpi 1 0        ; Vergleich mit 0
        jne label_1     ; Falls ungleich, Sprung nach label_1
        lodi 1 0        ; Konstante 0 in R1
        jump  label_2   ; Sprung auf naechste Anweisung
label_1:
        lodi 1 1        ; Konstante 1 in R1
label_2:                ; Sprungmarke fuer Folgeanweisung
```

Hier könnte die Anweisung *lodi 1 0* sogar entfallen, da in diesem Fall bereits sichergestellt ist, dass das (Ziel-)Register 1 den Wert 0 enthält.

Auch der (MS-C-) Compiler für die Intel-Prozessoren erzeugt keinen wesentlich anderen Code:

```
        cmp  _a,0       ; Falls a von Null verschieden
        jne  label_3    ; Sprung nach label_3
        cmp  _b,0       ; Falls b gleich Null
        je   label_1    ; Sprung nach label_1
label_3:
        mov  ax,1       ; 1 in ax-Register
        jmp  label_2    ; Sprung auf Folgeanweisung
label_1:
        sub  ax,ax      ; Optimierte Version von mov ax,0
label_2:
```

Eine Besonderheit bildet der Adress-Operator, der im Code keine Entsprechung hat, da seine Wirkung nur darin besteht, eine Adresse nicht mehr als Adresse sondern als Wert anzusehen. Umgekehrt dazu bewirkt die Auswertung einer Variablen die Erzeugung von Code etwa für *lod 1 1*, dieser Code entsteht auch beim Dereferenzieren eines Pointers.

Abschließen wollen wir diesen Abschnitt mit der Bemerkung, dass die Berechnung von Offsets innerhalb von Strukturen durch den Compiler aufgrund der Information in der Symboltabelle erfolgt, und der Code letzlich nur in einer Addition des Offsets zur Basisadresse besteht. Ähnliches gilt für die Adressberechnung von Feldelementen, nur dass hier zum einen der Offset ggf. erst zur Laufzeit ermittelt wird, und zum anderen aus dem Produkt von Feldindex und Speicherbedarf des Feldtyps entsteht.

Ist also x eine Struktur und z ein Feld, so ergibt sich für die Zuweisung

```
x.y = z[w]
```

folgender Code:

```
lodi 1 x          ; Adresse von x in R1
addi 1 4          ; erhöhen um Offset 4 von y
lodi 2 z          ; Adresse von z in R2
lodi 3 w          ; Adresse von w in R3
lod 3 3           ; Auswerten
muli 3 2          ; mit Konstante 2 multiplizieren (2 Byte Feld-Typ)
add 2 3           ; Summe
lod 2 2           ; Auswerten (der rechten Seite)
sto 2 1           ; Zuweisen (auf linke Seite)
```

Hier spielt der Compiler für die 8086-Reihe die vollen Vorzüge des Befehlssatzes und der Adressierungsarten aus:

```
mov  bx,_w      ; Inhalt von w nach bx-Register
shl  bx,1       ; optimierte Multiplikation mit 2
mov  ax,_z[bx]  ; Inhalt in ax-Register
mov  _x+4,ax    ; Inhalt von ax nach x.y
```

7.3 Funktionsaufrufe

Anders als bei der Auswertung von Ausdrücken bietet sich für die Verarbeitung von Funktionsaufrufen die Nutzung eines Memory-Stacks an – und zwar sowohl für die Übergabe von Funktionsparametern und Funktionswert als auch für die Nutzung von lokalen Variablen, die insbesondere bei verschachtelten oder rekursiven Funktionsaufrufen den begrenzten Platz der Register sprengen würden.

So wird zur Vorbereitung eines Funktionsaufrufs zunächst der Platz für den Funktionswert auf dem Stack reserviert – wir nehmen der Einfachheit halber an, dass jede Funktion einen Funktionswert liefert – danach werden die Übergabeparameter auf dem Stack abgelegt, dorthin gelangt ebenso die Rücksprungadresse, und dann wird der Code der Funktion angesprungen. Nach der Rückkehr von der aufgerufenen Funktion ist der Stack zu restaurieren und ggf. der Funktionswert weiterzuverarbeiten.

Die heute üblichen Prozessoren verfügen dafür über Spezialbefehle wie *call* bzw. *jsr* für das Sichern der Rücksprungadresse und *ret n* bzw. *rts* für den Rücksprung, wobei im Fall von ret sogar noch die Stackkorrektur (n Bytes) in den Befehl integriert ist. Bei einer Sprache wie PASCAL, die Funktionsaufrufe mit einer variablen Parameteranzahl ausschließt, läßt sich dieses Feature dazu nutzen, die Stackkorrektur von der gerufenen Funktion durchführen zu lassen. In C hingegen, das (Beispiel printf) Funktionen mit variablen Parameteranzahlen zuläßt, muss die rufende Funktion bei jedem Aufruf diese Korrektur besorgen.

Ähnlich wie bei schon bei der Ansprache von Variablen kennt unsere Maschine zwei Funktionsbefehle *cll* und *clli*, die die Adresse der Funktion in einem Register bzw. direkt im Befehl erwarten. Mit dem Befehl *psh* bzw. *pshi* können Registerinhalte bzw. Konstante auf den Stack geschoben werden; der Befehl *pop* holt einen Wert vom Stack und legt ihn in einem Register ab.

In diesem Sinne wird beispielsweise der Funktionsaufruf

```
a = f(1,2);
```

für unsere Maschine in die Codesequenz

```
pshi 0          ; Platz für Funktionswert freimachen
pshi 1          ; 1. Parameter (Konstante 1) auf den Stack
pshi 2          ; 2. Paramer (Konstante 2) auf den Stack
clli f          ; Funktionsaufruf
pop 1           ; Abholen des Funktionswertes
stoi 1 a        ; Speichern des Funktionswertes
```

Etwa für die 8086-Prozessoren sieht der Code folgendermaßen aus:

```
mov  ax,2
push ax         ; Parameter 2
mov  ax,1
push ax         ; Parameter 1
call _f         ; Aufruf
add  sp, 4      ; Stack-Korrektur 2 x 2 Byte
mov  _a,ax      ; Zuweisung des Funktionswertes
```

Die gerufene Funktion muss als erstes den aktuellen Wert des Stackpointers im sog. *Framepointer* (oder auch *Basispointer*) und deshalb vorher auch den Framepointer selbst sichern, da über diesen zum einen die übergebenen Parameter und zum anderen die lokalen Variablen der Funktion zu adressieren sind. Zu sichern sind – wie bei einem realen Prozessor ggf. auch – die Register, bevor der eigentliche Code der Funktion beginnt. Entsprechend muss vor dem Rücksprung der ursprüngliche Inhalt von Framepointer, Stackpointer und den übrigen Registern wieder hergestellt werden. Die dazu notwendigen Codesequenzen nennt man auch Prolog und Epilog.

Unsere Register-Maschine kennt dafür die Befehle *mks* und *rtn*, so dass der Code einer Funktion mit 2 Parametern und 3 lokalen Variablen folgendes Aussehen hat:

```
mks 3           ; Platz fuer lokale Variable
...
rtn 2           ; Stackkorrektur und Ruecksprung
```

Wiederum fuer die 8086-Familie bekommen wir:

```
push bp         ; Framepointer -> Stack
mov  bp,sp      ; Stackpointer -> Framepointer
...
pop bp          ; Restaurieren des Framepointers
ret             ; Ruecksprung
```

Dabei ist zu beachten, dass die Zuweisung des Funktionswertes innerhalb der Funktion auf die korrekte Stelle des Stacks (oder ggf. in ein geeignetes Register) erfolgt. In gleicher Weise muss auch die Adressierung sowohl der Funktionsparameter als auch der lokalen Variablen relativ zum Framepointer erfolgen. Unsere Maschine verfügt dazu über den Befehl *lodf*, dessen zweites Argument für die Parameter und den Funktionswert einen negativen Werte annimmt, für lokale Variable hingegen einen positiven Wert. Diese Offsets sind natürlich dem Compiler zur Übersetzungszeit bekannt.

Um dies zum Abschluss des Abschnittes zu verdeutlichen, betrachten wir die einfache Funktion f, die die Summe der Quadrate ihrer zwei Argumente berechnet:

```
function f(x,y)
{
  var x2, y2;
  x2 = x*x;
  y2 = y*y;
  return x2 + y2;
}
```

Der vollständigen Code für unsere Maschine könnte so aussehen:

```
mks  2          ; Platz fuer lokale Variable
                ; Retten der Register

lodf 1 x2       ; Adresse von x2 in R1
lodf 2 x        ; Adresse von x in R1
lod 2 2         ; Auswerten
lodf 3 x        ; nochmal
lod 3 3
mul 2 3         ; Produkt
sto 2 1         ; Speichern (nach x2)

lodf 1 y2       ; Adresse von y2
lodf 2 y        ; Adresse von y
lod 2 2         ; Auswerten
lodf 3 y2       ; nochmal
lod 3 3
mul 2 3         ; Produkt
sto 2 1         ; Speichern (nach y2)

lodf 1 retval   ; Adresse des Funktionswertes
lodf 2 x2       ; Adresse von x2
lod 2 2         ; Auswerten
lodf 3 y3       ; Adresse von y2
```

```
        lod 3 3         ; Auswerten
        add 2 3         ; Summe
        sto 2 1         ; Speichern (Funktionswert)

        rtn 2           ; Stackkorrektur und Ruecksprung
```

Interessanterweise erzeugt der Compiler für die 8086-Architektur nichts wesentlich Anderes:

```
        _x$ = 8
        _y$ = 12
        _x2$ = -4
        _y2$ = -8

    _f PROC NEAR
        push    ebp         ; Framepointer -> Stack
        mov     ebp, esp    ; Stackpointer -> Framepointer
        sub     esp, 8      ; Platz für lokale Variable

        mov     eax, DWORD PTR _x$[ebp]    ; x -> eax
        imul    eax, DWORD PTR _x$[ebp]    ; Produkt eax * x -> eax
        mov     DWORD PTR _x2$[ebp], eax   ; eax -> x2

        mov     ecx, DWORD PTR _y$[ebp]    ; y -> ecx
        imul    ecx, DWORD PTR _y$[ebp]    ; Produkt ecx * y -> ecx
        mov     DWORD PTR _y2$[ebp], ecx   ; ecx -> y2

        mov     eax, DWORD PTR _x2$[ebp]   ; x2 -> eax
        add     eax, DWORD PTR _y2$[ebp]
; Summe eax + y2 -> eax

        mov     esp, ebp    ; Stack-Korrektur
        pop     ebp         ; Restauration des Framepointers
        ret     0           ; Rücksprung
```

Natürlich bietet schon dieser einfache Code genügend Spielraum für Optimierungen: Zum einen kann die doppelte Auswertung der Funktionsparameter x und y durch ein Kopieren der Register ersetzt werden; zum anderen wären auch die lokalen (temporären) Variablen $x2$ und $y2$ dadurch entbehrlich, dass der vollständige Ausdruck ohne explizite Speicherung von Zwischenergebnisse berechnet werden kann.

Auf weitergehende Optimierungsstrategien können wir hier aber nicht eingehen, sondern verweisen etwa auf [1] und [9].

Wie eingangs schon erwähnt, werden im Normalfall die Werte von Variablen oder allgemeiner die Werte von Ausdrücken als Parameter an die aufgerufene

Funktion übergeben und die Funktion kann ihrerseits nur einen Funktionswert zurückgeben – dies nennt man auch *call by value*. Es gibt aber auch Situationen, wo es wünschenswert ist, dass die aufgerufene Funktion die übergebenen Variablen direkt verändern kann – dieses Verhalten nennt man *call by reference*. Unsere Maschine unterstützt auch diese Aufrufmethodik; der Compiler muss lediglich beim Übersetzen darauf achten, statt der Inhalte der Variablen der Adressen auf den Stack zu schieben und innerhalb der Funktion bei der Ansprache dieser Variablen zusätzliche *lod*-Aufrufe zu erzeugen.

Dies wird durch das abschließende Beispiel einer Funktion, die die Werte der beiden ihr übergebenen Variablen vertauscht, verdeutlicht:

