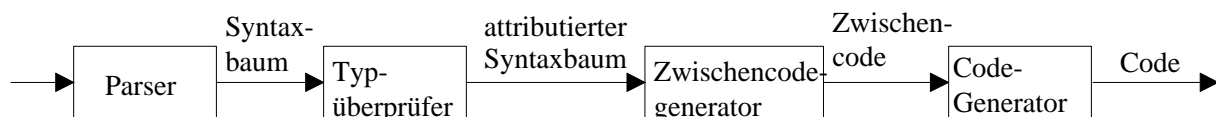


# 8 Code- und Zwischencode-Generierung

## 8.1 Syntaxorientierte Übersetzung

Der syntaktischen und semantischen Analyse folgt als nächste Compilerphase die Zwischencode-Erzeugung.



*Syntax-orientierte Übersetzung:* Den Produktionen der Grammatik, die bei der Syntaxanalyse erkannt werden, sind *semantische Aktionen* zugeordnet, die Zwischencode erzeugen.

Ein *Top-Down-Parser* führt also immer dann solche semantischen Aktionen durch, wenn ein *Teilauftrag* erledigt ist.

Ein *Bottom-Up-Parser* führt semantische Aktionen bei *Reduktionen* aus.

Der Prozeß der Zwischencode-Generierung kann, falls die Sprache es erlaubt und es erwünscht ist, in den Parsing-Prozeß eingebunden werden (Einpass-Compiler).

## 8.2 Zwischensprachen

Zwischencode steht in seiner Komplexität zwischen höheren Programmiersprachen und Maschinensprache.

Vorteile der Verwendung von Zwischencode sind:

- Maschinenunabhängigkeit
- Gute Ausgangsbasis für Optimierung und Codegenerierung
- Klare Schnittstelle zwischen verschiedenen Phasen, Modularisierung des Compilers

Wir betrachten im folgenden gebräuchliche Zwischencodetypen und ihre Eigenschaften.

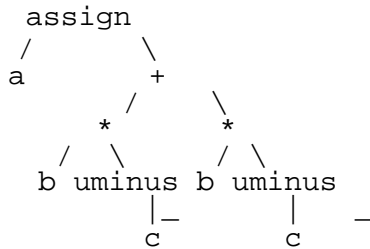
### 1. Graphen-Darstellungen

Ein *Syntaxbaum* beschreibt die natürliche hierarchische Struktur eines Quellprogramms.

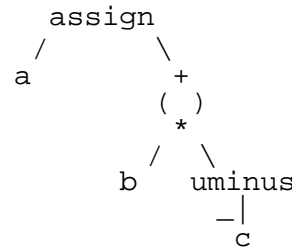
Ein GAG (Gerichteter azyklischer Graph) enthält die gleichen Informationen, aber in einer kompakteren Darstellung, da gemeinsame Unterausdrücke erkannt werden.

**Bsp.:**  $a := b * -c + b * -c$

a) *Syntaxbaum*



b) *GAG*



### 2. Quadrupelsprachen

Die Anweisungen sind Realisierungen abstrakter Dreiadreßbefehle im Format

OP	x	y	z
----	---	---	---

wobei  $x, y, z$  *Operanden*, dh. Konstanten, Variablen oder Compiler-generierte temporäre Werte sind. OP steht für einen Operator.

Die Anweisung

$$z := a * b + c * d$$

wird übersetzt in

MPY	a	b	t <sub>1</sub>
MPY	c	d	t <sub>2</sub>
ADD	t <sub>1</sub>	t <sub>2</sub>	t <sub>3</sub>
STORE	t <sub>3</sub>	z	

mit temporären Größen  $t_i$ . Dreiadreß-Code ist gut geeignet für Zielcode-Generierung und Code-Optimierung. Er stellt eine *linearisierte Darstellung* eines Syntaxbaums oder GAG dar:

**Bsp.:**  $a := b * -c + b * -c$

a) Code für Syntaxbaum

NEG	c	t <sub>1</sub>	
MPY	b	t <sub>1</sub>	t <sub>2</sub>
NEG	c	t <sub>3</sub>	
MPY	b	t <sub>3</sub>	t <sub>4</sub>
ADD	t <sub>2</sub>	t <sub>4</sub>	t <sub>5</sub>
STORE	t <sub>5</sub>	a	

b) Code für GAG

NEG	c	t <sub>1</sub>	
MPY	b	t <sub>1</sub>	t <sub>2</sub>
ADD	t <sub>2</sub>	t <sub>2</sub>	t <sub>5</sub>
STORE	t <sub>5</sub>	a	

### 3. Tripelsprachen

Bei der Erzeugung von Quadrupelcode wird jede Hilfsvariable nur ein einziges Mal zur Aufnahme eines Zwischenresultats verwendet. Man kann deshalb auf die explizite Angabe der Hilfsvariablen verzichten. Wird ein Zwischenresultat als Operand einer weiteren Anweisung gebraucht, so trägt man in dieser Anweisung einen *Zeiger* auf die das Zwischenresultat erzeugende Anweisung ein.

Format:

OP	X	y
----	---	---

wobei OP der Operator und x und y Operanden sind. Unsere Anweisung

$$z := a * b + c * d$$

wird übersetzt in

```

1: MPY      a      b
2: MPY      c      d
3: ADD      (1)    (2)
4: STORE    (3)    z

```

Diese Darstellung führt zu geringerem Platzbedarf, aber größerem Aufwand bei Optimierungen, wo man Zeiger umstellen muß.

