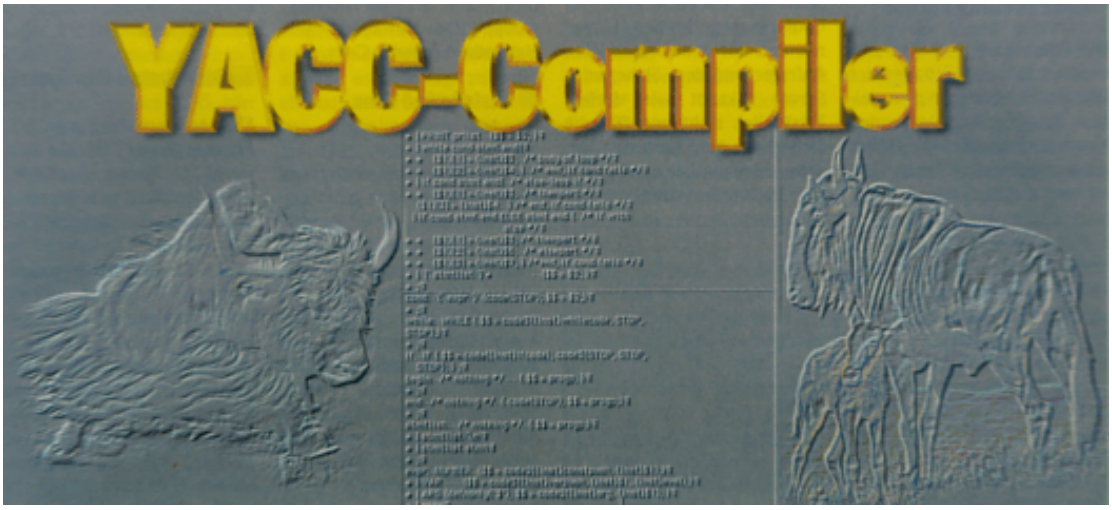

Lexikalische und syntaktische Analyse mit lex und yacc

Entwicklungshilfe

Oliver Müller • Die Leistungsfähigkeit einer Programmiersprache steht und fällt mit ihrer Grammatik. Somit auch mit dem Parser, der den Quelltext aufgrund dieser Grammatik analysiert. Wie man solche Parser mittels lex und yacc aufbaut, klärt dieser Beitrag.

In den frühen 70er Jahren herrschte eine wahre Popularitätswelle für sogenannte Compiler-Compiler. Als C. Johnson von den Bell Labs 1972 mit yacc einen weiteren Vertreter dieser Zunft veröffentlichte, trug er diesem Umstand mit der recht ironischen Abkürzung "yacc" (yet another compiler-compiler) Rechnung. Später folgte dann "lex" von M. E. Leck und E. Schmidt als Ergänzung zu yacc. Somit war ein perfektes Paar für das Parsing geboren: lex für die lexikalische und yacc für die syntaktische Analyse. Dabei gehen die Möglichkeiten von lex und yacc weit über den klassischen Compilerbau hinaus, was mit ein Grund sein mag, daß diese Veteranen der Computergeschichte immer noch aktuell sind.
Im folgenden sollen die Möglichkeiten von lex und yacc anhand von formint (formula interpreter) – einer Art mathematischer Programmiersprache oder Rechensystem – verdeutlicht werden.

■ Lexikalische Analyse

Jedem Quelltext liegt ein genau definiertes Alphabet zugrunde. Ein solches Alphabet besteht aus Buchstaben, Ziffern, Interpunktions-, Trenn-, Operator- und Layout-Zeichen (blank, EOL und EOF). Aus diesem Alphabet setzen sich die lexikalischen Symbole zusammen, die der Darstellung von Bezeichnern, Konstanten, Schlüsselwörtern etc. dienen. Die Regeln, weiche die Darstellung dieser Symbole definieren, heißen im Fachjargon Mikrosyntax. Diese können mittels automatentheoretischer Mechanismen einfach beschrieben werden. Für die Spezifikation eignen sich sogenannte reguläre Ausdrücke (regular expressions) und für deren Analyse endliche, nichtdeterministische Automaten. Diesen Automaten läßt sich nun unmittelbar eine Scanner-Routine entnehmen, die nur den Eingabequelltext mit den Automaten untersuchen muß. Für einen passenden Automaten wird während des Scan-Vorgangs jeweils ein sogenannter Token erzeugt. Token sind Paare (Typ, Wert), die Auskunft über den Typ und den Wert eines lexikalischen Symbols geben (zum Beispiel würde für die Konstante 1.45 ein Token der Form (NUMBER, 1.45) erzeugt). Nach diesem Prinzip arbeitet auch lex. Der Programmierer ist hierbei lediglich dafür verantwortlich, die Mikrosyntax in reguläre Ausdrücke zu fassen und das Token aufzubauen.