```
function f(ref x,ref y)
{
  var h;
  h = x;
  x = y;
  y = h;
}
...
...
f(ref a, ref b);
```

```
mkst 1          ; Platz fuer lokale Variable
lodf 1 h        ; Adresse von h in R1
lodf 2 x        ; Adresse von Adresse von x in R2
lod 2 2         ; Auswerten: jetzt Adresse von x
lod 2 2         ; Nochmal: jetzt Wert
sto 2 1         ; Speichern
lodf 1 x        ; Adresse von Adresse von x
lod 1 1         ; Auswerten: jetzt Adresse von x
lodf 2 y        ; Adresse von Adresse von y in R2
lod 2 2         ; Auswerten: jetzt Adresse von y
lod 2 2         ; Nochmal: jetzt Wert
sto 2 1         ; Spreichern
lodf 1 y        ; Adresse von Adresse von y in R1
lod 1 1         ; Auswerten: jetzt Adresse von y
lodf 2 h        ; Adresse von h  in R2
lod 2 2         ; Auswerten
sto 2 1         ; Speichern
rtn 2           ; Stackkorrektur und Ruecksprung

...
...

lodi 1 a        ; Adresse von a in R1
psh 1           ; auf den Stack
```

```
lodi 1 b        ; Adresse von b in R1
psh 1           ; auf den Stack
clli f          ; Aufruf von f
```

Übungsaufgaben

Die Übungsaufgaben ab Aufgabe 4 beziehen sich für die Codegenerierung auf die virtuellen Maschine der Vorlesung und verwenden den im Anhang aufgelisteten Befehlssatz. Dabei können (und sollen) Adressen in symbolischer Notation, also mit Buchstaben-Bezeichnern verwendet werden, dies gilt insbesondere auch für Parameter von Funktionen.

1. Nennen Sie mindestens 4 unterschiedliche Arten von Objekten einer Programmiersprache, die in einer Symboltabelle abzulegen sind.

   Nennen Sie dabei auch die abzulegenden Eigenschaften der Objekte.

2. Welche typischen Probleme treten für blockstrukturierte Sprachen beim Suchen von Namen in der Symboltabelle auf. Nennen Sie eine mögliche Strategie für deren Lösung.

3. Worin unterscheiden sich die Verzweigungskonstrukte von PASCAL (CASE) und C (switch)?

   Nennen Sie je einen Unterschied hinsichtlich der Syntax und der Abarbeitung.

4. Schreiben Sie eine Codesequenz für die virtuelle Maschine, die die Summe aller Quadratzahlen von 1 bis 400 berechnet (und ausgibt).

5. Disassemblierung:

   Die folgende Befehlssequenz wurde von einem Compiler erzeugt. Versuchen Sie eine Hochsprachen-Befehlsfolge anzugeben, aus der der Compiler diese Befehlsfolge hätte erzeugen können.

```
 lodi 1 10                    lodi 1 p
 stoi 1 e                     lod 1 1
 lodi 1 2                     lodi 2 b
 stoi 1 b                     lod 2 2
 lodi 1 1                     mul 1 2
 stoi 1 p                     stoi 1 p
.label label_1               .label label_3
 lodi 1 e                     lodi 1 b
 lod 1 1                      lod 1 1
 lodi 2 0                     lodi 2 b
 grt 1 2                      lod 2 2
 cmpi 1 0                     mul 1 2
 jeq label_2                  stoi 1 b
 lodi 1 e                     lodi 1 e
 lod 1 1                      lod 1 1
```

```
modi 1 2                        divi 1 2
lodi 2 1                        stoi 1 e
equ 1 2                         jmp label_1
cmpi 1 0                       .label label_2
jeq label_3                     lodi 1 p
                                lod 1 1
                                wrt 1
```

6. Fügen Sie in die die folgende Syntaxprozedur für eine if-Anweisung an den geeigneten Stellen Anweisungen ein, die eine Befehlssequenz für die virtuelle Maschine erzeugen. Sie dürfen (und sollen) dazu zwei Hilfsfunktionen verwenden:

   string labelMarke() (zur Bestimmung einer Sprungzielmarke)

   void emit(Befehlskürzel, Attribut) (zur Codeausgabe)

   Das mit der Funktion $labelMarke()$ erhaltene Label kann per $emit(label, \text{“}:\text{”})$ ausgegeben werden.

```
void ifStatement()
{
  scanner();
  mustbe(symKlAuf);
  scanner();
  mustbeIn(HeadCondition);
  condition();
  mustbe(symKlZu);
  scanner();
  mustbeIn(HeadStatement);
  statement();
  if ( vsymbol == symElse ) {
    scanner();
    mustbeIn(HeadStatement);
    statement();
  }
}
```

7. PASCAL (und auch C++) kennt Referenzparameter von Funktionen, die es der Funktion gestatten, auch die Inhalte der übergebenen Variablen zu modifizieren.

   Als Beispiel betrachten wir eine Funktion $f(ref\ a, ref\ b)$, die die Inhalte ihrer zwei Referenzparameter vertauscht.

