

# 9 Compilerbau-Werkzeuge

## 9.1 Überblick

In der Praxis des Compilerbaus werden heute vielfach Werkzeuge eingesetzt, die Routineaufgaben erleichtern sollen. In erster Linie ist dabei an die Punkte

- Generierung eines Scanners
- Generierung eines Parsers mit semantischen Aktionen

gedacht. Von besonderer Bedeutung sind heute wegen ihrer Verbreitung die Werkzeuge *Lex* und *Yacc*, die beim Betriebssystem UNIX als Utilities mitgeliefert werden.

*Lex* ist ein Scanner-Generator, der anhand von Spezifikationen der Symbole durch reguläre Ausdrücke eine Scanner-Funktion in C erzeugt.

*Yacc* ist ein Parser-Generator, der anhand einer EBNF-ähnlichen Spezifikation der Grammatik einen Parser in C erzeugt.

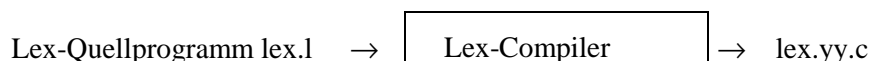
Diese beiden Tools sind jedoch nicht nur für UNIX/Linux und die Sprachen C/C++ erhältlich, sondern auch für andere Betriebssysteme wie Windows NT/2000/XP, OS/2, VMS und auch für andere Programmiersprachen wie z.B. Pascal und Java.

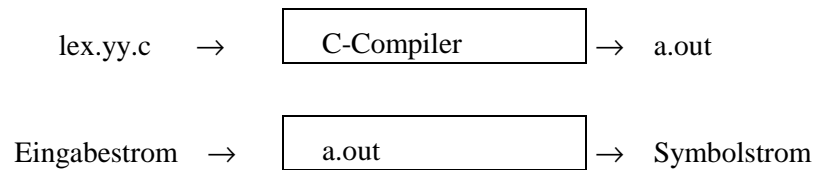
## 9.2 Der Scanner-Generator Lex

### Vorgehensweise

Hier wird gezeigt, wie *Lex* im allgemeinen Kontext von UNIX und C zu benutzen ist. Zunächst wird eine Spezifikation des Scanners in Form eines Programms *lex.l* erstellt, geschrieben in der Lex-Sprache. Dann wird mit Hilfe des *Lex-Compilers* ein C-Programm *lex.yy.c* daraus erzeugt. Das Programm *lex.yy.c* besteht aus einer Tabelle, die ein aus den regulären Ausdrücken von *lex.l* erstelltes Übergangdiagramm darstellt, zusammen mit einer Standardroutine, die diese Tabelle zum Erkennen von Lexemen benutzt. Die in *lex.l* an reguläre Ausdrücke gebundenen Aktionen sind C-Codestücke, die unverändert in *lex.yy.c* übernommen werden.

Zum Schluß wird *lex.yy.c* durch den C-Compiler in ein Objektprogramm übersetzt. Dieses stellt den Scanner dar, der einen Eingabestrom in eine Folge von Symbolen zerlegt.





## Lex-Spezifikationen

Ein Lex-Programm besteht aus drei Teilen:

```

Deklarationen
%%
Übersetzungsregeln
%%
Hilfsprozeduren

```

Der Deklarationsteil enthält Deklarationen von

- Variablen,
- symbolischen Konstanten,
- regulären Definitionen.

Eine symbolische Konstante ist ein Bezeichner, der als Repräsentant einer Konstante deklariert ist. Reguläre Definitionen dienen zur Bildung regulärer Ausdrücke, die innerhalb der Übersetzungsregeln vorkommen.

Die Übersetzungsregeln sind Anweisungen der Form

```

p1    {Aktion1}
p2    {Aktion2}
...     ...
pn    {Aktionn}

```

Dabei ist jedes  $p_i$  ein regulärer Ausdruck und jede Aktion <sub>$i$</sub>  ein Programmstück, das der Scanner ausführen soll, wenn ein Lexem für das Muster  $p_i$  gefunden wurde.

Der dritte Abschnitt eines Lex-Programmes besteht aus allen zur Durchführung der Aktionen benötigten Hilfsprozeduren.

## Zusammenarbeit von Scanner und Parser

Ein mit Lex erzeugter Scanner arbeitet folgendermaßen mit einem Parser zusammen. Sobald der Parser aktiviert wurde, beginnt der Scanner Zeichen für Zeichen den Eingabestrom zu lesen, bis er das längste Präfix der Eingabe (z.B. ist *Ban* Präfix von *Banane*) gefunden hat, das auf einen der regulären Ausdrücke  $p_i$  paßt. Anschließend führt er Aktion <sub>$i$</sub>  durch. Normalerweise gibt er dabei die Kontrolle an den Parser zurück. Falls dies jedoch nicht der Fall ist, versucht der Scanner weitere Lexeme zu finden,

bis irgendeine Aktion die Kontrolle an den Parser zurückgibt. Die wiederholte Suche nach Lexemen bis zu einem expliziten Rücksprung ermöglicht dem Scanner eine unkomplizierte Verarbeitung von Leerzeichen und Kommentaren. Der Scanner gibt dem Parser als einzigen Wert das gefundene Symbol zurück. Über eine globale Variable `yylval` kann ein zusätzlicher Attributwert zur näheren Beschreibung des Lexems übergeben werden.

