

F A C H A R B E I T

Mathematik / Informatik

von

Jochen Leidner

STAATLICHE KOOPERATIVE GESAMTSCHULE BAD BERGZABERN

- Staatliches Gymnasium Bad Bergzabern -

**Compilerbau in Theorie  
und Praxis**

### Danksagung

Mein Dank gilt Herrn Jürgen Knoll  
für die freundliche Betreuung und Korrektur  
dieser Facharbeit  
Daniel Pfeifer für die Diskussionen und die Pizza  
sowie meinem Rechner für sein  
fehlerfreies Funktionieren nicht  
nur während dieser Arbeit.

## Inhaltsverzeichnis

- 1) Zielsetzung der Arbeit
- 2) Einleitung
  - 2.1) Was ist ein Compiler?
  - 2.2) Verwendete Notationen und Abkürzungen
  - 2.3) Formale Sprachen
- 3) Die Programmiersprache LEAZY
  - 3.1) Sprachdefinition
    - 3.1.1) Lexikalische Definitionen
    - 3.1.2) Syntax
    - 3.1.3) Semantik
  - 3.2) Programmbeispiele
    - 3.2.1) Hello world
    - 3.2.2) Weihnachtsgrüße
    - 3.2.3) Quadratzahlen
    - 3.2.4) GGT-Berechnung
    - 3.2.5) Fakultät
    - 3.2.6) Dreieck
    - 3.2.7) Zahlenraten
- 4) Compilationstechniken
  - 4.1) Analyse
    - 4.1.1) Lexikalische Analyse
    - 4.1.2) Syntaktische Analyse
    - 4.1.3) Semantische Analyse
  - 4.2) Synthese
    - 4.2.1) Zwischencodeerzeugung
    - 4.2.2) Optimierung
    - 4.2.3) Codegeneration
- 5) Der LEAZY-Compiler
  - 5.1) Aufbau und Funktionsweise des Compilers
  - 5.2) Die C-Implementation
  - 5.3) Hinweise zur Benutzung
- 6) Bewertung
- 7) Kommentierter Compilerquelltext
- 8) Dokumentationsmaterial
- 9) Bibliographie

## 1) ZIELSETZUNG DER ARBEIT

Zweck dieser Arbeit ist es, in die Thematik und Techniken der Compilerkonstruktion einzuführen und anhand eines vollständig im Quelltext vorliegenden exemplarischen Compilers darzustellen. Dieser Compiler übersetzt die eigens für diese Facharbeit entwickelte Programmiersprache LEAZY in völlig rechnerunabhängiges ANSI-C. Da auch der Compiler selbst in ANSI-C geschrieben und 100% systemunabhängig programmiert wurde, ist es problemlos möglich, ihn auf alle Maschinen mit verfügbarem ANSI-C Compiler ohne Änderung zu portieren und so ein lauffähiges LEAZY-System zu erhalten (vgl. Abb. 5.2.1).

Als Hilfsmittel zur Erstellung dieser Facharbeit diente ein Atari ST 520+ Computer (mit Motorola 68000 Prozessor) und folgende Software: Ein ANSI-C Compiler, eine Textverarbeitung und ein Zeichenprogramm für das Manuskript sowie ein Public-Domain XREF-Tool.

## 2) EINLEITUNG

Diese Arbeit unterliegt folgender Gliederung: Nachdem oben ein kurzer Abriß des Zwecks dieser Arbeit gegeben wurde, werden in diesem einleitenden Kapitel zunächst einige grund-sätzliche Fragen geklärt. Es schließt sich Kapitel 3 an, wo die Sprache LEAZY formal definiert wird, unterstützt durch mehrere Beispiele vollständiger Programme. Kapitel 4 erläutert Kompilationstechniken im Allgemeinen. Den speziel- len Aufbau des LEAZY-Compilers beschreibt Kapitel 5 und erläutert ausführlich seine Bedienung. Kapitel 6 enthält eine zusammenfassende Bewertung, gefolgt vom C-Quelltext (Kapitel 7) sowie der Kreuzreferenzliste, den Strukto-grammen, usw. (Kapitel 8). Eine Bibliographie zum Thema Compilerbau schließt die Arbeit ab.

### 2.1) Was ist ein Compiler?

Compiler (dt. Übersetzer) nennt man Programme, deren Funktion darin besteht, ein *Quellprogramm* (engl. source program), das in einer *Quellsprache* geschrieben ist, in eine *Zielsprache* (engl. destination language) zu Übersetzen. Zusammen mit dem Betriebssystem (engl. operating system, OS) und den Hilfsprogrammen (engl. Utilities) bilden die Compiler die sog. Systemsoftware eines Rechners. Innerhalb der Klasse sprachverarbeiten- der Programme stellen Compiler den Höhepunkt einer langjährigen Forschung und Entwicklung dar (Abb. 2.1.1). Im Gegensatz zu Interpretern, die jeweils eine Anweisung lesen und die dadurch symbolisierten Aktionen in Form von Maschinenbefehlen direkt ausführen und die man sich daher als "Simultandolmetscher" vorstellen kann, nehmen Compiler einmalig eine Übersetzung des gesamten Programms vor, das dann sehr schnell ablaufen kann, da zur Laufzeit keine Interpretation (Zuordnung: Anweisung->symbolisierter Code) mehr erfolgen muß. Abb. 2.1.2 zeigt eine Modelldarstellung (sogenanntes "T- Modell", [23]) eines in der Sprache L<sub>c</sub> geschriebenen Compilers N, der ein in der Sprache L<sub>s</sub> geschriebenes Programm SRC in die Sprache L<sub>d</sub> übersetzt (und als DST ausgibt).

Compilerkonstruktion ist eines der bedeutendsten Gebiete der Informatik, da sich viele Probleme sehr leicht lösen lassen, sobald man sich sprachlicher Ansätze bedient, um dann bewährte Techniken aus der theoretischen Informatik (Automatentheorie und Formale Sprachen) anzuwenden. Aus diesem theoretischen Wert folgt natürlich auch ein weitgefächterter praktischer Nutzen, da Compilerbautechniken in Übersetzern von Programmiersprachen, automatischen Beweisprogrammen, syntaxgesteuerten Editoren, Textsatzsystemen, Quell- textformatierern (z.B. im UNIX-Kommando cb), und vielen anderen Programmen eingesetzt werden können. Sogar das Finden von Fehlern in gedruckten Schaltungen und genetische Mustererkennung wurde schon mit diesen Techniken automatisiert.

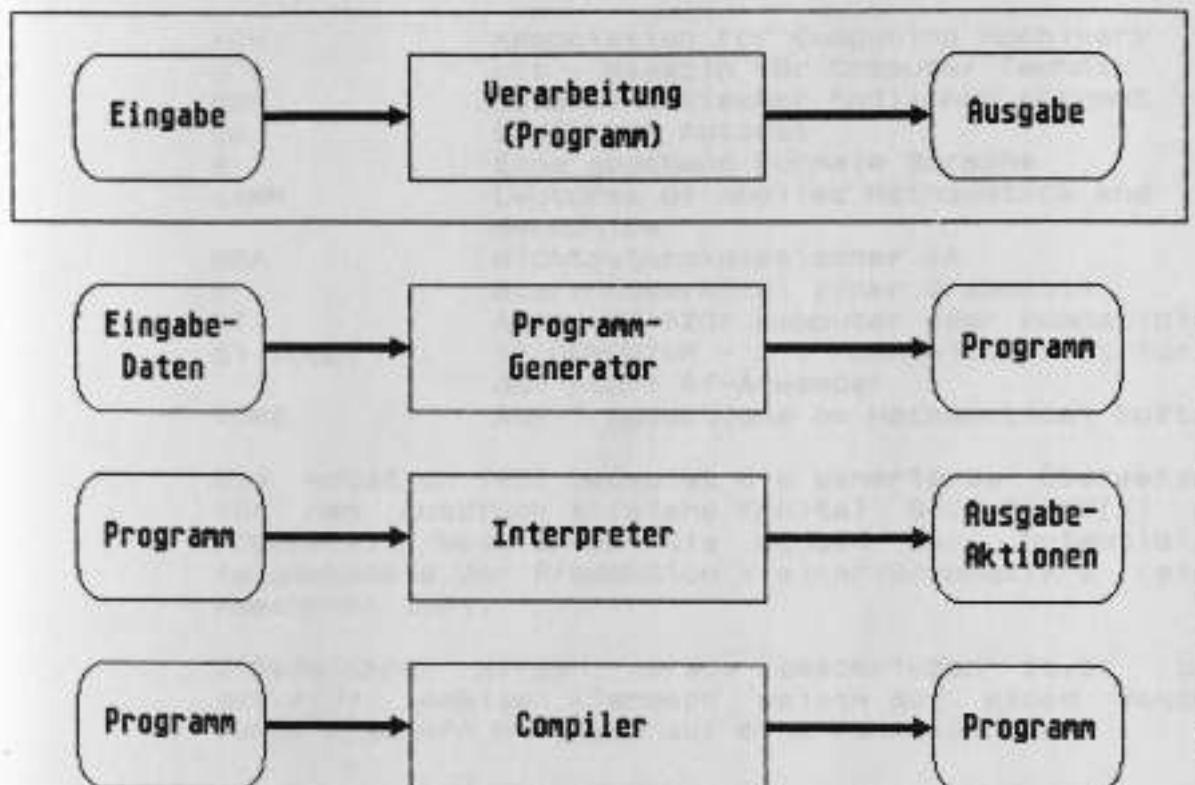


Abb. 2.1.1) Sprachverarbeitende Systemprogramme

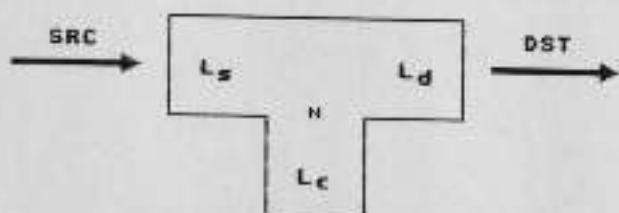


Abb. 2.1.2) T-Modell eines Compilers

## 2.2) Verwendete Notationen, Abkürzungen und Konventionen

In der vorliegenden Arbeit werden unter anderem folgende Abkürzungen verwendet (alphabetisch geordnet):

ACM	Association for Computing Machinery
c't	c't - Magazin für Computer Technik
DEA	Deterministischer Endlicher Automat
EA	Endlicher Automat
$\Sigma$	Eine gegebene Formale Sprache
LAMM	Lectures of Applied Mathematics and Mechanics
NEA	Nichtdeterministischer EA
S	Startnonterminal einer Grammatik
ST	Atari ST 520+ Computer oder kompatibler
ST Comp.	ST COMPUTER - Die Fachzeitschrift für den Atari ST-Anwender
TOMS	ACM Transactions on Mathematical Softw.

Die Notation  $T(x)$  bedeutet die generierte Übersetzung für den Ausdruck  $x$  (siehe Kapitel 8). FIRST( $x$ ) und FOLLOW( $x$ ) bezeichnen die Mengen der potentiellen Folgesymbole der Produktion  $x$  einer Grammatik  $\Sigma$  (siehe Abschnitt 2.3).

C-Bezeichner werden *kursiv* geschrieben (z.B. *p[]*, *error()*), eckige Klammern weisen auf einen Vektor, runde Klammern hingegen auf eine Funktion hin.

### 2.3) Formale Sprachen

Def. 2.2.1: Eine Menge  $A = \{ z_1, z_2, \dots, z_{n-1}, z_n \}$  von Zeichen heißt Alphabet.

Es wird hier nichts über die Zeichen selbst ausgesagt, weil hier lediglich der strukturelle Aspekt eine Rolle spielt (vgl. 'Punkt' in der Geometrie).

Def. 2.2.2: Eine endliche Sequenz  $f$  von Zeichen mit der Form  $z_1 z_2 z_3 \dots z_n$  aus einem Alphabet  $A$  heißt String über  $A$ .

Def. 2.2.3: Seien  $N$  und  $T$  zwei disjunkte Mengen von Nonterminalsymbolen bzw. Terminalsymbolen,  $P$  eine Menge von Produktionen (Stringersetzungsrregeln nach Emil Post) der Form

$$\delta \rightarrow \delta_1 ; \delta_2 ; \dots ; \delta_n$$

und  $S$  ein Startsymbol (Axiom).

Dann heißt ein Quadrupel  $\xi = (N, T, P, S)$  eine formale Grammatik.

Beispiel: Eine Grammatik für Telefonnummern lautet:

$\xi_1 = (N_1, T_1, P_1, \text{telnum})$  mit

$N_1 = \{ \text{telnum}, \text{land}, \text{ort}, \text{num} \}$   
 $T_1 = \{ '.', '(', ')', '/', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' \}$

Die Menge  $P_1$  enthält folgende Produktionen:

```
telnum ::= [ [ land '.' ] ort ] num ;
land   ::= '(' num ')';
ort    ::= num '/';
num   ::= '0' | '1' | '2' | '3' | '4' |
          '5' | '6' | '7' | '8' | '9' |
          num ;
```

Gewöhnlich genügt es, nur die Produktionen anzugeben, sofern man vereinbart, daß

- das erstgenannte Nonterminal als Startsymbol  $S$  fungiert
- alle Terminalen sich von ihrer Form her von den Nonterminalen abheben (Einschluß in '...')

Korrekte Telefonnummern wären z.B.

"(049) 06343/1677", "110", "089/321608";

Beispiele für falsche Nummern wären

"101-221", "\*339#", "(001) 20", "(1)2/3".

Def. 2.2.4: Ein String  $f$  heißt Satz einer Grammatik  $\mathcal{G}$  dann und nur dann, wenn er nur Terminal-symbole enthält und aus dem Startsymbol  $S$  durch Anwendung von Produktionen aus  $P$  hergeleitet werden kann.

Beispiel: Eine einfache Grammatik für arithmetische Ausdrücke lautet:

```
expr      ::= term { addop term }
term      ::= factor { mulop factor }
factor    ::= number | '(' expr ')'
addop    ::= '+' | '-'
mulop    ::= '*' | '/'
number   ::= '1' | '2' | '3' | '4' ...
```

Der Ausdruck  $2+3*4$  ist durch obige Regeln formal herleitbar:

```
expr => term addop term
=> term '+' term
=> factor '+' term
=> number '+' term
=> '2' '+' term
=> '2' '+' factor mulop factor
=> '2' '+' factor '*' factor
=> '2' '+' number '*' factor
=> '2' '+' '3' '*' factor
=> '2' '+' '3' '*' number
=> '2' '+' '3' '*' '4'
```

Diese baumförmige Herleitung beweist die Zugehörigkeit des Ausdrucks  $2 + 3 * 4$  zur angegebenen Grammatik. Es handelt sich um eine Top-Down-Herleitung, da von der Wurzel des Baumes ausgegangen wurde. Solch eine Baumstruktur findet man auch bei natürlichen Sprachen, wie z.B. dem Englischen (Abb. 2.3.1). Compiler prüfen die syntaktische Korrektheit von Programmen, indem sie versuchen, eine Herleitung zu finden; daher kann man Compiler auch als automatische Beweisfinder interpretieren.

Def. 2.2.5: Die Menge aller Sätze bzgl. einer Formalen Grammatik heißt Formale Sprache  $\mathcal{L}$ .

Durch unendlich wiederholtes Anwenden der Produktionen kann man unendlich viele Sätze herleiten. Die Gesamtheit aller herleitbaren Sätze einer Grammatik heißt Sprache zu dieser Grammatik.

Def. 2.2.6: Zwei Grammatiken heißen äquivalent genau dann, wenn die durch sie erzeugte Sprachen gleich sind.

So wie Scanner als Implementierungen Endlicher Automaten gesehen werden können, können Parser als Implementierungen von Stackautomaten gesehen werden, die Formale Sprachen erkennen können. Der Linguist Noam Chomsky [1956] erforschte Sprachen und teilte sie in folgende Klassen ("Chomsky-Hierarchien") ein:

- Typ 0: keine Einschränkungen (natürlich auch Leerproduktionen erlaubt)
- Typ 1: bei der Ableitung entstehen länger werdende Folgen von Satzformen (kontextsensitive Gramm.)
- Typ 2: Kontextfreie Grammatiken
- Typ 3: reguläre (einseitig lineare) Grammatiken

Für die praktische Verwendung sind lediglich die letzten beiden Klassen von Bedeutung: Kontextfreie Grammatiken werden benutzt, um die Syntax von Programmiersprachen zu beschreiben (vgl. 4.2.2) und reguläre Grammatiken lassen sich effizient für die Lexikalische Analyse nutzen (siehe Abb. 4.1.1 und Abb. 4.1.2).

Diese Darstellung Formaler Sprachen kann bedingt durch den Umfang dieser Arbeit nur an der Oberfläche kratzen. Deshalb sei an dieser Stelle auf Hopcroft [1979] und Peters [1974] verwiesen.