### 4. Postfixsprachen

Diese werden im Zusammenhang mit einer *abstrakten Stack-Maschine* benutzt. Sie besitzen in der Regel einen oder gar keinen Operanden.

#### **Prinzip:**

Dem Programm steht ein Stack zur Verfügung mit dem Operanden LOAD zum Transport der Daten auf den Stack und STORE für das Entfernen von Daten aus dem Stack.

#### **Konvention:**

- Besitzt eine Operation keinen expliziten Operanden, so setzt sie voraus, daß diese zum Zeitpunkt der Ausführung *oben auf* dem Stack liegt. Die Operation legt ihr Resultat wieder *auf den Stack* ab.
- Jede Operation besitzt eine *eindeutige* Zahl von Stackoperanden.

**Bsp.:** Die Anweisung  $z := a * b + c * d$  ergibt folgenden Postfixcode

```

LOAD  A
LOAD  B
MPY
LOAD  C
LOAD  D
MPY
ADD
STORE Z

```

**Frage:** Wie sieht der Stack jeweils aus?

## 8.3 Semantische Aktionen

Es soll an einigen wenigen Beispielen gezeigt werden, wie man den Produktionen einer Grammatik semantische Aktionen zuordnet und wie sich dies auf die Zwischencode-Erzeugung auswirkt. Wir betrachten hier nur die Übersetzung in Postfixcode.

### 1a. Übersetzung von Ausdrücken in Postfixcode

Grammatik  $G = (N, T, P, \text{EXPR})$  mit

$$\begin{aligned} N &= \{ \text{EXPR}, \text{TERM}, \text{FACT} \}; \\ T &= \{ \text{ident}, \text{number}, +, -, *, /, (, ) \}; \\ P &= \{ \text{EXPR} \rightarrow [+ | -] \text{TERM} \{ + \text{TERM} | - \text{TERM} \}; \\ &\quad \text{TERM} \rightarrow \text{FACT} \{ * \text{FACT} | / \text{FACT} \}; \\ &\quad \text{FACT} \rightarrow \text{ident} | \text{number} | (\text{EXPR}) \quad \quad \quad \} \end{aligned}$$

Der Recursive-Descent-Parser für arithmetische Ausdrücke aus Abschnitt 4.3 wird erweitert durch die Zwischencode-Generierung im Postfix-Format.

#### Zielsprache:

LOAD	v	bringt den Wert einer Variablen v in den Stack
LIT	c	bringt den Wert c einer Konstanten in den Stack
NEG		negiert das oberste Stackelement
ADD		addiert die obersten beiden Stackelemente und legt das Resultat wieder auf den Stack
SUB		subtrahiert die obersten beiden Stackelemente voneinander und legt das Resultat wieder auf den Stack
MPY		multipliziert die obersten beiden Stackelemente und legt das Resultat wieder auf den Stack
DIV		dividiert die obersten beiden Stackelemente durcheinander und legt das Resultat wieder auf den Stack

#### Semantik zu $\text{FACT} \rightarrow \text{ident} | \text{number} | (\text{EXPR})$

Wenn die syntaktische Analyse die Struktur `ident`, also eine Variable erkannt hat, so wird die Anweisung `LOAD v` erzeugt, wobei v der Name der Variablen ist.

Wenn die syntaktische Analyse die Struktur `number`, also eine Konstante erkannt hat, so wird eine Anweisung `LIT c` erzeugt, wobei c der Wert der Konstanten ist.

Wenn die syntaktische Analyse die Teilstruktur `( EXPR )` erkannt hat, so kann vorausgesetzt werden, daß der zugehörige Code bereits erzeugt ist und zur Laufzeit das Ergebnis des Ausdrucks schon im Stack vorliegt. Es sind daher keine semantischen Aktionen erforderlich.

#### Semantik zu $\text{EXPR} \rightarrow [+ | -] \text{TERM} \{ + \text{TERM} | - \text{TERM} \}$

Erkennt der Parser das Vorzeichen `-`, so generiert er eine `NEG`-Anweisung sobald der zugehörige Operand verarbeitet ist. Beim Vorzeichen `+` ist keine semantische Aktion erforderlich.

Jedesmal, wenn die Teilstruktur `+ TERM` oder `- TERM` zur Analyse ansteht, ist der linke Operand bereits analysiert und Code generiert, der ihn in den Stack lädt. Der Compiler muß sich jedoch die Operanden selbst solange merken, bis er den rechten Operanden analysiert und interpretiert hat. Erst dann kann er die entsprechende Zwischencode-Anweisung ausgeben.

**Semantik zu TERM  $\rightarrow$  FACT { \* FACT | / FACT }**

Jedesmal, wenn die Teilstruktur \* FACT oder / FACT zur Analyse ansteht, ist der linke Operand bereits analysiert und Code generiert, der ihn in den Stack lädt. Der Compiler muß sich jedoch die Operanden selbst solange merken, bis er den rechten Operanden analysiert und interpretiert hat. Erst dann kann er die entsprechende Zwischencode-Anweisung ausgeben.

