2 Allgemeines zu lex und yacc

Wer das erste Knopfloch verfehlt, kommt mit dem Zuknöpfen nicht zu Rande.

Goethe

Die Arbeitsweise eines Compilers, der die Übersetzung eines ihm vorgelegten Quellprogramms in verschiedenen Phasen vornimmt, eignet sich hervorragend zur Einführung in die beiden Werkzeuge **lex** und **yacc**, die bei der Realisierung der beiden Phasen »Lexikalische Analyse« (**lex**) und »Syntaxanalyse« (**yacc**)[1] eingesetzt werden können.

Hier werden zunächst alle Phasen eines Compilers vorgestellt, bevor die beiden Begriffe »*Lexikalische Analyse*« und »*Syntaxanalyse*« einer genaueren Betrachtung unterzogen werden, indem ein einfacher Taschenrechner sowohl ohne als auch mit Verwendung der beiden Tools **lex** und **yacc** realisiert wird.

Dieses Kapitel gibt somit bereits einen Einblick in die Einsatzmöglichkeiten und die Arbeitsweisen von **lex** und **yacc**.

2.1 Phasen eines Compilers

Im allgemeinen ist der Übersetzungsvorgang eines Compilers in verschiedene Phasen unterteilt, wobei in manchen Compilern mehrere Phasen zu einer zusammengefaßt sein können. Abbildung 2.1 zeigt die Phasen eines Compilers in der Reihenfolge, wie sie bei der Übersetzung eines Quellprogramms durchlaufen werden.

Die *Symboltabelle* ist die zentrale Datenstruktur, die für jeden Bezeichner eines Quellprogramms einen Platz vorsieht, in dem Name, Datentyp und sonstige Attribute festgehalten werden.

Neben dem Symboltabellen-Manager existiert meist noch ein weiteres zentrales Modul »*Fehlerbehandlung*«, das von allen Phasen benutzt wird. Das Fehlerbehandlungsmodul enthält üblicherweise die zu allen möglichen Fehlerarten gehörigen Nummern und Meldungen.

1. Das Werkzeug **yacc** wird im nächsten Kapitel 3 vorgestellt.

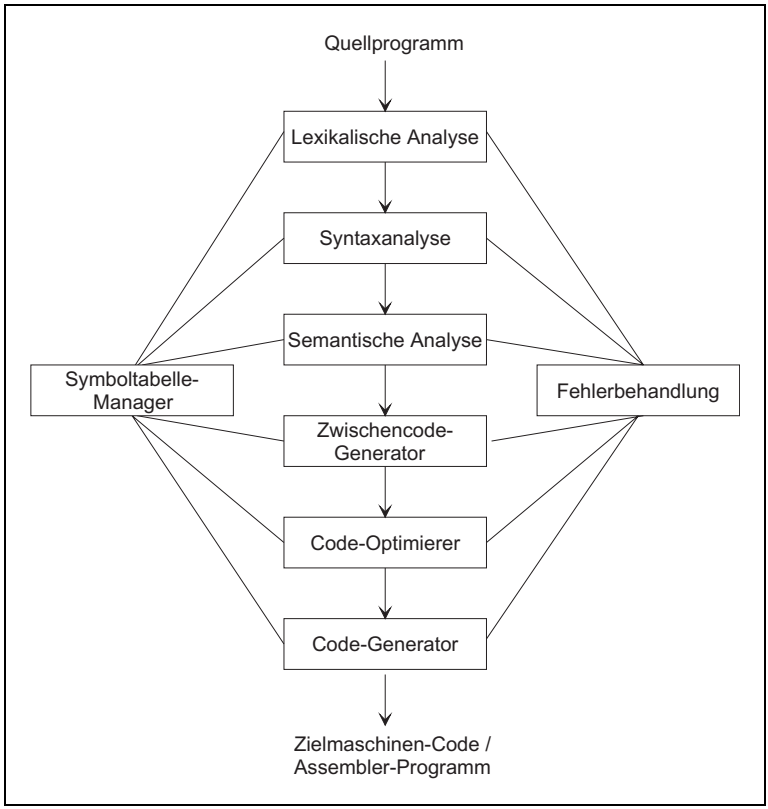

Abbildung 2.1: Typische Phasen eines Compilers

Abbildung 2.2 soll anhand eines einfachen Beispiels (Übersetzung einer Anweisung) die Funktionsweise jeder einzelnen Phase verdeutlichen.

In der Phase **Lexikalische Analyse** wird das vorgelegte Quellprogramm in sogenannte Token zerlegt. Ein Token ist ein Überbegriff für die Bestandteile eines Programms; z. B. könnte das Token *BEZEICHNER* für Variablennamen, das Token *INTEGER-KONSTANTE* für ganze Zahlen, das Token *OPERATOR* für die Zeichen +, –, *, / usw. stehen. So könnte die Anweisung `neuwert = wert + x * 50` in der lexikalischen Analyse eines C-Compilers wie folgt analysiert werden:

Komponente	*Token*
`neuwert`	*BEZEICHNER*
`=`	*OPERATOR*
`wert`	*BEZEICHNER*
`+`	*OPERATOR*