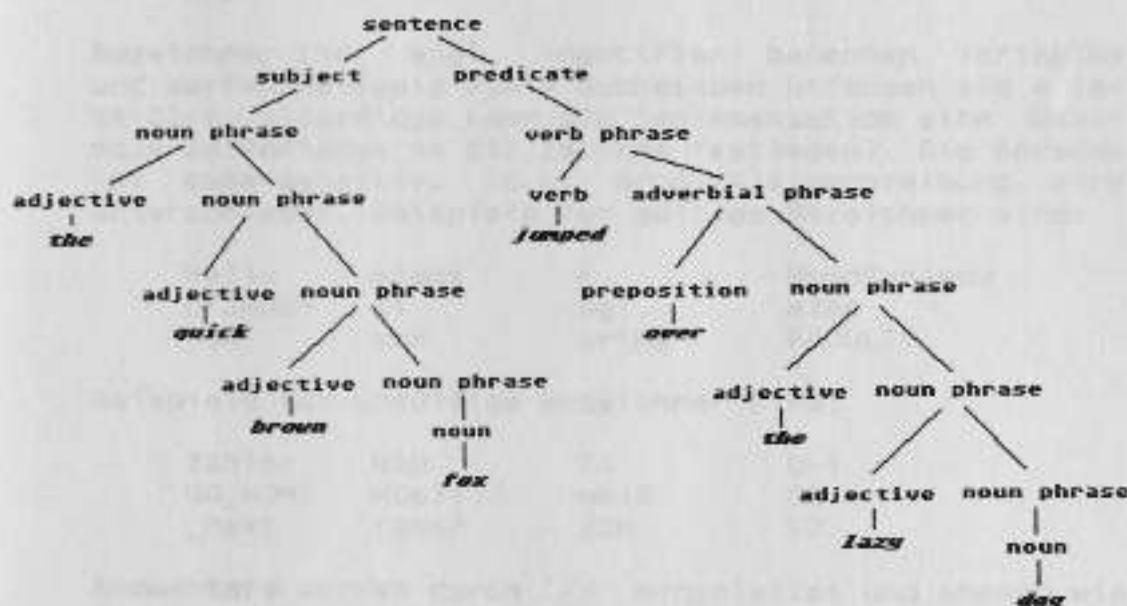


Abb. 2.3.1) Struktur des englischen Satzes "the quick brown fox jumped over the lazy dog." [11]

### 3) DIE PROGRAMMIERSPRACHE LEAZY

#### 3.1) Sprachdefinition

##### 3.1.1) Lexikalische Definitionen

Die **Schlüsselwörter** (engl. keywords) von LEAZY müssen in Kleinschrift eingegeben werden und lauten wie folgt:

clr	continue	cr	dec
do	else	end	exit
for	inc	if	loop
pop	print	push	quit
request	set	stop	swap
then	trace	untrace	while

Schlüsselworte in LEAZY sind nicht reserviert, d.h. man darf Variablen ohne Einschränkungen gleichlautend zu Schlüsselworten benennen. Ein Beispiel:

```
if (if > then) then
    set set = then * else
else
    set end = 2-end
end
```

Bezeichner (id, engl. identifier) benennen Variablen und dürfen beliebig viele Buchstaben umfassen (id = [a-zA-Z]+; allerdings kann die Implementation eine maximale Zeillänge  $\geq 512$  Zeichen festlegen). Die Sprache ist case-sensitiv, d.h. Groß-/Kleinschreibung wird unterschieden. Beispiele für gültige Bezeichner sind:

Hello	alpha	F	HomoSapiens
DFJHDDF	dT	GgT	else
FOR	xxx	prime	FACULTY

Beispiele für ungültige Bezeichner sind:

Zähler	H2O	T4	G-1
GO_HOME	Müsli	weiß	Ö1
_next	!\$%&/	22H	V0

Kommentare werden durch '//' eingeleitet und ebenso wie Anweisungen durch die Zeilenbegrenzung ('\n', ASCII: 13) beendet (anstelle z.B. von ';' in Pascal).

Zahlenkonstanten haben die Form num=[-]?[0-9]+, es existieren nur ganzzahlige Variablen und Konstanten, aber diese sind mit einer minimalen Breite von 32 Bit definiert (signed long integer), d.h. es sind Werte von mindestens -2147483648 bis +2147483647 möglich.

Zeichenkettenkonstanten (strlit, engl. string literal) werden durch Doppel-Anführungsstriche ("...") begrenzt, die natürlich im Literal selbst nicht vorkommen können.

### 3.1.2) Syntax in EBNF (Erweiterte Backus-Naur-Form):

```
S      ::= { [ label ] [ statem ] '\n' } ;
label ::= '<' id '>' ;
statem ::= 'cr' | 'quit' | request | print | set
           | if | 'continue' label | for | push | pop
           | 'exit' | 'stop' | loop | while | 'trace'
           | 'untrace' | inc id | dec id | cir id ;
print ::= 'print' (strlit | expr) { strlit | expr } ;
request ::= 'request' id { ',' id } ;
loop  ::= 'loop' S 'end' ;
while ::= 'while' '(' expr ')' 'do' S 'end' ;
set   ::= 'set' id '=' expr { ',' id '=' expr } ;
for   ::= 'for' id '=' '(' expr '..' expr ')' 'do'
           S 'end' ;
push  ::= 'push' (id : expr) { ',' (id : expr) } ;
pop   ::= 'pop' id { ',' id } ;
if    ::= 'if' '(' expr op expr ')'
           ( 'then' S [ 'else' S ] 'end'
             | 'continue' label ) ;
op   ::= '<' | '>' | '=' | '<=' | '>=' | '#' ;
expr ::= term { addop term } ;
term ::= factor { mulop factor } ;
factor ::= '(' expr ')' | (num | id) ;
addop ::= '+' | '-' ;
mulop ::= '*' | '/' | '%' ;
```

Erläuterungen zur Erweiterten BNF-Notation finden sich in Kapitel (2.3), bei Wirth [1985], Kopp [1988] und Aho et al. [1986].

### 3.1.3) Semantik

<b>clr</b>	Setzt Variablen auf null
<b>continue</b>	Unbedingte Verzweigung (Sprung)
<b>cr</b>	Setzt den Cursor an den Anfang einer neuen Zeile
<b>dec</b>	Erniedrigt den Wert einer Variable um 1
<b>for..do..end</b>	Wiederholungsschleife mit Zähler
<b>if..continue</b>	Bedingter Sprung
<b>if..then..end</b> <b>if..then..else..end</b>	Bedingte Ausführung von Anweisungsfolgen und Mehrfachverzweigung
<b>inc</b>	Erhöht den Wert einer Variablen um 1
<b>loop..exit..end</b>	Bedingungslose Schleife (Endlosschleife; Abbruch mit 'exit')
<b>push</b>	Wert auf Stack legen
<b>pop</b>	Wert vom Stack holen und speichern
<b>print</b>	Audrücke und Strings ausgeben
<b>quit</b>	Beendigung des Programms
<b>request</b>	Zahl von Tastatur einlesen
<b>set</b>	Zuweisung
<b>stop</b>	Synonym zu 'quit'
<b>swap</b>	Vertauschen von Variableninhalten
<b>text</b>	Zeichenkette ausgeben
<b>trace</b>	Automatische Ablaufverfolgung (Trace-Modus) einschalten: Es werden zusätzliche Anweisungen für Quasi-Source-Code-Debugging mitkompliiert: Beim Ablauf wird jede LEAZY-Anweisung angezeigt und auf Drücken der RETURN-Taste gewartet (Einzel-schrittmodus)

**untrace** Trace-Modus ausschalten (genau wie 'tron' keine ausführbare Anweisung, sondern ein Compiler-Pragma)

**while..do..end** Offene ('abweisende') Schleife

Es folgt die Bedeutung der Operatoren, wobei Klammern hierarchisch höher stehen als '\*', '/', '%', die wiederum vor '+' und '-' rangieren. Die geringste Priorität haben die Vergleichsoperatoren.

( und ) Klammern zum Festlegen der Auswertungsreihenfolge

+

Addition

-

Subtraktion

\*

Multiplikation

/

Ganzzahlige Division (DIV)

%

Rest der Division (MOD)

>

Größer als

>=

Größer als oder gleich

<

Kleiner als

<=

Kleiner als oder gleich

=

"ist gleich"-Relation oder Zuweisung (je nach Kontext)

#

ungleich

### 3.2) PROGRAMMBEISPIELE

Folgende Beispielprogramme in LEAZY sollen dazu dienen, sich an die Syntax der Sprache zu gewöhnen:

```
//  
//      Demotext ausgeben - von Jochen Leidner  
//  
print "Hello world"  
cr
```

Programm 3.2.1: HELLO.LZY, Das obligatorische Programm

```
//  
//      Tannenbaum Zeichnen - von Jochen Leidner  
//  
print "Tannenbaum"  
cr  
cr  
print "Bitte Länge und Breite eingeben (z.B. 3 5): "  
request x, y  
cr  
print "Jahr (z.B. 1991): "  
request year  
cr  
for a = (1..x) do  
    for b = (1..y) do  
        for u = (1..40 - y - b) do  
            print "  
        end  
        set c = 1  
        while (c <= b * 2 - 1) do  
            print "*"  
            inc c  
        end  
        cr  
    end  
end  
for b = (1..x) do  
    for a = (1..38 - y) do  
        print "  
    end  
print "***"  
cr  
end  
cr  
print " I wish You a merry Christmas & a wonderful "  
print year + 1, "!"  
cr  
cr  
print "                Jochen Leidner"  
cr
```

Programm 3.2.2: XMASTREE.LZY wünscht Frohe Weihnachten

```
//      Quadratzahlen - von Jochen Leidner
//  
print "Quadratzahloberechnung"  
cr  
cr  
print "Zahlensbereich: 1.."  
request m  
cr  
for i = (1..m) do  
    print i * i, "  
end  
print "..."  
cr
```

Programm 3.2.3: SQUARES.LZY berechnet Quadratzahlen

```
//      GGT - von Jochen Leidner
//  
print "Größter gemeinsamer Teiler"  
cr  
cr  
print "Bitte m und n eingeben: "  
request m, n  
set x = m, y = n  
while (x # y) do  
    if (x > y) then  
        set x = x - y  
    else  
        set y = y - x  
    end  
end  
cr  
print "GgT(", m, ", ", n, ") = ", x  
cr
```

Programm 3.2.4: GCD.LZY: Größter Gemeinsamer Teiler

```
//          Fakultät - von Jochen Leidner
//  
print "Fakultät von "
request num
set n = num
cr
set f = 1
while (n > 0) do
    set f = f * n
    dec n
end
print num, " ! = ", f
cr
```

Programm 3.2.5: FACULTY.LZY berechnet n-Fakultät

```
//          Dreieck - von Jochen Leidner
//  
print "Gleichschenkliges Dreieck"
cr
cr
print "Hypothenuse: "
request hypothenuse
cr
print "*"
cr
for i = (2..hypothenuse-1) do
    print "*"
    if (i <= hypothenuse / 2) then
        for j = (1..i-1) do
            print ""
        end
    else
        for j = (1..hypothenuse-i) do
            print ""
        end
    end
    print "*"
    cr
end
print "*"
cr
```

Programm 3.2.6: TRIANGLE.LZY zeichnet Dreiecke

```
//          Zahlenraten - von Jochen Leidner
//  
set rand = 63434533  
print "Zahlenraten"  
cr  
cr  
print "Ich denke mir eine Zahl zwischen 1 und 100,"  
print " die Du erraten mußt."  
cr  
loop  
    cr  
    clr versuche  
    loop  
        set rand = rand * 314159263 + 1877  
        set rand = rand % (2 * 1073741824)  
        set zahl = rand % 100  
        if (zahl > 0) then  
            if (zahl <= 100) then  
                exit  
            end  
        end  
    end  
    loop  
    cr  
    print "Dein Tip: "  
    inc versuche  
    request tip  
    if (tip = zahl) then  
        print "*** Bravo, Du hast die Zahl in "  
        print versuche, " Versuchen erraten !! ***"  
        cr  
        exit  
    else  
        if (tip > zahl) then  
            print "zu groß"  
        else  
            print "zu klein"  
        end  
    end  
end
```

Programm 3.2.7: ZRATEN.LZY ist ein Glücksspiel

## 4) COMPILATIONSTECHNIKEN

Bei der Beschreibung der Compilationstechniken teilt man den Compiler oftmals in Analyse- und Syntheseteil ein (auch Front- und Back-End genannt):

*"Compiler sind in der Regel sehr umfangreiche Programme von hoher Komplexität. Es liegt daher nahe, ihre Aufgabenstellung durch Modularisierung in einzelne überschaubare Phasen zu unterteilen." [25]*

Beide Phasen (engl. passes) werden in weitere Teilphasen unterteilt, die man sich allesamt als sequentiell ablaufend vorstellt, wobei die Ausgabe jeder Phase als Eingabe für die darauffolgende dient (Abb. 4.1); in der Praxis jedoch werden alle Phasen meist verschränkt abgewickelt.

### 4.1) Analyse

#### 4.1.1) Lexikalische Analyse

Die Lexikalische Analysephase eines Compilers, die der sogenannte Scanner durchführt, ist derjenige Teil, der das Quellprogramm tokenisiert, d.h. den Zeichenstrom des Inputs zu elementaren Einheiten zusammenfaßt: Buchstabenfolgen werden als Bezeichner aufgefaßt und in die Symboltabelle eingetragen oder als Schlüsselworte interpretiert, aufeinanderfolgende Ziffern werden als Zahlen aufgefaßt; Dabei werden auch Kommentare und Leerräume entfernt. Scanner liefern gewöhnlich Tupel zurück. Tupel sind Paare aus jeweils einem Token und einem Attributwert, z.B. stellt

(id, "hallo")

ein Tupel dar, bestehend aus dem Token id, das gewöhnlich für eine scannerinterne Zahl steht, die die lexikalische Klasse Bezeichner symbolisiert, sowie einem Zeiger auf den Symboltabelleneintrag des Variablenbezeichners "hallo".

Aus den beiden Anweisungen

```
print ( 2 * (1+1) ), "= 4" // Ausgabe
set hallo = 1           // Zuweisung
```

wird so die Tupel-Sequenz

```
(print, -) (obra, -) (num, 2) (mulop, *) (obra, -)
(num, 1) (addop, +) (num, 1) (cbra, -) (cbra, -)
(comma, -) (strlit, "= 4") (newln, -) (set, -)
(id, "hallo") (eq1, -) (num, 1).
```

Zur Vereinfachung der Scannerprogrammierung wurden sogenannte Scannergeneratoren entwickelt, wie z.B. das UNIX-Kommando Lex von Lesk [1975] (zur Einführung siehe Weber [1989]). Sie erlauben das einfache und bequeme Erstellen von komplexen Scannern, ohne den Benutzer mit Verwaltungsoverhead zu belästigen. Die Benutzung solcher Generatoren erhöht auch die Wartbarkeit, jedoch wird die Portabilität eventuell stark eingeschränkt, da solche Compiler immer von der Verfügbarkeit des Generators abhängig sind.

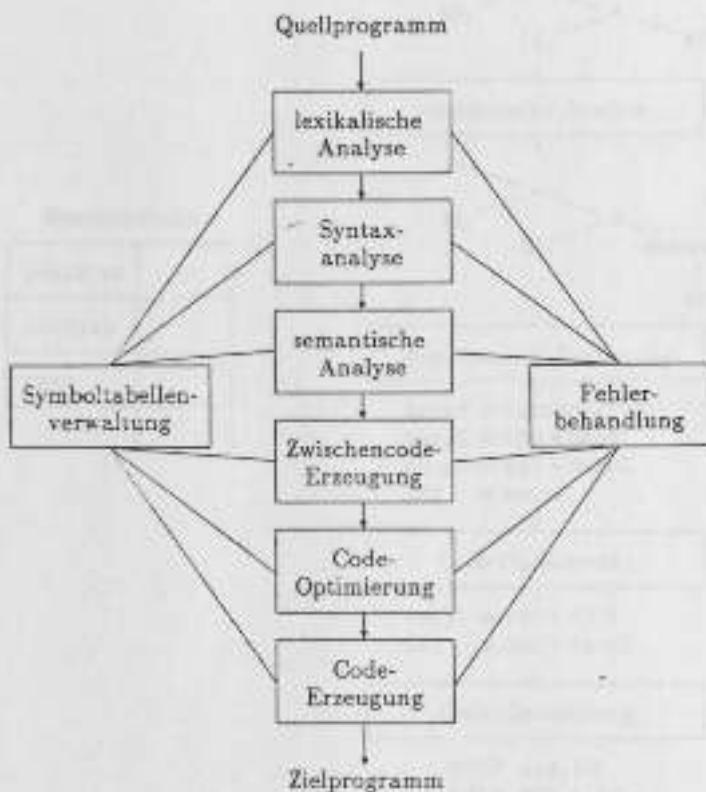


Abb. 4.1) Compilerstruktur nach dem Phasenmodell [1]

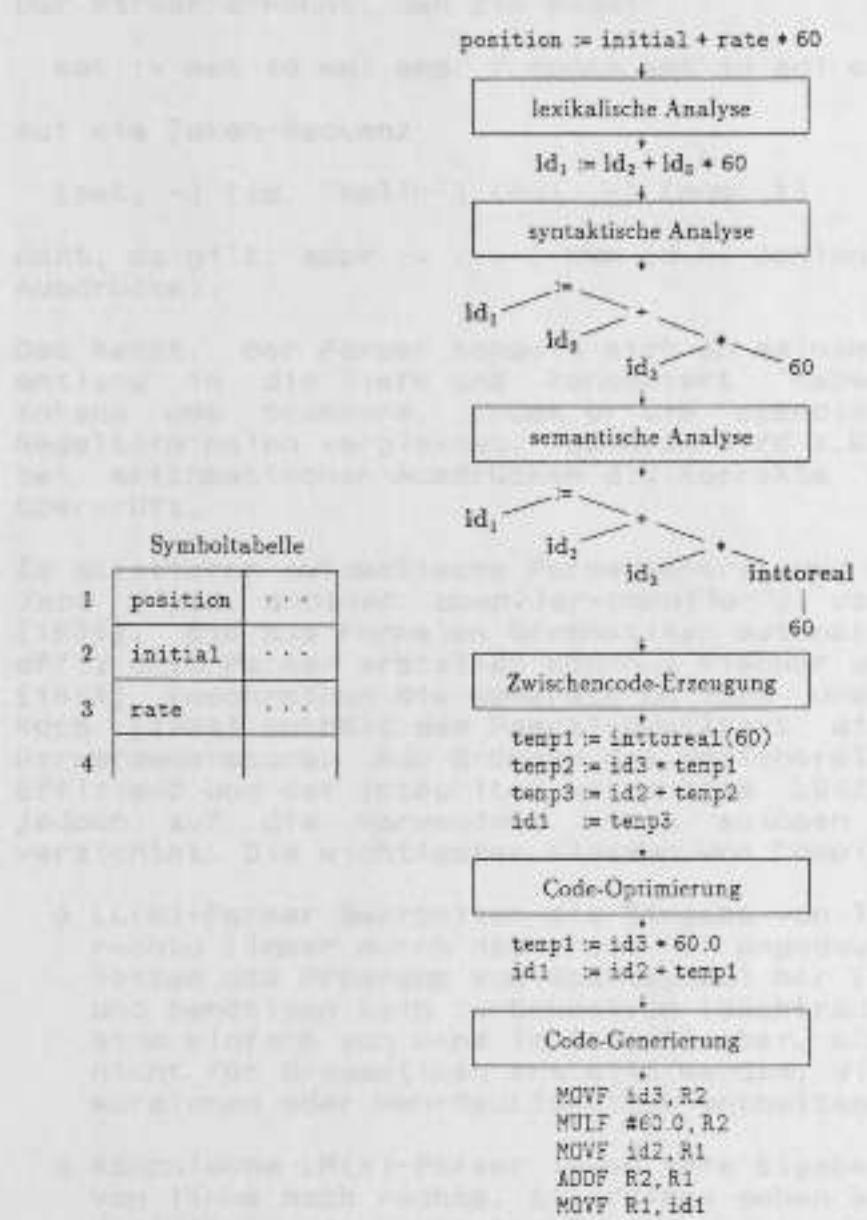


Abb. 4.2) Phasenstruktur und Datenfluß [1]

#### 4.1.2) Syntaktische Analyse

Die Tokens, die vom Scanner geliefert werden, dienen dem Parser als Eingabe, der ihnen in der Syntaktischen Analysephase eine hierarchische Struktur unterlegt, die benutzt wird, um (explizit oder wie bei LEAZY implizit) einen Syntaxbaum zu generieren (das entspricht der Herleitung aus Kap. 2.3):

Der Parser erkennt, daß die Regel

```
set := set id eql expr { comma set id eql expr } ;
```

auf die Token-Sequenz

```
(set, -) (id, "hallo") (eql, -) (num, 1)
```

paßt, da gilt:  $\text{expr} := \dots \mid \text{num}$  (d.h. Zahlen sind auch Ausdrücke).