Der *Expression-Compiler* hat dann folgendes Aussehen:

```

/* compiler */
int symbol;

void expression() {
    int min;
    if (symbol == plus || symbol == minus) {
        if (symbol == minus) min = 1; else min = 0;
        getsym(); term();
        if (min == 1) puts ("NEG");
    } else term();
    while (symbol == plus || symbol == minus) {
        if (symbol == minus) min = 1;
        else min = 0;
        getsym(); term();
        if (min == 1) puts("SUB"); else puts("ADD");
    }
} /* expression */

void term() {
    int mpy;
    factor();
    while (symbol == times || symbol == slash)
    {
        if (symbol == times) mpy = 1; else mpy = 0;
        getsym(); factor();
        if (mpy == 1) puts("MPY"); else puts("DIV");
    }
} /* term */

void factor() {
    switch (symbol) {
        case ident : puts("LOAD v"); getsym(); break;
        case number : puts("LIT c"); getsym(); break;
        case lparen : getsym(); expression();
                    if (symbol == rparen)getsym(); else error();
                    break;
    }
} /* factor */

main()
{
    getsym();
    expression();
} /* compiler */

```

Für die Eingabe

$$(v+v) * (v-v) / v$$

liefert das Programm das Ergebnis

```

LOD v
LOD v
ADD
LOD v
LOD v
SUB
MPY
LOD v
DIV

```

## 1b. Übersetzung von Ausdrücken in Dreiadreib-Code

Wir betrachten weiterhin die obige *Grammatik*  $G = (N, T, P, \text{EXPR})$  für Ausdrücke.

Der Recursive-Descent-Parser für arithmetische Ausdrücke aus Abschnitt 4.3 wird erweitert durch die Zwischencode-Generierung im Dreiadreib-Format.

### Zielsprache:

Dreiadreib-Befehle der Form  $\text{op3} := \text{INST op1 op2}$ ,  $\text{INST} = \text{ADD, SUB, MPY, DIV}$ .

Die Codegenerierung ist hier ein wenig komplexer als im Fall des Postfixcodes. Es ist erforderlich, den syntaktischen Prozeduren einen Parameter zu geben, in dem der Platz des Resultates des Dreiadreib-Befehls zurückgeliefert wird.

```

#define PLUS '+'
#define MINUS '-'
#define TIMES '*'
#define SLASH '/'
#define IDENT 'v'
#define LPAREN '('
#define RPAREN ')'

char symbol;
int temp = 0;

void term();
void factor();

void newtemp(char *s) {
    char z[4];
    temp++;
    itoa(temp, z, 10);
    strcpy(s, "t");
    strcat(s, z);
}

void error() {
    puts("Error");
    exit(1);
}

void getsym() {
    symbol = getchar();
}

```

```
void expression(char *place) {
    int min;
    char place1[10], place2[10], temp[10], code[30];

    if (symbol == PLUS || symbol == MINUS) {
        if (symbol == MINUS) min = 1;
        else min = 0;
        getsym();
        term(place1);
        newtemp(temp);
        if (min == 1) {
            strcpy(code, temp);
            strcat(code, " := ");
            strcat(code, " NEG ");
            strcat(code, place1);
            puts(code);
            strcpy(place, temp);
        }
    }
    else term(place1);
    strcpy(place, place1);
    while (symbol == PLUS || symbol == MINUS) {
        if (symbol == MINUS) min = 1;
        else min = 0;
        getsym();
        term(place2);
        newtemp(temp);
        strcpy(code, temp);
        strcat(code, " := ");
        if (min == 1) strcat(code, " SUB ");
        else strcat(code, " ADD ");
        strcat(code, place1);
        strcat(code, " ");
        strcat(code, place2);
        puts(code);
        strcpy(place, temp);
        strcpy(place1, place);
    }
} /* expression */

void term(char *place) {
    char place1[10], place2[10], temp[10], code[30];
    int mpy;
    factor(place1);
    strcpy(place, place1);
    while (symbol == TIMES || symbol == SLASH){
        if (symbol == TIMES) mpy = 1;
        else mpy = 0;
        getsym();
        factor(place2);
        newtemp(temp);
        strcpy(code, temp);
        strcat(code, " := ");
        if (mpy == 1) strcat(code, " MPY ");
        else strcat(code, " DIV ");
        strcat(code, place1);
        strcat(code, " ");
    }
}
```

```
        strcat(code, place2);
        puts(code);
        strcpy(place, temp);
        strcpy(place1, place);
    }
} /* term */

void factor(char *place) {
    switch (symbol) {
        case IDENT:    strcpy(place, "v");
                       getsym();
                       break;
        case LPAREN:  getsym();
                       expression(place);
                       if (symbol == RPAREN) getsym();
                       else error();
                       break;
        default:      error();
    }
} /* factor */

main()
{
    char place[10];
    getsym();
    expression(place);
    printf("Resultat in %s\n", place);
    return 0;
}
```

Für die Eingabe

$(v+v) * (v-v) / v$

liefert das Programm das Ergebnis

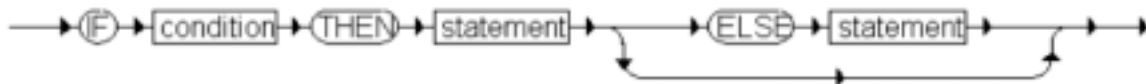
```
t1 := ADD v v
t2 := SUB v v
t3 := MPY t1 t2
t4 := DIV t3 v
```

## 2. Übersetzung bedingter Anweisungen

Die oben betrachtete Methode zur syntaxorientierten Übersetzung läßt sich auf alle Strukturen einer Programmiersprache verallgemeinern.