```
function f(ref x, ref y)
{
  var h;
```

```
  h = x;
  x = y;
  y = h;
  return 0;
}
```

Wie muss der Code für die entsprechende Funktion mit Wertparametern (, die natürlich die Werte der übergebenen Variablen unverändert läßt) an der Aufrufstelle und innerhalb der Funktion f verändert werden, damit der gewünschte Effekt eintritt?

Aufrufstelle:

```
lodi 1 a
psh 1
lodi 1 b
psh 1
clli exchg
```

Funktionskörper

```
mks 1
lodf 1 h
lodf 2 x
lod 2 2
sto 2 1
lodf 1 x
lodf 2 y
lod 2 2
sto 2 1
lodf 1 y
lodf 2 h
lod 2 2
sto 2 1
rtn 2
```

Kapitel 8

Praxis

In diesem abschließenden Kapitel sollen die bis hierher erzielten Ergebnisse in einem konkreten Compiler-Bau-Projekt zusammengetragen werden.

8.1 Quellsprache

Quellsprache für das Projekt ist eine für unsere Zwecke stark vereinfachte Sprache die sich an die Sprachen C und Pascal anlehnt und deren Ausdrucks- und Kontrollstrukturen umfasst, sich aber auf ganzzahlige Variablentypen beschränkt. Sie unterstützt aber Funktionen – der Einfachheit halber nur mit Wertparametern.

Die Grammatik verlangt die Deklaration von globalen Konstanten, Variablen und Funktionen vor dem eigentlichen Hauptprogramm als Folge von Statements, die durch das Dateiende abgeschlossen werden.

```
parse ::= { constDecl } { varDecl } { funcDecl } { stmt } eof
```

Konstantendeklarationen werden mit dem Schlüsselwort *const* eingeleitet und enthalten eine durch Kommata getrennte Liste von einfachen Abkürzungsvereinbarungen für Namen, die als Konstanten Verwendung finden sollen. Die Werte der Konstanten beschränken sich der Einfachheit halber auf ggf. vorzeichenbehaftete Ganzzahlen.

```
constDecl    ::= const constPart { "," constPart } ";"
```

```
constPart    ::= identifier "=" signedNumber
signedNumber ::= [ "-" ] number
```

Variablendeklarationen werden mit dem Schlüsselwort *var* eingeleitet und enthalten eine durch Kommata getrennte Liste von einfachen (Integer-)Variablen oder Arrays.

```
varDecl  ::= var varPart { "," varPart } ";"
varPart  ::= identifier [ "[" constant "]" ]
constant ::= identifier | signedNumber
```

Funktionsdeklarationen werden mit dem Schlüsselwort *function* eingeleitet und enthalten den Funktionsnamen, die Liste der Parameter und den eigentlichen Funktionskörper. Funktionsparameter haben Value-Semantik. Im Funktionskörper müssen zu Beginn ggf. verwendete lokale Variablen vor den eigentlichen Anweisungen deklariert werden.

```
funcDecl      ::= function identifier "(" [ paramDeclList ] ")"
                  funcBody
paramDeclList ::= paramDecl { "," paramDecl }
paramDecl     ::= identifier
funcBody      ::= "{" { varDecl } { stmt } "}"
```

Anweisungen sind zum einen die sog. einfachen Anweisungen (simple statement), wie die Ausgabe eines Ausdrucks oder das Einlesen eines Variablenwertes oder die Rückgabe eines Funktionswertes. Aus syntaktischen Gründen müssen Zuweisungen von Variablen und Funktionsaufrufe in einer gemeinsamen syntaktischen Funktion abgehandelt werden. Einfache Anweisungen werden mit ';' abgeschlossen und hinterlassen – anders als in C – keinen Wert.

Ebenfalls als Anweisungen gelten die zusammengesetzten Anweisungen (compound statement) und eine in geschweifte Klammern eingeschlossene Folge von Anweisungen.

```
stmt ::= print orExpr ";" | read identifier [ index ] ";" |
         assCall ";" | return orExpr ";" |
         ifStmt | whileStmt | forStmt | doStmt | switchStmt |
         block
```

Die Zuweisung einer einfachen Variablen oder eines Arrays mit entsprechendem Index geschieht mit dem Gleichheitszeichen; Funktionsaufrufe werden durch einer in runden Klammern eingeschlossenen Parameterliste – die ggf. leer sein

kann - ausgedrückt. Diese Form des Funktionsaufrufs hinterlässt keinen Wert – der Compiler muss dafür sorgen, dass in diesem Fall der Funktionswert vom Stack entfernt wird.

```
assCall   ::= identifier ( actParams | [ index ] "=" orExpr )
actParams ::= "(" [ paramList ] ")"
paramList ::= param { "," param }
param     ::= orExpr
index     ::= "[" orExpr "]"
```

Die zusammengesetzten Anweisungen ermöglichen die üblichen Bedingungs- und Schleifenkonstrukte. Bei der switch-Anweisung verzichten wir auf eine break-Anweisung wie in C, sondern erlauben für jeden Zweig ein einziges Statement, das natürlich auch ein Block sein kann. Vergleichswerte für die einzelnen Case-Zweige müssen Konstanten sein; eine Zusammenfassung mehrerer solcher Vergleichswerte ist nicht vorgesehen.

```
block      ::= "{" { stmt } "}"
ifStmt     ::= if "(" orExpr ")" stmt [ else stmt ]
whileStmt  ::= while "(" orExpr ")" stmt
forStmt    ::= for "(" assCall ";" orExpr ";" assCall ")" stmt
doStmt     ::= do stmt while "(" orExpr ")" ";"
switchStmt ::= switch "(" orExpr ")" "{" caseList "}"
caseList   ::= caseStmt { caseStmt } [ default ":" stmt ]
caseStmt   ::= case constant ":" stmt
```

Ausdrücke hinterlassen ihre Werte in einem Register. Die üblichen Präzedenzregeln spiegeln sich in den Abhängigkeiten der Produktionen. Konstante Ausdrücke, wie sie bei Case-Labels vorkommen, sind auf vorzeichenbehaftete Zahlen beschränkt. Die schon bei den Anweisungen vermerkte syntaktische Besonderheit mit Variablenzugriffen und Funktionsaufrufen treffen wir hier wieder.