Das heißt, der Parser hängt sich an seinem Regelbaum entlang in die Tiefe und konsumiert "nebenbei" die Tokens des Scanners, indem er sie ständig mit den Regelterminalen vergleicht. Dadurch wird z.B. implizit bei arithmetischen Ausdrücken die korrekte Klammerung überprüft.

Es existieren automatische Parsergeneratoren wie *UNIX-Yacc* ("yet another compiler-compiler") von Johnson [1975], die aus Formalen Grammatiken automatisch hocheffiziente Parser erstellen können. Fischer und LeBlanc [1991] beschreiben die Generatoren Yacc und *ScanGen*. Kopp [1988] enthält den Pascal-Quelltext eines LALR-Parsergenerators. Aus Gründen des Speicherplatzes, der Effizienz und der Integrität wurde beim LEAZY-Compiler jedoch auf die Verwendung eines solchen Programms verzichtet. Die wichtigsten Klassen von Compilern sind:

- o LL(k)-Parser bearbeiten die Eingabe von links nach rechts (immer durch das erste 'L' angedeutet); sie leiten das Programm vom Startsymbol her (Top-down) und benötigen kein Zurücksetzen (Backtracking). Sie sind einfach von Hand implementierbar, können aber nicht für Grammatiken erstellt werden, die Linksrekursionen oder Mehrdeutigkeiten enthalten.
- o Kanonische LR(k)-Parser lesen ihre Eingabe ebenfalls von links nach rechts, allerdings gehen sie bei der Herleitung vom konkreten Programm aus, und versuchen, durch Anwendung der Produktionen von Rechts nach Links (daher auch das 'R' im Namen) Terminalgruppen (sogenannte "Handles") zu Nonterminalen zu reduzieren, bis sie an das Startsymbol gelangen. Daher nennt man sie auch Shift-Reduce- oder Bottom-Up-Parser. LR(k)-Parser können Grammatiken aller geläufiger Programmiersprachen erkennen; leider ist es fast unrealisierbar, sie von Hand zu entwickeln, doch existieren Compilergeneratoren für alle hier genannten Klassen.

- o SLR( $k$ )-Parser ("Simple Left-Right") sind einfacher zu implementierende Bottom-Up-Parser, aber auch nicht so mächtig wie kanonische LR. Dennoch erlauben sie es, eine größere Klasse von Grammatiken als LL( $k$ ) zu analysieren, bei besserer Effizienz als kanonischer LR.
- o LALR( $k$ )-Parser ("Look-Ahead Left-Right") arbeiten Bottom-Up, liegen von der Stärke her zwischen SLR und kanonischer LR und ermöglichen effiziente Analysen (daher sind sie in Parsergeneratoren am weitesten verbreitet; auch Yacc generiert LALR(1)-Parser).

Die Konstante  $k$  gibt jeweils die Anzahl der Zeichen an, die der Parser maximal vorrausschauen muß (Look-Ahead), um zu entscheiden, welche Produktion zu benutzen ist; fehlt die Angabe, so wird ein Look-Ahead von 1 angenommen. In der Praxis werden in den meisten Fällen LL(1)- und LALR(1)-Techniken verwendet.

#### 4.1.3) Semantische Analyse

Die Semantische Analyse spürt syntaktisch korrekte, aber sinnlose Konstrukte auf, wie z.B. Realwerte als Array-Indices oder Typunverträglichkeiten. So wird beispielsweise in FORTRAN IV ein Fehler angezeigt, wenn einer Integer-Variablen eine komplexe Zahl zugewiesen wird.

LEAZY fängt fast alle semantischen Fehler bereits während der Syntaxanalyse ab (z.B. set a = "Hallo"); dies ist eine weitere positive Eigenschaft aller Compiler, die Teile des Scannprozesses in die Syntaxanalyse verlagern.

## 4.2) Synthese

### 4.2.1) Zwischencodeerzeugung

Im Zuge der Syntax-/Semantikanalyse wird gewöhnlich eine Zwischendarstellung des Quellprogramms erzeugt, deren Komplexität zwischen Quellsprache und Zielsprache liegt. Solche Zwischencodes ermöglichen maschinenunabhängige Optimierungen, verbessern die Compilerstruktur und erleichtern die Fehlersuche im Compiler. Es eignen sich vor allem folgende Darstellungen:

- UPN-Code
- Syntaxbäume / GAGs
- Quadrupel

- o UPN-Code (Umgekehrte Polnische Notation, nach dem polnischen Mathematiker Lukasiewicz so benannt) ist eine Darstellung mathematischer Ausdrücke, wobei der Operator nach den Operanden geschrieben wird. Dadurch werden Klammern überflüssig. So wird der Infix-Ausdruck

$$(4 - 7) * 2$$

in UPN als

$$4 \ 7 \ - \ 2 \ *$$

notiert.

UPN-Code kann leicht durch einen Stapelmaschinen-Interpreter abgearbeitet werden. Ein Beispiel für UPN-Code ist der an der *University of California at San Diego* (UCSD) entwickelte P-Code; er wurde als virtuelle Maschinensprache entworfen, um darauf aufbauend das hardwareunabhängige UCSD-P-OS zu entwickeln, ein Public-Domain-Betriebssystem für Mikrocomputer mit einem Pascalcompiler, der selbst wieder P-Code erzeugt (daher das P wie Pascal im Namen).

Anmerkung des Verfassers: Das UCSD-System ist ob seines antiquierten Editors zu recht gefürchtet und sollte tunlichst gemieden werden.

- o Syntaxbäume sind abstrakte Bäume, deren Blätter Operanden sind, und deren Kanten die zu den Operanden gehörenden Operationen enthalten. GAGs (Gerichtete Azyklische Graphen) enthalten die gleiche Information; der Unterschied besteht darin, daß in GAGs gemeinsame Teilausdrücke nur einmal dargestellt werden (siehe dazu die aus [1] entnommene Abb. 4.2.1.1).

Man beachte, daß man UPN-Code erhält, wenn der entsprechende Syntaxbaum in Post-Order durchlaufen wird. Knuth [1973a] gibt hierzu folgenden rekursiven Algorithmus, der ebenso kurz wie prägnant formuliert ist:

"We can visit the node in (...) postorder by doing (...) three steps: (...)

1. Traverse the left subtree
2. Traverse the right subtree
3. Visit the root" [15]

o Quadrupelcode besteht aus Anweisungen der Form

Erg := Arg<sub>1</sub> Op Arg<sub>2</sub>

das heißt, ein Quadrupel enthält einen Operator Op, der auf seine beiden Operanden Arg<sub>1</sub> und Arg<sub>2</sub> angewandt wird, wonach das Ergebnis in Erg gespeichert wird. Daher wird Quadrupelcode auch Drei-Adress-Code genannt. Er eignet sich hervorragend zum Einsatz in Hochoptimierenden Compilern.

```
t1 := uminus(c)      { Vorzeichen umdrehen }
t2 := t1 * b
t3 := t2 + t2
a := t3              { weise das Ergebnis a zu }
```

Abb. 4.2.1.2) Bsp. für Quadrupel aus Abb. 4.2.1.1)

#### 4.2.2) Optimierung

Die Aufgabe der Optimierungsphase ist es, durch Transformationen auf dem Zwischencode dessen Effizienz (idealerweise zu einem Optimum) zu steigern, ohne daß sich das Verhalten des Programmes dadurch ändert (Funktionserhaltende Transformationen). Dieses Optimum wird nur selten und unter großen Anstrengungen erreicht. Der optimierte Code aus Abb. 4.2.1.2 ist in Abb. 4.2.2.1 dargestellt. Neben der Ersparnis eines temporären Symbols wurde die Addition t2+t2 durch a<<1 ersetzt, was äquivalent ist, da ein Links-Shift um 1 Bit einer Multiplikation mit 2 entspricht, die wiederum äquivalent mit der Addition eines Wertes zu sich selbst ist, aber schneller ausgeführt werden kann.

```
t1 := uminus(c)
a := t1 * b
a := a << 1          { Links-Shift um 1 Bit }
```

Abb. 4.2.2.1) Der optimierte Code nach Abb. 4.2.1.2)

Rechnerabhängige Optimierungen sind verbesserte Registervergabe und die Ausnutzung von Spezialbefehlen einer Maschine.

Was die Konstruktion einer Rechnung aus den Prinzipien der Syntax und Semantik erfordert.

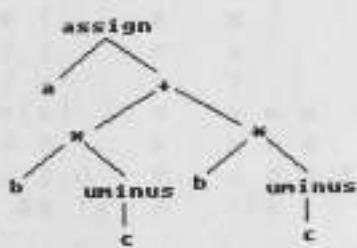
Die Konstruktion einer Rechnung ist eine Kette von Schritten, die die Struktur der Rechnung aufbauen. Diese Schritte sind:

1. Die Konstruktion des Syntaxbaums (Syntaxbaumkonstruktion).

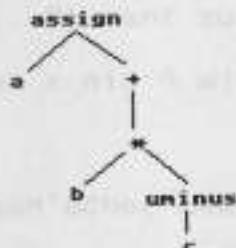
2. Die Konstruktion des Semantikbaums (Semantikbaumkonstruktion).

3. Die Konstruktion des Ausdrucksbaums (Ausdrucksbaumkonstruktion).

Die Konstruktion des Syntaxbaums ist die Basis für alle anderen Schritte. Sie besteht aus der Konstruktion eines Baums, der die Struktur der Rechnung darstellt. Der Baum hat einen Wurzelknoten, der die Wurzel der Rechnung ist. Dieser Knoten verzweigt sich in verschiedene Kinderknoten, die wiederum in weitere Kinderknoten verzweigen. Dieser Prozess wird fortgesetzt, bis alle Kinderknoten Blattknoten sind. Ein Blattknoten ist ein Knoten, der keine Kinderknoten mehr hat. Ein Blattknoten kann ein Terminalsymbol oder ein Nichtterminalsymbol sein.



(a) Syntaxbaum



(b) GAG

Abb. 4.2.1.1) Die Darstellung von  $a := b * (-c) + b * (-c)$

Wird beispielsweise beim Motorola 68000 Prozessor eine Funktion aufgerufen, sollte statt

```
MOVE.W #$10, -(SP)
MOVE.W #$200, -(SP)
MOVE.W #3, -(SP)
JSR    FUNC
```

besser

```
MOVE.L #$02000010, -(SP)
MOVEQ.W #3, -(SP)
BSR    FUNC
```

erzeugt werden.

Rechnerunabhängige Optimierungen sind die Auswertung konstanter Ausdrücke wie  $2\pi$  zur Kompilationszeit, Ersetzung teurer Instruktionen durch billigere, Entfernung redundanten Codes bis hin zur kompletten Datenflußanalyse, die Endlosschleifen und nicht erreichbare Codesegmente aufspürt und in der Lage ist, ganze Programme wegzooptimieren (Wachtmeister [1989]). Diagramm 4.2.2.2 zeigt geläufige funktionserhaltende, effizienzverbessernde Transformationen.

code	->	$T^\circ(\text{code})$	
$x + 0$	=	$x$	(neutral. Element zu '+')
$0 - x$	=	$-x$	
$x * 1$	=	$x$	(neutral. Element zu '*')
$x / 1$	=	$x$	
$x / x$	=	1	(nur wenn $x$ nie 0 wird !)
$\text{sqr}(x)$	=	$x * x$	
$x * 2$	=	$x + x$	
$x / 2.0$	=	$x * 0.5$	
$!x \&& !y$	=	$!(x    y)$	(De Morgan'sches Gesetz)

Abb. 4.2.2.2) Gängige verbesserrnde Transformationen

Rechnerunabhängige Optimierungen werden breiträumig in Aho et al. [1986] behandelt. Wachtmeister [1989] erläutert das Modell des Datenflußrechners mit seinen hervorragenden Optimierungsmöglichkeiten. Rechnerabhängige Optimierungen für den mc68k-Prozessor von Motorola behandelt Seimet [1989].

#### 4.2.3) Codegeneration

Die Codegeneration erzeugt schließlich aus der (ggf. optimierten) Zwischencodedarstellung des Programms den Zielcode in Form von

- relativem (linkbarem) Objektcode
- absolutem Objektcode
- Assemblercode
- einem Programm in einer Hochsprache

Wichtige Eigenschaften eines guten Codegenerators sind Korrektheit und Effizienz des erzeugten Codes. Daher ist die Codegeneration meist die einzige Phase eines Compilers, die nicht rechnerunabhängig entwickelt werden kann. Dies trifft jedoch nicht für LEAZY zu, da hier von der letzten Möglichkeit Gebrauch gemacht wird: Der Compiler erzeugt Programme in der portablen Hochsprache C und ist daher ohne Änderungen voll Übertragbar. Abb. 4.2.3.1 zeigt den Code, den ein Codegenerator für die Prozessoren MOS-6502 bzw. Motorola 68000 erzeugen könnte.

MOS 6502	MC 68000
LDA C	move c, d7
JSR UMINUS	not d7
LDX B	muls b, d7
JSR IMULT	asl d7
ASL	move d7, a
STA A	

Abb. 4.2.3.1) Der erzeugte Code aus Abb. 4.2.2.)

Beträchtlich erschwert wird eine gute Codegeneration, wenn die Zwischendarstellung des Programms hohen Portabilitätsansprüchen genügen muß, z.B. zur Benutzung auf mehreren Maschinen und/oder für mehrere Sprachen; man spricht dann von einer UNCOL (Universal Code Language).

In der Praxis kann die C-Codegeneration am LEAZY-Compiler studiert werden. Da [1] anführt, daß sehr selten Compilerquellcode veröffentlicht wird, sei noch bemerkt, daß [8] den kompletten Quellcode eines Entwicklungssystems für eine Untermenge der Sprache C enthält (aber leider nur für den ehemaligen de-facto-Standard nach Kernighan/Ritchie [1978]).

## 5) DER LEAZY-COMPILER

### 5.1) Aufbau und Funktionsweise des Compilers

Für Grammatik 3.1.2 wurde ein datengesteuerter LL(0)-Parser für alle Konstrukte außer Ausdrücke (Expressions) gebaut; für letztere wurde ein LL(1)-Parser mit rekursivem Abstieg (engl. predictive top-down recursive-descend parser) entworfen. Dies war möglich, da die Grammatik so konzipiert wurde, daß für alle Produktionen

$p := t_1 X ; t_2 Y ; \dots ; t_n Z ;$

gilt: Alle  $t_k$  für  $k=(1..n)$  bilden eine Untermenge von T, die nur Schlüsselworte enthält. LL(k)-Parser besitzen die sog. "variable prefix"-Eigenschaft, d.h. sie können Syntaxfehler zum frühestmöglichen Zeitpunkt erkennen.

Der Compiler (Abb. 5.1.1) ist ein 1-Pass-Compiler, jedoch übernimmt nicht wie sonst üblich der Parser, sondern die vom Scanner gelesenen Daten die Steuerung des Kompilationsvorganges. Außerdem liefert der Scanner nicht wie üblich Tupel aus Tokens und Attributen, sondern einen Funktionszeiger auf eine Parserfunktion und einen globalen Zeiger cp auf einen String, der die aktuelle Quelltextzeile enthält (ohne Zwischenraum und Kommentare).

Da in LEAZY Variablen nicht deklariert werden müssen, wohl aber in der Zielsprache ANSI-C, wären grundsätzlich zwei Durchläufe über den Quellcode erforderlich. Um dies zu vermeiden, wird der Zielcode in zwei Dateien (Code + Deklarationen) abgelegt, was eine Geschwindigkeitssteigerung (um Faktor 2) bringt: Während der Kompilation werden alle Bezeichner und Labels in drei Symboltabellen gespeichert; erstere werden nachträglich in die Deklarationsdatei ("\*.h") geschrieben.

Die Codeerzeugung erfolgt syntaxgesteuert und wird als Seiteneffekt der vom Scanner aktivierten Parserprozeduren erzeugt und direkt gespeichert. Daher ist eine Codeoptimierung nicht ohne weiteres möglich, was aber auch im Rahmen der kurzen für das Projekt zur Verfügung stehenden Zeit kaum möglich gewesen wäre.