Komponente	*Token*
x	*BEZEICHNER*
*	*OPERATOR*
50	*INTEGER-KONSTANTE*

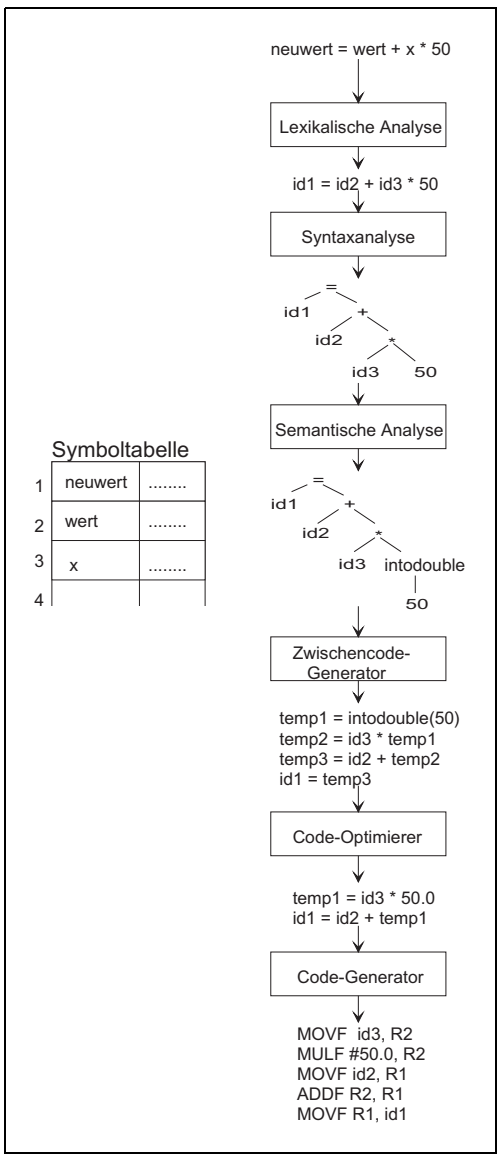

*Abbildung 2.2: Compiler-Phasen für die C-Anweisung »neuwert = wert + x*50«*

Die **Syntaxanalyse** ist für die Gruppierung der einzelnen Token und die Prüfung auf syntaktische Richtigkeit verantwortlich.

Die **Semantische Analyse** muss z. B. erkennen, dass ein Operand zuerst in **double** umzuwandeln ist, bevor er mit den anderen Operanden eines Ausdrucks verknüpft werden kann.

Der **Zwischencode-Generator** übersetzt das Quellprogramm in eine Zwischensprache, die dann vom **Code-Optimierer** verbessert wird, indem z. B. redundante Um- und Zwischenspeicherungen erkannt und beseitigt werden. Der **Codegenerator** schließlich übersetzt das ihm in der Zwischensprache vorgelegte Programm in den Code der Zielmaschine oder in ein Assemblerprogramm.

Da im Zusammenhang mit den beiden Werkzeugen **lex** und **yacc** nur die beiden ersten Phasen »Lexikalische Analyse« und »Syntaxanalyse« von Interesse sind, wird in den nächsten beiden Kapiteln vertieft auf diese beiden Begriffe eingegangen.

2.2 Lexikalische Analyse

Die lexikalische Analyse ist für die Zerlegung eines Textes verantwortlich, indem sie entsprechende Textteile nach vorgegebenen Regeln erkennt und klassifiziert. Für jede Klasse von Textteilen werden dabei sogenannte Token definiert. Ein Token ist der Klassenname, wie z. B. *BEZEICHNER* für Variablennamen, *PLUS* für den Operator **+**, *SCHLÜSSELWORT* für **if**, **for**,... usw. Da die lexikalische Analyse einen Eingabetext nach bestimmten Textmustern abtastet, wird sie oft auch als *Scanner* bezeichnet.

Anhand eines Beispiels soll die lexikalische Analyse verdeutlicht werden. Dazu wird nun ein einfacher Taschenrechner entwickelt, der die folgenden Operatoren kennt:

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
( )	Klammerung von Ausdrücken

Als Operanden sind ganze Zahlen erlaubt.

Der Wert jedes in einer Zeile angegebenen arithmetischen Ausdrucks soll unmittelbar ausgegeben werden.

Bei den nachfolgenden Realisierungen mit der Programmiersprache C werden folgende globalen Definitionen verwendet:

```
$ cat global.h ↵    [enthält programmglobale Typ- und Konstanten-Definitionen]
#include <stdio.h>
#include <ctype.h>

#define FALSE    0
#define TRUE     1

#define ZAHL   256
#define PLUS   257  /* + */
#define MINUS  258  /* - */
#define MULT   259  /* * */
#define DIV    260  /* / */
#define AUF    261  /* ( */
#define ZU     262  /* ) */

#define NICHTS     263
#define ZEILENENDE 264
#define FERTIG     265
$
```

Lexikalische Analyse für einen Taschenrechner mit C

Die lexikalische Analyse für diesen Taschenrechner wird mit dem folgenden C-Modul *lexan.c* durchgeführt:

```
$ cat lexan.c ↵
#include "global.h"

int tokenwert = NICHTS; /* Programmglobale Variable, der im Falle   */
                        /* einer Zahl Zahlenwert zugewiesen wird.   */
int zeilennr = 1;  /* Programmglobale Variable:                    */
                   /* enthaelt immer Nummer der akt. Eingabezeile   */

int lexan(void)    /* Lexikalische Analyse */
{
   char zeich;
   while (TRUE) {
      zeich = getchar();
      if (zeich==' ' || zeich=='\t') /*Leer- u. Tabzeich. ueberlesen*/
          ;
      else if (zeich == '\n') {
         return(ZEILENENDE);
      } else if (isdigit(zeich)) {
         ungetc(zeich, stdin);  /* Zuviel gelesenes Zeichen zurueck */
                                /* in den Eingabepuffer             */
         scanf("%d", &tokenwert);
         return(ZAHL);
      } else if (zeich == EOF) {
         return(FERTIG);
      } else {
```

```
            switch (zeich) {
                case '+': return(PLUS);
                case '-': return(MINUS);
                case '*': return(MULT);
                case '/': return(DIV);
                case '(': return(AUF);
                case ')': return(ZU);
                default : tokenwert=zeich; return(NICHTS);
            }
        }
    }
}
$
```

