Compiler-Baukasten

Wie man mit Compiler-Compilern arbeitet

Werkzeuge zur Compiler-Generierung gibt es schon seit einiger Zeit auf UNIX Systemen. Mit solch einem Compiler-Compiler lassen sich nicht nur Compiler für Programmiersprachen herstellen. Diese Werkzeuge können auch zum Entwurf einer Eingabesprache als Benutzerschnittstelle verwendet werden. Man kann damit Interpreter oder sogar Assembler erzeugen. Grund genug, Compiler-Compiler genauer zu beleuchten.

Für MS-DOS gibt es heute interessante Implementationen von Compiler-Compilern. Sie tragen tierische Namen, zum Beispiel BISON, der GNU-C erzeugt, oder PCYACC, der mit umfangreichen Sprachbeispielen ebenfalls für den PC verfügbar ist. BISON ist wie auch GNU-C ein Public-Domain-Programm. BISON und YACC erzeugen ein C-Programm, das man später noch mit einem normalen C-Compiler in den Code der Zielmaschine übersetzen muß.
Die Programme BISON und YACC erzeugen genaugenommen gar nicht den ganzen Compiler, sondern einen speziellen Teil des Compilers, den sogenannten Parser. Scanner und den eigentlichen Codegenerator muß man selbst programmieren. So wurde zum Beispiel der Parser des GNU-C-Compilers mit Hilfe von BISON erzeugt und gestattet es damit relativ leicht, Änderungen an der Sprachsemantik durchzuführen. Ein kleines Schema in *Bild 1* zeigt die Arbeitsstufen eines Compilers auf. Der Scanner hat die Aufgabe, den Quelltext in sogenannte Token zu zerlegen. Der Scanner muß zum Beispiel Datenwerte, Strings, Kommentare (die gleich entfernt werden), Namen und Schlüsselwörter aus der Menge der Grundsymbole erkennen. Wenn er eine gelesene Zeichenfolge nicht als Datum, String oder Kommentar erkannt hat, wird in der Liste aller reservierten Wörter nach Schlüsselwörtern (also IF, THEN, BEGIN etc...) gesucht und dann das zugehörige Token zugeordnet.
Die anderen Elemente markiert er so, daß Speicherplatz reserviert und bei Konstanten auch gefüllt werden kann.
So ein Scanner ist vom Aufbau nicht allzu schwierig zu schreiben. Er wird per Funktion vom Parser aufgerufen. Zu YACC gibt es sogar ein Hilfsprogramm Namens LEX zur Erzeugung eines Scanners. Der erzeugte Code ist jedoch meist nicht sehr effizient.
Ein Parser bekommt die einzelnen Token vom Scanner und generiert daraus einen Parserbaum, der sich dann vom Codegenerator in den Maschinencode umsetzen läßt. Der Compiler-Compiler wird zur Erzeugung des Parsers mit der syntaktischen Beschreibung der Sprache gefüttert, die der Parser erkennen soll. Dazu muß ihm als erstes die Menge der Grundsymbole der neuen Sprache mitgeteilt werden. Dann müssen die Grammatikregeln aufgestellt werden. Dabei werden in unserem Beispiel die Token zur Unterscheidung von den in den Regeln neu definierten syntaktischen Variablen mit Großbuchstaben geschrieben oder (bei Operatorensymbolen) in Hochkommata eingefaßt. Ein Beispiel für eine Regel:

```
Ausdruck: ZAHL
        | Ausdruck '+' Ausdruck
        | Ausdruck '-' Ausdruck
        | '-' Ausdruck
        | Ausdruck '*' Ausdruck
        | Ausdruck '/' Ausdruck
        | '(' Ausdruck ')'
        ;
```

Hier wird die Syntax für arithmetische Ausdrücke beschrieben. In diesem Beispiel fehlt jedoch ein wichtiger Aspekt: Es wurden keine Prioritäten für die Operatoren +, *, −, / festgelegt. Deshalb ist hier nicht definiert, wie der Ausdruck 2 + 4 * 5 zu berechnen ist.