```
orExpr    ::= andExpr { "||" andExpr }
andExpr   ::= relation { "&&" relation }
relation  ::= expr [ relOp Expr ]
expr      ::= term { addOp term }
term      ::= factor { mulOp factor }
factor    ::= "(" orExpr ")" | number |
              identifier [ index | actParams ] | unOp factor
relOp     ::= "==" | "!=" | "<" | ">" | "<= | ">=
addOp     ::= "+" | "-"
mulOp     ::= "*" | "/" | "%"
unOp      ::= "+" | "-" | "!"
```

8.2 Zielsprache

Die Zielsprache orientiert sich an den üblichen Assemblersprachen - sie ist zeilenorientiert und kennt Anweisungen, die Prozessorbefehle ausdrücken und Direktiven, die Zusatzinformationen beinhalten.

```
parse ::= { directive | newline | stmt } eof
```

Funktions- und End-Direktiven markieren den Namen sowie Beginn und Ende des Codes für eine Funktion, Label-Direktiven stehen für Sprungmarken und Variablen-Direktiven enthalten Adress-Informationen für globale und lokale Variable, wobei bei letzteren auch wegen der Adressierung von Funktionsparametern auch negative Attribute vorkommen können.

```
directive ::= "." ( funcDir | endDir | globalDir | localDir |
                    labelDir ) newline
funcDir   ::= function identifier
endDir    ::= end identifier
globalDir ::= global identifier number
localDir  ::= local identifier signedNumber
labelDir  ::= label identifier
signedNumber ::= [ "-" ] number
```

Die eigentlichen Assembler-Anweisungen enthalten den Befehlcode und optional eine Ergänzung durch einen Variablen- oder Funktionsnamen oder eine – ggf. auch negative – Zahl. Der Einfachheit halber werden die Befehlscode-Varianten nicht in Form von Scanner-Symbolen codiert, sondern die Zuordung findet erst auf Parser-Ebene durch die Auswertung der Bezeichner statt.

```
stmt :: identifier [ identifier | signedNumber ]
          [ identifier | signedNumber ] newline
```

An Stellen, an denen symbolische Bezeichner für Variablen-Adressen im Code vorkommen, kann der Assembler einfach die aus den zugehörigen Direktiven gemerkten Werte einsetzen. Bei Sprunglabeln (oder ggf. auch Funktionsbezeichnern), deren wirkliche Adressen erst später ermittelt werden, muss der Assembler über eine geeignete Buchführung diese Adressen nachträglich im Code einfügen.

8.3 Projektorganisation

Im Projekt werden zwei Applikationen realisiert: Das ist zum einen der Compiler selbst, der den Quellcode in den Zielcode übersetzt, und zum anderen ein Assembler-Interpreter. Dieser erzeugt im ersten Schritt aus dem Zielcode den internen Code der virtuellen Maschine und lädt ihn in deren Speicher. Im zweiten Schritt wird dann die virtuelle Maschine gestartet, die den so abgelegten Code ausführt.

8.4 Compiler

Der Compiler besteht neben der Programm-Klasse, die die statische Main-Methode enthält, aus den Klassen *Parser*, *Scanner*, *Symtab*, *Code*.

8.4.1 Parser

Die Klasse *Parser* realisiert einen Top-Down-Recursive-Descent-Parser für die Quellsprache und stellt neben dem öffentlichen Konstruktor nur noch die syntaktische Methode *parse* zur Verfügung, die dem Startsymbol der Grammatik zugeordnet ist.

```
public Parser(String inputName, String outputName)

public bool parse()
```

Im Konstruktor werden private Objekte der Klassen *Scanner*, *Symtab* und *Code* erzeugt, und mit den Namen von Ein- und Ausgabedateien versorgt.

Der Rückgabewert der Methode *parse* gibt Auskunft über die Anzahl der festgestellten Syntax- und Semantik-Fehler.

8.4.2 Scanner

Die Klasse *Scanner* ist für die lexikalische Analyse des Quelltextes verantwortlich. Sie stellt dem Parser Datentypen und Methoden zur Verfügung.

```
public enum SymbolType { SymNull, SymEof,
```

```
    SymNumber, SymIdentifier,
    SymIf, SymElse, SymWhile, SymDo, SymFor, SymSwitch,
    SymCase, SymDefault,
    SymPrint, SymRead, SymFunction, SymVar, SymReturn,
    SymConst,
    SymPlus, SymMinus, SymTimes, SymDiv, SymMod,
    SymLeftParen, SymRightParen, SymLeftBrace, SymRightBrace,
    SymLeftBracket, SymRightBracket,
    SymEqual, SymSemicolon, SymColon, SymComma, SymNot,
    SymEqualEqual, SymNotEqual, SymLess, SymGreater,
    SymLessEqual, SymGreaterEqual,
    SymLogicalAnd, SymLogicalOr
};
```

Der Aufzähltyp *SymbolType* ist der grundlegenden Typ für die syntaktischen Symbole.

```
public Scanner(String fileName)

public SymbolType GetNextSymbol()

public int GetNumber()

public String GetIdentifier()
```

Im Konstruktor wird die Quelldatei geöffnet und das Einlesen der Symbole vorbereitet. *GetNextSymbol* liest so lange Einzelzeichen aus dem Quelltext, bis ein vollständiges Symbol vorhanden ist. *GetNumber* bzw. *GetIdentifier* geben die Zahlen- und String-Werte zurück, die den entsprechenden Symbolen zugeordnet sind.

8.4.3 Symtab

Die Klasse *Symtab* realisiert eine einfache dreistufige Namenstabelle; die drei Stufen entsprechen dabei den Namensräumen für globale Konstanten, Variablen und Funktionen, für Funktionsparameter und lokale Variablen von Funktionen.