■ Aufbau eines lex-Programms

Ein lex-Programm gliedert sich in drei Teile:
- Definitionsteil
- Regelteil
- Benutzerdefinierte Routinen

Der Definitionsteil nimmt reguläre Definitionen (eine Art Makro für reguläre Ausdrücke) und sprachspezifische Definitionen auf. Reguläre Definitionen bestehen aus einem Namen, gefolgt von ein oder mehreren Leer- oder Tabulatorzeichen und einem regulären Ausdruck. Diese "Makros" können im Regelteil durch {Name} eingebunden werden und erleichtern so das Definieren von ähnlichen Regeln. Unter sprachspezifischen Definitionen versteht lex C- oder Ratfor-Deklarationen. Man unterscheidet zwei Arten von Eingaben: Zum einen können Definitionen von Variablen oder Funktionsprototypen direkt angegeben werden. Zu beachten ist allerdings, daß die erste Spalte der Zeile mit einem Leer- oder Tabulatorzeichen beginnen muß. Die andere Art ist die Kapselung des Codes in %{ und %}. Hierbei ist es gleichgültig, in welcher Spalte die Definitionen beginnen. Außerdem dürfen in einem solchen Block auch Präprozessordirektiven angegeben werden. Diese müssen allerdings aufgrund der Präprozessorsyntax in Spalte 1 beginnen. Das Herzstück eines jeden lex-Programms ist der Regelteil. Dieser wird mit %% vom Definitionsteil abgegrenzt. Diese Abgrenzung muß in jedem lex-File angegeben sein, selbst wenn kein Definitionsteil benötigt wird. Eine lex-Regel besteht im wesentlichen immer aus einem Pattern und den zugehörigen Aktionen. Das Pattern wird in lex ausschließlich als Regulärer Ausdruck interpretiert (siehe auch untenstehenden Textkasten). Dieser Reguläre Ausdruck muß in der ersten Spalte beginnen. Die Aktionen sind C-Statements geklammert in {...}
Am Ende jeder Aktion sollte, so fern es sich bei dem erkannten Muster um ein Token handelt, auch ein Token-Wert zurückgeliefert werden. Die Werte für die Token werden von yacc erzeugt und liegen im Falle von formint in der Datei y.tab.h beziehungsweise y_tab.h.

■ Interne Variablen von lex

Innerhalb dieser Aktionen stellt lex verschiedene Variablen bereit, die die Verarbeitung des lexikalischen Symbols ermöglichen. Die wohl wichtigste ist yytext. Diese Variable enthält den durch das Pattern abgedeckten Eingabestring. Eine weitere Variable ist yyleng, die die Stringlänge von yytext enthält. Diese wird zum Beispiel in der Regel für Stringkonstanten innerhalb von formint benutzt, um Speicher für das Symbol zu allokieren. formint führt eine Symboltabelle, in der sämtliche Variablen, Unterprogramme und Konstanten vom lexikalischen Analysator eingetragen werden.
Zwei weitere wichtige Variablen sind die Filestreams yyin und yyout. Diese beiden Variablen stellen die Eingabe- und Ausgabedatei dar. Standardmäßig sind diese mit stdin respektive stdout verbunden und müssen bei Bedarf mit Dateien verknüpft werden. In formint geschieht dies in der Funktion moreinput() in Module formint.y, welche jeweils eine über die Kommandozeile übergebene Datei mit yyin verbindet. yyout ist in formint immer noch mit stdout gleichgesetzt.
Im dritten Teil können nach dem %%-Trenner benutzerdefinierte Routinen angegeben werden. Der dritte Teil ist wie dieses zweite %% optional.
Die Funktion, welche den lexikalischen Analysator aufnimmt, nennt sich yylex() und wird von yacc aufgerufen. yylex() liefert einen Wert vom Typ int zurück. Dieser Wert entspricht bei positivem Vorzeichen einem Token (vergleichlich Rückgabewert bei Aktionen); bei Null oder kleiner ist das Dateiende erreicht. lex legt yylex() unter Unix in einer Datei namens lex.yy.c ab. Da dieser Dateiname unter DOS im FAT-System (altbekanntes 8+3) nicht möglich ist, nennt sich diese Datei zum Beispiel bei GNU flex unter DOS lexyy.c. Diese Datei muß zusammen mit den anderen Modulen vom C-Compiler compiliert werden.

Syntaktische Analyse

Der nächste Schritt auf dem Pfad in Richtung ablauffähiger Code ist die syntaktische Analyse. Die Grundlage hierfür bildet die Grammatik, mit deren Hilfe hierarchische Abhängigkeiten der syntaktischen Elemente spezifiziert werden. Zur Spezifikation einer Grammatik werden Regeln aufgebaut. Diese Regeln können in verschiedenen Arten dargestellt werden, etwa mit Hilfe von Syntaxdiagrammen oder der Backus-Naur-Form (BNF). Eine kontextfreie Grammatik für einen, die vier Grundrechenarten umfassenden Taschenrechner würde wie folgt definiert:

```
start → empty | expression

expression → term '+' expression | term '-' expression | term

term → factor '*' term | factor '/' term | factor

factor → NUMBER | '(' expression ')'
```