Für die Festlegung von Prioritäten gibt es für YACC eine eigene Anweisung:

```
%left '+' '-'
%left '*' '/'
%left VZMINUS
```

Das Beispiel sagt: Die Token in einer Zeile sollen im Zielcompiler von links nach rechts ausgewertet werden, die in späteren Zeilen angegebenen Token werden mit höherer Priorität als die in den vorhergehenden versehen. Zusammen mit der Grammatik oben gilt Punkt vor Strich. Der Ausdruck VZMINUS wird für das Vorzeichen benötigt, wofür die Definition von Ausdruck abgeändert wird:

```
| '-' Ausdruck %prec VZMINUS
```

Damit erhält das einfache Vorzeichen die höchste Priorität.

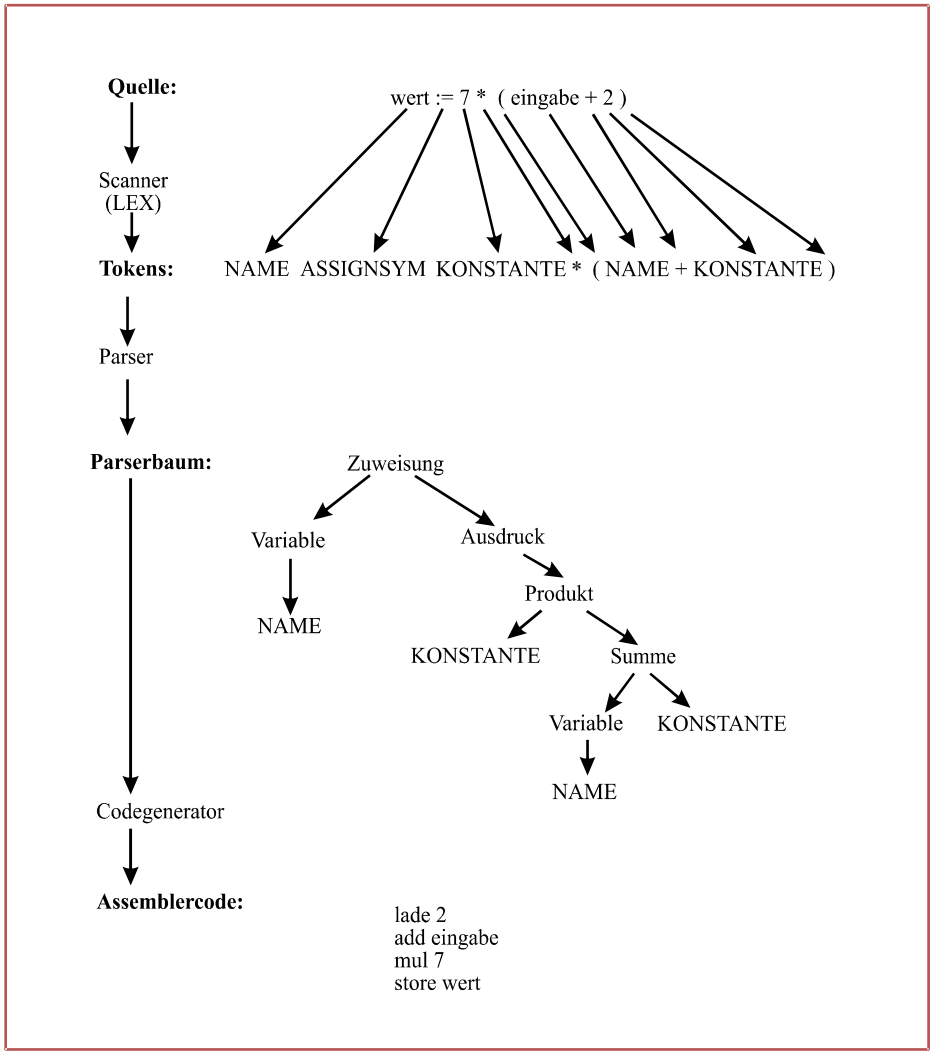

Bild 1. Diese Schritte führt ein Compiler beim Übersetzen aus

Bei einem Ausdruck wie -4 + 3 * 4 sorgt der erzeugte Compiler jetzt dafür, daß zuerst -4 berechnet wird, dann 3 * 4 und zuletzt die Summe -4 + 12, wenn der Compiler-Compiler die oben genannten Anweisungen und Regeln bekommen hat.
Um diesen Ausdruck auswerten zu können, fehlt jetzt noch der eigentliche Zielcode. Man kann zwischen den Definitionsteilen des zu erzeugenden Compilers bei YACC richtige C-Programmteile in geschweiften Klammern einsetzen, die die semantischen Aktionen des Zielcompilers für die vorhergehende Zeile beschreiben. Dabei ist

vereinbart, daß Ausdrücke oder Token über die Symbole $1 .. $n Werte übernehmen können und daß sie mit $$ einen Wert als Ergebnis liefern. Wieder ein Beispiel:

```
%token <integer> ZAHL
%type <integer> Ausdruck

Ausdruck: ZAHL
            { $$ = $1; }
        | Ausdruck '+' Ausdruck
            { $$ = $1 + $3; }
        | Ausdruck '-' Ausdruck
            { $$ = $1 -$3; }
        | '-' Ausdruck
            { $$ = -$1; }
        | Ausdruck '*' Ausdruck
            { $$ = $1 * $3; }
        | Ausdruck '/' Ausdruck
            { $$ = $1 / $3; }
        | '(' Ausdruck ')'
            { $$ = $2; }
        ;
```

Die ersten beiden Definitionen %token und %type legen fest, welcher Ergebnistyp für den $-Zugriff geliefert werden soll. Ein Ausdruck in der hier definierten Syntax soll also immer Werte vom Typ integer liefern und gegebenenfalls einen oder zwei Werte vom Typ integer übernehmen. Entweder wird der von Ausdruck übernommene Wert unverändert unter ZAHL oder (Ausdruck) zur Weitergabe bereitgehalten oder die beiden Operanden $1 und $3 werden addiert (subtrahiert, multipliziert oder dividiert) und dann das Ergebnis bereitgehalten. Die $-Platzhalter in den kurzen C-Sequenzen (die nur aus „=“, „+“ , ... bestehen) sind die formalen Parameter auf Compiler-Compiler-Ebene, die der erzeugte Compiler mit den aktuellen Bezügen ausfüllen muß.