Lexikalische Analyse für einen Taschenrechner mit lex

Mit der Verwendung von **lex** kann die lexikalische Analyse für diesen Taschenrechner wesentlich einfacher realisiert werden:

```
$ nl -ba lexan.l ↵      [lex-Programm mit Zeilennummern ausgeben]
     1  %{
     2  #include <stdlib.h>
     3  #include "global.h"
     4
     5  int tokenwert = NICHTS; /*Programmglob. Var., der im Falle */
     6                          /* einer Zahl Zahlenwert zugew. Wird */
     7  int zeilennr = 1;  /* Programmg. Variable: enthaelt immer  */
     8                     /* Nummer der aktuellen Eingabezeile    */
     9  %}
    10  %%
    11  [ \t]+   ;  /* Leer- und Tabzeichen ueberlesen */
    12  \n       { return(ZEILENENDE);   }
    13  [0-9]+   { tokenwert = strtol(yytext,NULL,10); return(ZAHL); }
    14  "+"      { return(PLUS); }
    15  "-"      { return(MINUS); }
    16  "*"      { return(MULT); }
    17  "/"      { return(DIV); }
    18  "("      { return(AUF); }
    19  ")"      { return(ZU); }
    20  .        { tokenwert = yytext[0]; return(NICHTS); }
    21  %%
$
```

An diesem einfachen Beispiel ist bereits der grundsätzliche Aufbau eines lex-Programms erkennbar:

%{
C-Deklarationen und C-Definitionen, die für lexikalische Analyse benötigt werden.
%}

```
%%
   Regelteil
%%
```

Im Regelteil wird für jedes mögliche Token ein regulärer Ausdruck angegeben. Kommt im Eingabetext ein String vor, der durch einen der angegebenen regulären Ausdrücke abgedeckt wird, so wird der danach angegebene C-Code (meist in { } eingebettet) ausgeführt:

Zeilennummer	*Bedeutung*
11	Alle Leer- und Tabzeichen ignorieren.
12	Für Neuezeilezeichen das Token ZEILENENDE zurückgeben.
13	*[0-9]+* deckt alle ganzen Zahlen im Eingabetext ab. Ein durch einen regulären Audruck abgedeckter String wird von **lex** im char-Array *yytext* gespeichert. Mit dem Aufruf der C-Bibliotheksfunktion *strtol* wird der String aus *yytext* in eine Zahl umgewandelt und der Variablen *tokenwert* zugewiesen, bevor das Token ZAHL zurückgegeben wird.
14 bis 19	Decken die Zeichen +, –, *, /, (und) ab und geben entsprechend eines der Token PLUS, MINUS, MULT, DIV, AUF und ZU zurück.
20	Hierbei handelt es sich um eine Default-Angabe. Der Punkt deckt ein beliebiges Zeichen ab. Diese Regel wird nur ausgeführt, wenn ein Zeichen nicht durch einen der zuvor angegebenen regulären Ausdrücken abgedeckt wird. In diesem Fall wird das Token NICHTS zurückgegeben.

Dieses lex-Programm *lexan.l* muss nun **lex** vorgelegt werden. **lex** erzeugt hieraus ein C-Programm *lex.yy.c*. Der Inhalt dieses automatisch von **lex** erzeugten C-Moduls ist hier nicht von Interesse. Entscheidend ist, dass der Regelteil von **lex** in eine C-Funktion *yylex* umgeformt wird, die die lexikalische Analyse für den Taschenrechner durchführt. *yylex()* liefert bei jedem Aufruf eines der im Regelteil mit **return** zurückgegebenen Token. Am Ende des Eingabetextes liefert *yylex()* den Wert 0. Um die lexikalische Analyse durchführen zu lassen, wird deshalb meist folgendes angegeben:

```
while (token=yylex()) {
   ....
}
```

```
$ lex lexan.l ↵      [lex erzeugt aus lex-Programm lexan.l ein C-Programm lex.yy.c]
$ ls lex.yy.c ↵
lex.yy.c
$ cat lexantest.c ↵  [Treiber-Modul zum Testen der von lexan.l durchgeführt. lex. Anal.]
#include "global.h"
```

```
extern int zeilennr, tokenwert;

int main(void)
{
    int token;
    printf("%5d : ", zeilennr);
    while (token=yylex()) {
        switch (token) {
          case NICHTS    : printf("%c (in Zeile "
                                   "%d nicht klassifizierbar)\n",
                                   tokenwert, zeilennr); break;
          case ZAHL      : printf("%d (ZAHL)\n", tokenwert); break;
          case PLUS      : printf("+ (PLUS)\n"); break;
          case MINUS     : printf("- (MINUS)\n"); break;
          case MULT      : printf("* (MULT)\n"); break;
          case DIV       : printf("/ (DIV)\n"); break;
          case AUF       : printf("( (AUF)\n"); break;
          case ZU        : printf(") (ZU)\n"); break;
          case ZEILENENDE: printf("-----\n");
                           printf("%5d : ", ++zeilennr); break;
          default        : printf("Programmfehler\n"); break;
        }
    }
    return(0);
}
$ cc -o lexan lex.yy.c lexantest.c -ll ↵   (bei LINUX ist -lfl statt -ll anzugeb.)
$
```