### Ein Beispiel

Gegeben seien folgende Symbolmuster in Form regulärer Ausdrücke:

Regulärer Ausdruck	Symbol	Attributwert
<b>ws</b>	-	-
<b>if</b>	if	-
<b>then</b>	then	-
<b>else</b>	else	-
<b>id</b>	id	Verweis auf Tabelleneintrag
<b>num</b>	num	Verweis auf Tabelleneintrag
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Dabei sind die von den Terminalen **if**, **then**, ..., **relop**, **id** und **num** erzeugten Stringmengen durch folgende reguläre Definitionen gegeben:

```

if      → if
then   → then
else   → else
relop  → < | <= | = | <> | > | >=
id     → letter ( letter | digit ) *
num    → digit+ ( .digit+ ) ? ( E ( + | - ) ? digit+ ) ?
letter → A | B | ... | Z | a | b | ... | z
digit  → 0 | 1 | ... | 9
delim → blank | tab | newline
ws    → delim+

```

Es folgt das zugehörige Lex-Programm:

```

%{
    /* Definition der symbolischen Konstanten
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
}%

/* Reguläre Definitionen */
delim    [\t\n]
ws       {delim}+
letter   [A-Za-z]
digit    [0-9]
id       {letter}({letter}|{digit})
number   {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

%%

{ws}     { /* keine Aktion, keiner Rückkehr */ }
if       { return(IF); }
then     { return(THEN); }
else     { return(ELSE); }
{id}     { yylval = install_id(); return(ID); }
{number} { yylval = install_num(); return(NUMBER); }
"<"     { yylval = LT; return(RELOP); }
"<="    { yylval = LE; return(RELOP); }
"="      { yylval = EQ; return(RELOP); }
"<>"    { yylval = NE; return(RELOP); }
">"     { yylval = GT; return(RELOP); }
">="    { yylval = GE; return(RELOP); }

%%

install_id()
{ /* Funktion zum Eintragen eines Lexems in die Symboltabelle.
  yytext zeigt auf das erste Zeichen des Lexems, yyleng gibt
  seine Länge an. Rückgabewert ist ein Verweis auf den
  Symboltabelleneintrag */
}

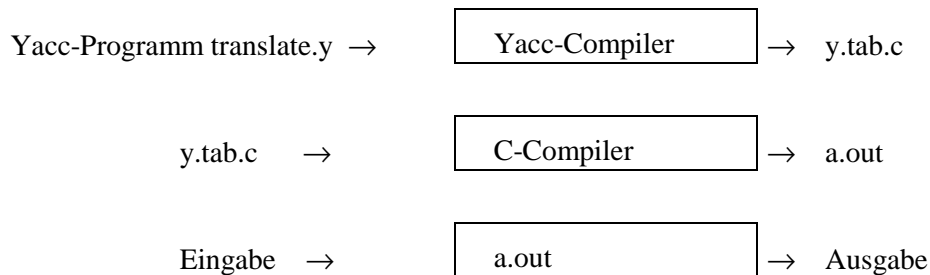
install_num()
{ /* Ähnliche Funktion zum Eintragen eines Lexems, das eine
  Zahl darstellt */
}

```

Im *Deklarationsteil* erkennen wir die Deklaration einiger symbolischer Konstanten, die in `%{` und `%}` eingeschlossen werden. Was zwischen diesen Klammern steht, wird unbesehen in `lex.yy.c` kopiert. Genauso ergeht es den Hilfsprozeduren im dritten Abschnitt, die hier durch `install_id` und `install_num` vertreten sind; sie werden ebenfalls nach `lex.yy.c` kopiert. Daneben enthält der Deklarationsteil noch einige reguläre Definitionen.

## 9.3 Der Parser-Generator Yacc

Yacc ist ein Parser-Generator für LR-Grammatiken, die gegenüber LL-Grammatiken gewisse Vorteile haben. Ein Compiler kann unter Verwendung von Yacc konstruiert werden, so wie folgendes Diagramm zeigt



Das UNIX-Kommando

```
yacc translate.y
```

überführt die Syntaxspezifikationen im Yacc-Programm `translate.y` in ein C-Programm `y.tab.c`. Dieses stellt einen LALR-Parser zusammen mit anderen C-Routinen dar, die vom Benutzer verwendet werden können. Mit Hilfe von

```
c y.tab.c -ly
```

wird `y.tab.c` übersetzt und mit der `ly`-Bibliothek, die das LR-Analyseprogramm enthält, zusammengebunden. Werden noch andere Routinen gebraucht, so können diese zusammen mit `y.tab.c` übersetzt werden.