```
public enum Type { TypeVar, TypeFunc, TypeLVar, TypePar, TypeConst };

public class SymEntry
{
  public SymEntry(String n, Type t)
  {
```

```
    name = n;
    type = t;
  }
  public String name;
  public Type type;
}

public class VarEntry : SymEntry
{
  public VarEntry(String n)
    : base(n, Type.TypeVar)
  {
    dim = 1;
  }
  public int address;
  public int dim;
  public bool local;
}

public class FuncEntry : SymEntry
{
  public FuncEntry(String n)
    : base(n, Type.TypeFunc)
  {
    parameters = new List<SymEntry>();
  }
  public List<SymEntry> parameters;
}

public class ParEntry : SymEntry
{
  public ParEntry(String n)
    : base(n, Type.TypePar)
  {
  }
  public int address;
}

public class ConstEntry : SymEntry
{
  public ConstEntry(String n)
    : base(n, Type.TypeConst)
  {
    constVal = 0;
  }
  public int constVal;
};
```

Jeder Namenseintrag wird durch eine Instanz einer der Klassen *VarEntry*, *FuncEntry*, *ParEntry*, *ConstEntry* repräsentiert, die Attribute für den Namen selbst, den Typ, die Adresse, die Dimension, den Konstanten-Wert und ggf. die Parameterliste enthalten. Die Typen der Einträge sind im Aufzählltyp *Type* codiert.

```
public Symtab()

public void IncLevel(FuncEntry func)

public void DecLevel()

public bool PutSym(SymEntry entry)

public SymEntry GetSym(String name)
```

Der Konstruktor initialisiert die drei Tabellen, die Methoden *IncLevel* und *DecLevel* erhöhen bzw. vermindert den Tabellen-Level – wobei *IncLevel* der Funktionseintrag, das dem Level zugeordnet ist, mit übergeben wird. *PutSym* trägt einen Eintrag in die aktuelle Tabelle ein und gibt zurück, ob der Eintrag neu ist; *GetSym* gibt einen passenden Eintrag zurück oder auch *null*, wobei die Namenssuche vom aktuellen Level aus in absteigender Reihenfole stattfindet.

8.4.4 Code

Die Klasse *Code* stellt Parser Datentypen und Methoden zum Erzeugen des Assembler-Codes zur Verfügung.

```
public enum OpType
{
  lod, lodi, lodf, sto, stoi,
  jmp, jeq, jne, cll, clli, rtn, mks,
  wrt, rea,
  not, neg,
  cmp, cmpi,
  add, sub, mul, div, mod,
  equ, neq, grt, lth, geq, leq,
  addi, subi, muli, divi, modi,
  equi, neqi, grti, lthi, geqi, leqi,
  psh, pshi, pop, popi,
  nop, brk
}
```

Der Aufzählltyp *OpType* enthält die Befehlsworte des virtuellen Prozessors.

```
public Code(String fileName)

public void CloseOutput()

public void EmitSingle(OpType opcode)

public void EmitArg(OpType opcode, int arg)

public void EmitArg2Name(OpType opcode, int arg1, String name)

public void EmitArgName(OpType opcode, String name)

public void EmitArg2(OpType opcode, int arg1, int arg2)

public void EmitJump(OpType opcode, int labelNo)

public void EmitLabelInfo(int labelNo)

public void EmitVarInfo(bool local, String name, int address)

public void EmitFuncInfo(String name)

public void EmitEndInfo(String name)
```

Der Konstruktor öffnet die Ausgabe-Datei zum Schreiben, durch die Methode *CloseOutput* wird diese wieder geschlossen. Die Methoden *EmitSingle*, *EmitArg*, *EmitArgName*, *EmitArg2*, *EmitArg2Name*, *EmitJump* erzeugen Assemberbefehle ohne Argumente, Assemblerbefehle mit Zahlen-Argumenten bzw. Namensargumenten und Sprungmarken. Die Methoden *EmitLabelInfo*, *EmitVarInfo*, *EmitFuncInfo*, *EmitEndInfo* erzeugen die entsprechenden Assembler-Direktiven.

8.5 Assembler-Interpreter

Der Assembler-Interpreter besteht neben der Programm-Klasse, die die statische Main-Methode enthält, aus den Klassen *Parser*, *Scanner*, *Symtab*, *Code*. Im der Main-Methode werden Objekte der Klassen *Parser* und *Code* angelegt, die öffentliche Methode *parse* von *Parser* aufgerufen und im Erfolgsfalle anschließend die öffentliche Methode *Interprete* von *Code*.

8.5.1 Parser

Die Klasse *Parser* realisiert einen Parser für die Zielsprache des Projektes, der trotz der Einfachheit und der Zeilenorientierung ebenfalls als Top-Down-

Recursive-Descent-Parser angelegt ist. Auch er stellt neben dem öffentlichen Konstruktor nur noch die syntaktische Methode *parse* zur Verfügung, die dem Startsymbol der Grammatik zugeordnet ist.

```
public Parser(String inputName, Code aCode)

public bool parse()
```

Im Konstruktor wird ein privates Objekte der Klasse *Scanner* erzeugt und mit den Namen der Eingabedatei versorgt.

Der Rückgabewert der Methode *parse* gibt Auskunft darüber, ob das in der vorgelegten Datei enthaltene Assemblerprogramm syntaktisch korrekt ist oder nicht.

8.5.2 Scanner

```
public enum SymbolType { SymNull, SymEof,
  SymNumber, SymIdentifier, SymMinus,
  SymFunction, SymEnd, SymLocal, SymGlobal, SymLabel,
  SymDot, SymEol
};
```

Der Aufzähltype *SymbolType* enthält die grundlegenden Symbole der Assembler-Sprache, wobei der Einfachheit halber allerdings die Befehlsworte des Prozessors nicht als Symbole der Grammatik codiert sind, sondern dort nur in der Form eines Identifiers auftauchen. die

```
public Scanner(String fileName)

public SymbolType GetNextSymbol()

public int GetNumber()

public String GetIdentifier()
```

Im Konstruktor wird die Quelldatei geöffnet und das Einlesen der Symbole vorbereitet. *GetNextSymbol* liest so lange Einzelzeichen aus dem Quelltext, bis ein vollständiges Symbol vorhanden ist. *GetNumber* bzw. *GetIdentifier* geben die Zahlen- und String-Werte zurück, die den entsprechenden Symbolen zugeordnet sind.

8.5.3 Code

Die Klasse *Code* stellt Datentypen und Methoden zum Erzeugen und Interpretieren des Maschinencodes zur Verfügung.