Die Kommandozeile

cc -o lexan lex.yy.c lexantest.c -ll bzw.
cc -o lexan lex.yy.c lexantest.c -lfl (unter LINUX)

bewirkt, dass zunächst die beiden C-Quelldateien *lex.yy.c* und *lexantest.c* kompiliert werden, bevor die daraus resultierenden Objektdateien *lex.yy.o* und *lexantest.o* mit den aus der lex-Bibliothek */usr/lib/libl.a* (**-ll**) bzw. unter Linux */usr/lib/libfl.a* (**-lfl**) benötigten Funktionen und Definitionen zusammengebunden (gelinkt) werden. Das daraus entstehende (ausführbare) Programm wird in die Datei *lexan* (**-o lexan**) geschrieben. Nun soll kurz getestet werden, ob *lexan* die lexikalische Analyse für den Taschenrechner richtig durchführt:

```
$ lexan ↵
    1 : (2 + 3) *20 ↵
( (AUF)
2 (ZAHL)
+ (PLUS)
3 (ZAHL)
) (ZU)
* (MULT)
20 (ZAHL)
-----
```

```
    2 : 100/3^2↵
100 (ZAHL)
/ (DIV)
3 (ZAHL)
^ (in Zeile 2 nicht klassifizierbar)
2 (ZAHL)
-----
    3 : Strg-D
$
```

Die lexikalische Analyse zerlegt zwar den Eingabetext in einzelne Token, macht aber keinerlei Prüfung, ob die Reihenfolge der Token sinnvoll ist. Auch können Vorrangsregeln wie »Punkt vor Strich« bei der lexikalischen Analyse nicht berücksichtigt werden. Zur Prüfung solcher Regeln ist die Syntaxanalyse erforderlich.

Unter Linux existiert kein eigenes Programm **lex**, sondern das frei verfügbare GNU-Programm **flex**, das weitgehend identisch zu dem in diesem Kapitel beschriebenen **lex** ist. Ruft man unter Linux **lex** auf, so ruft man das Shellskript **/usr/bin/lex** auf, das meist das folgende Aussehen hat:

```
#!/bin/sh
/usr/bin/flex -l $*
```

Dieses Shellskript ruft also **flex** mit der Option **-l** auf. Diese Option macht **flex** kompatibel zum Standard-**lex**.

2.2.1 Syntaxanalyse

Die für einen Eingabetext vorgegebene Syntax legt die Regeln und die Grammatik fest, die für diesen Text gelten. Um eine Syntax zu beschreiben, existieren verschiedene Darstellungsmöglichkeiten:

Syntaxdiagramm

Ein Beispiel für Syntaxdiagramme zeigt die Abbildung 2.3.

Backus-Naur Form

Eine andere und sehr häufig benutzte Darstellungsform ist die *Backus-Naur Form* (**BNF**), die oft auch *kontextfreie Grammatik* genannt wird.

Eine kontextfreie Grammatik hat 4 Komponenten:

1. Eine Menge von *terminalen Symbolen*.
   Terminale Symbole sind die Token.
2. Eine Menge von *nichtterminalen Symbolen*.
   Nichtterminale Symbole sind Überbegriffe für Konstruktionen, die sich aus terminalen und/oder nichtterminalen Symbolen zusammensetzen.

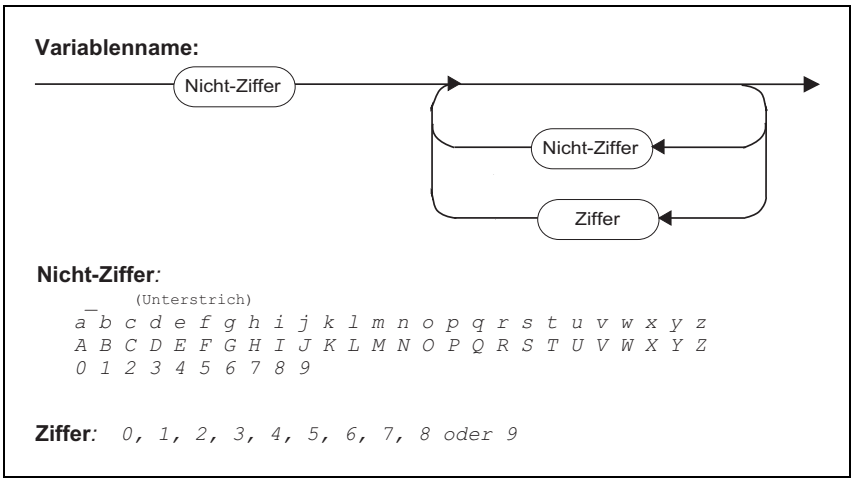

Abbildung 2.3: Syntax-Diagramm für eine C-Variable

3. Eine Menge von *Produktionen*
   Jede Produktion wird dabei wie folgt angegeben:
   linke Seite ---> *rechte Seite*
   linke Seite muss ein nichtterminales Symbol sein.
   rechte Seite ist eine Folge von terminalen und/oder nichtterminalen Symbolen.

4. Startsymbol
   Ein nichtterminales Symbol legt immer das Startsymbol fest.

Die Produktion mit dem Startsymbol wird bei der Angabe von Produktionen immer zuerst angegeben.

Produktionen mit dem gleichen nichtterminalen Symbol auf der linken Seite, wie z. B.

op ---> **+**
op ---> **-**
op ---> *****
op ---> **/**

können zu einer Produktion zusammengefaßt werden, indem die einzelnen rechten Seiten mit | voneinander getrennt angegeben werden:

op ---> **+** | – | ***** | **/**

Oft wird eine solche Produktion auch wie folgt angegeben:

op ---> **+**
| **-**
| *****
| **/**

Es gibt auch ein leeres Symbol ε.