wobei S = start; empty, NUMBER, ´(´, ´)´, ´+´, ´-´, ´/´, ´*´ g T; start, expression, term, factor g N gilt. Das ”→” entspricht der Produktion und das ”|” einer Alternation (oder). empty ist ein spezielles Symbol, welches einem “leeren Wort” entspricht. Der Taschenrechner würde ohne empty immer ein expression erwarten, so arbeitet er auch mit leeren Zeilen zwischen den Anweisungen.
Aus diesem Beispiel ist zu ersehen, daß die Punkt-vor-Strich und die Klammern-Regel befolgt werden. Bevor zum Beispiel eine Addition erfolgen kann wird zuerst term ausgewertet und term kann eben eine solche Multiplikation und Division enthalten. Ebenfalls wird deutlich, daß auch rekursive Definitionen möglich sind. Um einer unterminierten Rekursion allerdings vorzubeugen, muß immer auch eine Produktion ohne Rekursion erfolgen. Ein Beispiel wäre “| factor” in der term-Produktion.

Das yacc-Programm

Auch der yacc-Quelltext gliedert sich ähnlich dem von lex in drei Teile:
- Definitionsteil
- Regelteil
- Benutzerdefinierte Funktionen

Im Definitionsteil können C-Definitionen analog zu lex in %{...%} geklammert werden. Mit der Anweisung %start wird das Startsymbol festgelegt. Falls kein Startsymbol angegeben ist, nimmt yacc automatisch das linke Symbol der ersten Regel als S an. Es empfiehlt sich jedoch, schon aus Lesbarkeitsgründen und der Übersichtlichkeit halber, das %start-Statement zu verwenden.
Durch %union wird der Datentyp der Kommunikationsvariable yylval festgelegt. Diese Variable enthält die Werte für ein Token (Typ, Wert). Der voreingestellte Typ für vylval ist int. Dieser Typ genügt jedoch in den allerwenigsten Fällen, da zum Beispiel Strings, Fließkommazahlen oder Bezeichner zurückgeliefert werden müssen. Zum Beispiel muß ein Token vom Typ STRING in formint einen Zeiger auf einen Speicherbereich mit dem Stringinhalt zurückgeben. Daher wird im lexikalischen Analysator der yylval-Komponente sym ein Zeiger auf eine Symbol-struct übergeben. Diese Symbol-struct ist in formint ein Datentyp, der Informationen über eine Funktion, Prozedur, Bezeichner oder Konstante aufnehmen kann und verweist immer auf ein Element der Symboltabelle.
Mit %token werden die terminalen Symbole definiert. Hier hat sich durchgesetzt, daß in yacc terminale Symbole in Großbuchstaben und nichtterminale in Kleinbuchstaben angegeben werden. Obwohl auch ein umgekehrtes Verfahren prinzipiell möglich wäre, da alle Symbolnamen lediglich den C-Bezeichner-Regeln genügen müssen, gilt es als schlechter Stil sich dieser „Quasinorm" nicht zu unterwerfen. Zwischen dem Schlüsselwort %token und dem oder den Tokenbezeichner(n) kann dem Token in <...> ein Komponententyp (Variablenbezeichner in yylval, %union) zugewiesen werden. So wird in formint zum Beispiel dem Token STRING die Komponente sym zugewiesen; dies bedeutet, daß STRING vom Typ Symbol ist.

■ Assoziative Operatoren in yacc

Mittels %left, %right und %nonassoc, kann den Token eine Assoziativität zugewiesen werden. Exemplarisch soll hier 60/6/2 Betrachtung finden. Die Auswertung hängt von der Assoziativität des Divisionsoperators / ab:

Linksassoziativität (Auswertung von links her) 60/6/2 = (60/6) / 2 = 5
Rechtsassoziativität (Auswertung von rechts her) 60/6/2 = 60 / (6/2) = 20