### Bsp.: Übersetzung bedingter Anweisungen in Postfixcode

In PL/0 lautet diese:



Falls kein ELSE-Teil vorhanden ist, muß der zu erzeugende Code folgende Struktur haben:

Code zur Auswertung von "Condition"	
JUMP ON FALSE L <sub>1</sub>	← <i>Fixup1</i>
Code zur Ausführung von "then-Statement"	
L1: nachfolgender Code	

Falls der ELSE-Teil vorhanden ist, muß er folgende Struktur haben:

Code zur Auswertung von "Condition"	
JUMP ON FALSE L1	← <i>Fixup1</i>
Code zur Ausführung von "then-Statement"	
JUMP L2	← <i>Fixup2</i>
L1: Code zur Ausführung von "else-Statement"	
L2: nachfolgender Code	

Dies ergibt folgende Codesequenz in der Pascal-Version des PL/0-Compilers:

```

if symbol = ifsym then
begin
  getsym;
  condition(...);
  if sym = thensym then getsym else error(...);
  fixup1:= codeindex;           { Adr. des Sprungbefehls merken }
  generate(JPC, 0, 0);          { Bed. Sprung mit undef. Adr. }
  statement(...);
  if sym = elsesym then
  begin
    getsym;
    fixup2:= codeindex;         { Adr. des Sprungbefehls merken }
    generate(JPU, 0, 0);        { Unbed. Sprung mit undef. Adr. }
    code[fixup1].a := codeindex; { Adr. für Sprung über then }
    statement(fsys);
    code[fixup2].a := codeindex { Adr. für Sprung über else }
  end
  else
    code[fixup1].a := codeindex; { Adr. für Sprung über then }
  end;

```

In der C-Version lautet der Programmtext

```

void ifstmnt()
{
    int fixup1, fixup2;

    getsym();
    condition();
    match(thensym, 16);
    fixup1 = cdindex();
    emit(JPC, 0, 0);
    statement();
    if (sym == elsesym) {
        getsym();
        fixup2 = cdindex();
        emit(JPU, 0, 0);
        fixup(fixup1, cdindex());
        statement();
        fixup(fixup2, cdindex());
    } else {
        fixup(fixup1, cdindex());
    }
}

```

### 3. Übersetzung von Schleifen

**Bsp.:** Übersetzung der while-Schleife in Postfixcode

In PL/0 lautet diese:



Der zu erzeugende Code muß folgende Struktur haben:

L1:	Code zur Auswertung von "Condition"	← <i>Fixup</i>
	JUMP ON FALSE L2	
	Code zur Ausführung von "Statement"	
	JUMP L1	
L2:	nachfolgender Code	

Dies ergibt folgende Codesequenz im Recursive-Descent-Compiler für PL/0 (Pascal-Version):

```
if symbol = whilesym then
begin
  fixup1:=codeindex;           {Adresse des Befehls merken}
  getsym;
  condition(...);
  fixup2:=codeindex;         {Adresse des Befehls merken}
  generate(JPC,0,0);         {Sprung mit undef. Adresse}
  if sym=dosym then
    Getsym
  else
    Error(...);
    statement(fsyst);
    generate(JPU,0,fixup1);   {Rücksprung an Anfang}
    code[fixup2].a := codeindex; {Sprungadresse nachtragen}
end;
```

In der C-Version lautet der Programmtext

```
void whilestmnt()
{
  int fixup1, fixup2;

  fixup1 = cdindex();
  getsym();
  condition();
  fixup2 = cdindex();
  emit(JPC, 0, 0);
  match(dosym, 18);
  statement();
  emit(JPU, 0, fixup1);
  fixup(fixup2, cdindex());
}
```

## 8.4 Die Zwischensprache von PL/0

Die abstrakte Maschine, die dem erzeugten Zwischencode zugrunde liegt, wird charakterisiert durch ihre Datenstruktur und die zur Verfügung stehenden Operationen.

### Adressierung von Objekten

Dynamische Speicherverwaltung, Adressierung über die Niveaudifferenz der Static-Link-Kette und ihre Relativdistanz zum Datenraumanfang. Konstanten werden zum Zeitpunkt ihrer Verwendung auf den *Hilfsvariablenstack* gelegt.

### Aufbau der Symboltabelle

Pascal-Version

```

symtable: ARRAY [0..tmax] OF RECORD
    name: alfa;
    CASE kind OF
        constobj: (value: integer);
        varobj  : (level, addr, size: integer);
        procobj : (level, addr, size: integer);
    END;

```

C-Version

```

struct symttype{
    char name[cmax];
    int  kind;
    int  level;
    int  addr;
    int  size;
} symtable [txmax];

```

Einträge für Konstanten haben also diese Form

Konstantenname	constobj	Konstantenwert
----------------	----------	----------------

Einträge für *Variablen* sehen so aus:

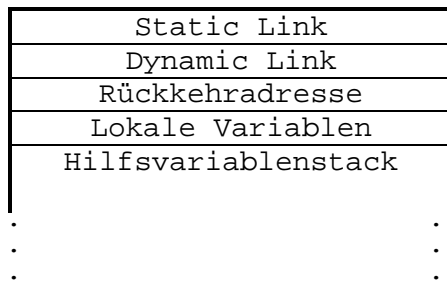
Variablenname	varobj	Niveau der Deklaration	Relativadresse im Datenraum	size (unbenutzt)
---------------	--------	------------------------	-----------------------------	------------------

Einträge für *Prozeduren* schließlich:

Prozedurname	procobj	Niveau der Deklaration	Adresse der ersten Anweisung	Größe des Datenraums
--------------	---------	------------------------	------------------------------	----------------------

### Datenstruktur

Bei jedem Prozeduraufruf wird ein Activation Record neu auf den Laufzeit-Stack gelegt. Er sieht folgendermaßen aus:



Nur der oberste Activation Record im Stack besitzt einen Hilfsvariablenstack, da dieser zum Zeitpunkt eines Prozeduraufrufs stets vollständig abgebaut ist.