In den nachfolgenden Beispielen werden terminale Symbole fett und nichtterminale Symbole kursiv angegeben.

Beispiel Ein Beispiel einer kontextfreien Grammatik für eine erlaubte C-Variable ist:

```
Bezeichner  --->  Nicht-Ziffer  Buchziffer

Buchziffer  --->  Nicht-Ziffer  Buchziffer
             |     Ziffer  Buchziffer
             |     ε

Nicht-Ziffer --->  _
              |    A | B | C | D | E | F | G | H | I | J | K | L | M
              |    N | O | P | Q | R | S | T | U | V | W | X | Y | Z
              |    a | b | c | d | e | f | g | h | i | j | k | l | m
              |    n | o | p | q | r | s | t | u | v | w | x | y | z

Ziffer       --->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Die Syntax für den in Kapitel 2.1.2 beschriebenen Taschenrechner hat in BNF z. B. folgendes Aussehen:

```
zeilen     --->  ausdruck  ZEILENENDE  zeilen
            |    ε

ausdruck   --->  ausdruck  PLUS  term
            |    ausdruck  MINUS  term
            |    term

term       --->  term  MULT  factor
            |    term  DIV  factor
            |    factor

factor     --->  ZAHL
            |    MINUS  ZAHL
            |    AUF  ausdruck  ZU
```

Die Herleitung einer Eingabezeile für den Taschenrechner kann dabei sehr anschaulich an einem sogenannten *Parse tree* gezeigt werden. Zum Beispiel ergibt sich für die Eingabezeile

4 + 5 * 2

der Parse tree aus Abbildung 2.4.

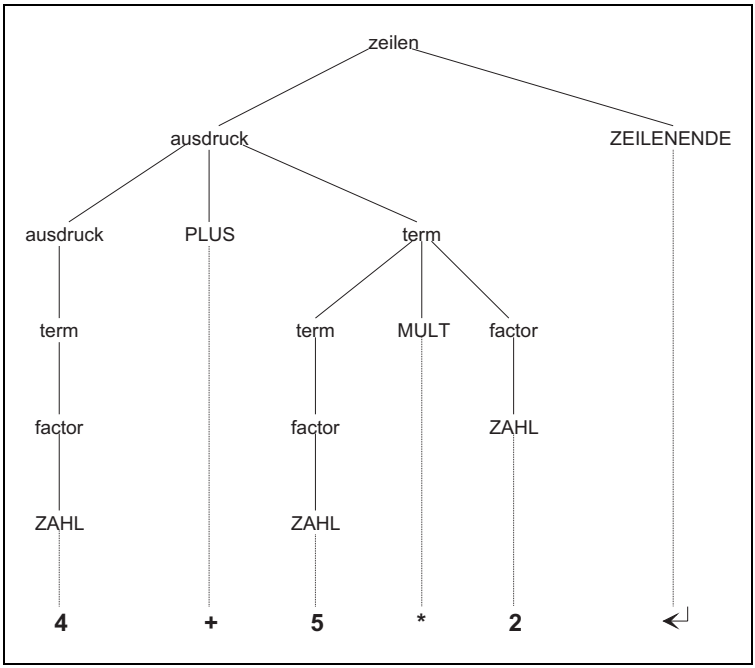

*Abbildung 2.4: Parse tree für die Eingabezeile »4+5*2«*

Ein Parse tree ist ein Baum mit folgenden Eigenschaften:

1. Die Wurzel ist das Startsymbol.
2. Jedes Blatt ist ein Token (terminales Symbol) oder **ε**.
3. Jeder innere Knoten (nicht die Blätter) ist ein nichtterminales Symbol.
4. Wenn *A* ein innerer Knoten ist, der die Kinder *b1*, *b2*, ..., *bn* hat, stellt dies die Produktion *A* ---> *b1 b2 ... bn* dar.

Aus dem Parse tree in Abbildung 2.4 wird deutlich, dass zuerst *5*2* zu berechnen ist, bevor auf dieses Ergebnis der Wert *4* addiert wird.

Syntaxanalyse für einen Taschenrechner mit C

Der Vorteil von kontextfreien Grammatiken ist, dass man für die nichtterminalen Symbole Funktionen schreiben kann, die dann nur noch entsprechend den vorgegebenen Regeln aufzurufen sind. Dabei wird die rekursive Definition in einer Grammatik, wie z. B. bei

factor ---> **AUF** *ausdruck* **ZU**

durch einen rekursiven Funkionsaufruf realisiert. So kann z. B. die Syntaxanalyse für den Taschenrechner aus Kapitel 2.1.2 mit dem folgenden C-Modul

parser.c durchgeführt werden. Die Phase der Syntaxanalyse wird oft auch als Parsing bezeichnet, weshalb die betreffende Routine in diesem C-Modul auch *parse()* genannt wird.