Aus der Syntax oben läßt sich im Prinzip das Arbeiten mit einem kleinen Taschenrechner ableiten, bei dem der Benutzer eine Formel eingibt und dann sofort ein Ergebnis bekommt. Der so generierte Compiler würde also aus einer Taschenrechner-Eingabe-Sprache den Code „Ergebniszahl“ erzeugen.
Den mit YACC erzeugten Parser kann man von einem C-Hauptprogramm aus mit dem Aufruf yyparser() aufrufen. Er ruft seinerseits das Unterprogramm yylex() auf. In yylex, muß man die Eingabe, das Programm in der Sprache des Zielcompilers, analysieren und entsprechende Token erzeugen. Jetzt ein komplettes Beispiel, das als *Listing 1* abgedruckt ist. Es stellt einen Minicompiler dar, mit dem man Zuweisungen mit konstanten Ausdrücken an Variable durchführen kann. Als Code wird Assembler für den i860 erzeugt. Beachten Sie, daß man in der Grammatik-Definition für YACC noch eine Union-Datenstruktur vorgeben kann, in die man Typendeklarationen einbringen kann. Die Elemente dieser Union-Definition werden dann bei %token und %type zitiert, um den Ergebnistyp festzulegen.
Die ersten Zeilen, die mit %{ und %} eingeschlossen sind, geben Standard-Include-Dateien an, die später im C-Programm, das bereits der Zielcompiler ist, direkt übernommen werden, die der Compiler-Compiler also nur durchreicht. Danach kommt die Definition der Union-Struktur, die alle geplanten Datentypen von Token und Definitionen festlegen muß. Mit %start programm wird dem Compiler-Compiler gesagt, wo es ernst wird. Bei %% beginnen die eigentlichen Definitionen.
Die Definition für Programm ist in *Listing 1* nur trivial ausgeführt. Ein Programm besteht nur aus Anweisungen, „anweisungen“ sind dann wieder definiert und können zum einen eine „Zuweisung“ gefolgt von einem Strichpunkt sein, oder „anweisungen“ gefolgt von „Zuweisung“ gefolgt von einem Strichpunkt. Durch diese rekursive Definition ist es möglich, in der so definierten Sprache eine beliebige Anzahl von Zuweisungen zu einem Programm zu kombinieren. Sie müssen jeweils durch einen Strichpunkt getrennt sein.
Zuweisung kann nun ein IDENTIFIER sein (also ein Variablenname), dem ein Gleichheitszeichen folgt, dem wiederum ein „constausdruck“ folgt. Wenn dies der Fall ist, so wird der in Klammern stehende nachfolgende C-Ausdruck in das Compilat eingefügt und mit den aktuellen Bezügen ergänzt. Hier handelt es sich um ein Codegeneratorstück. Es erzeugt die Sequenz „mov wert,r16" und „st.l r16, variable“ – was Originalton i860-Assembler ist. Wobei für Wert der errechnet Wert von „constausdruck“ ($3) verwendet wird und als Name der String der zu IDENTIFIER gehört und als $1 zur Verfügung steht. Das ist der Grund, weshalb IDENTIFIER mit dem %token-<ch>-Befehl definiert wurde, „ch“ ist dabei ein Zeiger auf eine Zeichenfolge, „constausdruck“ ähnelt dem