## Die Zwischencode-Anweisungen

Der Zwischencode-Befehlssatz, der auf dieser Datenstruktur operiert, ist ein Postfixcode und umfaßt folgende Anweisungen:

### *Transportoperationen:*

- LIT n bringe einen Literalwert n in den Hilfsvariablenstack.  
 LOD l,a bringe den Wert einer Variablen in den Stack mit der Relativedistanz a zur Datenraumbasis und einer Niveaudifferenz l zwischen dem aktuellen Niveau und dem ihrer Deklaration.  
 STO l,a bringe das oberste Stack-Element auf den Platz einer Variablen mit der Relativedistanz a zur Datenraumbasis und einer Niveaudifferenz l zwischen dem aktuellen Niveau und dem ihrer Deklaration.  
 INT n Reserviere n Plätze im Stack für die Variablen des nächsten Datenraumes.

### *Sprungbefehle:*

- JPU a springe zur Anweisung mit der Adresse a.  
 JPC a springe zur Anweisung mit der Adresse a, falls im Hilfsvariablenstack der Wert FALSE liegt. Entferne diesen vom Stack.  
 CAL l, a Aufruf der Prozedur mit der Codeadresse a und einer Niveaudifferenz l zwischen dem aktuellen Niveau und dem ihrer Deklaration. Lege einen neuen Datenraum im Stack an, initialisiere den Header und springe zu Adresse a.  
 RET Rückkehr aus einer Prozedur: der aktuelle Datenraum wird freigegeben und danach die Programmausführung hinter der Aufrufstelle fortgesetzt.

### *Arithmetische Operationen:*

Sie entnehmen grundsätzlich ihre Operanden vom Stack und legen ihr Resultat dort wieder ab.

- ADD addiere zwei Summanden.  
 SUB subtrahiere das oberste Stackelement vom darunter stehenden.  
 MPY multipliziere zwei Faktoren.  
 DIV dividiere das vorletzte Stackelement durch das oberste.  
 NEG negiere das oberste Stackelement.

**Logische Operationen:**

Sie vergleichen die beiden obersten Stackelemente und legen als Ergebnis TRUE (=1) bzw. FALSE (=0) im Stack ab. Eine Ausnahme ist die Operation OD, die nur einen Operanden besitzt.

OD	ist der Operand ungerade?
EQ	sind beide Operanden gleich?
NE	sind beide Operanden ungleich?
LE	ist der erste Operand kleiner oder gleich dem zweiten?
GE	ist der erste Operand größer oder gleich dem zweiten?
LS	ist der erste Operand kleiner als der zweite?
GT	ist der erste Operand größer als der zweite?

**Input-Output-Operationen:**

Sie beeinflussen zum Teil das oberste Stackelement.

INP	lies eine Integer-Zahl ein und lege sie auf die Stack.
PRN	nimm den obersten Wert von Stack und gib ihn aus.
PRL	gib einen Zeilevorschub aus.
PRS a	gib a ASCII-Zeichen aus, nimm diese aus den a folgenden ASC-Befehlen.
ASC a	ASCII-Zeichen a

In Abschnitt 8.6 finden wir ein Beispiel für die Code-Erzeugung.

## Leistungen des Recursive-Descent-Compilers für PL/0

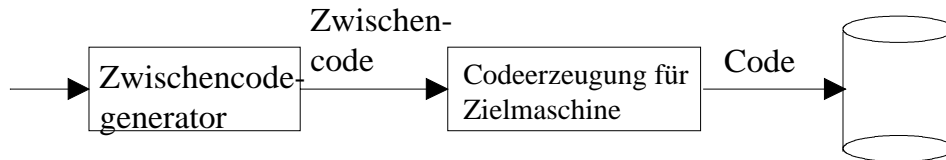
- syntaktische Analyse der Quelle mit Fehlerbehandlung
- Übersetzung korrekter Programme in Postfixcode
- Protokoll der während der Analyse durchlaufenden Prozeduren
- Auflistung der Symboltabelle zu verschiedenen Zeitpunkten der Übersetzung, immer nach der Bearbeitung eines vollständigen Blocks.

Die Erzeugung des Postfixcode geschieht in den syntaktischen Prozeduren mit Hilfe der Prozedur "Emit".