```
public class Instruction
{
  public OpType opcode;
  public int arg1;
  public int arg2;
}

public Instruction[] code;
public int pc;
```

Die Klasse CodeRec enthält die Datenstruktur, die den Maschinenbefehlen zugrunde liegt, wobei der Aufzählтyp *OpType* mit dem des Parser-Projektes übereinstimmt.

Das Memory des virtuellen Prozessors ist als einfaches Integer-Array angelegt und unterteilt sich in das Memory für globale Variablen und den Stack.

```
public Code()

public void GenSingle(OpType op)

public void GenOne(OpType op, int arg)

public void GenTwo(OpType op, int arg1, int arg2)

public void List(String fileName)

public void Interprete(int start, int trace)
```

Im Konstruktor werden die Maschinencode und Memory-Objekte angelegt. Die Methoden *GenSingle*, *GenOne*, *GenTwo* erzeugen die Maschinenbefehle ohne Argumente bzw. mit einem oder zwei Argumenten sowie Sprungbefehle. Die Methode *List* schreibt ein Listing des Maschinencodes (mit aufgelösten Adressen) in die angegebene Datei und die Methode *Interprete* schließlich führt den vom Parser abgelegten Code aus; über Parameter lassen sich die Startadresse sowie ein Einzelbefehlstrace einstellen.

8.6 Beispiel

Das folgende einfachen Beispiel illustriert noch einmal die Verwendung von Quell- und Zielsprache:

Es enthält eine einfache Funktion, zwei globale Variablen, die zugewiesen und als Argumente an die Funktion übergeben werden.

```
var a, b;

function f(x, y)
{
  return x * y - 1;
}

a = 2;
b = 4;

print f(a,b);
```

Der Code beginnt mit einem Sprung zum ersten auszuführenden Befehl, enthält Direktiven zum Bereitstellen des Speichers für die globalen Variablen, die Funktionsdirektive mit dem anschließenden Code der Funktion und danach das durch *brks* abgeschlossene Hauptprogramm.

```
 jmp label_0
.global a 0
.global b 1

.function f
.local rtn -4
.local x -3
.local y -2
 mks 0
 lodf 1 rtn
 lodf 2 x
 lod 2 2
 lodf 3 y
 lod 3 3
 mul 2 3
 subi 2 1
 sto 2 1
 rtn 2
 rtn 2
.end f
```

```
.label label_0
 lodi 1 2
 stoi 1 a
 lodi 1 4
 stoi 1 b
 psh 0
 lodi 2 a
 lod 2 2
 psh 2
 lodi 2 b
 lod 2 2
 psh 2
 clli f
 pop 1
 wrt 1
 brk
```

Der Interpreter gibt als erstes ein Listing des erzeugten Maschinencodes – mit den von ihm aufgelösten Adressen – in die angegebene Datei aus.

```
0: jmp 12
1: mks 0
2: lodf 1 -4
3: lodf 2 -3
4: lod 2 2
5: lodf 3 -2
6: lod 3 3
7: mul 2 3
8: subi 2 1
9: sto 2 1
10: rtn 2
11: rtn 2
12: lodi 1 2
13: stoi 1 0
14: lodi 1 4
15: stoi 1 1
16: psh 0
17: lodi 2 0
18: lod 2 2
19: psh 2
20: lodi 2 1
21: lod 2 2
22: psh 2
23: clli 1 0
24: pop 1
25: wrt 1
26: brk
```

Bei der Interpretation wird der Funktionswert gedruckt und abschließend die aktuellen Werte von program counter, stack pointer und frame pointer.

```
 7
pc=27, sp=1001, fp=0
End of Program
```

Anhang: Befehlssatz

Hier ist noch einmal der verwendete Befehlssatz für unsere Register-Maschine zusammengefasst.

In der kompakten Code-Notation stehen *Regs* für die normalen Register R0 - R15, *Mem* für das Memory, *pc*, *fl*, *sp*, *fp*, für die Spezialregister Program-Counter, Flags, Stack-Pointer und Frame-Pointer.

Register-Manipulation

```
lod <r1> <r2>     // load register r1 with memory at contents(r2)
                  // Regs[r1] = Mem[Regs[r2]]

lodi <r> <num>    // load register r with constant num (also address)
                  // Regs[r] = num

lodf <r> <num>    // load register r with contents(fp)+num
                  // Regs[r] = fp+num

sto <r1> <r2>     // store content(r1) to memory at contents(r2)
                  // Mem[Regs[r2]] = Regs[r1]

stoi <r1> <adrs>  // store content(r1) to memory at adrs
                  // Mem[adrs] = Regs[r1]

psh <r>           // Push contents(r) onto the stack
                  // sp=sp + 1; Mem[sp] = Regs[r]

pop <r>           // Pop contents(r) from the stack
                  // Regs[r] = Mem[sp]; sp = sp - 1
```

Input und Output

```
wrt <r>          // write contents(r)

rea <r>          // read into r
```

Vergleichsoperationen

Verglichen werden die Inhalte zweier Register bzw. der Inhalt eines Registers mit einer Konstanten. Das Ergebnis wird im Flag-Register abgelegt.

```
cmp <r1> <r2>    // Compares contents(r1) with contents(r2), result in Flag-Register fl
                 // fl = Regs[r1] - Regs[2]

cmpi <r1> <num>  // Compares contents(r1) with contents(r2), result in Flag-Register fl
                 // fl = Regs[r1] - num
```

Relationaloperatoren

Im Gegensatz dazu werden werden die Inhalte zweier Register relational verknüpft und das Ergebnis im ersten Register abgelegt.