```
$ cat parser.c ↵
#include "global.h"

int lookahead;
extern int tokenwert, zeilennr;

static void  fehler(int z);   /* Prototyp-Deklarationen */
static void  passt(int token);
static int   ausdruck(void);
static int   term(void);
static int   factor(void);
       void  parse(void);

static void fehler(int z)
{
   printf("Syntaxfehler in Zeile %d\n", z);
   exit(1);
}

static void passt(int token)
{
    if (lookahead == token)
       lookahead = lexan();
    else
       fehler(zeilennr);
}

static int ausdruck(void)
{
   int ergeb = term();
   while (TRUE) {
      switch(lookahead) {
         case PLUS : passt(lookahead); ergeb += term(); break;
         case MINUS: passt(lookahead); ergeb -= term(); break;
         default   : return(ergeb);
      }
   }
}

static int term(void)
{
   int erg = factor();
   while (TRUE) {
      switch(lookahead) {
         case MULT: passt(lookahead); erg *= factor(); break;
         case DIV : passt(lookahead); erg /= factor(); break;
         default  : return erg;
```

```
      }
   }
}

static int factor(void)
{
   int erg;
   switch (lookahead) {
      case ZAHL : erg = tokenwert; passt(ZAHL); return(erg);
      case MINUS: passt(MINUS);
                  switch (lookahead) {
                     case ZAHL : erg = tokenwert; passt(ZAHL);
                                 return(-erg);
                     default   : fehler(zeilennr);
                  }
      case AUF  : passt(AUF); erg=ausdruck(); passt(ZU); return(erg);
      default   : fehler(zeilennr);
   }
}

void parse(void)
{
    int ergeb;
    lookahead = lexan();
    while (lookahead != FERTIG) {
        ergeb = ausdruck();
        printf("%d\n%5d : ", ergeb, ++zeilennr);
        passt(ZEILENENDE);
    }
}
$
```

Zusammen mit dem folgenden C-Modul *rechner.c*, das die Routine *parse()* aufruft, und dem C-Modul *lexan.c* aus Kapitel 2.2.1, das die lexikalische Analyse durchführt, ist nun der Taschenrechner fertig. Es müssen nur noch die einzelnen Module übersetzt und gelinkt werden:

```
$ cat rechner.c ↵
extern int zeilennr;

int main(void)
{
   printf("%5d : ", zeilennr);
   parse();
   return(0);
}
$ cc -o rechner rechner.c lexan.c parser.c ↵
$ rechner ↵                [Testen des in C realisierten Rechners]
    1 : 2+3   *5 ↵
17
    2 : (2+3) *5 ↵
```

```
25
    3 : 4+-12*6+3 ↵
-65
    4 : Strg-D
$
```

Abbildung 2.5 zeigt nochmals die grundsätzliche Arbeitsweise des Programms *rechner*: Das Modul *lexan.c* zerlegt den Eingabetext in Token, die an das Modul *parser.c* weitergeleitet werden. Das Modul *parser.c* führt die Syntaxanalyse entsprechend der Reihenfolge der gelieferten Token durch und berechnet das Ergebnis des Ausdrucks, bevor es dieses ausgibt.

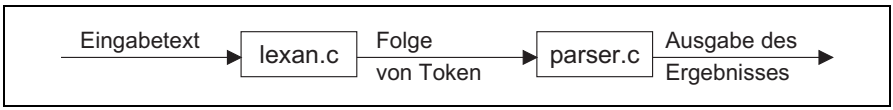

Abbildung 2.5: Arbeitsweise des mit C realisierten Taschenrechners

Syntaxanalyse für einen Taschenrechner mit yacc

Unter Verwendung von **yacc** kann die Syntaxanalyse für diesen Taschenrechner wesentlich einfacher und verständlicher formuliert werden, da im yacc-Programm nur die zuvor in BNF vorgestellten Produktionsregeln anzugeben sind. **yacc** formt diese Regeln dann in ein C-Programm *y.tab.c* um.

Das yacc-Programm, das die Syntaxanalyse für den Taschenrechner durchführt, wird hier in der Datei *parser.y* angegeben:

```
$ nl -ba parser.y ↵
 1  %{
 2  #define prompt   printf("\n%5d : ",++zeilennr)
 3  %}
 4
 5  %start  zeilen
 6  %token PLUS  MINUS  MULT  DIV  AUF  ZU  ZAHL  ZEILENENDE  NICHTS
 7
 8  %%
 9
10  zeilen :   /* leeres Symbol (Epsilon) */
11         |   ausdruck {printf("%d",$1); prompt;} ZEILENENDE zeilen
12         ;
13
14  ausdruck:   ausdruck  PLUS  term  { $$ = $1 + $3; }
15          |   ausdruck  MINUS  term  { $$ = $1 - $3; }
16          |   term  { $$ = $1; }
17          ;
18
19  term    :   term  MULT  factor { $$ = $1 * $3; }
20          |   term  DIV  factor { $$ = $1 / $3; }
```