Beispiel zur Bearbeitung arithmetischer Ausdrücke. Die Codeerzeugung beschränkt sich auf die Umrechnung von zurückgelieferten Werten.
Nach %% endet die Definitionsphase für den Compiler-Compiler. Der nachfolgende C-Code wird direkt in den generierten Compiler übernommen. Hier ist eine Routine getit(), die immer ein Zeichen nach dem anderen liefert. Um einfache Tests durchführen zu können, wird hier eine Benutzereingabe für jede einzulesende Zeile angefordert. Die so eingegebenen Zeichen werden bei jedem Aufruf einzeln geliefert, bis die Eingabezeile beendet ist und eine neue Zeile angefordert wird. Der Programmteil yylex() wird von dem erzeugten Parser aufgerufen – er muß also anwesend sein. Von ihm werden die einzelnen Token geliefert.
Weil Leerzeichen von unserem Programm überlesen werden sollen, werden am Anfang des Programms yylex() zunächst Leerzeichen aufgespürt und solange von der Eingabe überlesen, bis ein Zeichen ungleich einem Leerzeichen auftritt. Falls EOF gefunden wird, wird dies als Ergebnis weitergereicht, um dem Parser das Ende des Eingabestromes mitzuteilen.
Danach wird geklärt, ob eine Zahl eingegeben wurde. Wenn ja, wird das Token NUMBER zurückgeliefert und der Zahlenwert in yylval.in gespeichert, yylval ist eine im Parser definierte Datenstruktur, auf die man im Scanner zurückgreifen darf. Bei einer alphanumerischen Zeichenfolge wird IDENTIFIER als Ergebnis geliefert und die Zeichenfolge in einem Puffer bereitgehalten. Der Puffer wird dazu zuvor mit malloc angelegt. Ein Pointer auf diesen Puffer wird in yylval.ch an den Parser übergeben und steht damit im Parserteil zur Verfügung. Bei einem sonstigen Zeichen wird einfach der Wert dieses Zeichens als Ergebnis von yylex geliefert.
Im Hauptprogramm main() wird zunächst die Variable nextch mit dem ersten von Space verschiedenen Zeichen der Eingabe vorbelegt. Dann wird yyparse() aufgerufen, also der eigentliche Compiler. Das Unterprogramm yyerror() wird vom Parser im Fehlerfall verwendet. Es wird mit SYNTAX ERROR als Parameter aufgerufen, wenn der Parser feststellt, daß keine passende Definition gefunden werden kann. Es ist auch möglich, im Parser selbst Fehlermeldungen auszugeben, um so dem Benutzer Hinweise auf die Fehlerursache zu geben. In yyerror sollte man dann auch die fehlerhafte Eingabezeile mit einer Zeilennummer ausgeben, um so die Fehlersuche zu erleichtern.
In unserem einfachen Beispiel mit der direkten Eingabe über die Konsole ist das nicht notwendig.
Ein einfacher Test für den Minicompiler lautet im Protokoll:

```
Eingabe: ergebnis = 3 * 4 + 5 * ( 19 - 234 );
```