C wie auch formint definieren zum Beispiel den Divisonsoperator (und auch die anderen Grundrechenarten zuzüglich Modulo) als linksassoziativ. Ein rechtsassoziativer Operator wäre das für die Zuweisung definierte Gleichheitszeichen. a=b=c*d würde in C wie in formint ausgewertet wie a=(b=(c*d)).
Ein mit %nonassoc definiertes Token besitzt keinerlei Assoziativität. Ein Beispiel hierfür wäre der Vergleichsoperator < in Pascal. Vergleiche wie a<10 sind zulässig; im Gegensatz zu 2<a<10, welche gegen die Pascal-Syntax verstoßen.
Die Reihenfolge der %left-, %right- und %nonassoc-Angaben bestimmt deren Priorität, wobei die zuletzt geschriebene Zeile die höchste Priorität besitzt. Sehr gut zu sehen ist dies bei der Implementierung der Grundrechenarten in formint:
Zuerst findet sich die Angabe %left '+' '-' und danach %left '*' '/', woraus folgt, daß Punkt-vor-Strich gilt. Finden sich mehrere Token oder Literale (zum Beispiel '+') in einer Assoziativitätsangabe, so besitzen sie gleiche Priorität. In formint zum Beispiel '+' und '-'. Bei %left, %right und %nonassoc kann ebenfalls, wie bei %token in <...> eine Komponentendefinition erfolgen.
Um den Rückgabewert eines nichtterminalen Symbols auf einen bestimmten Type zu setzen beziehungsweise einer Komponente von yylval zuzuordnen existiert noch die %type-Angabe.
Der Regelteil – abgegrenzt durch %% – besteht im wesentlichen aus Produktionen, deren Syntax sich stark an die BNF anlehnt. Eine yacc-Regel hat den im Kasten “Aufbau eines yacc-Programms” beschriebenen Mustern zu folgen. lSymbol ist Element der nichtterminalen Symbole. rSymbol kann entweder ein terminales oder nichtterminales Symbol sein. Zu jedem rSymbol können Aktionen in {...} angegeben werden, sie sind jedoch optional. In den Aktionen kann auf die Werte der Symbole mit sogenannten Pseudovariablen zugegriffen werden. Mit $1 auf den Wert des ersten .rSymbol, $2 auf den des zweiten, und so weiter. Hier ist jedoch anzumerken, daß Aktionen bei dieser Angabe von Pseudovariablen ebenfalls mitgerechnet werden. Wenn eine Rückgabe an ein hierarchisch höheres Symbol erfolgen soll, so kann dies durch Übergabe an die Variable $$ geschehen. yacc expandiert beim Compilierungsvorgang diese $x-Ausdrücke zu den Typenangaben entsprechenden C-Variablen. Beispiel:
STRING ist in formint mittels %token als sym definiert worden. Wird jetzt auf ein rSymbol STRING zum Beispiel .mit $1 zugegriffen, so expandiert yacc dies zu yylval.sym in y.tab.c.
Ein besonderes Symbol ist error, es paßt auf jede fehlerhafte Quelltextzeile und ermöglicht dem Programmierer ein Errorhandling zu implementieren.
Der dritte Teil ist (wenn überhaupt) in gleicher Weise wie bei lex zu füllen.
Noch ein paar Worte zu formint. Die Syntax von formint kann diesem Quelltext und dem yacc-File entnommen werden. Zur Fehlerbeseitigung findet sich in run() (formint.y) ein setjmp, das es bei kleineren Fehlern formint ermöglicht, in einen definierten Anfangszustand zurückzukehren. Bei der Compilierung von formint.y wird yacc “16 shift/reduce conficts” melden. Dies ist in diesem Fall auf verschiedene Interpretationsmöglichkeiten zurückzuführen, in anderen Fällen könnte hier ein schwerer Parser-Fehler vorliegen.

Literatur

[1] *Helmut Herold*: lex und yacc – Lexikalische und syntaktische Analyse. Bonn, u.a.: Addison Wesley 1995
[2] *Brian Kernighan, Rob Pike*: Der UNIX-Werkzeugkasten, München, Wien: Carl Hanser 1987
[3] *Dietmar Nentwick, Christian Becker*: 1×1 des Compilerbaus, mc extra in DOS International 8/95
[4] *Volker Penner*: Konzepte und Praxis des Compilerbaus – Eine Einführung, Braunschweig, Wiesbaden: Vieweg 1994
[5] *Aho, Sethi, Ullmann*: Compilerbau. Bonn, u.a.: Addison Wesley 1988

Aufbau eines lex-Programms

Aufbau	Beispiel
Definitionen:	
Name Reguläre Definition	HexDigit [0-9a-fA-F]
C-Definition (1. Zeichen blank oder tab)	int lineno
%{	%{
C-Definitionen und Präprozessor-Direktiven	#include <stdlib.h> short var;
%}	%}
%%	
lex-Regeln:	
Regulärer Ausdruck Aktion	[\f\t\v] ;
oder	
Regulärer Ausdruck Forward	"exit" |
oder	
Regulärer Ausdruck {Aktionen}	\"([^"] | \\.)\" {string_zaehler++; printf ("%s", yytext);}
%%	
Benutzerdefinierte Routinen:	
C-Code	

