Compilerbau

P. Köhler

Sommersemester 2019

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

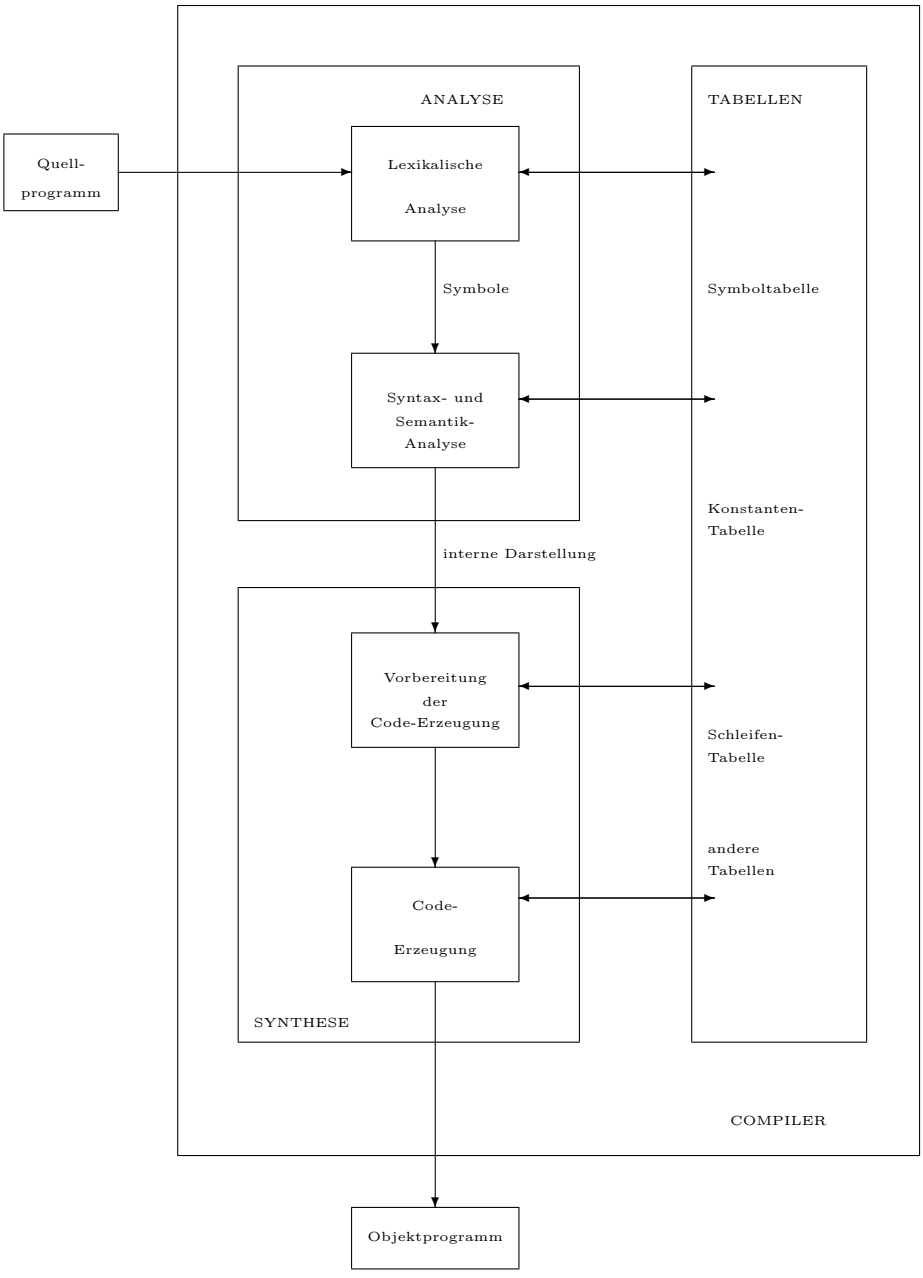


Abbildung 2.1: Schematischer Aufbau eines Compilers

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) $\quad stmt ::= expr \ =$

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

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 startWert)
    {
      int value = 10 * startWert + nextchar - '0';
      nextchar = Console.Read();
      if (isdigit(nextchar))
      {
        value = number(value);
      }
      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(0);
  }
  // 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 == '=')
      {
        nextchar = Console.Read();
        Console.WriteLine(value);
      }
      else error();
    }