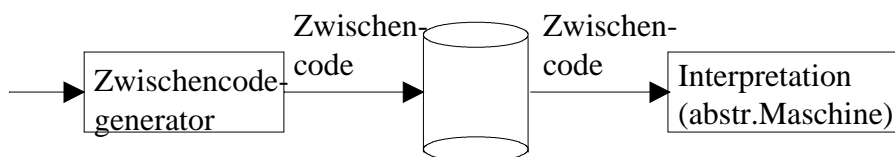
```
void emit(int opc, int lv, int addr)
{
    if (codeindex == codemax) error(31);
    code[codeindex].o = opc;
    code[codeindex].l = lv;
    code[codeindex].a = addr;
    codeindex++;
}
```

## 8.5 Code-Erzeugung und Interpretation

Bei Native-Code-Compilern gilt das Diagramm



Es gibt jedoch auch den Fall, daß der Zwischencode direkt ausgeführt wird auf einer abstrakten Maschine, d.h. von einem geeigneten Programm interpretiert wird.



Dies hat einige Vorteile:

- Geringer Codeumfang
- Bessere Behandlung von Laufzeitfehlern

aber auch einen gewichtigen Nachteil:

- Geringere Effizienz

gegenüber der Erzeugung von Native-Code.

Im Anhang findet sich der Interpreter für den vom PL/0-Compiler erzeugten Postfixcode.

## 8.6 Erzeugung von Assemblercode aus dem Zwischencode

Die Generierung von Zielcode für einen *echten* Prozessor ist eine erheblich schwerere Aufgabe als die bisher behandelte Interpretation. Man benötigt dazu ausgezeichnete Kenntnisse des Prozessors, seiner Assemblersprache sowie des Betriebssystems, unter dem die Zielprogramme schließlich ablaufen sollen.

Die Generierung von Maschinen- bzw. Assemblercode für den Zielprozessor kann unmittelbar nach der semantischen Analyse an die Stelle der Zwischencodegenerierung treten. Damit büßt der Compiler jedoch einen Großteil seiner Portabilität ein.

Die Zielcodeerzeugung kann sich jedoch auch an die Generierung eines Zwischencodes und eine etwaige Optimierung dieses Zwischencodes als weitere Phase anschließen. Als Möglichkeiten des Zwischencodes sind Postfixcode, Dreiadreß-Code oder Zweiadreß-Code gebräuchlich.

Die Vorgehensweise ist dabei so, daß zu jeder Instruktion des Zwischencodes eine äquivalente Folge von Instruktionen des Zielcodes generiert wird. Dies kann in Form von Maschinencode oder von Assemblercode geschehen.

Wir betrachten hier exemplarisch den Fall, daß aus dem Postfixcode des PL/0-Compilers sog. Assembler-Makros für den Zielprozessor 80X86 von Intel unter dem Betriebssystem Windows 95, NT bzw. 2000 generiert werden soll, für PC-Besitzer eine natürliche Wahl! Diese Assemblercode-Generierung wird von einem vom Compiler getrennten Programm PLOASM vorgenommen. Ihr schließt sich dann eine Assemblierung und ein Linkerlauf an. Dies bedingt folgenden Ablauf:

```
pl0scan test    Scannen von test.pl0 → test.tok
pl0pars test    Parsen, Code-Erz. von test.pl0 → test.cod
pl0asm test     Umwandlung von test.cod → test.asm
tasm test;     Assemblierung mit Turbo Assembler → test.obj
tlink test;    Linken mit Turbo Linker → test.exe
```

Dem Stack der abstrakten Maschine entspricht entspricht der Hardwarestack des 80x86-Mikroprozessors. Für jeden Postfixcode-Befehl wird ein Folge von äquivalenten Assemblerbefehlen bzw. ein gleichwertiges Assembler-Makro benötigt. Ein so definiertes Assembler-Programm wird noch durch einen Assembler-Prolog zu Beginn und einen Assembler-Epilog am Ende ergänzt. Diese haben hauptsächlich organisatorische Aufgaben.

Wir zeigen exemplarisch Epilog, Prolog und Assembler-Makos zu den Befehlen

ADD, MPY OD, EQ, LOD und JPU.

```
; Assembler prolog
dosseg
.model small
.stack 1000h
.data
base dw 0
str1 db 0, '$'
buffer db 80 dup (?)
blen db 80
blank db ' ', '$'
minus db '-', '$'
crlf db 13, 10, '$'
.code

start:
mov ax, @data
mov ds, ax
mov ax, 0
push ax
mov [base], sp
mov ax, sp
push ax
mov ax, offset stop
push ax
add sp, 6

; Assembler epilog
stop:
mov ah, 4Ch
int 21h
include rts.inc
```

```

end    start

p_add  macro    ; Addition der beiden obersten Stackelemente
pop    ax          ; oberstes poppen
mov    bp,sp      ; Adresse des neuen obersten
add    [bp],ax    ; beide addieren --> oberstes
endm

p_mpy  macro    ; Multiplikation der beiden obersten
           ; Stackelemente
pop    ax          ; oberstes poppen
mov    bp,sp      ; Adresse des neuen obersten
imul   word ptr [bp] ; ax mit zweitobersten
           ; multiplizieren
mov    [bp],ax    ; Resultat --> oberstes
endm

p_od   macro    ; Ist Top of Stack ungerade?
pop    ax          ; oberstes
cwd                ; Dividend dx:ax
mov    bx, 2      ; Division durch 2
idiv   bx         ; Rest in dx
push   dx         ; Resultat
endm

p_eq   macro    ; Vergleich der beiden obersten Stackelemente
           ; =)
local  eq1, eq2
pop    ax          ; oberstes
pop    bx          ; zweitoberstes
cmp    ax, bx     ; vergleichen
je     eq1        ; Sprung, wenn beide gleich sind
sub    ax, ax     ; <> : 0
jmp    eq2
eq1:   mov    ax, 1 ; = : 1
eq2:   push   ax
endm

p_lod  macro    l, a ; Variable mit Relativedifferenz l und
           ; Offset a holen und auf Stack pushen
p_sbs  l          ; Base als Resultat in dx
mov    bp, dx     ; Adressierung
push   word ptr [bp-2*a] ; Variable holen und pushen
endm

p_jpu  macro    n ; Unbedingter Sprung nach n
jmp    far ptr n
endm