Reguläre Ausdrücke von lex

Zeichen	Bedeutung	Beispiel
^	Zeilenanfang	^A deckt A am Zeilenanfang ab.
$	Zeilenende	A$ deckt A am Zeilenende ab.
.	entspricht jedem beliebigen Zeichen außer \n	
[...]	Klasse von Zeichen	[ABC] paßt auf Zeichen A, B oder C [0-9] paßt auf Zeichen 0..9 [+–] entspricht + oder –
[^...]	Komplementklasse von Zeichen	[^ABC] deckt jedes Zeichen außer A, B und C ab [^0-9] paßt auf jede Nichtziffer [^^] entspricht jedem Zeichen außer ^
r_1r_2	Konkatenation	AB deckt A unmittelbar gefolgt von B ab
$r_1\|r_2$	Alternation	der|die|das entspricht den Strings "der", "die" und "das" Turbo\(C\+\+|Pascal) deckt "Turbo C++" oder "Turbo Pascal" ab
(r)	Klammerung	
r*	deckt kein oder beliebig viele r ab	A* deckt "", "A", "AA", etc. ab
r+	deckt ein oder beliebig viele r ab	A+ deckt "A", "AA", "AAA" etc. ab
r?	deckt null oder ein r ab	A? deckt "" oder "A" ab
\z	Hebt Sonderbedeutung des Zeichens z auf oder stellt eine Escape-Sequenz z dar	\r entspricht Return * entspricht *
"..."	Stringbildung (Klammerung) mit Aufhebung der Sonderbedeutung der Zeichen (außer \) entspricht C-String	
r{m,n}	deckt beliebig viele Vorkommen von r zwischen m- und n-mal ab	A{10,20} entspricht einer Folge von 10 bis 20 A
{rd}	deckt reguläre Definition mit dem Namen rd ab	
r_1/r_2	Kontextsensitivität (Lookahead). Deckt r_1 nur dann ab, wenn r_2 folgt. In yytext befindet sich dann nur r_1!	IF/\(.*\)[a-zA-Z] deckt Schlüsselwort IF in Fortran ab (in Fortran ist es möglich, Schlüsselwörter als Variablenbezeichner zu (miß)gebrauchen)

Aufbau eines yacc-Programms

Definitionen:

```
%{
C-Definitionen und Präprozessor-Direktiven
%}
%union{
          Komponenten (Deklaration in C)
}
%start    startsymbol
%token    <Komponente>  TOKEN1 TOKEN2
oder
%token    TOKEN1 TOKEN2
%type     <Komponente>  nichtterminales Symbol
%right    <Komponente>  TOKEN1 TOKEN2
oder
%right    TOKEN1 TOKEN2
%left     <Komponente>  TOKEN1 TOKEN2
oder
%left     TOKEN1 TOKEN2
%nonassoc <Komponente>  TOKEN1 TOKEN2
oder
%nonassoc TOKEN1 TOKEN2
%%
```

yacc-Regeln:

lSymbol: $rSymbol_{10}$ $\{Aktion_{10}\}$ $rSymbol_{11}$ $\{Aktion_{11}\}$ ...
| $rSymbol_{20}$ $\{Aktion_{20}\}$ $rSymbol_{21}$ $\{Aktion_{21}\}$ ...
...
;

oder

lSymbol: /*epsilon*/
| $rSymbol_{10}$ $\{Aktion_{10}\}$ $rSymbol_{11}$ $\{Aktion_{11}\}$ ...
...
;

%%

Benutzerdefinierte Routinen:

C-Code

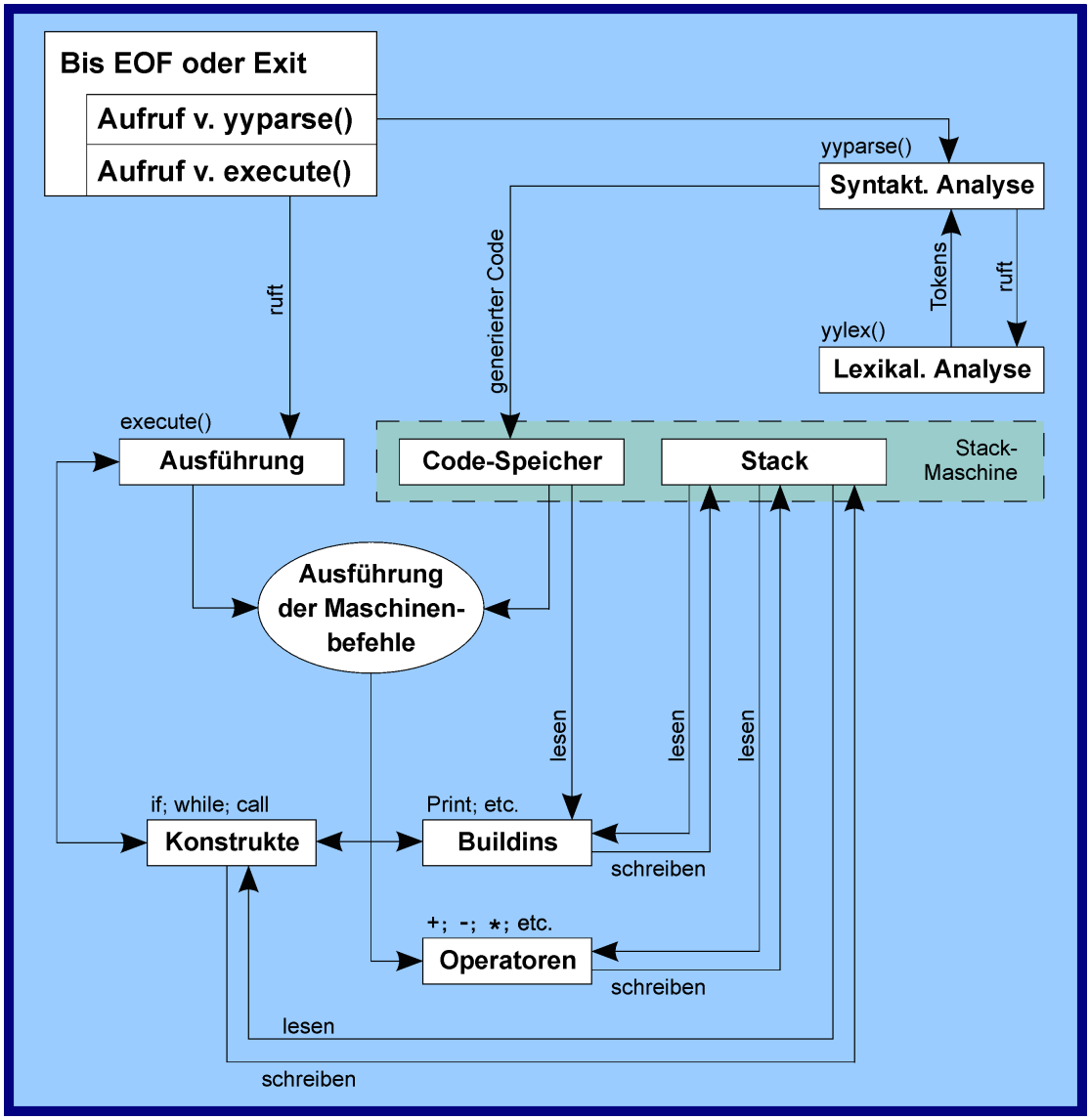

Bild 1. Die prinzipielle Funktionsweise von formint

Die Stackmaschine

formint ist als Interpreter konzipiert. Anstatt die Resultate der einzelnen Operationen gleich zu berechnen, generiert formint zuerst Code für einen Block und führt diesen anschließend aus. Ein solcher Block ist eine Produktion des Startsymbols, im Falle von formint sind dies die Produktionen von list: defn, asgn, stmt und expr (nicht error!). Die Stackmaschine ist ein einfaches Computermodell: Ein Operand wird jeweils auf dem Stack abgelegt (genauer gesagt, Code wird generiert, der den Operanden auf den Stack befördert). Für die Zuweisung y = 2 * x wurde beispielsweise folgender Code generiert werden:

constpush	Konstante auf den Stack bringen
2	... Wert 2
varpush	Zeiger auf Symbol auf den Stack bringen
x	... Variable x
eval	Evaluation: Zeiger durch Wert ersetzen
mul	oberste Werte multiplizieren; Produkt ersetzt beide
varpush	Zeiger auf Symbol auf den Stack bringen
y	... Variable y
assign	Wert in Variable speichern, Zeiger entfernen
pop	obersten Wert vom Stack entfernen
STOP	Ende der Befehlsfolge

Nach Abarbeitung dieser Befehlsfolge wird der arithmetische Ausdruck evaluiert und das Ergebnis in y gespeichert. Der zehnte Befehl - pop - entfernt den Resultatwert vom Stack, da dieser in diesem Statement nicht mehr benötigt wird. (Anders wäre dies bei einem Konstrukt ähnlich z = y = 2 * x)

Stackmaschinen ergeben im Normalfall einfache Interpreter. Die Maschine von formint bildet hier keine Ausnahme. Ein Array prog für die Maschinenbefehle, ein weiteres Feld stack für den Stapel und einige Zeiger auf relevante Stellen Innerhalb dieser Vektoren. In der Funktion run() aus Modul formint.y wird durch den Aufruf von yyparse() der Code erzeugt und in prog abgelegt. execute() führt diesen unter mehr oder minder starken Zugriff auf stack aus. Hier werden auch die beiden Phasen eines Interpreters deutlich – die Analyse und die Auswertung.

Ein drittes Array soll nicht verschwiegen werden: frame. Bei der Ausführung eines entsprechenden Unterprogramms wird ein Rahmen für diesen Aufruf generiert. Dieser Rahmen enthält Informationen über die übergeben Argumente, die Rücksprungadresse und so weiter.

Bild 1 stellt die Funktionsweise von formint grafisch dar und soll bei eigenen Experimenten mit formint hilfreich zur Seite stehen.