    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

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

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,

(3.10) $$expr ::= number$$

(3.11) $$expr ::= expr \; + \; expr$$

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

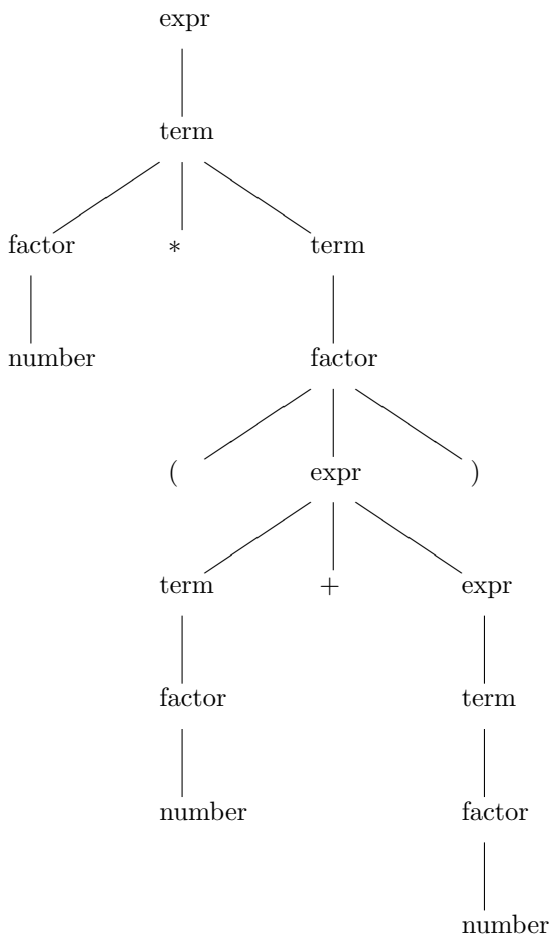


Abbildung 3.1: Syntaxbaum für $number * (number + number)$

(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) $stmt ::= expr\ =$

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

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

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

(3.18) $number ::= digit\{\ digit\ \}$

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.*

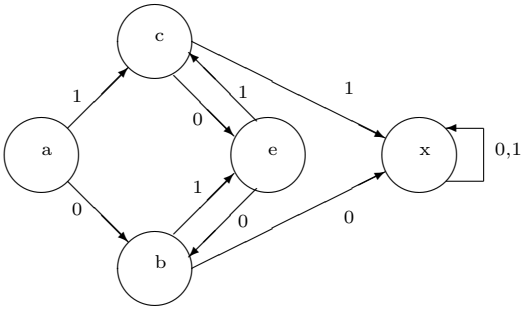


Abbildung 4.1: Einfacher Zustandsautomat

(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:

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

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

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

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;
```

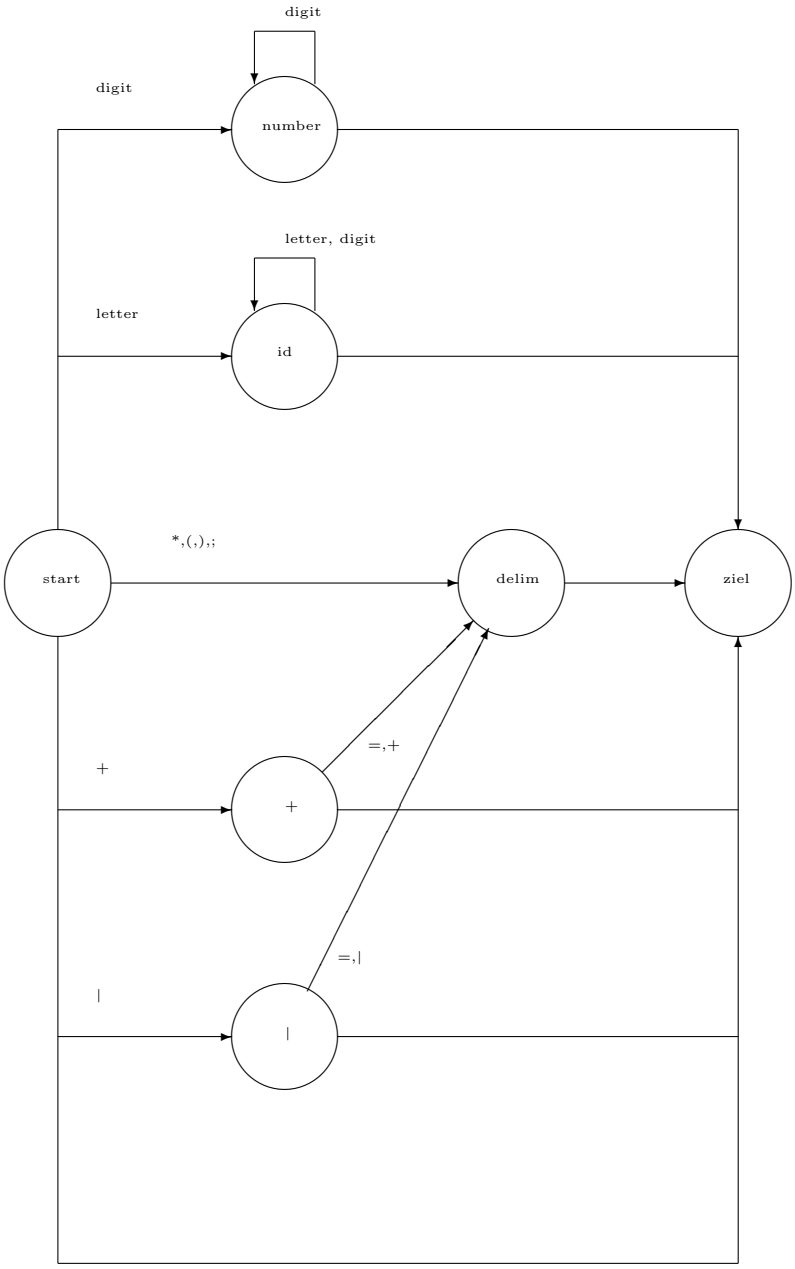


Abbildung 4.2: Zustandsübergangsdiagramm für den Scanner

```
        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.

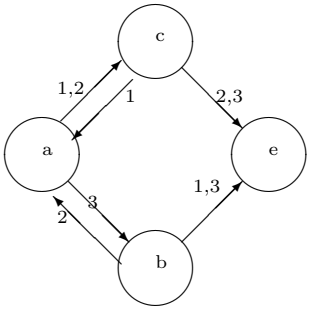
Für eine weitergehende Einführung in Lex (und Yacc) verweisen wir auf [5].

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.

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