## Yacc-Spezifikationen

Ein Yacc-Programm besteht aus drei Teilen:

```

Deklarationen
%%
Übersetzungsregeln
%%
Hilfsprozeduren
  
```

Dies ähnelt dem Aufbau von Lex-Programmen sehr.

### **Beispiel:**

Das folgende Yacc-Programm dient zur Konstruktion eines einfachen Tischrechners, der einen arithmetischen Ausdruck einliest, ihn auswertet und dann den entsprechenden Zahlenwert ausgibt. Es werden also schon semantische Aktionen in den Parser einbezogen. Die Grammatik dazu lautet

```

E → E + T | T
T → T * F | F
F → (E) | digit
  
```

Das Token (Symbol) digit ist eine Ziffer zwischen 0 und 9. Es folgt das Yacc-Programm:

```
%{
#include <ctype.h>
}%

%token DIGIT

%%
line      :  expr '\n'                { printf("%d\n", $1); }
          ;
expr      :  expr '+' term            { $$ = $1 + $3; }
          |  term
          ;
term      :  term '*' factor          { $$ = $1 * $3; }
          |  factor
          ;
factor    :  '(' expr ')'             { $$ = $2 }
          |  DIGIT
          ;

%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

Der *Deklarationsteil* enthält optional gewöhnliche C-Deklarationen, begrenzt durch `%{` und `%}`. Ferner können dort Deklarationen von Grammatikzeichen stehen, die im zweiten und dritten Teil benutzt werden.

Der *Übersetzungsregelteil* besteht aus Regeln. Jede Regel besteht aus einer Produktion der Grammatik in EBNF-ähnlicher Form und der assoziierten semantischen Aktion. Die Menge der Produktionen

$$\langle \text{linke Seite} \rangle \rightarrow \langle \text{alt 1} \rangle | \langle \text{alt 2} \rangle | \dots | \langle \text{alt n} \rangle$$

wird in Yacc als

```
< linke Seite > :      < alt 1 >          { semant. Aktion 1 }
                  |      < alt 2 >          { semant. Aktion 2 }
                  ...
                  |      < alt n >          { semant. Aktion n }
                  ;
```

geschrieben. Ein gequotetes einzelnes Token 'c' wird als Terminalsymbol c genommen. Ungequotete Strings, die nicht als Token deklariert sind, werden als Nichtterminale betrachtet. Die erste Produktion wird als Startsymbol genommen.

Die semantischen Aktionen werden in { und } eingeklammert. Sie sind Folgen von C-Anweisungen. Das Symbol \$\$ in einer semantischen Aktion verweist auf den Attributwert des auf der linken Seite stehenden Nichtterminals. Das Symbol \$i verweist auf den Wert des i-ten Grammatiksymbols auf der

rechten Seite der Produktion (Terminal oder Nichtterminal). Die semantische Aktion wird bei der Reduktion der zugehörigen Produktion ausgeführt.

Der *Hilfsprozedurteil* besteht aus optionalen C-Funktionen. Ein Scanner mit dem Namen `yylex` muß zur Verfügung stehen. Er kann entweder als C-Funktion selbst programmiert werden oder durch Lex erzeugt sein und dann z.B. durch ein `#include`-Statement eingebunden werden.

## 9.4 Problematik der Benutzung von Lex und Yacc

Es darf nicht der Eindruck entstehen, mit Tools wie Lex und Yacc könnte der Compilerbau automatisiert werden. Durch Eingabe einer Spezifikation der kontextfreien Grammatik der Sprachsyntax und durch Eingabe von regulären Ausdrücken für die lexikalische Analyse erhält man zwar den Quellcode des Gerüsts eines Compilers. Jedoch sind damit hauptsächlich nur Routineaufgaben erledigt und es fehlt noch sehr viel zum fertigen Compiler!

Die semantischen Aktionen des Parsers müssen ja in der Yacc-Sprache nach wie vor im C-Quellcode von Hand eingetragen werden. Dazu gehören

- Symboltabellenmanipulationen
- Semantische Prüfungen (Typprüfungen etc.)
- Zwischencode-Erzeugung

Gegenüber der Technik des rekursiven Abstiegs, wo verschiedene Verwaltungsaufgaben durch den Compiler erleichtert werden, müssen hier manche Stacks selbst verwaltet werden.

Weitere Punkte von Interesse, die nicht der Automatisierung zugänglich sind, sind die Optimierung, die Codeerzeugung und der Runtime-Support.