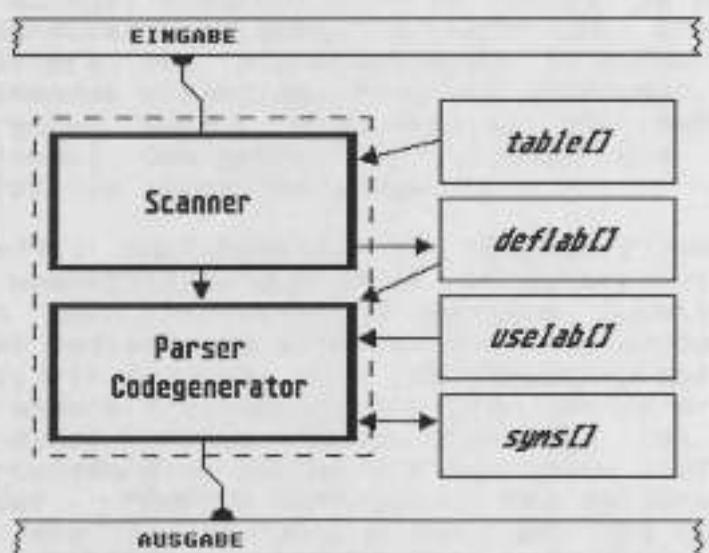


Abb. 5.1.1) Der LEAZY-Compiler im Modell

## 5.2) Die C-Implementation

Der Compiler (Quelltext in "LEAZY.C") wurde in der Systemprogrammiersprache "C" (American National Standard ANS X3.159-1989) geschrieben, die von Dennis M. Ritchie in den siebziger Jahren entwickelt wurde (Kernighan und Ritchie [1978]). C dient auch als Zielsprache, so daß der Compiler als Komponente in einem Programmiersystem gesehen werden sollte (Abb. 5.2.1).

Vorab einige grundsätzliche Bemerkungen: Die Quelltextdatei des Compilers ("LEAZY.C") enthält eigentlich zwölf verschiedene Versionen des Programms; durch #define-Preprozessor-Kommandos läßt sich eine Version selektieren. Zur Auswahl stehen folgende Schalter, die (fast) beliebig kombiniert werden können:

INTERACTIVE	-	Benutzeroberfläche einbinden
PROTOCOL	-	Protokollfähige Version
DEBUG	-	Fehlersuchhilfen einbinden ( <i>dbgcall()</i> , <i>dbgsyms()</i> ...)
UNIX	-	UNIX – Betriebssystem benutzen

Zum Ablauf: Zunächst wird in *main()* je nach Anzahl der Kommandozeilenparameter entweder der Interaktionsmodus aktiviert und die Benzershell aufgerufen oder im Batchmodus direkt das Programm übersetzt, dessen Name übergeben wurde. Wird mehr als ein Parameter (zzgl. Optionen) übergeben, so erfolgt die Ausgabe einer Hilfszeile (Funktion *usage()*).

*ishell()* implementiert die Menüoberfläche und besteht im wesentlichen aus einer Fallunterscheidung, die je nach gewähltem Menüpunkt weitere Funktionen aufruft. Dabei bedient sie sich der Funktion *pattex()*, die durch *exec()* in der Lage ist, Musterersetzung vorzunehmen und andere Programme aufzurufen (child process calls). Da *exec()* keine Standardfunktion ist, enthält der Sourcecode eine portable Alternative (ggf. vor *exec()* #undef \_\_PROCESS einfügen). Des Weiteren wird *show()* von *isHELL()* benutzt, um die Codefiles der Arbeitsdatei auf dem Bildschirm auszugeben. *options()* behandelt das gleichnamige Menü. *help()* gibt eine Auswahlhilfe zum jeweils aktuellen Menü aus.

*compile()* ist nun bei der Kompilation für das Öffnen und Schließen aller Dateien, die Speicherverwaltung und die Erzeugung des Init-Codes verantwortlich: Der Quelltext des zu kompilierenden Programms wird anschließend vom Scanner (Funktionen *leazypyrg()* und *getln()*) zeilenweise in den Puffer *p[]* gelesen, jegliche Leerräume (außer in Stringliteralen) entfernt und alle Kommentare ignoriert.

Jetzt wird das erste Schlüsselwort der Zeile in der Tabelle `table[]` ermittelt (die hierfür zuständige Funktion `keyword()` verwendet einen extra für LEAZY entwickelten, kollisionsfreien Hashingalgorithmus) und es erfolgt ein Sprung über den im betreffenden Eintrag gespeicherten Funktionszeiger (Abb. 5.2.2 und 5.2.3). In dieser Funktion (`docxxxxx()`, bzw. `dofxxxxx()`) erfolgt die übrige Lexikalische Analyse, der Syntax-Check und die Codeerzeugung für das ermittelte Konstrukt. Die Fehlerbehandlung erfolgt gegebenenfalls durch `error()`. Nachdem mit `ocdec1()` die für C nötigen Deklarationen geschrieben wurden, werden durch `closeio()` alle Dateien geschlossen.

```
struct tabentry /* Keyword-Tabelleneintrag */
{
    char *lex;
    int len;
    FUNPTR cfunc;
    FUNPTR ffunc;
};
```

Abb. 5.2.2) Die Hauptdatenstruktur des Compilers

Soll während dieses Ablaufs ein arithmetischer Ausdruck (engl. expression) analysiert werden, so wird die Funktion `expr()` gerufen, die zusammen mit `term()` und `factor()` den Recursive-Descend-Parser implementiert. Man beachte, daß mit Ausnahme obiger drei Funktionen für die Ausdrucksbehandlung alle Rekursionen aus dem Programm entfernt wurden. Bei Anweisungen wie z.B. `if` geschah dies beispielsweise durch Einführung eines Zählers `stage` für die Schachtelungstiefe.

Die Funktion `lookup()` speichert in der Symboleitabelle `syms[]` alle entdeckten Variablenbezeichner in HASHTABSIZ dynamischen linearen Listen, die über eine Hashtabelle verbunden sind (vergleiche Abb. 5.2.3). Der Hashwert wird durch `hashpjw()` berechnet, eine modifizierte Funktion (frei nach [1]), die ursprünglich von P.J. Weinberger in K&R-C entwickelt wurde.

Zwei weitere Symboleitabellen namens `deflab[]` und `uselab[]`, die ebenso aufgebaut sind, dienen zur Aufnahme der definierten bzw. benutzten Labels. Die Funktionen `checklabs()` und `defp()` werden von `compile()` benutzt, um mit Hilfe dieser Tabellen unbenutzte und undefinierte Labels aufzuspüren, d.h. diese Funktionen implementieren die semantische Analyse. `outsyms()` gibt die gesamte Symboleitabelle auf dem Bildschirm aus. `freetab()` gibt den Speicherplatz für eine Tabelle wieder frei.

Wallwitz und Stäudel [1989] beschreiben einen kommerziellen Pascal-nach-C-Transformator. Fischer [1985] beschreibt einen in Pascal geschriebenen Compiler für die experimentelle Sprache DOIT. Wirth [1986] und Kopp [1988] implementieren die Sprache PL/O, die Wirth entwickelte.

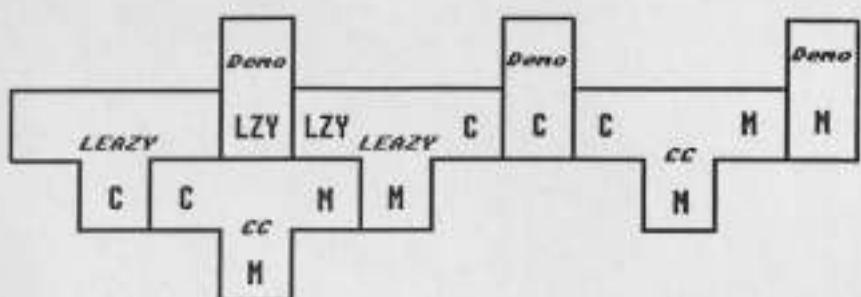


Abb. 5.2.1) Der LEAZY-Compiler als Systemkomponente

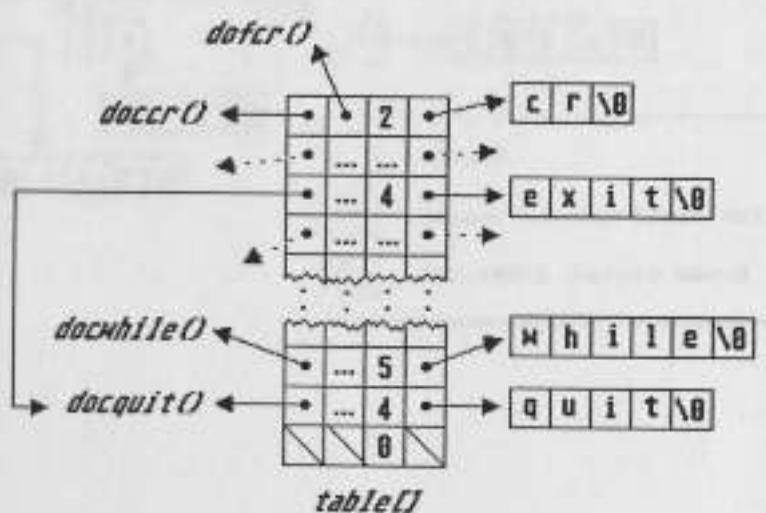


Abb. 5.2.2) Aufbau der Schlüsselworttabelle

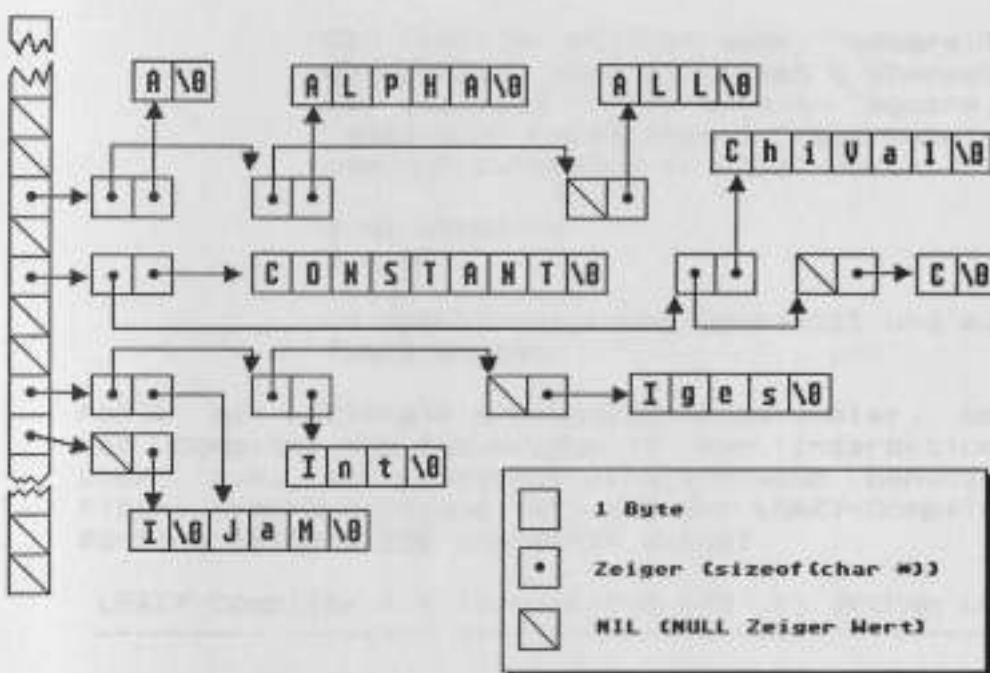


Abb. 5.2.3) Aufbau der Symboltabelle

### 5.3) Hinweise zur Benutzung

Der Compiler befindet sich vollständig in der Datei "LEAZY.TOS" (UNIX: "leazy"); Als Kommandozeilenparameter kann der Name einer Quelldatei angegeben werden (die Extension ".lzy" wird automatisch angefügt). Dieses Programm wird dann in Dateien gleichen Namens kompiliert (für das Zielprogramm werden die Extensions ".c" und ".h" benutzt).

Usage: LEAZY [ <filename> [-t;-n] ]

Der optionale Switch "-n" bewirkt, daß lediglich eine Syntaxanalyse erfolgt, also kein Code generiert wird. Durch Angabe des wahlfreien Schalters "-t" wird der TRACE-Modus aktiviert (bis zum nächsten untrace). "-n" und "-t" schließen sich wechselseitig aus.

Beispiel: Man gibt über die UNIX-C-Shell folgende Zeile ein (gefolgt von CR):

```
$ leazy square
```

Der Compiler wird geladen, "square.lzy" eingelesen, von LEAZY nach C übersetzt und der Zielcode in die Dateien "square.c" und "square.h" geschrieben. Diese Dateien können nun (wiederum in UNIX) durch

```
$ cc square.c  
$ a.out
```

in Maschinensprache übersetzt und ausgeführt werden.

Fehlt der optionale Kommandozeilenparameter, so geht der Compiler vom Batchmodus in den Interaktionsmodus über, d.h. es erscheint eine einfache Benutzeroberfläche (Voraussetzung ist, daß der LEAZY-Compiler mit #define INTERACTIVE übersetzt wurde):

```
LEAZY Compiler 1.4 (C)opyright 1991 by Jochen Leidner
```

---

```
[W]orkfile name: "workfile"  
[E]dit [C]ompile / Sy[N]tax check  
[M]ake [R]un  
[X]ecute [O]ptions  
[H]elp [S]how code  
[Q]uit
```

```
Ok> _
```

Workfile name:

Hier läßt sich eine Datei als Arbeitsdatei festlegen, so daß nicht bei jeder Kompilation die erneute Eingabe eines Namens erforderlich wird. Als Voreinstellung wurde 'workfile' gewählt. Eine Extension darf nicht mit eingegeben werden (Es wird immer ".lzy" automatisch angehängt).

Edit:

es wird ein externer Editor geladen und gestartet, um ein Bearbeiten der Arbeitsdatei zu ermöglichen.

Compile:

Die Arbeitsdatei wird in C übersetzt. Im Fehlerfall wird eine Fehlermeldung ausgegeben, der Editor aufgerufen und der Cursor an die fehlerhafte Stelle gesetzt.

Syntax check:

Das Programm in der Arbeitsdatei wird auf syntaktische Korrektheit hin überprüft.

Make:

Wie 'Compile', jedoch werden anschließend ein separater C-Compiler und Linker aufgerufen, um ein ausführbares Programm zu erzeugen.

Run:

Wie 'Make', jedoch wird das erzeugte Programm anschließend direkt gestartet.

Execute:

Das zuletzt erzeugte ausführbare Programm mit dem Namen der Arbeitsdatei wird geladen und gestartet.

Options:

In diesem Untermenü lassen sich die benötigten externen Programme wie C-Compiler, Linker und Editor mit Pfadangaben und Parametern festlegen:

LEAZY Compiler 1.4 (C)opyright 1991 by Jochen Leidner

[S]yntax tree output is off

[E]ditor	file name:	'ED.PRG'..
	parameters:	'%f.lzy %1 %c'..
[C]ompiler	file name:	'TC\TCC.PRG %'..
	parameters:	'-A -HTC\INCL'..
[L]inker	file name:	'TC\TLINK.PRG'..
	parameters:	'-O=%f.TOS LI'..

[T]race is off [H]elp

[Q]uit ( back to main menu )

Ok> \_

Es folgt ein Beispiel für eine mögliche Konfiguration unter UNIX:

```
[E]ditor
    file name: vi <cr>
    parameters: %f.lzy %l %c <cr>

[C]ompiler
    file name: cc <cr>
    parameters: %f.c -o %f.tos <cr>

[L]inker
    file name: <cr>
    parameters: <cr>
```

Bei den Parametern steht

- o '%f' für die aktuelle Arbeitsdatei (ohne Extension), z.B. 'workfile'
- o '%l' für die Zeile, in der ein Fehler entdeckt wurde, z.B. '126'
- o '%c' für die Spalte des Fehlers, z.B. '35'
- o '%' für ein einzelnes '%',

d.h. es wird beispielsweise automatisch bei

```
EDIT %%n %f.lzy %f.c -L %l -C %c
```

der Aufruf

```
EDIT %n test.lzy test.c -L 126 -C 35
```

ausgeführt.

Im 'Options'-Menü können auch die Ausgabe des traversierten Syntaxbaumes sowie der TRACE-Modus ein- bzw. ausgeschaltet werden (ersteres nur, wenn der LEAZY-Compiler mit #define PROTOCOL übersetzt wurde).

#### Show:

Der zuletzt für die aktuelle Arbeitsdatei erzeugte C-Zielcode wird ausgegeben. Nach jeder Seite wird auf den Druck von <RETURN> gewartet (Abbruch durch 'q' + <RETURN>). Wurde noch kein neuer Code erzeugt, so erscheint eine Warnung.

#### Help:

Durch 'Help' wird das Titelmenü erneut ausgegeben.

#### Quit:

Die interaktive Umgebung wird verlassen (Programmende).

Um die Arbeit mit der integrierten Benutzerumgebung (und der protokollfähige Version) zu veranschaulichen, folgt hier eine ausführliche Beispielsitzung:

Beispiel:

Starten des Programms LEAZY.TOS (o.ä.)

Es erscheint das Hauptmenü (siehe oben)

Ok> e <Return> eingeben

Der Editor wird geladen und das Programm kann eingegeben werden:

```
///
//      Countdown - von Jochen Leidner
///
print "Countdown starts: "
for counter = (0..10) do
    print 10-counter, "... "
end
print "running!! "
```

Z.B. durch Drücken von <F10> kann nun das Programm gespeichert und der Editor verlassen werden (Dies hängt vom verwendeten Editor ab, notfalls <HELP> drücken!).

Ok> r <Return>

Es erfolgt jetzt die Kompilation nach C, anschließend werden der C-Compiler und der Linker aufgerufen, die ein ausführbares Programm erzeugen, daß dann gestartet wird:

LEAZY Compiler 1.4 (C)opyright 1991 by Jochen Leidner

COMPILER LISTING Sun Feb 11 22:38:17 1991  
=====

\*\*\*\* compiling workfile.lzy..

```
1 //  
2 //      Countdown - von Jochen Leidner  
3 //  
4 print "Countdown starts: "  
5 for counter = (0..10) do  
6     print 10-counter, "... "  
7 end  
8 print "running!! "  
9
```

SYMBOL TABLE  
=====

counter

\*\*\*\* writing declaration file.  
\*\*\*\* no errors found in 267 ms.