Listing 1 lex-Programm: formint.l

```
/* Program : formint
   File    : formint.l
   Language: lex
   Purpose : Definition of lexical analyser
   Author  : Oliver Mueller
*/
%{
#include <stdio.h>
#include <string.h>
#ifndef __TURBOC__
  #include <malloc.h>
#endif
#include "formint.h"
#if defined(__MSDOS__) || defined(MSDOS)
  #include "y_tab.h"
#else
  #include "y.tab.h"
#endif

extern int lineno;

void backslash (char *s);
%}

%%

[ \t\v\f]                 ;  /* ignore whitespaces */

"exit"                    return 0;
"proc"                    return PROC;
"func"                    return FUNC;
"return"                  return RETURN
"if"                      return IF;
"else"                    return ELSE;
"while"                   return WHILE;
"print"                   return PRINT;
"read"                    return READ;

[0-9]+\.?|[0-9]*\.[0-9]+  { double d;
                            sscanf (yytext, "%lf", &d);
                            yylval.sym = install ("",
                                                  NUMBER, d);
                            return NUMBER;
                          }

[_a-zA-Z][_a-zA-Z0-9]*    {
                            Symbol *s;
                            if ((s=lookup(yytext)) == 0)
                              s=install(yytext,UNDEF,0.0);
                            yylval.sym=s;
                            return (s->type == UNDEF)
                                        ? VAR : s->type;
                          }

">="                      return GE;
">"                       return GT;
"<="                      return LE:
"<"                       return LT;
"=="                      return EQ;
"!="                      return NE;
"!"                       return NOT;
"||"                      return OR;
"&&"                      return AND;
```

```
"$"[0-9]+                   { sscanf (yytext, "$%d",
                                 &yylval.narg); return ARG; }

\"(\\.|[^\\"])*\"           {
                              char *strptr=emalloc(yyleng);
                              strcpy(strptr, &yytext[1]);
                              strptr[strlen(strptr) - 1]='\0';
                              backslash(strptr);
                              yylval.sym=install(strptr,
                                                 STRING, 0);
                              free(strptr);
                              return STRING;
                            }

\n                          { lineno++; return '\n'; }

.                           return yytext[0];

%%
void backslash (char *s)
{
  int i=0, j=0;
  while (s[i] != '\0')
  {
    if (s[i] == '\\')
    {
      switch (s[++i])
      {
        case 'n' : s[j]='\n'; break;
        case 'r' : s[j]='\r'; break;
        case 'a' : s[j]='\a': break;
        case 'b' : s[j]='\b'; break;
        case 'v' : s[j]='\v'; break;
        case 'f' : s[j]='\f'; break;
        case 't' : s[j]='\t'; break;
        default  : s[j]=s[i];
      }
    }
    else s[j]=s[i];
    j++;
    i++;
  }
  s[j]='\0';
}
```