```
21          |   factor  { $$ = $1; }
22          ;
23
24  factor  :   AUF  ausdruck  ZU  { $$ = $2; }
25          |   ZAHL  { $$ = tokenwert; }
26          |   MINUS  ZAHL { $$ = -tokenwert; }
27          ;
28  %%
29
30  #include "lex.yy.c"
31
32  void  yyerror(char *s)  /* Routine wird bei Fehler aufgerufen   */
33  {                       /* und gibt Fehlermeld. `s' aus, was    */
34     printf("%s\n", s);   /* die Meldung 'syntax error' ist.      */
35  }
$
```

An diesem einfachen Beispiel ist bereits der grundsätzliche Aufbau eines yacc-Programms erkennbar:

%{
C-Deklarationen und C-Definitionen, die für Syntaxanalyse benötigt werden.
%}
yacc-Deklarationen (**%start** .., **%token** .., ...)
%%
yacc-Regeln
%%
C-Programm

Alle in **%{** ... **%}** angegebenen C-Deklarationen und C-Definitionen werden direkt in das von **yacc** generierte C-Programm *y.tab.c* kopiert.

Die *yacc-Deklarationen* legen hier das Startsymbol und die Token fest. Dazu werden die yacc-Schlüsselwörter **%start** und **%token** verwendet.

Im Regelteil werden die *yacc-Regeln* in einer Form angegeben, die weitgehend der BNF entspricht. yacc-Regeln sind eine Menge von Produktionsangaben der folgenden Form:

lsymbol : $rsymbol_{11}$ {$aktion_{11}$} $rsymbol_{12}$ {$aktion_{12}$} ...
| $rsymbol_{21}$ {$aktion_{21}$} $rsymbol_{22}$ {$aktion_{22}$} ...
.....
;

Anders als bei der BNF trennt ein Doppelpunkt das Symbol der linken Seite von der rechten Seite. Unterschiedlich zur BNF ist auch, dass jede yacc-Regel mit Semikolon abzuschließen ist.

Im Unterschied zur BNF kann außerdem zu jedem *rsymbol* (Symbol auf der rechten Seite) eine Aktion angegeben werden. Die in {..} angegebene Aktion wird in Form von C-Anweisungen angegeben, die ausgeführt werden, wenn das links davon angegebene *rsymbol* abgedeckt wird. In den C-Anweisungen einer Aktion kann auf den Wert, den das erste *rsymbol* liefert, mit **$1**, auf den Wert, den das zweite *rsymbol* liefert, mit **$2** usw. zugegriffen werden. Soll eine Aktion einen Wert zurückgeben, muss in dieser Aktion der speziellen Variablen **$$** der entsprechende Wert zugewiesen werden.

Für den Taschenrechner sind hier 4 Regeln angegeben:

```
1. Regel:   zeilen  :  /* leeres Symbol (Epsilon) */
                    |  ausdruck { printf("%d",$1); prompt; } ZEILENENDE zeilen
                    ;
```

legt fest, dass mehrere Eingabezeilen zugelassen sind, wobei jede Eingabezeile einen *ausdruck* enthalten kann. Ist *ausdruck* vollständig abgearbeitet, wird der zugehörige Wert (**$1**) ausgegeben, bevor mit der Ausgabe der Zeilennummer zur nächsten Eingabe aufgefordert wird.
Es handelt sich hier um eine rekursive Definition, da das nichtterminale Symbol *zeilen* auf der rechten Seite wieder angegeben ist. Um eine endlose Rekursion zu vermeiden, ist für *zeilen* auch eine **ε**-Produktion angegeben. **ε**-Produktionen werden in den yacc-Regeln üblicherweise in Form eines Kommentars angegeben.

```
2. Regel:   ausdruck:  ausdruck PLUS term { $$ = $1 + $3; }
                    |  ausdruck MINUS term { $$ = $1 – $3; }
                    |  term { $$ = $1; }
                    ;
```

legt fest, dass *ausdruck* wieder ein *ausdruck* sein kann, nach dem ein Plus- bzw. Minus-Zeichen gefolgt von einem *term* vorkommt. Der Rückgabewert für diese beiden Produktionen ist dabei der von *term* gelieferte Wert (**$3**) addiert mit bzw. subtrahiert vom Wert, den *ausdruck* liefert (**$1**).
Um die hier vorhandene rekursive Definition zu einem Ende kommen zu lassen, wird festgelegt, dass *ausdruck* auch nur ein *term* sein kann. In diesem Fall ist der Rückgabewert der Wert von *term*.

```
3. Regel:   term  :  term MULT factor { $$ = $1 * $3; }
                  |  term DIV factor { $$ = $1 / $3; }
                  |  factor { $$ = $1; }
                  ;
```

legt fest, dass ein *term* erneut ein *term* sein kann, der mit einem *factor* multipliziert bzw. durch einen solchen *factor* dividiert wird. Der Rückgabewert für diese beiden Produktionen ist dabei der von *term* gelieferte Wert (**$1**) multipliziert mit bzw. dividiert durch den Wert, den *factor* liefert (**$3**).

Um die hier vorhandene rekursive Definition enden zu lassen, wird festgelegt, dass *term* auch nur ein *factor* sein kann. In diesem Fall ist der Rückgabewert der Wert von *factor*.

4. Regel: *factor* : *AUF ausdruck ZU { $$ = $2; }*
| *ZAHL { $$ = tokenwert; }*
| *MINUS ZAHL { $$ = -tokenwert; }*
;

legt fest, dass *factor* eine Zahl sein kann, vor der optional ein Minuszeichen angegeben ist. Der Rückgabewert ist hier die Zahl selbst bzw. deren Negativwert. Zusätzlich legt diese Regel fest, dass *factor* wieder ein geklammerter *ausdruck* sein kann. Der Rückgabewert ist in diesem Fall der von *ausdruck* gelieferte Wert: **{$$=$2;}**.

Schließlich kann in einem yacc-Programm ein *C-Programm* angegeben werden. Hier wird mit *#include "lex.yy.c"* festgelegt, dass später der C-Präprozessor an dieser Stelle das von **lex** generierte C-Programm einkopieren soll. Somit enthält das von **yacc** generierte C-Programm *y.tab.c* zugleich auch die lexikalische Analyse für den Taschenrechner.

Die lexikalische Analyse wird durch das folgende lex-Programm *lexer.l* durchgeführt, das bis auf das Fehlen von

#include "global.h"

identisch zu dem in Kapitel 2.2.2 verwendeten lex-Programm *lexan.l* ist. Mit dem Vorhandensein von **%token ...** im yacc-Programm sind die expliziten Token-Definitionen in *global.h* überflüssig geworden.