Executing 'workfile'..

Countdown starts: 10,... 9,... 8,... 7,... 6,... 5,... 4,...  
3,... 2,... 1,... 0,... running!!

Ok> s <Return>

Es erscheint das Übersetzte Programm in C:

```
#include <stdio.h>  
#include <stdlib.h>  
  
long _counter;  
  
void main(void)  
{  
    printf("%s", "Countdown starts: ");  
    for(_counter=0L; _counter<=10L; _counter++)  
    {  
        printf("%ld", 10L-_counter);  
        printf("%s", "... ");  
    }  
    printf("%s", "running!! ");  
    exit(0);  
}
```

Nun kann durch Eingabe von

Ok> q <Return>

die Arbeit mit LEAZY beendet werden.

## 6) BEWERTUNG

Der Compiler konnte trotz der kurzen zur Verfügung stehenden Zeit (Zum Vergleich: Die Erstellung des ersten FORTRAN-Compilers dauerte nach [1] volle 18 Mannjahre) komplett fertiggestellt werden und übersetzte alle beigefügten sowie einige weitere Testprogramme fehlerfrei. Diese Tatsache ist unter anderem auch den Untersuchungen zur Projektplanung aus Brooks [1975] zu verdanken. Auf einem Atari ST mit 8 Mhz (Motorola 68000 Prozessor) wurde ein 18 KBytes großes Benchmarkprogramm mit 1000 Zeilen representativem LEAZY-Sourcecode in 0,885 Sekunden auf syntaktische Korrektheit überprüft (incl. Zeit für Ram-Disk-Ein/Ausgabe). Damit beträgt die Analysegeschwindigkeit des Parsers etwa

67797 Zeilen pro Minute,  
( 210012 Bytes/s oder 1,2 MB/min )

Der Compiler übersetzt Programme von LEAZY nach C mit einer Geschwindigkeit von über 15000 Zeilen pro Minute. Der Compiler selbst ("LEAZY.TOS") ist 23057 Bytes groß (Non-interaktiv: 18343 Bytes; mit Protokollfähigkeit: 25785 Bytes); der Quelltext bringt 56801 Bytes in 2593 Zeilen 'auf die Waage'. Eine Analyse der Codequalität ist schlecht möglich, weil sie von dem verwendeten Back-End-Compiler und dessen Bibliotheken abhängt (zu den Techniken der Laufzeitanalyse sei hier nur auf Detert [1988] und Bailey und Jones, TOMS 1 [1975], S. 196 ff. verwiesen).

## 7) DOKUMENTIERTER COMPILERQUELLTEXT

```
*****  
*  
*          LEAZY.C  
*  
*      Experimenteller LEAZY Compiler 1.4  
*  
*  
*      (C)opyright 1990, 1991 - Written by Jochen Leidner  
*  
*      Dieser Compiler wurde im Rahmen der Facharbeit  
*      "Compilerbau in Theorie und Praxis"  
*      im Bereich Mathematik/Informatik am Staatlich Kooperativen  
*      Gymnasium Bad Bergzabern erstellt.  
*  
*          Jochen Leidner  
*          Friedensstrasse 10  
*          W6748 Bad Bergzabern  
*          Bundesrepublik Deutschland  
*          Tel.: (06343) 1677  
*  
*****  
  
#include <stdio.h>                                /* Standard-Headerfiles */  
#include <stdlib.h>  
#include <string.h>  
#include <ctype.h>  
#include <time.h>  
#include <process.h>                               /* ( exec()-Routine ) */  
#include <setjmp.h>  
#include <errno.h>  
  
#define INTEGRATED                                     /* Integrierte Umgebung ein */  
  
#undef PROTOCOL                                      /* ( Syntaxbaum/Protokoll ) */  
#undef DEBUG                                         /* ( Debugging-Hilfe ) */  
#undef UNIX                                          /* ( '/' als Pfadtrenner ) */  
  
#define DEZ             10                           /* Basis des Dezimalsystems */  
#define ERR             11                           /* Error Return Number */  
#define L_PER_S          20                           /* Zeilen pro Seite (CRT) */  
#define LINEMAX         512                          /* Laenge des Zeilenpuffers */  
#define FNAMEMAX        128                          /* Laenge des Dateinamens */  
#define MATCHSTRLEN     12                           /* Laenge match()-Buffer */  
#define MAXLABEL        33                           /* Speicherlaenge f. Labels */  
#define BUFSRC          11                           /* Puffergroesse n * BUFSIZ */  
#define BUFDST          10                           /* fuer Source-, C- und */  
#define BUFHDF          5                            /* Headerfile */  
#define HASHTABSIZ       998                          /* Groesse der Hashtabelle */  
#define KEYTABSIZ        41                           /* (prim + 1 erforderlich) */  
#define HASHTABGROESSE  /* Schluesselwort- */  
                           /* Hashtabellengroesse */
```

```

#define NO          0           /* Logisches NEIN */
#define YES         1           /* Logisches JA */
#define EOLST       -1          /* Listenende - Markierung */
#define EVER        ;;          /* fuer Endlosschleifen */
#define ESTR         (char *)estr /* Leere Zeichenkette */
#define OPLEN        3           /* RELOP-Laenge + 1 */

#ifndef _QC
    #define EDITOR      "vised.exe" /* Default-Editor: PD Visual-Editor */
    #define CCOMP       "qcl.exe"   /* Default-Compiler: QuickC 2.5 */
    #define LINKER     ""
    #define EDPAR      "%f.lzy %1 %c"
    #define CCPAR      "/batch /Gs /I \\\qc25\\\include /J /Ox /W4" \
                     "/Za %f.c /link \\\qc25\\\lib\\slibce.lib /BA"
    #define LNPAR      ""
#else
    #ifdef UNIX
        #define EDITOR      "vi"
        #define CCOMP       "cc"
        #define LINKER     "date"
        #define EDPAR      "%f.lzy"
        #define CCPAR      "%f.c -o %f"
        #define LNPAR      ""
    #else
        #define EDITOR      "ED.PRG"
        #define CCOMP       "E:\\TC1\\\\TCC.PRG"
        #define LINKER     "E:\\TC1\\\\TLINK.PRG"
        #define EDPAR      "%f.LZY %1 %c"
        #define CCPAR      "-A -HE:\\\\TC1\\\\INCLUDE -K %f.C"
        #define LNPAR      "-O=%f.TOS E:\\\\TC1\\\\LIB\\\\TCSTA" \
                         "RT.O %f.O E:\\\\TC1\\\\LIB\\\\TCSTDLIB.LIB"
    #endif
#endif

#if !(__STDC__)
    #error leazy compiler requires ANSI-C compiler
#endif

#ifndef PROTOCOL
    #ifndef INTEGRATED
        #error PROTOCOL facility requires INTEGRATED to be defined
    #endif
#endif

```

```

/*
 * Deklaration der neuen Datentypen
 */
/*-----*/
typedef void (*FUNPTR)(void);           /* Typ: Zeiger auf Prozedur */

typedef struct symbol                 /* Typ: Symboltabellenentry */
{
    struct symbol *next;              /* Zeiger auf naechstes El. */
    char *lex;                      /* Zeiger auf Bezeichner */
} symbol_t;

typedef enum
{
    MENU_MAIN,                      /* Kennung fuer Hauptmenue */
    MENU_OPT                         /* Options-M. */
} context_t;

enum prgid
{
    IED,                            /* Kennung fuer Editor */
    ICC,                            /* C-Compiler */
    ILN,                            /* Linker */
    IEX,                            /* Programm */
    INONE                           /* nichts */
};

enum extid
{
    EXT_LZY,                         /* LZY-Dateierweiterung */
    EXT_C,                           /* C (C-Code) */
    EXT_H                            /* H (Headerfile) */
};

/*-----*/
/* Globale Variablen definieren und z.T. Initialisieren
*/
/*-----*/

FILE *fs, *fd, *fh;                  /* Quell- und Zielfile */

long lineno = 0;                    /* Zeilennummer */

int exitflag = NO;                 /* Abbruch? */
errocc = NO;                       /* Fehler aufgetreten? */
wprotect = NO;                     /* Fehlerhaftes File */
stackused = NO;                   /* Stack benutzt? */
swapused = NO;                    /* SWAP-Anweisung benutzt? */
stage, trace = NO;                /* Schachtelungstiefe */
quick = NO;                        /* Ablaufverf. mitkomp. ? */
valid = NO;                        /* Flag 'syntax check only' */
valid = NO;                        /* (S)how-Versionskontrolle */
interactive = NO;                 /* Interaktiv - Flag */

#endif PROTOCOL
    int syntax_tree = NO,
        depth;                      /* Syntaxbaum ausgeben? */
    /* PROTOCOL - Einrueckung */
#endif

```

```

static const char *ext[] = /* File Extensions */ {
    ".lzy",
    ".c",
    ".h"
};

#ifndef INTEGRATED

static char *parname[] = { "[E]ditor",
                           "[C]ompiler",
                           "[L]inker" },
    inp[82], /* Eingabezeile */
    fname[5][FNAMEMAX] = { EDITOR, /* Dateinamen */
                           CCOMP,
                           LINKER },
    patt [3][FNAMEMAX] = { EDPAR, /* Schablone */
                           CCPAR,
                           LNPAR },
    workfile[FNAMEMAX] = "workfile", /* Arbeitsdatei */
    param[3][FNAMEMAX]; /* Parameter */

static const char
*ret = "[return] ",
*head2 = "\t-----\n";

*menu = "\n\t\t[W]orkfile name: \"%s\"\n"
        "\n\t\t[E]dit          [C]ompile / Sy[N]tax check"
        "\n\t\t[M]ake          [R]un"
        "\n\t\t[X]ecute        [O]ptions"
        "\n\t\t[H]elp          [S]how code"
        "\n\n\t\t[Q]uit\n",
*opts = "\n\t\t%<s>\tfile name: '%.13s'..."
        "\n\t\t\tparameters: '%.13s'...",
*opts2 = "\n\n\t\t[T]race is %s"
        "\t\t[H]elp\n\n\t\t[Q]uit ( back to main menu )\n",
*on = "on",
*off = "off",
*fss = "%s: %s",
*prompt= "\nOk> ";

#endif PROTOCOL
    static char *opts1 = "\n\t\t[S]yntax tree output is %s\n";
#endif
#endif

```

```

static const char
*err_mem    = "not enough memory", /* 'Speicher voll'-Meldung */
*err_open   = "open error",      /* I/O-Fehlermeldung */
*err_op     = "unknown operator", /* falscher Operator */
*err_id     = "identifier expected", /* Bezeichner erwartet */
*err_num    = "number expected", /* Zahl erwartet */
*err_numid  = "number or identifier expected",
*err_strnt = "unterminated string", /* String nicht beendet */
*err_lab   = "label expected", /* Label erwartet */
*err_cont   = "'continue' or 'then' expected",
*err_else   = "unexpected 'else'", /* nichterlaubtes 'else' */
*err_endne  = "unexpected 'end'", /* nichterlaubtes 'end' */
*err_exit   = "'exit' only within loops",
*err_eolex  = "end of line expected",
*err_unst   = "unknown statement", /* unbekannte Anweisung */
*err_ext   = "I don't like extensions",
*err_fd    = "can't open code file",
*err_fh    = "can't open header file",
*err_end   = "'end' missing", /* Fehlendes 'end' */
*estr      = "",
*cname     = "LEAZY",
*head      = "LEAZY Compiler 1.4 "
                "(C)opyright 1991 by Jochen Leidner\n";

static char name[FNAMEMAX],           /* Dateiname */
          sline[LINEMAX];        /* Quelltextzeile */

static char *pext,                   /* Zeiger auf Extension */
          *p,                      /* Zeiger -> Src-Line-Start */
          *cp,                     /* Zeiger -> Quelltextpos. */
          *start,                  /* dto., aber ohne f. Space */
          *sname,                  /* Quelltextname */
          *fsbuf,                  /* Pufferzeiger fuer */
          *fdbuf,                  /* Source-, C- und */
          *fhbuf;                 /* Headerfile */

#endif INTEGRATED

static jmp_buf procenv,             /* Prozess-Environments f. */
          repenv;                /* Fehler-Spruenge */

#endif

static symbol_t *syms [HASHTABSIZ], /* Symboltafel */
               *deflab[HASHTABSIZ], /* Label-Definitionstab. */
               *uselab[HASHTABSIZ]; /* Tab. benutzter Labels */

static const size_t tabsiz = sizeof(syms);

```

```
-----*/  
/* Prototypen */-----*/  
  
int exec(char *pathname, char *cmdlstr, char *envstr, int *retcode);  
  
void error(const char *);  
void tab(int);  
void dtab(void);  
void match(const char *);  
void closeio(void);  
void dbgcall(FUNPTR);  
void dbgsyms(void);  
void outsyms(void);  
void freetab(symbol_t *[]);  
FUNPTR keyword(void);  
unsigned short hashpjw(register const char *);  
char *lookup(symbol_t *[], const char *);  
int getln(void);  
  
void op(void);  
char *id(void);  
void num(void);  
void numid(void);  
  
void relation(void);  
void factor(void);  
void term(void);  
void expr(void);  
  
void doclabel(char *);  
void doccr(void);  
void docprint(void);  
void docquit(void);  
void doccontinue(void);  
void docrequest(void);  
void docif(void);  
void docwhile(void);  
void docend(void);  
void docpush(void);  
void docpop(void);  
void docfor(void);  
void docelse(void);  
void docloop(void);  
void docexit(void);  
void docswap(void);  
void docset(void);  
void docclr(void);  
void docdec(void);  
void docinc(void);  
void doctrace(void);  
void docuntrace(void);  
  
void qop(void);  
void qid(void);  
void qidlist(void);  
void qnum(void);  
void qnumid(void);
```

```

void qrelation(void);
void qfactor(void);
void qterm(void);
void qexpr(void);
void qexprlist(void);

void dofdummy(void);
void dofcr(void);
void dofprint(void);
void dofquit(void);
void dofcontinue(void);
void dofrequest(void);
void dofif(void);
void dofwhile(void);
void dofend(void);
void dofpush(void);
void dofpop(void);
void doffor(void);
void dofelse(void);
void dofloop(void);
void dofexit(void);
void dofswap(void);
void dofset(void);

void leazyprg(void);
void ocdecl(void);
void defp(const char *);
void checklabs(void);
void compile(char *);
void pattex(int);
void help(context_t);
void options(void);
void show(void);
void ishell(void);
void usage(void);
int main(int, char *[]);

/*
/* Schluesselworttabelle mit assoziierten Funktionszeigern */
*/

const struct tabentry                                /* Keyword-Tabelleneintrag */
{
    char      *lex;
    int       len;
    FUNPTR   ffunc;
    FUNPTR   cfunc;
}
table[] =                                              /* Keyword-/FUNPTR-Tabelle */
{
    { "while",      5,  dofwhile,    docwhile   },
    { NULL,         0,  NULL,        NULL       },
    { "clr",        3,  qidlist,    docclr    },
    { NULL,         0,  NULL,        NULL       },
    { NULL,         0,  NULL,        NULL       }
};

```

```
{ NULL, 0, NULL, NULL },
{ "exit", 4, dofexit, docexit },
{ NULL, 0, NULL, NULL },
{ NULL, 0, NULL, NULL },
{ "for", 3, doffor, docfor },
{ NULL, 0, NULL, NULL },
{ "push", 4, qexprlist, docpush },
{ NULL, 0, NULL, NULL },
{ "if", 2, dofif, docif },
{ NULL, 0, NULL, NULL },
{ NULL, 0, NULL, NULL },
{ "request", 7, qidlist, docrequest },
{ NULL, 0, NULL, NULL },
{ "print", 5, dofprint, docprint },
{ "continue", 8, dofcontinue, doccontinue },
{ "dec", 3, qidlist, docdec },
{ NULL, 0, NULL, NULL },
{ "set", 3, dofset, docset },
{ NULL, 0, NULL, NULL },
{ NULL, 0, NULL, NULL },
{ NULL, 0, NULL, NULL },
{ "quit", 4, dofdummy, docquit },
{ "else", 4, dofelse, docelse },
{ NULL, 0, NULL, NULL },
{ "trace", 5, dofdummy, doctrace },
{ NULL, 0, NULL, NULL },
{ "untrace", 7, dofdummy, docuntrace },
{ NULL, 0, NULL, NULL },
{ "pop", 3, qidlist, docpop },
{ "loop", 4, dofloop, docloop },
{ NULL, 0, NULL, NULL },
{ "swap", 4, dofswap, docswap },
{ "inc", 3, qidlist, docinc }
```

```

{ NULL,          0,  NULL,      NULL      },
{ NULL,          0,  NULL,      NULL      },
{ "cr",          2,  dofdummy,  docr     },
{ NULL,          0,  NULL,      NULL      },
{ "end",         3,  dofend,    docend   },
{ NULL,          0,  NULL,      NULL      },
{ "stop",        4,  dofdummy,  docquit },
{ NULL,        EOLST,  NULL,      NULL      }
};

/*-----*/
/* error()
 */
/* Fehlermeldung ausgeben + terminieren
 */
/*-----*/

void error(register const char *msg)
{
    register exists = cp > p;

    printf( "%s%c%c\n%s: %s in line %ld, file \"%s\".\n",
            (exists ? start : ESTR),
            (exists ? '\n' : ' '),
            (int)(cp - start) + 1,
            (exists ? '^' : ' '),
            cname,
            msg,
            lineno,
            sname );

#define INTEGRATED
    if(interactive)                                /* Fehlerbehandlung */
    {
        closeio();
        errocc = YES;
        longjmp(procenv, ERR);
    }
    else
#endif
        exit(ERR);
}

#define PROTOCOL

/*-----*/
/* tab()
 */
/* n * 2 Leerzeichen ausgeben
 */
/*-----*/

void tab(register n)
{
    printf("%*c", n << 1, ' ');
    /* oder:  n <= 1;
               while(n--)
                   putchar(' '); */
}