```
lth <r1> <r2>    // less
                 // Regs[r1] = Regs[r1] < Regs[r2]

grt <r1> <r2>    // greater
                 // Regs[r1] = Regs[r1] > Regs[r2]

equ <r1> <r2>    // equal
                 // Regs[r1] = Regs[r1] == Regs[r2]

neq <r1> <r2>    // not equal
                 // Regs[r1] = Regs[r1] != Regs[r2]

leq <r1> <r2>    // less or equal
                 // Regs[r1] = Regs[r1] <= Regs[r2]

geq <r1> <r2>    // greater or equal
                 // Regs[r1] = Regs[r1] <= Regs[r2]
```

```
lthi <r1> <num>   // less
                  // Regs[r1] = Regs[r1] < num

grti <r1> <num>   // greater
                  // Regs[r1] = Regs[r1] > num

equi <r1> <num>   // equal
                  // Regs[r1] = Regs[r1] == num

neqi <r1> <num>   // not equal
                  // Regs[r1] = Regs[r1] != num


leqi <r1> <num>   // less or equal
                  // Regs[r1] = Regs[r1] <= num

geqi <r1> <num>   // greater or equal
                  // Regs[r1] = Regs[r1] >= num
```

Binäre arithmetische Operationen

Miteinander verknüpft werden die Inhalte zweier Register, bzw. der Inhalt eines Registers mit einer Konstanten. Das Ergebnis wird im ersten Register abgelegt.

```
add <r1> <r2>      // addition
                   // Regs[r1] = Regs[r1] + Regs[r2]

sub <r1> <r2>      // subtraction
                   // Regs[r1] = Regs[r1] - Regs[r2]

mul <r1> <r2>      // multiplication
                   // Regs[r1] = Regs[r1] * Regs[r2]

div <r1> <r2>      // division
                   // Regs[r1] = Regs[r1] / Regs[r2]
                   // runtime error if Regs[r2] == 0

mod <r1> <r2>      // modulo
                   // Regs[r1] = Regs[r1] % Regs[r2]
                   // runtime error if Regs[r2] <= 0

addi <r1> <num>    // addition
                   // Regs[r1] = Regs[r1] + num

subi <r1> <num>    // subtraction
```

```
                    // Regs[r1] = Regs[r1] - num

muli <r1> <num>     // multiplication
                    // Regs[r1] = Regs[r1] * num

divi <r1> <num>     // division
                    // Regs[r1] = Regs[r1] / num
                    // runtime error if num == 0

modi <r1> <num>     // modulo
                    // Regs[r1] = Regs[r1] + num
                    // runtime error if num <= 0
```

Unäre arithmetische Operationen

Geändert wird das genannte Register.

```
neg <r>             // unary arithmetic inversion
                    // Regs[r] = - Regs[r]

not <r>             // unary logical negation
                    // Regs[r] = ! Regs[r]
```

Sprungbefehle

Die Ausführung wird unbedingt, bzw. bedingt an der angegebenen Adresse fortgesetzt.

```
jmp <adrs>          // jump unconditionally
                    // pc = adrs - 1

jne <adrs>          // jump if flag register contains not equal
                    // if (fl != 0) pc = adrs - 1

jeq <adrs>          // jump if flag register contains equal
                    // if (fl == 0) pc = adrs - 1
```

Funktionsbefehle

```
cll <r>            // push return address
                   // sp = sp + 1; Mem[sp] = pc;
                   // jump to address contained in r
                   // pc = Regs[r] - 1

clli <adrs>        // push return address
                   // sp = sp + 1; Mem[sp] = pc;
                   // jump to address adrs
                   // pc = adrs - 1

mks <num>          // push the frame pointer
                   // sp = sp + 1; Mem[sp] = fp;
                   // set up the new frame pointer
                   // fp = sp;
                   // make room for <num> local variables
                   // sp = sp + num;
                   // and save the registers
                   // for(n=0;n<16;n++) Mem[sp+n+1] = Regs[n]; sp = sp + 16;

rtn <num>          // return from function with <num> parameters
                   // restore the registers
                   // sp = sp - 16; for (n=0;n<16;n++) Regs[n] = Mem[sp+1+n];
                   // restore the stack pointer (from frame pointer)
                   // sp = fp;
                   // pop the frame pointer
                   // fp = Mem[sp]; sp = sp - 1;
                   // pop the return address to pc
                   // pc = Mem[sp]; sp = sp - 1;
                   // adjust the stack pointer (remove <num> parameters)
                   // sp = sp - num
```

Ausführungsende

```
brk                // stop execution, go to monitor mode
```

Literaturverzeichnis

[1] A.V. Aho, M.S. Lam, R. Sethi, J. D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison Wesley, Boston 2006

[2] D. Gries. *Compiler Construction for Digital Computers*. Wiley, New York-London-Sydney-Toronto 1971

[3] D. Grune and C.J.H. Jacobs. *A Programmer-friendly LL(1) Parser Generator*. Software–Practice and Experience, 18(1), 29-38, 1988

[4] A.C. Hartmann. *A Concurrent Pascal Compiler for Minicomputers*. Springer Lecture Notes in Computer Science (50), Berlin-Heidelberg-New York 1977

[5] J.R. Levine, T. Mason, D. Brown. *lex & yacc*. O'Reilly, Sebastopol 1992

[6] R. Mak. *Writing Compilers & Interpreters*. Wiley, New York-Chichester-Brisbane-Toronto-Singapore 1991

[7] D.P. Scarpazza. *Practical Parsing for ANSI C*. Dr. Dobb's Journal 1-2007, 48-55

[8] N. Wirth. *Compilerbau*. Teubner Studienbücher Informatik, Stuttgart 1984

[9] N. Wirth. *Grundlagen und Techniken des Compilerbaus*. Addison Wesley, Bonn, Paris [u.a.] 1996

[10] X/Open Company. *X/Open Portability Guide – Programming Languages*. Prentice Hall, Englewood Cliffs, 1988