```
$ cat lexer.l ↵
%{
#include <stdlib.h>

int tokenwert = NICHTS; /* Programmglobale Variable, der im Falle   */
                        /* einer Zahl Zahlenwert zugewiesen wird.   */
int zeilennr = 1;  /* Programmglob. Variable:                      */
                   /* enthaelt immer Nummer der akt. Eingabezeile   */
%}
%%
[ \t]+     ;  /* Leer- und Tabzeichen ueberlesen */
\n         { return(ZEILENENDE);   }
[0-9]+     { tokenwert = strtol(yytext,NULL,10); return(ZAHL); }
"+"        { return(PLUS); }
"-"        { return(MINUS); }
"*"        { return(MULT); }
"/"        { return(DIV); }
"("        { return(AUF); }
")"        { return(ZU); }
```

```
.           { tokenwert = yytext[0]; return(NICHTS); }
%%
$
```

Nun muss zunächst **lex** aufgerufen werden, um aus *lexer.l* das C-Programm *lex.yy.c* generieren zu lassen. Danach wird **yacc** aufgerufen, um aus dem yacc-Programm *parser.y* das C-Programm *y.tab.c* generieren zu lassen.

```
$ lex lexer.l ↵
$ ls lex.yy.c ↵
lex.yy.c
$ yacc parser.y ↵
$ ls y.tab.c ↵
y.tab.c
$
```

Der Inhalt von *y.tab.c* ist hier wieder nicht von Interesse. Wesentlich ist, dass sich in *y.tab.c* eine Funktion *yyparse()* befindet, die die Syntaxanalyse durchführt. Dazu ruft diese Routine ständig die von **lex** generierte Routine *yylex()* auf, um die lexikalische Analyse durchführen zu lassen. Es fehlt jetzt nur noch ein Hauptmodul, das in der Funktion *main()* die Routine *yyparse()* aufruft. Das Hauptmodul wird hier *lexyacctest.c* genannt.

```
$ cat lexyacctest.c ↵
extern int zeilennr;

int main(void)
{
    printf("%5d : ", zeilennr);
    yyparse();
    return(0);
}
$
```

Nun muss noch **cc** aufgerufen werden, um die beiden Module *lexyacctest.c* und *y.tab.c* kompilieren und linken zu lassen. *lex.yy.c* muss bei diesem **cc**-Aufruf nicht angegeben werden, da dieses Modul durch

#include "lex.yy.c" (in *parser.y*)

automatisch mit kompiliert und gelinkt wird.

```
$ cc -o rechner2 lexyacctest.c y.tab.c -ly -ll ↵
$
```

Unter Linux ist der folgende Aufruf erforderlich:

```
$ cc -o rechner2 lexyacctest.c y.tab.c -lfl ↵
$
```

Die Angaben von **-ly** bzw. **-ll** bewirken, dass die aus der yacc-Bibliothek (*/usr/lib/liby.a*[1]) bzw. lex-Bibliothek (*/usr/lib/libl.a*) benötigten Funktionen und Definitionen mit eingebunden werden. Nun kann der mit **lex** und **yacc** realisierte Taschenrechner verwendet werden:

```
$ rechner2 ↵
    1 : 2+3   *5 ↵
17
    2 : (2+3) *5 ↵
25
    3 : 4+-12*6+3 ↵
-65
    4 : Strg-D
$
```

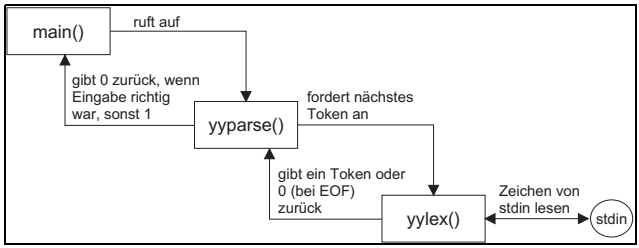

Abbildung 2.6: Die beiden automatisch generierten yylex() und yyparse() im Zusammenspiel

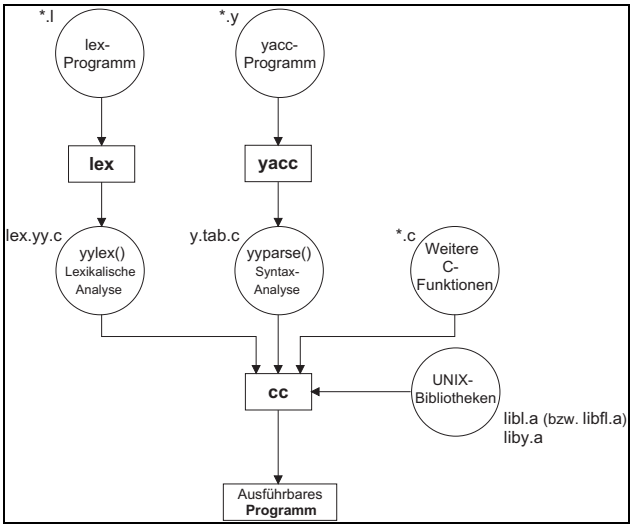

Abbildung 2.7: Verwendung von lex, yacc und cc

1. Existiert nicht unter Linux, weswegen man dort eigene Definitionen für die Funktionen *yyerror* und *main* angeben muss.

Während das Abbildung 2.6 das Zusammenspiel der beiden automatisch generierten C-Routinen *yylex()* und *yyparse()* zeigt, faßt die Abbildung 2.7 nochmals die wesentlichen Schritte zusammen, die bei der Verwendung von **lex** und **yacc** zu beachten sind.

Eine ausführliche Beschreibung zum Werkzeug **lex** befindet sich in den anschließenden Teilkapiteln, die Beschreibung zu **yacc** in Kapitel 4.