```

Wir beschließen diesen Abschnitt mit einem exemplarischen **Vergleich** zwischen Interpretation und Assemblercode-Generierung anhand eines PL/0-Programms.

Das kleine PL/0-Programm test.pl0

```

const n=10000;
var s, i, j;

begin
  write('Test');
  writeln;
  i:= 0;
  while i < n do
  begin
    i:= i + 1;
    write(i);
    writeln;
    s:= 0;
    j:= 0;
    while j < n do
    begin
      j:= j + 1;
      s:= s + j;
    end;
  end;
end.

```

Es wurde kompiliert mit dem PL0-Compiler durch die Kommandos

```

pl0scan test
pl0pars test

```

und ergab ein Postfixcode-File test.cod der Länge 760 Bytes:

000	CAL	0	1	021	LIT	0	0
001	INT	0	3	022	STO	0	3
002	PRS	0	4	023	LIT	0	0
003	ASC	0	84	024	STO	0	5
004	ASC	0	101	025	LOD	0	5
005	ASC	0	115	026	LIT	0	10000
006	ASC	0	116	027	LSS	0	0
007	PRL	0	0	028	JPC	0	38
008	LIT	0	0	029	LOD	0	5
009	STO	0	4	030	LIT	0	1
010	LOD	0	4	031	ADD	0	0
011	LIT	0	10000	032	STO	0	5
012	LSS	0	0	033	LOD	0	3
013	JPC	0	39	034	LOD	0	5
014	LOD	0	4	035	ADD	0	0
015	LIT	0	1	036	STO	0	3
016	ADD	0	0	037	JPU	0	25
017	STO	0	4	038	JPU	0	10
018	LOD	0	4	039	RET	0	0
019	PRN	0	0				
020	PRL	0	0				

Es wurde ausgeführt mit Hilfe des Interpreters:

```
pl0int test
```

Die Ausführungszeit war ca. 81 sec, auf der anderen Seite entsteht daraus mittels

```
pl0asm test
```