Listing 2 yacc-Programm: formint.y

```
/* Program : formint
   File    : formint.y
   Language: yacc
   Purpose : grammar definition
   Author  : Oliver Mueller
*/

%{
#ifdef __TURBOC__
  #include <alloc.h>
#else
  #include <malloc.h>
#endif
#include <string.h>
#include "formint.h"
#include "code.h"

#define code2(c1,c2) code(c1); code(c2)
#define code3(c1,c2,c3) code(c1); code(c2); code(c3)
int indef;

void yyerror (char *s);
int  run();
void warning (char *s; char *t);
int  moreinput();
int  execerror (char *str, char *str1);
int  defnonly (char *s);
int  follow (int expect, int ifyes, int ifon);
int  init();  /* defined in init.c */
%}
%union {
  Symbol *sym;  /* symbol table pointer */
  Inst *inst;   /* machine instructions */
  int narg;     /* number of arguments  */
}

%start list

%token <sym> NUMBER STRING PRINT VAR BLTIN UNDEF WHILE IF ELSE
%token <sym> FUNCTION PROCEDURE RETURN FUNC PROC READ
%token <narg> ARG
%type <inst> stmt asgn expr stmtlist cond while if end prlist begin
%type <sym> procname
%type <narg> arglist
%right '='
%left OR
%left AND
%left GT GE LT LE EQ NE
%left '+' '-'
%left '*' '/'
%left UNARYMINUS NOT
%right '^'

%%

list: /* epsilon */
  | list '\n'
  | list defn '\n'
  | list asgn '\n' {code2((Inst)pop,STOP); return 1;}
  | list stmt '\n' {code(STOP); return 1;}
  | list expr '\n' {code2((Inst)print, STOP); return 1;}
  | list error '\n' {yyerror("at list\n");}
  ;
```

```
asgn: VAR '=' expr {code3((Inst)varpush,(Inst)$1,(Inst)assign); $$=$3;}
  | ARG '=' expr {defnonly("$");code2((Inst)argassign,(Inst)$1); $$=$3;}
  ;
stmt:  expr  {code((Inst)pop);}
  | RETURN {defnonly("return"); code((Inst)procret);}
  | RETURN expr {defnonly("return"); $$ = $2; code((Inst)funcret); }
  | PROCEDURE begin '(' arglist ')' {$$ = $2; code3((Inst)call,
                                          (Inst)$1, (Inst)$4); }
  | PRINT prlist   {$$ = $2; }
  | while cond stmt end{
      ($1)[1] = (Inst)$3;            /* body of loop */
      ($1)[2] = (Inst)$4; }          /* end, if cond fails */
  | if cond stmt end{                /* else-less if */
      ($1)[1] = (Inst)$3;            /* thenpart */
      ($1)[3] = (Inst)$4; }          /* end, if cond fails */
  | if cond stmt end ELSE stmt end{  /* if with else */
      ($1)[1] = (Inst)$3;            /* thenpart */
      ($1)[2] = (Inst)$6;            /* elsepart */
      ($1)[3] = (Inst)$7; }          /* end, if cond fails */
  | '{' stmtlist '}'      {$$ = $2; }
  ;
cond:  '(' expr ')' {code(STOP); $$ = $2;}
  ;
while:  WHILE { $$ = code3((Inst)whilecode, STOP, STOP);}
  ;
if:  IF { $$ = code((Inst)ifcode); code3(STOP, STOP, STOP); }
  ;
begin:     /* nothing */  {$$ = progp; }
  ;
end:       /* nothing */  { code(STOP); $$ = progp;}
  ;
stmtlist:  /* nothing */  { $$ = progp;}
  | stmtlist '\n'
  | stmtlist stmt
  ;
expr:  NUMBER {$$ = code2((Inst)constpush, (Inst)$1);}
  | VAR       {$$ = code3((Inst)varpush, (Inst)$1, (Inst)eval);}
  | ARG {defnonly("$"); $$ = code2((Inst)arg, (Inst)$1); }
  | asgn
  | FUNCTION begin '(' arglist ')' {$$ = $2;
                           code3((Inst)call,(Inst)$1,(Inst)$4); }
  | READ '(' VAR ')'   {$$ = code2((Inst)varread, (Inst)$3);}
  | BLTIN '(' expr ')' {$$ = $3; code2((Inst)bltin, (Inst)$1->u.ptr);}
  | '(' expr ')' {$$ = $2;}
  | expr '+' expr  {code((Inst)add);}
  | expr '-' expr  {code((Inst)sub);}
  | expr '*' expr  {code((Inst)mul);}
  | expr '/' expr  {code((Inst)div);}
  | expr '^' expr  {code((Inst)power);}
  | '-' expr %prec UNARYMINUS   { $$=$2; code((Inst)negate);}
  | expr GT  expr {code((Inst)gt);}
  | expr GE  expr {code((Inst)ge);}
  | expr LT  expr {code((Inst)lt);}
  | expr LE  expr {code((Inst)le);}
  | expr EQ  expr {code((Inst)eq);}
  | expr NE  expr {code((Inst)ne);}
  | expr AND expr {code((Inst)and);}
  | expr OR  expr {code((Inst)or);}
  | NOT expr {$$ = $2; code((Inst)not); }
  ;
string:  STRING    { code2((Inst)varpush,(Inst)$1); }
```

```
  ;
prlist: expr  {code((Inst)prexpr);}
  | string    {code((Inst)prstr);}
  | prlist ',' expr    {code((Inst)prexpr);}
  | prlist ',' string  {code((Inst)prstr);}
  ;
defn:  FUNC procname { $2->type=FUNCTION; indef=1;}
    '(' ')' stmt   {code((Inst)procret); define($2); indef=0;}
  | PROC procname { $2->type=PROCEDURE; indef=1; }
    '(' ')' stmt {code((Inst)procret); define($2); indef=0; }
procname:  VAR
  | FUNCTION
  | PROCEDURE
arglist: /* nothing */  { $$ = 0; }
  | expr                { $$ = 1; }
  | arglist ',' expr    { $$ = $1 + 1; }
  ;
%%

#include <ctype.h>
#include <signal.h>
#include <setjmp.h>

char *progname;
int lineno = 1;
char *infile;       /* input file same */
extern FILE *yyin;  /* input file pointer */
char **gargv;       /* global argument list */
int gargc;
int c;              /* global for use by warning() */

jmp_buf begin;

int follow(int expect, int ifyes, int ifno)
{
  int c=getc(yyin);

  if (c == expect)
    return ifyes;
  ungetc(c, yyin);
  return ifno;
}

int defnonly(char *s) /* warn if illegal definition */
{
  if (!indef)
    execerror(s, "used outside definition");
}

void yyerror(char *s)
{
  puts(s);
  exit(1);
}

int execerror(char *str, char *str1)
{
  warning (str, str1);
  fseek (yyin, 0L, 2);   /* flush rest of file */
  longjmp (begin, 0);
}
```

```
int main(int argc, char *argv[])
{
  int i;
  progname = argv[0]
  if (argc = 1)   /* fake an argument list */
  {
    static char *stdinonly[] = {"-"};
    gargv = stdinonly;
    gargc = 1;
  }
  else
  {
    gargv = argv + 1;
    gargc = argc - 1;
  }
  init();
  while (moreinput())
    run();
  return 0;
}

int moreinput()
{
  if (gargc-- <= 0)
    return 0;
  if (yyin && yyin != stdin)
    fclose(yyin);
  infile = *gargv++;
  lineno = 1;
  if (strcmp(infile, "-") = 0)
  {
    yyin = stdin;
    infile = 0;
  }
  else if ((yyin=fopen(infile, "r")) == NULL)
  {
    fprintf (stderr, "can't open %s\n", infile);
    return moreinput();
  }
  return 1;
}

int run() /* execute until EOF */
{
  setjmp(begin);
  for (initcode(); yyparse(); initcode())
    execute(progbase);
}

void warning(char *u,char *t)
{
  fprintf(stderr, "%s: %s", progname, s);
  if (t)
    fprintf(stderr, " %s", t);
  if (infile)
    fprintf (strerr, " is %s", infile);
  fprintf (stderr, " near line %d\n", lineno);
}
```