```

```
/*
 * protocol()
 */
/* Syntaxbaum ausgeben (implizit)
*/
void protocol(const char *str)
{
    if(syntax_tree)
    {
        tab(depth++); /* Seiteneffekt !! */
        puts(str);
    }
#endif

/*
 * dtab()
 */
/* (stage + 1 Tabulatoren ('\t') in Zielfile schreiben
*/
void dtab(void)
{
    register n = stage + 1;
    while(n--)
        putc('\t', fd);
}

/*
 * match()
 */
/* Erzwinge str in der Eingabe
*/
void match(register const char *str)
{
    static char emsg[MATCHSTRLEN];
    register size_t len = strlen(str);

    if(!strncmp(cp, str, len)) /* str in der Eingabe? */
        cp += len; /* ok, dann weiter */
    else
    {
        *emsg = '\\';
        emsg[1] = '\0';
        strcat(emsg, str);
        strcat(emsg, " expected");
        error(emsg);
    }
}
```

```

/*
/* closeio()
*/
/* Schliesse alle Dateien
*/
*/

void closeio(void)
{
    fclose(fs);                      /* Quelltextdatei schliessen */
    if(wprotect)                     /* Wenn Schreibfehler.. */
        wprotect = NO;              /* Dann Flag loeschen + fertig */
    else                            /* Sonst: */
        if(!quick)
    {
        fputc('\n', fh);           /* Zeilenvorschub */
        fclose(fh);                /* Deklarationsdatei schliessen */
        fclose(fd);                /* C-Zieldatei schliessen */
    }
}

#ifndef DEBUG

#define D(s) { puts(s); getchar(); }      /* Debugging-Hilfsausgabe */

/*
/* dbgcall()
*/
/* Symbolische Debugging-Hilfe: rekonstruiert aus der Adresse
/* einer Funktion deren Quelltext-Namen
*/
*/

void dbgcall(FUNPTR f)
{
    register const struct tabentry *tp = table;
    static char flex[35];             /* static, sonst dangling reference */

    while(tp->len != EOLST)          /* Listenende? */
        if(f == (quick ? tp->ffunc : tp->cfunc))    /* Name finden */
            break;
        else
            ++tp;
    strcpy(flex, tp->lex);           /* bauen */
    printf("DEBUG-CALL do%c%s()\t", quick ? 'f' : 'c', flex);
    getchar();
}

```

```
/*
 *  dbgsyms()
 */
/* Debugging-Hilfe: Gibt die Symboleiste syms[] aus
*/
#define CR1 { printf(sympfmt, p->lex); \
    ++n; \
    if(n % 10UL == 0) \
        getchar(); \
} /* Warten auf <CR> */

void dbgsyms(void)
{
    register symbol_t **symp = syms;
    register const symbol_t **top = symp + HASHTABSIZ;
    register unsigned long n = 0UL;

    printf("\nsyms == 0x%p:\n", symp);
    while(symp < top) /* [0..HASHTABSIZ-1] */
    {
        static const char symfmt[] = "\"%s\"\t";
        register symbol_t *p = *symp++;

        if(p)
        {
            CR1;
            do
            {
                p = p->next;
                if(p)
                    CR1
                else
                    break;
            }
            while(p);
            puts("NIL");
        }
    }
    puts("\n");
}

#else
#endif PROTOCOL
```

```
/*
 *  outsyms()
 *
 /*  Symboltabelle syms[] ausgeben
 */
void outsyms(void)
{
    register symbol_t **symp = syms;
    register const symbol_t **top = symp + HASHTABSIZ;

    printf( "\n      SYMBOL TABLE"
            "\n      =====\n" );

    while(symp < top)                                /* [0..HASHTABSIZ-1] */
    {
        static const char symfmt[] = "%10.10s\t";
        register symbol_t *p = *symp++;

        if(p)
        {
            printf(symfmt, p->lex);
            do
            {
                p = p->next;
                if(p)
                    printf(symfmt, p->lex);
                else
                    break;
            }
            while(p);
        }
    }
    puts("\n");
}

#endif
#endif
```

```

/*
/*  freetab()
/*
/* Loescht Symboletabellen
/*
*/
void freetab(symbol_t *table[])
{
    register symbol_t **symp = table;
    register const symbol_t **top = symp + HASHTABSIZ;

    while(symp < top)           /* [0..HASHTABSIZ-1] */
    {
        register symbol_t *p = *symp++, *q;
        while(p)
        {
            free(p->lex);
            q = p;
            p = p->next;
            free(q);
        }
    }
}

/*
/* keyword()
/*
/* Prueft, ob aktuelles Symbol ein Schluesselwort ist. Benutzt
/* kollisionsfreies, speziell fuer LEAZY entwickeltes Hashing
/*
*/
FUNPTR keyword(void)
{
    register const unsigned short hval = ((*cp * *(cp + 1)) % 89) - 5;
    register const struct tabentry *tp = table + hval;
    register short r = tp->len;

    if(r && !strncmp(tp->lex, cp, r))
    {
        cp += r;
        return quick ? tp->ffunc : tp->cfunc;
    }
    return NULL;
}

```

```

/*
/* hashpjw()
*/
/* Hashing-Funktion frei nach P. J. Weinberger
/* ( optimierte ANSI-C Version von Jochen Leidner )
*/
*/

unsigned short hashpjw(register const char *s)
{
    register const char *p = s;
    register unsigned long h = 0L, g;           /* Nie negativ (Index!) */

    while(*p)
    {
        h <= 4;
        h += *p++;
        if((g = h & 0xFFFFFFFFL) != 0)
        {
            h ^= g >> 24;
            h ^= g;
        }
    }
    return (unsigned short)(h % HASHTABSIZ);/* -> [0..HASHTABSIZ-1] */
}

/*
/* lookup()
*/
/* Prueft, ob aktueller Bezeichner/Label schon in der uebergebenen */
/* Symboltafel enthalten ist und traegt ihn ggf. ein. Zum Suchen */
/* wird die Hashfunktion hashpjw() benutzt. Gibt einen Zeiger auf */
/* das Bezeichnerlexem in table[] zurueck.
*/
*/

#define NEW(ObjectType, ObjectSize) ((ObjectType *)malloc(ObjectSize))

char *lookup(register symbol_t *table[], const char *ident)
{
    register const short hval = hashpjw(ident);
    register symbol_t *p = table[hval];
    char *cptr;

    while(p)
    {
        if(!strcmp(p->lex, ident))
            return p->lex;
        p = p->next;
    }
    if((table[hval] = NEW(symbol_t, sizeof(symbol_t))) == NULL)
        error(err_mem);
    table[hval]->next = p;
    if((cptr = table[hval]->lex = NEW(char, strlen(ident) + 1)) == NULL)
        error(err_mem);
    return strcpy(cptr, ident);
}

```

```

/*
 *  getln()
 */
/*
 * Zeile holen, z.T. Lexikalische Analyse
 */
*/

int getln(void)
{
    int noquote = YES;
    register char *n, *u, *q, *v;

    do
    {
        if(fgets(p, LINEMAX, fs) == NULL) /* Zeile lesen */
        {
            #ifdef DEBUG
                puts("(EOF)\n");
            #endif

            exitflag++;
            return NO; /* Dateiende */
        }
        else
        {
            linenono++; /* Zeilennummer inkrem. */

            #ifdef PROTOCOL
                if(!quick)
                    printf("%4ld %s", linenono, p);
            #endif

            u = p;
            while(*u == ' ' || *u == '\t') /* fuehrende Leeraeume */
                ++u;

            start = u; /* merken fuer error() */
        }
    }
    while(!strncmp(u, "//", 2) || *u == '\n'); /* Kommentare entf. */

    v = q = u;
    *(strrchr(u, '\n')) = '\0';

    if(trace) /* Ablaufverfolgung */
    {
        dtab();
        fprintf(fd, "puts(\"\\n\"");
        n = p;
        while(*n != '\0')
        {
            if(*n == '\\')
                /* Escapemechanismus */
                fputc('\\', fd);
            fputc(*n++, fd);
        }
        fprintf(fd, "\\");\n");
        dtab();
        fprintf(fd, "getchar();\\n");
    }
}

```

```

while(*q)
{
    while(noquote && (*q == ' ' || *q == '\t')) /* Leerzeume */
        ++q;

    if(*q == '\"' && ((q == v) ? YES : (*(q-1) != '\\')))
        noquote = !noquote;

    *v++ = *q++;
} /* Ziel-/Quellptr inkrt. */

*v = '\0'; /* Endmarkierung neu */
cp = u;

if(*u == '<') /* Label bearbeiten */
{
    register n = (int)strcspn(cp, ">"); /* ab Zeilenanfang */
    cp = u + n + 1;
    *(u + n) = '\0';

    if(*u++ && !quick)
        doclabel(u);
}

return YES; /* noch nicht EOF */
}

/*-----
/* dofdummy() */
/*
/* Dummy-Funktion fuer schnelle Syntaxanalyse */
-----*/
void dofdummy(void)
{
    /* Diese Funktion muss leer sein */
}

/*-----
/* doclabel()
/*
/* Generiere C-Labels */
-----*/
void doclabel(char *p)
{
    #ifdef PROTOCOL
        protocol("doclabel()");
    #endif

    fprintf(fd, "\nL%s:\n", lookup(deflab, p));

    #ifdef PROTOCOL
        depth--;
    #endif
}

```

```
/*
 * op()
 */
/* Binaere Operatoren
*/
void op(void)
{
    auto char op[OPLEN] = "";
    #ifdef PROTOCOL
        protocol("op()");
    #endif

    if(*cp == '<')
    {
        if(++cp == '=')
        {
            strcpy(op, "<=");
            ++cp;
        }
        else
            strcpy(op, "<");
    }
    else
    {
        if(*cp == '>')
        {
            if(++cp == '=')
            {
                strcpy(op, ">=");
                ++cp;
            }
            else
                strcpy(op, ">");
        }
        else
        {
            if(*cp == '=')
            {
                strcpy(op, "==");
                ++cp;
            }
            else
            {
                if(*cp == '#')
                {
                    strcpy(op, "!=");
                    ++cp;
                }
                else
                    error(err_op);
            }
        }
    }
    fprintf(fd, op);
    #ifdef PROTOCOL
        depth--;
    #endif
}
```

```

void qop(void)
{
    if(*cp == '<')
    {
        if(*++cp == '=')
            ++cp;
    }
    else
    {
        if(*cp == '>')
        {
            if(*++cp == '=')
                ++cp;
        }
        else
            if(*cp == '=')
                ++cp;
        else
            if(*cp == '#')
                ++cp;
        else
            error(err_op);
    }
}

/*-----
/*  id()                               */
/*-----*/
/*  Bezeichner                         */
/*-----*/

char *id(void)
{
    static char identifier[LINEMAX];
    register char *p = identifier, *q;

#ifndef PROTOCOL
    protocol("id()");
#endif

    if(!isalpha(*cp))
        error(err_id);

    do
    {
        *p++ = *cp++;
    }
    while(isalpha(*cp));
    *p = '\0';

    fprintf(fd, "_%s", q = lookup(syms, identifier));

#ifndef PROTOCOL
    depth--;
#endif

    return q;
}

```

```

void qid(void)
{
    if(!isalpha(*cp))
        error(err_id);

    do
    {
        cp++;
    }
    while(isalpha(*cp));
}

void qidlist(void)
{
    while(YES)
    {
        qid();
        if(*cp == ',')
            ++cp;
        else
            break;
    }
}

/*-----*/
/* num() */
/* Zah1 */
/*-----*/
void num(void)
{
    #ifdef PROTOCOL
        protocol("num()");
    #endif

    if(isdigit(*cp) || *cp == '-')
        fprintf(fd, "%ldL", strtol(cp, &cp, DEZ)); /* Dez.-Konst. */
    else
        error(err_num);

    #ifdef PROTOCOL
        depth--;
    #endif
}

void qnum(void)
{
    if(isdigit(*cp) || *cp == '-')
        strtol(cp, &cp, DEZ);
    else
        error(err_num);
}

```

```
/*-----*/
/*  numid()
/*
/*  Zahl ; Bezeichner
/*-----*/
/* */

void numid(void)
{
    #ifdef PROTOCOL
        protocol("numid()");
    #endif

    if(isdigit(*cp) || *cp == '-')
    {
        fprintf(fd, "%ldL", strtol(cp, &cp, DEZ)); /* Dez.-Konst. */
        /* ^ direkt statt num()-Call, weil Pruefungen unnoetig sind */
    }
    else
        if(isalpha(*cp))
            id();
        else
            error(err_numid);

    #ifdef PROTOCOL
        depth--;
    #endif
}

void qnumid(void)
{
    if(isdigit(*cp) || *cp == '-')
        strtol(cp, &cp, DEZ);
    else
        if(isalpha(*cp))
            qid();
        else
            error(err_numid);
}

/*-----*/
/*  relation()
/*
/*  relation ::= expr op expr ;
/*-----*/
/* */

void relation(void)
{
    #ifdef PROTOCOL
        protocol("relation()");
    #endif

    match("(");
    expr();
    op();
    expr();
    match(")");
}

    #ifdef PROTOCOL
        depth--;
    #endif
}
```

```
void qrelation(void)
{
    match("(");
    qexpr();
    qop();
    qexpr();
    match(")");
}

/*-----*/
/* factor() */
/*
/* Rekursiver Ausdrucksparser I
/*-----*/

void factor(void)
{
    #ifdef PROTOCOL
        protocol("factor()");
    #endif

    if(*cp == '(')
    {
        fputc('(', fd);
        cp++;
        expr();
        match(")");
        fputc(')', fd);
    }
    else
        numid();
    #ifdef PROTOCOL
        depth--;
    #endif
}

void qfactor(void)
{
    if(*cp == '(')
    {
        cp++;
        qexpr();
        match(")");
    }
    else
        qnumid();
}
```

```

/*
/*  term()
/*
/* Rekursiver Ausdrucksparser II
/*
void term(void)
{
    #ifdef PROTOCOL
        protocol("term()");
    #endif

    factor();
    while(*cp == '*' || (*cp == '/' && *(cp + 1) != '/') || *cp == '%')
    {
        fputc(*cp++, fd);
        factor();
    }

    #ifdef PROTOCOL
        depth--;
    #endif
}

void qterm(void)
{
    qfactor();
    while(*cp == '*' || (*cp == '/' && *(cp + 1) != '/') || *cp == '%')
    {
        cp++;
        qfactor();
    }
}

/*
/*  expr()
/*
/* Rekursiver Ausdrucksparser III
/*
void expr(void)
{
    #ifdef PROTOCOL
        protocol("expr()");
    #endif

    term();
    while(*cp == '+' || *cp == '-')
    {
        fputc(*cp++, fd);
        term();
    }

    #ifdef PROTOCOL
        depth--;
    #endif
}

```

```
void qexpr(void)
{
    qterm();
    while(*cp == '+' || *cp == '-')
    {
        cp++;
        qterm();
    }
}

void qexprlist(void)
{
    do
    {
        qexpr();
    }
    while(*cp == ',' ? ++cp : 0);
}

/*-----*/
/* doccr() */
/* cr - Anweisung */
/*-----*/

void doccr(void)
{
    #ifdef PROTOCOL
        protocol("doccr()");
    #endif

    dtab();
    fprintf(fd, "putchar('\\\\n');\\n");

    #ifdef PROTOCOL
        depth--;
    #endif
}
```

```

/*
 * docprint()
 *
 * print - Anweisung
 */
 */

void docprint(void)
{
    #ifdef PROTOCOL
        protocol("docprint()");
    #endif

    do
    {
        if(*cp == '\\')
        {
            register char *bemit = cp;
            register short neos = YES;           /* Innerh. String? */

            dtab();
            while(*++cp)                      /* ' " ' gefunden? */
            {
                *cp++ = '\0';
                fprintf(fd, "printf(\"%s\\", %s\\");\n", bemit);
                neos = NO;
                break;                         /* Ja: Ende */
            }
            if(neos)
                error(err_strnt);          /* Nein: Fehler */
        }
        else
        {
            dtab();
            fprintf(fd, "printf(\"%ld\\", );
            expr();
            fprintf(fd, ");\n");
        }
    }
    while(*cp++ == ',');
    --cp;

    #ifdef PROTOCOL
        depth--;
    #endif
}

```

```
void dofprintf(void)
{
    do
    {
        if(*cp == '\\')
        {
            register short neos = YES;

            while(*++cp)
                if(*cp == '\\')
                {
                    cp++;
                    neos = NO;
                    break;
                }
            if(neos)
                error(err_strnt);
        }
        else
            qexpr();
    }
    while(*cp++ == ',');

    --cp;
}
```

```
/*
/*  docquit()
/*
/*  quit - Anweisung
/*
*/
```

```
void docquit(void)
{
    #ifdef PROTOCOL
        protocol("docquit()");
    #endif

    dtab();
    fprintf(fd, "exit(0);\n");

    #ifdef PROTOCOL
        depth--;
    #endif
}
```

```
/*-----*/
/*  doccontinue()
*/
/*  continue - Anweisung
*/
/*-----*/
void doccontinue(void)
{
    #ifdef PROTOCOL
        protocol("doccontinue()");
    #endif

    dtab();
    if(*cp++ == '<' && isalpha(*cp))
    {
        char *bemit = cp;

        cp = strchr(cp, '>');
        *cp++ = '\0';
        lookup(uselab, bemit);
        fprintf(fd, "goto L%s;\n", bemit);
    }
    else
        error(err_lab);

    #ifdef PROTOCOL
        depth--;
    #endif
}

void dofcontinue(void)
{
    if(*cp++ == '<' && isalpha(*cp))
        cp = strchr(cp, '>') + 1;
    else
        error(err_lab);
}
```

```

/*
/* docrequest()
*/
/* request - Anweisung
*/
void docrequest(void)
{
    #ifdef PROTOCOL
        protocol("docrequest()");
    #endif

    while(YES)
    {
        dtab();
        fprintf(fd, "scanf(\"%ld\", &"); 
        id();
        fprintf(fd, ");\n");
        if(*cp == ',', )
            ++cp;
        else
            break;
    }

    #ifdef PROTOCOL
        depth--;
    #endif
}

/*
/* docif()
*/
/* if - Anweisung
*/
void docif(void)
{
    #ifdef PROTOCOL
        protocol("docif()");
    #endif

    dtab();
    fprintf(fd, "if(");
    relation();
    fprintf(fd, ")\n");

    if(*cp == 'c')
    {
        match("continue");
        dococontinue();
    }
    else
        if(*cp == 't')
        {
            match("then");
            dtab();
            fprintf(fd, "(\n");
            stage++;
        }
        else
            error(err_cont);
}