ein 80X86-Assembler-File test.asm:

```
; Assembler prolog
;Generated by PL0ASM for MS-DOS 3.X, 4.X, 5.X
        dosseg
        .model  small
;
        .stack  10240
;
        .data
base     dw      0
buffer  db      80 dup (?)
blen    db      80
blank   db      '$'
minus   db      '-$'
crlf    db      13, 10, '$'
bytecnt dw      ?
pflag   db      ?
str01   db      'Test$'
;
        .code
start:
        mov     ax, @data
        mov     ds, ax
;
        mov     ax, 0
        push   ax
        mov     [base], sp
        mov     ax, sp
        push   ax
;
        mov     cx, 0
        call   p_sbs
        push   dx
        mov     bp, sp
        mov     ax, [base]
        push   ax
        mov     [base], bp
        call   pr001
        pop    dx
        mov     [base], dx
        pop    dx
;
        call   endprocess
pr001   proc
        sub    sp, 6
        mov    bx, offset str01
        call   outstring
        mov    bx, offset crlf
        call   outstring
        mov    ax, 0
```

```
    push    ax
    pop     ax
    mov     bp, [base]
    mov     word ptr [bp-2*4], ax
1010:
    mov     bp, [base]
    push   word ptr [bp-2*4]
    mov     ax, 10000
    push   ax
    pop     ax
    pop     bx
    cmp     bx, ax
    jl     m001
    sub     ax, ax
    jmp     m002
m001:  mov     ax, 1
m002:  push   ax
    pop     ax
    cmp     ax, 0
    je     m003
    jmp     m004
m003:  jmp     far ptr 1039
m004:
    mov     bp, [base]
    push   word ptr [bp-2*4]
    mov     ax, 1
    push   ax
    pop     ax
    mov     bp, sp
    add     [bp], ax
    pop     ax
    mov     bp, [base]
    mov     word ptr [bp-2*4], ax
    mov     bp, [base]
    push   word ptr [bp-2*4]
    pop     ax
    mov     cx, 5
    call   printint
    mov     bx, offset crlf
    call   outstring
    mov     ax, 0
    push   ax
    pop     ax
    mov     bp, [base]
    mov     word ptr [bp-2*3], ax
    mov     ax, 0
    push   ax
    pop     ax
    mov     bp, [base]
    mov     word ptr [bp-2*5], ax
1025:
    mov     bp, [base]
    push   word ptr [bp-2*5]
    mov     ax, 10000
    push   ax
    pop     ax
    pop     bx
    cmp     bx, ax
```

```
        jl      m005
        sub     ax, ax
        jmp     m006
m005:   mov     ax, 1
m006:   push    ax
        pop     ax
        cmp    ax, 0
        je     m007
        jmp    m008
m007:   jmp     far ptr 1038
m008:   mov     bp, [base]
        push   word ptr [bp-2*5]
        mov    ax, 1
        push   ax
        pop    ax
        mov    bp, sp
        add    [bp], ax
        pop    ax
        mov    bp, [base]
        mov    word ptr [bp-2*5], ax
        mov    bp, [base]
        push   word ptr [bp-2*3]
        mov    bp, [base]
        push   word ptr [bp-2*5]
        pop    ax
        mov    bp, sp
        add    [bp], ax
        pop    ax
        mov    bp, [base]
        mov    word ptr [bp-2*3], ax
        jmp    far ptr 1025
1038:   jmp     far ptr 1010
1039:   add     sp, 6
        ret
pr001   endp
;
;
; Assembler epilog
;
; Runtime Support f r PL0-Compiler mit
; Assembler-Code-Generierung
;
p_sbs   proc
; Base-Adresse des DSA suchen
; Eingabeparameter:
; CX Leveldifferenz
; Ausgabeparameter:
; DX Resultat Base
        mov    dx, [base]
sbs2:   cmp     cx, 0
        je     sbs1
        mov    bp, dx
        mov    dx, word ptr [bp]
        dec    cx
```

```
        jmp      sbs2
sbs1:   ret
p_sbs  endp
;
endprocess proc
;   Prozess durch DOS-Funktion beenden
        mov     ah, 4Ch
        int     21h
        ret
endprocess endp
;
printint proc
;   Integer-Zahl ausgeben
;   Eingabeparameter:
;   AX  Zahl
;   CX  Laenge
        mov     bx, offset buffer
        cmp     ax, 32767
        jbe    prn1
        neg     ax
        call   number2string
        mov     bx, offset buffer
        cmp     [bx], byte ptr ' '
        jne    prn5
        inc     bx
prn3:   mov     cx, 4
        cmp     [bx], byte ptr ' '
        jne    prn4
        inc     bx
        loop   prn3
prn5:   mov     bx, offset minus
        call   outstring
        jmp    prn6
prn4:   dec     bx
        mov     [bx], byte ptr '-'
        mov     bx, offset blank
        call   outstring
prn6:   mov     bx, offset buffer
        call   outstring
        jmp    prn2
prn1:   call   number2string
        mov     bx, offset blank
        call   outstring
        mov     bx, offset buffer
        call   outstring
prn2:   mov     bx, offset blank
        call   outstring
        ret
printint endp
;
number2string proc
;   Umwandeln einer Dualzahl in einen String
;   Eingabeparameter:
;   AX  Zahl
;   DS:BX Zeiger auf Puffer zum Speichern d. Strings
;   CX  Anzahl der Zeichen
        mov     si, 10
```

```

        mov     di, cx
        add     bx, cx
        mov     [bx], byte ptr '$'
        dec     bx
wandeln:
        sub     dx, dx
        div     si
        add     dl, '0'
        mov     [bx], dl
        dec     bx
        loop    wandeln
        mov     cx, di
        inc     bx
        dec     cx
blanks:
        cmp     [bx], byte ptr '0'
        jne     fertig
        mov     [bx], byte ptr ' '
        inc     bx
        loop    blanks
fertig:
        ret
number2string endp
;
outstring proc
; Ausgabe eines Strings unter MS-DOS oder OS/2
; Eingabeparameter:
; DS:BX Zeiger auf String
        mov     dx, bx
        mov     cx, 0
anfang:
        inc     cx
        inc     bx
        cmp     [bx], byte ptr '$'
        jne     anfang
        mov     ah, 40H
        mov     bx, 1
        int     21h
        ret
outstring endp
;
string2number proc
; Umwandeln einer String in eine Dualzahl
; Eingabeparameter:
; DS:BX Zeiger auf Puffer mit String
; CX Anzahl der Zeichen
; Ausgabeparameter:
; AX Zahl
        mov     pflag, 0
        cmp     [bx], byte ptr '-'
        jne     pos
        mov     pflag, 1
        inc     bx
        dec     cx
pos:
        mov     si, 10
        sub     ax, ax
mult:

```

```
        mul     si
        mov     dh, 0
        mov     dl, [bx]
        sub     dl, byte ptr '0'
        mov     di, dx
        add     ax, di
        inc     bx
        loop   mult
        cmp     pflag, 1
        je     nega
        ret
nega:   neg     ax
        ret
string2number endp
;
instring proc
;   Einlesen eines Strings vom Keyboard (MS-DOS)
;   bzw. von stdin (MS OS/2)
;   Eingabeparameter:
;   DS:BX   Zeiger auf Puffer
;   AL      Länge des Puffers
;   Ausgabeparameter:
;   AL      Anzahl der gelesenen Bytes
        mov     ah, 3FH
        mov     dx, bx
        mov     bx, 0
        mov     ch, 0
        mov     cl, al
        int     21h
        ret
instring endp
;
        end     start
```

Durch die Kommandos

```
tasm test
tlink test
```

erhält man ein EXE-File test.exe für DOS/Windows der Länge 1122 Byte mit Ausführungszeit ca. 3 sec.

Zum Vergleich wurde das trivial geändertes Programm test.pas mit dem Turbo Pascal Compiler (7.0) compiliert und ausgeführt, Zeitbedarf < 1 sec. (Alle Rechnungen mit Pentium 4-Prozessor mit 2 GHz unter Windows 2000).