erzeugt den Assemblercode:

```
mov -1063,r16
st.l r16,ergebnis

Eingabe:
```

Bei Eingabe eines Returns wird das Programm beendet, da dann der Parser ein EOF erhält.
Listing 2 zeigt eine Headerdatei, die man sich erzeugen lassen sollte, wenn man zum Beispiel yylex in anderen C-Modulen unterbringt und auf die TOKEN-Definitionen zugreifen will. *Listing 3* zeigt eine Make-Datei für die Erzeugung unseres Minicompilers, wobei hier PCYACC als Parsergenerator und MSC 6.0 als C-Compiler verwendet wurden. Abschließend zeigen wir noch einen Ausschnitt aus der Parserdefinition von GNU-C *Listing 4*. Dort wird ebenfalls C-Code zwischen die Definitionen geschrieben. Der C-Code wird immer genau nach Erkennung der davor stehenden Definitionen in den Zielprogrammtext eingefügt.
Statt PCYACC kann auch BISON verwendet werden. BISON wird als Public Domain Produkt geliefert. Compiler, die mit BISON generiert wurden, dürfen jedoch auch nur wieder als Public Domain verwendet werden, da Codeteile von BISON in den Compiler übertragen werden. BISON wird komplett mit Quelle geliefert, so daß auch Anpassungen möglich sind. Bei PCYACC bekommt man umfangreiche Beispiele mitgeliefert. Da gibt es Parser für Fortran 77, Ansi-C, Pascal, PROLOG, PostScript, dBase und in der Profiversion sogar ein kompletter C++ nach C Compiler – mit Codegenerator und allen Quellen. Programme, die man damit erzeugt hat, darf man beliebig vermarkten.

PCYACC wird von der Firma ABRAXAS SOFTWARE INC, USA, hergestellt.
Telefon: 001-(503)-244-5253.
BISON kann komplett aus der mc Mailbox abgerufen werden.

Rolf-Dieter Klein/ro

Listing 1: Sprachdefinition des Mini-Compilers für PCYACC

```
/* Kleines Beispiel  fuer Mini-Compiler mit PCYACC geschrieben */
%{
  #include <stdio.h>
  #include <stdlib.h>
%}
%union {
  long in;
  char *ch;
}
%start programm
%left '+' '-'
%left '*' '/'
%left UNARYMINUS

%token <ch> IDENTIFIER
%token <in> NUMBER
%type <in> constausdruck

%%

programm:
                anweisungen
                ;
anweisungen:
                zuweisung ';'
                | anweisungen zuweisung ';'
                ;
zuweisung:
                IDENTIFIER '=' constausdruck
                  { printf("mov %d,r16\n", $3);
                    printf("st.l r16,%s\n", $1); }
                ;
constausdruck:
                NUMBER
                  { $$ = $1; }
                | constausdruck '+' constausdruck
                  { $$ = $1 + $3; }
                | constausdruck '-' constausdruck
                  { $$ = $1 - $3; }
                | constausdruck '*' constausdruck
                  { $$ = $1 * $3; }
                | constausdruck '/' constausdruck
                  { $$ = $1 / $3; }
                | '-'constausdruck %prec UNARYMINUS
                  { $$ = -$2; }
                | '(' constausdruck ')'
                  { $$ = $2; }
                ;
%%

int nextch;  /* global verwendet */
int chcnt = 0;
int chpos = 0;
char linebuff[160];
```

```
int getit()
{
  if (chcnt == 0)
  {
    printf("\nEingabe:");
    gets(linebuff);
    chcnt = strlen(linebuff);
    if (chcnt != 0) /* nur wenn keine leere Eingabe */
    {
      linebuff[chcnt] = ' '; /* zeilenende als blank uebertragen */
      chcnt++;  /* incl blank */
    }        /* sonst wird EOF uebertragen */
    chpos = 0;
  }
  if (chcnt == 0)
    return(EOF);
  chcnt--;
  return(linebuff[chpos++]);
}

char buffer[160];

yylex()
{
  int i,val;
  while (nextch == ' ')
    nextch = getit();
  if (nextch == EOF)
    return(EOF);
  if (isdigit(nextch))
  {
    yylval.in = 0;
    while (isdigit(nextch))
    {
      yylval.in = yylval.in * 10l + (nextch - '0');
      nextch = getit();
    }
    return(NUMBER);
  }
  else if (isalpha(nextch))
  {
    i = 0;
    while (isalnum(nextch))
    {
      buffer[i++] = nextch;
      nextch = getit();
    }
    buffer[i] = 0;
    yylval.ch = malloc(strlen(buffer)+1);
    strcpy(yylval.ch, buffer);
    return(IDENTIFIER);
  }
  else
  {
    val = nextch;
    nextch = getit();
    return(val);
```

```
  }
}

main()
{
  chcnt = 0;
  nextch = getit();
  if (yyparse())
  {
    printf("Fehler im Programm\n");
  }
}

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

/* end */
```