```

```

#define PROTOCOL
    depth--;
#endif
}

void dofif(void)
{
    qrelation();
    if(*cp == 'c')
    {
        match("continue");
        dofcontinue();
    }
    else
        if(*cp == 't')
        {
            match("then");
            stage++;
        }
        else
            error(err_cont);
}

/*-----
/*  docswap()
/*
/*  swap - Anweisung
/*-----*/
void docswap(void)
{
    char *a, *b;

    #ifdef PROTOCOL
        protocol("docswap()");
    #endif

    swapused = YES;

    dtab();
    fprintf(fd, "_temp=");
    a = id();
    match(",");
    fprintf(fd, "; _%s=", a);
    b = id();
    fprintf(fd, "; _%s=_temp;\n", b);

    #ifdef PROTOCOL
        depth--;
    #endif
}

void dofswap(void)
{
    qid();
    match(",");
    qid();
}

```

```
/*
 * docset()
 */
/*
 * set - Anweisung
 */
/*
void docset(void)
{
    #ifdef PROTOCOL
        protocol("docset()");
    #endif

    do
    {
        dtab();
        id();
        match("=");
        fputc('=', fd);
        expr();
        fprintf(fd, ";\n");
    }
    while(*cp++ == ',');
    --cp;

    #ifdef PROTOCOL
        depth--;
    #endif
}

void dofset(void)
{
    do
    {
        qid();
        match("=");
        qexpr();
    }
    while(*cp++ == ',');
    --cp;
}
```

```
/*
 * docclr()
 */
/*
 * clr - Anweisung
 */
void docclr(void)
{
    #ifdef PROTOCOL
        protocol("docclr()");
    #endif

    do
    {
        dtab();
        id();
        fprintf(fd, "=OL;\n");
    }
    while(*cp++ == ',');
    --cp;

    #ifdef PROTOCOL
        depth--;
    #endif
}

/*
 * docdec()
 */
/*
 * dec - Anweisung
 */
void docdec(void)
{
    #ifdef PROTOCOL
        protocol("docdec()");
    #endif

    do
    {
        dtab();
        id();
        fprintf(fd, "--;\n");
    }
    while(*cp++ == ',');
    --cp;

    #ifdef PROTOCOL
        depth--;
    #endif
}
```

```

/*
 *-----*
 * docinc()
 */
 */
 */
/* inc - Anweisung
 */
/*
void docinc(void)
{
    #ifdef PROTOCOL
        protocol("docinc()");
    #endif

    do
    {
        dtab();
        id();
        fprintf(fd, "++;\n");
    }
    while(*cp++ == ',');
    --cp;

    #ifdef PROTOCOL
        depth--;
    #endif
}
*/
/*
 *-----*
 * docwhile()
 */
 */
/*
 * while..end - Schleife
 */
/*
void docwhile(void)
{
    #ifdef PROTOCOL
        protocol("docwhile()");
    #endif

    dtab();
    fprintf(fd, "while(");
    relation();
    match("do");
    fprintf(fd, ")\\n");
    dtab();
    fprintf(fd, "{\\n");
    stage++;

    #ifdef PROTOCOL
        depth--;
    #endif
}
*/
void dofwhile(void)
{
    qrelation();
    match("do");
    stage++;
}

```

```
/*
 * docelse()
 */
/*
 * ..else..
 */
/*
void docelse(void)
{
    #ifdef PROTOCOL
        protocol("docelse()");
        depth--;
    #endif

    if(stage < 1)
        error(err_else);
    docend();
    dtab();
    fprintf(fd, "else\n");
    dtab();
    fprintf(fd, "{\n");
    stage++;
}

void dofelse(void)
{
    if(stage < 1)
        error(err_else);
    dofend();
    stage++;
}

/*
 * docend()
 */
/*
 * ..end
 */
/*
void docend(void)
{
    #ifdef PROTOCOL
        protocol("docend()");
        depth--;
    #endif

    stage--;
    dtab();
    fprintf(fd, "}\n");
    if(stage < 0)
        error(err_endne);
}

void dofend(void)
{
    stage--;
    if(stage < 0)
        error(err_endne);
}
```

```

/*
/* docpush()
/*
/* push - Anweisung
*/

void docpush(void)
{
    #ifdef PROTOCOL
        protocol("docpush()");
    #endif

    stackused = YES;

    do
    {
        dtab();
        fprintf(fd, "*SP++=");
        expr();
        fprintf(fd, ";\n");
    }
    while(*cp == ',' ? ++cp : 0);

    #ifdef PROTOCOL
        depth--;
    #endif
}

/*
/* docpop()
/*
/* pop - Anweisung
*/
void docpop(void)
{
    #ifdef PROTOCOL
        protocol("docpop()");
    #endif

    stackused = YES;

    while(YES)
    {
        dtab();
        id();
        fprintf(fd, "=*--SP;\n");
        if(*cp == ',')
            ++cp;
        else
            break;
    }

    #ifdef PROTOCOL
        depth--;
    #endif
}

```

```

/*
/* docfor()
*/
/* for..do..end - Schleife
*/
----- */

void docfor(void)
{
    char *cp;

    #ifdef PROTOCOL
        protocol("docfor()");
    #endif

    dtab();
    fprintf(fd, "for(");
    cp = id();
    match("=");
    fputc('=', fd);
    expr();
    fprintf(fd, "; _%s<=", cp);
    match("..");
    expr();
    fprintf(fd, "; _%s++)\n", cp);
    dtab();
    fprintf(fd, "{\n");
    match(")do");
    stage++;
    /* Erzwingt 'end' */

    #ifdef PROTOCOL
        depth--;
    #endif
}

void doffor(void)
{
    qid();
    match("=(\"");
    qexpr();
    match("..");
    qexpr();
    match(")do");
    stage++;
    /* Erzwingt 'end' */
}

```

```
/*
 * docloop()
 */
/*
 * loop - Anweisung
 */
void docloop(void)
{
    #ifdef PROTOCOL
        protocol("docloop()");
    #endif

    dtab();
    fprintf(fd, "for(;;)\n");
    dtab();
    fprintf(fd, "{\n");
    stage++;

    #ifdef PROTOCOL
        depth--;
    #endif
}

void dofloop(void)
{
    stage++;
}

/*
 * docexit()
 */
/*
 * exit - Anweisung
 */
void docexit(void)
{
    #ifdef PROTOCOL
        protocol("docexit()");
        depth--;
    #endif

    dtab();
    if(stage > 0)
        fprintf(fd, "break;\n");
    else
        error(err_exit);
}

void dofexit(void)
{
    if(stage < 1)
        error(err_exit);
}
```

```
/*-----*/
/*  doctron( )                                */
/*                                              */
/*  Automatische Ablaufverfolgung mitkompilieren */
/*-----*/
void doctrace(void)
{
    #ifdef PROTOCOL
        protocol("doctron()");
        depth--;
    #endif

    trace = YES;
}

/*-----*/
/*  doctroff()                                 */
/*                                              */
/*  Automatische Ablaufverfolgung nicht mitkompilieren */
/*-----*/
void docuntrace(void)
{
    #ifdef PROTOCOL
        protocol("doctroff()");
        depth--;
    #endif

    trace = NO;
}
```

```
/*
/* leazyprg()
*/
/* Hauptprogramm
*/
void leazyprg(void)
{
    while(YES)
    {
        if(getln())
        {
            register void (*f)(void) = keyword();

            if(f)
            {
                #ifdef DEBUG
                    dbgcall(f); /* Debugging-Hilfe */
                #endif

                (*f)();
                /* Funktionszeiger */

                switch(*cp) /* Zeilenende */
                {
                    case '\0': break;

                    case '/': if(*(cp + 1) == '/')
                                break; /* oder Kommentar */
                    default:
                        #ifdef DEBUG
                            printf("\n%s\\0\\n", cp);
                        #endif
                        error(err_eolex);
                }
            }
            else
            {
                #ifdef DEBUG
                    if(!cp)
                    {
                        printf("\n%s: internal error in leazyprg()"
                               " at $%lxh\\n", cname, leazyprg);
                        exit(1);
                    }
                #endif

                error(err_unst);
            }
        }
        else
            return;
    }
}
```

```

/*
/*  ocdecl()
/*
/* Deklarationsdatei ("*.h") schreiben
/*
void ocdecl(void)
{
    register symbol_t *walk, **symp = syms;
    register const symbol_t **top = symp + HASHTABSIZ;

    while(symp < top)
        if((walk = *symp++) != NULL)
        {
            fprintf(fh, "\nlong _%s", walk->lex); /* '_' voranst. */
            while(YES)
                if((walk = walk->next) != NULL)
                    fprintf(fh, ", _%s", walk->lex);
                else
                    break;
            fputc(';', fh); /* ';' schreib. */
        }
    fputc('\n', fh);

    if(stackused)
        fprintf(fh, "\nlong S[8192], *SP=S;\n");

    if(swapused)
        fprintf(fh, "long _temp;\n");
}

/*
/* defp()
/*
/* Ueberpruefe einen Label auf Deklaration
/*
void defp(register const char *s)
{
    static char message[16 + MAXLABEL] = "undefined label <";
    register symbol_t *walk = deflab[hashpjw(s)];

    while(walk)
        if(!strcmp(s, walk->lex))
            return;
        else
            walk = walk->next;

    strcat(message, s);
    strcat(message, ">");
    error(message);
}

```

```

/*
 * checklabs()
 */
/* Ueberprufe alle benutzten Labels auf ihre Deklarationen */
*/
void checklabs(void)
{
    register symbol_t *walk, **symp = uselab;
    register const symbol_t **top = symp + HASHTABSIZ;

    while(symp < top)
        if((walk = *symp++) != NULL)
        {
            defp(walk->lex);
            while(YES)
                if((walk = walk->next) != NULL)
                    defp(walk->lex);
                else
                    break;
        }
}

/*
 * compile()
 */
/* Dateien oeffnen, Speicherverwaltung, Hauptprogramm rufen,
 * I/O-Pufferung, File kompilieren, Dateien wieder schliessen */
*/
void compile(char *arg)
{
    #ifdef PROTOCOL
        time_t tida = time(NULL);
        clock_t tm = clock();
    #endif

    if(strchr(strcpy(sname = name, arg), '.'))
    {
        #ifdef INTEGRATED
            if(interactive)
            {
                printf("%s: %s\n", cname, err_ext);
                longjmp(repenv, ERR);
            }
            else
        #endif
            error(err_ext);
    }

    p = start = NULL;                      /* Pointer initialisieren! */
    lineno = 0;                            /* Zeilennummer */
}

```

```

pext = strrchr(name, '\0');
strcpy(pext, ext[EXT_LZY]);           /* Extension '.lzy' */
if((fs = fopen(sname, "r")) == NULL)
    #ifdef INTEGRATED
        if(interactive)
        {
            perror((char *)cname);
            longjmp(repenv, ERR);
        }
        else
    #endif
        error(err_open);
setbuf(fs, fsbuf);                  /* IO-Pufferung */

#ifndef PROTOCOL
printf("      COMPILER LISTING %s      ====="
      "=====\\n\\n**** compiling %s..\\n\\n",
      ctime(&tida),
      sname );
depth = 2;
#endif

p = sline;                         /* Zeiger muss hier initialisiert werden */

if(!quick)
{
    char *ps;

    memset(syms, 0, tabszie);
    memset(deflab, 0, tabszie);     /* Tabellen initialisieren */
    memset(uselab, 0, tabszie);

    strcpy(pext, ext[EXT_C]);       /* Extension '.c' */
    if((fd = fopen(sname = name, "w")) == NULL)
    {
        wprotect = YES;           /* Schreibfehler */
        error(err_fd);
    }
    setbuf(fd, fdbuf);           /* IO-Pufferung */

    strcpy(pext, ext[EXT_H]);       /* Extension '.h' */
    if((fh = fopen(name, "w")) == NULL)
        error(err_fh);           /* Fehler ? */
    setbuf(fh, fhbuf);           /* IO-Pufferung */

    fprintf(fd, "#include <stdio.h>\\n#include <stdlib.h>\\n\\n"
              "#include \"%s\"\\n\\nvoid main(void)\\n{\\n",
              ((ps = strrchr( name,
                               #ifdef UNIX
                                   '/' /* path separator */
                               #else
                                   '\\\\' /* Pfadtrennzeichen */
                               #endif
                           ) ) != NULL ? ps + 1 : name)
                           /* Pfad entfernen */
              );
    strcpy(pext, ext[EXT_LZY]);     /* Extension '.lzy' */
}

```

```

stage = 0;                                /* Hauptprogramm      */
leazyprg();                               /* Struktur O.K. ?   */
if(stage)
    error(err_end);

if(!quick)
{
    if(stackused)
        fprintf(fd, "\tif(SP<S; !SP>&(S[8191]))\n\t{\n\t\tputs(\""
                    "\"stack error\");\n\t\texit(1);\n\t}\n");
    fprintf(fd, "\texit(0);\n)\n\n");

#define DEBUG
    dbgsyms();
#else
#define PROTOCOL
    outsyms();
#endif
#endif
#define PROTOCOL
    puts("**** writing declaration file.");
#endif

ocdecl();                                 /* Deklarationsfile   */
checklabs();                              /* Labels dekl.??     */
freetab(syms);
freetab(deflab);                          /* Speicherfreigabe   */
freetab(uselab);
}

#ifndef PROTOCOL
    printf("**** no errors found in %ld ms.\n", clock() - tm);
#endif

closeio();                                /* Files schliessen   */
}

#ifndef INTEGRATED

/*-
 * exec()
 *
 * Laden und Starten eines anderen Programms (child process call)
 *-
 */

int exec(char *pathname, char *cmdistr, char *envstr, int *retcode)
#endif !defined(__PROCESS)
{
    auto char command[FNAMEMAX];

    if(*envstr != '\0')
        puts("LEAZY.exec(): Environment is being ignored");
    strcpy(command, pathname);
    strcat(command, " ");
    strcat(command, cmdistr);
    return ((*retcode = system(command)) != 0) ? -1 : 0;
}
#endif
;      /* PROCESS.H vorhanden: Dann nur Prototyp deklarieren */
#endif

```

```

/*
/*  pattex()
/*
/*  Musterschablone (PATTERn) fuellen und ausfuehren (EXecute)
/*
void pattex(register index)
{
    auto char num[11];
    register char *patptr = &(patt[index][0]), *paramptr = param[index];
    register c;
    int retcode;

    if(!!(index !! errocc)) /* Bei 'Edit' od. Dateiwechsel Zeile 0 */
    {
        lineno = 0;
        cp = start;
    }
    *paramptr = '\0';
    while((c = *patptr++) != '\0')
        if(c == '%')
        {
            switch(toupper(*patptr))
            {
                case 'F':   strcat(paramptr, workfile);
                patptr++;
                break;
                case 'L':   strcat(paramptr, itoa(lineno, num, DEZ));
                patptr++;
                break;
                case 'C':   strcat(paramptr,
                    itoa((long)(cp - start) + 1, num, DEZ));
                patptr++;
                break;
                case '%':   strcat(paramptr, "%");
                default:    break;
            }
        }
        else
        {
            auto char onechar[] = " ";
            *onechar = (char)c;
            strcat(paramptr, onechar);
        }
    }
    if(exec(fname[index], paramptr, ESTR, &retcode) == -1)
    {
        perror((char *)cname);
        longjmp(repenv, ERR);
    }
}