Listing 2: Die Header-Datei mit den generierten Definitionen

```
typedef union
{
  long in;
  char *ch;
} YYSTYPE;

extern YYSTYPE yylval;
#define UNARYMINUS 257
#define IDENTIFIER 258
#define NUMBER 259
```

Listing 3: Ein Make zur Erzeugung des Minicompilers

```
#makefile fuer erzeugung des Beispielprogramms

YFLAG=-c -d
CFLAG=-c -AC
LFLAG=/Fecppc

all:  simple.exe

simple.c  : simple.y
            c:\pcyacc\pcyacc -Dsimple.h simple.y

simple.exe : simple.c
        cl /Od /AL simple.c
```

Listing 4: Ein Ausschnitt aus der Parserdefinition für GNU C

```
/* Ausschnitt aus GNU-C Compiler Parser */


structsp:
          STRUCT identifier '{'
                { $$ = start_struct (RECORD_TYPE, $2);
                  /* Start scope of tag before parsing components.  */
                }
          component_decl_list '}'
                { $$ = finish_struct ($<ttype>4, $5);
                  /* Really define the structure.  */
                }
        | STRUCT '{' component_decl_list '}'
                { $$ = finish_struct (start_struct (RECORD_TYPE,
                                NULL_TREE),$3); }
        | STRUCT identifier
                { $$ = xref_tag (RECORD_TYPE, $2); }
        | UNION identifier '{'
                { $$ = start_struct (UNION_TYPE, $2); }
          component_decl_list '}'
                { $$ = finish_struct ($<ttype>4, $5); }
        | UNION '{' component_decl_list '}'
                { $$ = finish_struct (start_struct (UNION_TYPE,
                                NULL_TREE),$3); }
        | UNION identifier
                { $$ = xref_tag (UNION_TYPE, $2); }
        | ENUM identifier '{'
                { $<itype>3 = suspend_momentary ();
                  $$ = start_enum ($2); }
          enumlist maybecomma_warn '}'
                { $$ = finish_enum ($<ttype>4, nreverse ($5));
                  resume_momentary ($<itype>3); }
        | ENUM '{'
                { $<itype>2 = suspend_momentary ();
                  $$ = start_enum (NULL_TREE); }
          enumlist maybecomma_warn '}'
                { $$ = finish_enum ($<ttype>3, nreverse ($4));
                  resume_momentary ($<itype>2); }
        | ENUM identifier
                { $$ = xref_tag (ENUMERAL_TYPE, $2); }
        ;

maybecomma:
          /* empty */
        | ','
        ;

maybecomma_warn:
          /* empty */
        | ','
                { if (pedantic) warning ("comma at end of enumerator
list"); }
        ;
```