```

```
/*
 *  show()
 */
/* Code - Datei anzeigen
 */
-----*/



#define CR2 {
    printf("%s", sline);
    if(++i == L_PER_S)
    {
        i = 0;
        printf(ret);
        if(getchar() == 'q')
        {
            getchar();
            break;
        }
    }
}

void show(void)
{
    if((fd = fopen(strcat(strcpy(fname[4], workfile),
                           ext[EXT_C])), "r")) == NULL)
        printf(fss, cname, err_fd);
    else
    {
        short i = 0;

        puts(valid ? ESTR : "\nwarning: old code!\n");
        while(fgets(sline, LINEMAX, fd) != NULL)
            if(!strncmp(sline, "#include \"", (size_t)10))
            {
                if((fh = fopen(strcat(strcpy(fname[4], workfile),
                                      ext[EXT_H]), "r")) == NULL)
                    printf(fss, cname, err_fh);
                else
                {
                    while(fgets(sline, LINEMAX, fh) != NULL)
                        CR2;
                    fclose(fh);
                }
            }
        else
            CR2;
    }
    fclose(fd);
}
```

```
-----*/  
/* help()  
/* Hilfsfunktion -- Gibt Menue-Uebersicht aus  
/*-----*/  
  
void help(context_t menu_context)  
{  
    printf("\n\t%s%s", head, head2);  
    if(menu_context == MENU_MAIN)  
        printf(menu, workfile);  
    else  
    {  
        short i = -1;  
  
        #ifdef PROTOCOL  
            printf(opts1, syntax_tree ? on : off);  
        #endif  
  
        while(++i < 3)  
            printf(opts, parname[i], fname[i], patt[i]);  
        printf(opts2, trace ? on : off);  
    }  
}
```

```
/*
 *  options()
 */
/* Options - Menue der Benutzerschale */
/*
void options(void)
{
    short index;

    help(MENU_OPT);
    for(EVER)
    {
        index = INONE;
        printf(prompt);
        switch(toupper(*gets(inp)))
        {
            case 'Q':    return;
            case 'E':    index = IED;
                          break;
            case 'C':    index = ICC;
                          break;
            case 'L':    index = ILN;
                          break;
            case 'T':    trace = !trace;
                          break;
            #ifdef PROTOCOL
            case 'S':    syntax_tree = !syntax_tree;
                          break;
            #endif
            case 'H':    help(MENU_OPT);
                          break;
            default:    break;
        }
        if(index < IEX && index > -1)
        {
            printf("%s:\nold file name:\n\"%s\"\nnew file name:\n",
                   parname[index], fname[index]);
            gets(fname[index]);
            printf("old parameters:\n\"%s\"\nnew parameters:\n",
                   patt[index]);
            gets(patt[index]);
        }
    }
}
```

```
/*
 *  ishell()
 *
 *  Interaktive Benutzerschale (dialogorientierte Umgebung)
 */
void ishell(void)
{
    int retcode;

    printf("\n\t%s%s", head, head2);
    printf(menu, workfile);
    setjmp(repenv);
    cp = start;
    for(EVER)
    {
        printf(prompt);
        switch(toupper(*gets(inp)))
        {
            case 'E':  pattex(IED);           /****** Edit *****/
                        valid = errocc = NO;
                        break;

            case 'C':  if(!setjmp(procenv))   /**** Compile ****/
                        {
                            quick = NO;
                            compile(workfile);
                            valid = YES;
                        }
                        else
                        {
                            pattex(IED);
                            valid = errocc = NO;
                        }
                        break;

            case 'N':  if(!setjmp(procenv))   /* Syntaxcheck */
                        {
                            quick = YES;
                            compile(workfile);
                        }
                        else
                        {
                            pattex(IED);
                            errocc = NO;
                        }
                        valid = NO;
                        break;
        }
    }
}
```

```

case 'M': if(!setjmp(procenv))           /***** Make *****/
{
    quick = NO;
    compile(workfile);
    pattex(ICC);
    pattex(ILN);
    valid = YES;
}
else
{
    pattex(IED);
    quick = valid = errocc = NO;
}
break;

case 'R': if(!setjmp(procenv))           /***** Run *****/
{
    quick = NO;
    compile(workfile);
    pattex(ICC);
    pattex(ILN);
    valid = YES;
}
case 'X':
printf("Executing %s'%s'..\n\n",
       valid ? ESTR : "old ", workfile );
if(exec(strcat(strcpy(fname[IEX], workfile),
      #ifdef _QC
          ".exe"
      #else
          #ifdef UNIX
              ""
          #else
              ".tos"
          #endif
      #endif
      ), ESTR, ESTR, &retcode) == -1)
{
    perror((char *)cname);
    longjmp(repenv, ERR);
}
valid = YES;
}
else
{
    pattex(IED);
    valid = errocc = NO;
}
break;

case 'W': while(YES)
{
    printf("Work file: ");   /*** Workfile ***/
    gets(workfile);
    if(strchr(workfile, '.')) 
        printf(fss, cname, "No '.' please!\n\n");

    else
        break;
}
break;

```

```
    case 'S': show();                                /** Show file ***/
    break;

    case 'O': options();                            /** Options ****/
    break;

    case 'H': help(MENU_MAIN);                     ***** Help ****/
    break;

    case 'Q': return;                             ***** Quit ****/
    break;
}
}

#endif

/*
 * usage()
 */
/* Syntax der Benutzung ausgeben + terminieren
*/
void usage(void)
{
    printf("Usage: %s [ <filename> [-n|-t] ]\n", cname);
    exit(1);
}
```

```

/*
 * main()
 */
/* Behandle Kommandozeillenswitches, reserviere Speicher fuer
 * File-buffering und verzweige anhand der Parameteranzahl:
 * Interaktivshell / Batchmodus / falsche Parameter
*/
int main(int argc, char *argv[])
{
    #ifdef DEBUGALL
        printf( "%s\\nentering debugging mode.\n"
                "Last compilation %s %s\\n"
                "I have counted %d argument(s); trace is %s;\\n"
                "quick syntax check is %s.\\n",
                argv[0],
                __DATE__,
                __TIME__,
                argc,
                trace ? on : off,
                quick ? on : off );
    #endif

    if( (fsbuf = (char *)malloc((size_t)(BUFSRC * BUFSIZ))) == NULL ||
        (fdbuf = (char *)malloc((size_t)(BUFSTD * BUFSIZ))) == NULL ||
        (fhbuf = (char *)malloc((size_t)(BUFHDF * BUFSIZ))) == NULL )
        error(err_mem);

    switch(argc)
    {
        #ifdef INTEGRATED
            case 1:      interactive = YES;           /* Interaktiv */
                        ishell();
                        break;
        #endif

            case 3:      if(*argv[2] == '-')
                            switch(toupper(*(argv[2] + 1)))
                            {
                                case 'T':  trace = YES;
                                break;
                                case 'N':  quick = YES;
                                break;
                                default: usage();
                            }
                            else
                                usage();
            case 2:      putchar('\\n');
                        puts(head);
                        compile(argv[1]);
                        break;
            default:     usage();
        }
    return 0;                                     /* Normales Ende */
}

/*
 *      end of file 'leazy.c'
*/

```

8) DOKUMENTATION

Es folgt ein Cross-Reference-Listing des Compilers, das mit einem Public-Domain-XREF-Tool von Gunter Sanders erstellt und von Hand nachbearbeitet wurde, gefolgt der formalen Spezifikation des Codegenerators und Struktogrammen:

BUFDST	2556	49			
BUFHDF	2557	50			
BUFSIZ	2555	2556	2557		
BUFSRC	2555	48			
CCOMP	176	65	74	81	
CCPAR	179	68	77	84	
CR1	600	621	626		
CR2	2272	2304	2309		
D()	570				
DEBUG	1947	1960	1968	2162	38
	568	796			
DEBUGALL	2542				
DEZ	2240	2244	41		
EDITOR	175	64	73	80	
EDPAR	178	67	76	83	
EOLST	442	57	584		
ERR	2082	2098	2261	42	471
	475				
ESTR	59				
ESTR	2258	2294	457		
EVER	2350	2408	58		
EXT_C	132	2123	2288		
EXT_H	133	2131	2299		
EXT_LZY	131	2092	2147		
FILE	141				
FNAMEMAX	175	178	181	182	2200
	231	45			
FUNPTR	106	268	272	362	363
HASHTABSIZ	1994	2047	250	251	252
	51	610	649	688	747

ICC	123	2361	2447		
IED	122	2358	2413	2425	2437
	2453				
IEX	125				
ILN	124	2364	2448		
INONE	126	2352			
INTEGRATED	169	2077	2094	2189	243
	2562	35	466	96	97

KEYTABSIZ	53				
-----------	----	--	--	--	--

LINEMAX	2295	2303	232	44	794
LINKER	177	66	75	82	
LNPAR	180	70	78	85	
L_PER_S	2273	43			

MATCHSTRLEN	46	533			
MAXLABEL	2024	47			
MENU_MAIN	116	2323	2508		
MENU_OPT	117	2349	2377		

NEW()	759	773	776		
NO	1306	145	146	147	148
	149	151	152	153	154
	157	1928	2419	2426	2438
	2440	2445	2454	55	558
	801				
NULL	1997	2001	2050	2054	2071
	2088	2093	2124	2132	2144
	2288	2295	2299	2303	2555
	2556	2557	368	369	370
	371	372	373	375	376
	377	379	380	382	384
	385	386	387	388	389
	390	391	393	394	396
	397	398	399	400	401
	402	406	408	409	410
	413	415	417	418	419
	420	421	422	425	426
	427	428	429	432	433
	435	437	438	439	440
	442	722	773	776	794

OPLEN	60				
ObjectSize()	759				
ObjectType()	759				

PROTOCOL	1323	1435	1448	1473	1506
	1520	1540	1556	156	1581
	1594	1607	1620	1633	1646
	1659	1672	1692	1723	1750
	1765	1778	1795	1810	1819
	1853	1863	1881	1907	1923
	202	2070	2105	2165	2170
	2182	2329	2370	37	478
	638	807	95	97	

UNIX	2139	39	72	
------	------	----	----	--

YES	1510	1754	1782	1784	1912
	1939	2000	2053	2126	2421
	2432	2449	2564	2572	2574
	470	56	789		

	1010
<u>QC</u>	63
<u>DATE</u>	2548
<u>PROCESS</u>	2198
<u>STDC</u>	91
<u>TIME</u>	2549

a	1504	1514	1516	
arg	2068	2075		
argc	2540	2550	2560	
argv	2540	2547	2569	2570
				2582

b	1504	1517	1518	
be	97			
bemit	1305			
c	2223	2232	2233	2255
cfunc	363	585	720	
checklabs()	2044	2175	343	
clock()	2072	2183		
clock_t	2072			
closeio()	2186	267	469	554
cmdlstr	2197	2206	261	
cname	1972	2080	2097	2260
	2289	2300	2528	461
command	2200	2204	2205	2206
compile()	2068	2420	2433	2446
	344			2582
context_t	118	2320	346	
cp	1304	1320	1321	1332
	1430	1457	1463	1481
	1553	1554	1569	1570
	1592	1617	1618	1643
	1763	1789	1790	1808
	1820	1823	1953	1957
	1969	2229	2244	236
	454	459	536	537
	717	719		713
cptr	765	776	778	
ctime()	2108			

dbgcall()	1948	268	579		
dbgsyms()	2163	269	607		
deflab	2025	2120	2178	251	
defp()	2022	2052	2055	342	
depth	1324	1436	1474	1521	1557
	158	1595	1621	1647	1673
	1694	1725	1766	1796	1830
	1864	1883	1909	1925	2110
	504				
docclr()	374	1579	304		
doccontinue()	404	1382	1460	291	
doccr()	434	1270	288		
docdec()	405	1605	305		
docelse()	412	1690	299		
docend()	436	1699	1721	295	
docexit()	378	1879	301		
docfor()	381	1806	298		
docif()	392	1446	293		
docinc()	431	1631	306		
doxlabel()	287	888			
docloop()	423	1851	300		
docpop()	423	1776	297		
docprint()	403	1290	289		
docpush()	383	1748	296		
docquit()	411	441	1362	290	
docrequest()	395	1420	292		
docset()	407	1538	303		
docswap()	430	1502	302		
doctrace()	414	1905	307		
docuntrace()	416	1921	308		
docwhile()	367	1657	294		
dofcontinue()	404	1484	326		
dofcr()	323				
dofdummy()	411	414	416	434	441
	322				
dofelse()	412	1707	334		
dofend()	436	1711	1735	330	
dofexit()	378	1893	336		
doffor()	381	1834	333		
dofif()	392	1478	328		
dofloop()	424	1868	335		
dofpop()	332				
dofprint()	403	1328	324		
dofpush()	331				
dofquit()	325				
dofrequest()	327				
dofset()	407	1561	338		
dofswap()	430	1525	337		
dofwhile()	367	1677	329		
dtab()	1314	1452	1466	1512	1546
	1587	1613	1639	1663	1668
	1700	1702	1729	1758	1786
	1814	1824	1857	1859	1886
	265	517	826	836	

emsg	533	540	541	542	543
	544				
envstr	2197	2202	261		
err_cont	1471	1493	216		
err_else	1698	1710	217		
err_end	2153	225			
err_endne	1732	1739	218		
err_eolex	1963	220			
err_exit	1890	1896	219		
err_ext	2080	2085	222		
err_fd	2127	223	2289		
err_fh	2133	224	2300		
err_id	211				
err_lab	215				
err_mem	208	2558	774	777	
err_num	212				
err_numid	213				
err_op	210				
err_open	209	2102			
err_strnt	1310	214			
err_unst	1977	221			
errocc	146	2226	2426	2438	2454
	470				
error()	92	97	1310	1471	1493
	1698	1710	1732	1739	1890
	1896	1963	1977	2035	2085
	2102	2127	2133	2153	2558
	263	452	544	774	777
estr	226	59			
exec()	2197	2258	261		
exists	454	457	458	460	
exit()	1366	1973	2529	475	
exitflag	145	800			
expr()	1227	1316	1550	1760	1819
	1822	285			
ext	162	2092	2123	2131	2147
	2288	2299			
extid	129				

f	1943	1945	1948	1951	585
factor()	1153	283			
fclose()	2305	2310	556	563	564
fd	1305	1315	1317	141	1428
	1453	1455	1467	1513	1516
	1518	1549	1551	1589	1615
	1641	1664	1667	1669	1701
	1703	1730	1759	1761	1788
	1815	1818	1820	1823	1825
	1858	1860	1888	2124	2129
	2136	2158	2160	2287	2295
	2310	522	564	827	837
fdbuf	2129	240	2556		
ffunc	362	585	720		
fgets()	2295	2303	794		
fh	141	1999	2002	2005	2007
	2010	2013	2132	2134	2298
	2303	2305	562	563	
fhbuf	2134	241	2557		
flex	582	589	590		
fname	175	2258	2287	2298	2334
	2385	2386			
fopen()	2093	2124	2132	2287	2298
fprintf()	1305	1315	1317	1428	1453
	1455	1467	1513	1516	1518
	1551	1589	1615	1641	1664
	1667	1669	1701	1703	1730
	1759	1761	1788	1815	1820
	1823	1825	1858	1860	1888
	1999	2002	2010	2013	2136
	2158	2160	827	837	
fputc()	1549	1818	2005	2007	562
free()	696	699			
freetab()	2177	2178	2179	271	685
fs	141	2093	2103	556	794
fsbuf	2103	239	2555		
fss	199	2289	2300		

g	735	741	743	744	
getchar()	2277	2279	570	591	604
getln()	1941	275	787		
gets()	2354	2386	2389	2411	

h	735	739	740	741	743
	744	747			
hashpjw()	2025	273	732	763	
head	228	2322	2404	2581	
head2	186	2322	2404		
help()	2320	2349	2377	2508	346
hval	713	714	763	764	773
	775	776			

i	2273	2275	2292	2327	2333
	2334				
id()	1427	1514	1517	1547	1588
	1614	1640	1787	1816	278
	997				
ident	761	763	769	776	778
index	2222	2226	2258	2347	2352
	2358	2361	2364	2382	2385
	2386	2388	2389		
inp	174	2354	2411		
interactive	154	2078	2095	2564	467
ishell()	2400	2565	349		
<hr/>					
jmp_buf	245				
keyword()	1943	272	711		
<hr/>					
leazyprg()	1972	1937	2151	340	
len	361	534	536	537	584
	715				
lex	111	1999	2002	2028	2052
	2055	360	589	601	661
	666	696	717	769	770
	776				
lineno	143	2089	2228	2240	463
	805	809			
longjmp()	2081	2098	2261	471	
lookup()	274	761			
ltoa()	2240	2244			
<hr/>					
main()	2540	351			
malloc()	2555	2556	2557	759	
match()	1459	1465	1483	1489	1515
	1528	1548	1566	1666	1680
	1817	1821	1826	1837	1839
	1841	266	531		
memset()	2119	2120	2121		
menu	187	2324	2405		
menu_context	2320	2323			
message	2024	2033	2034	2035	
msg	452	462			

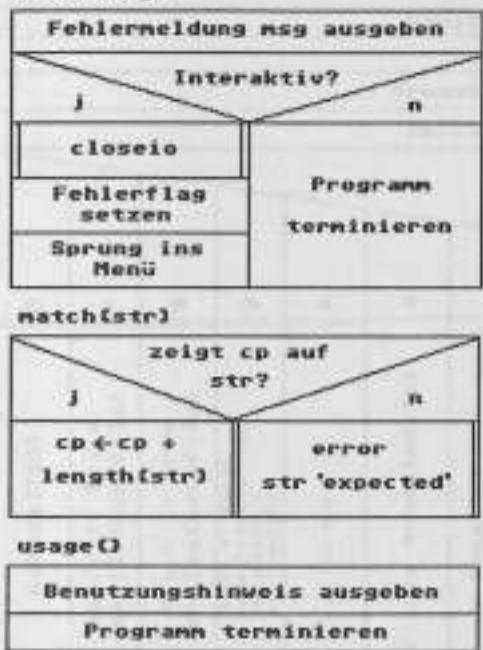
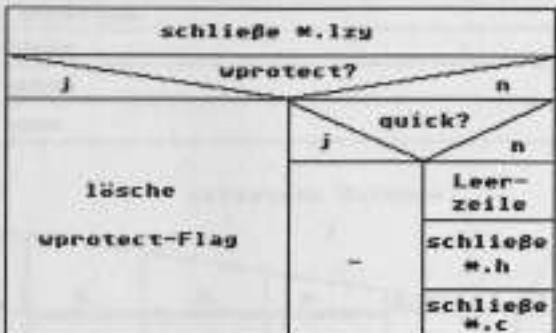
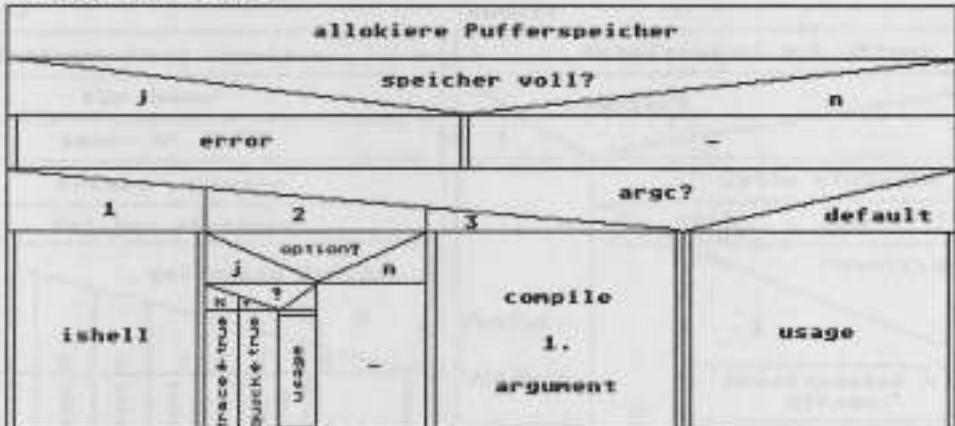
n	1274	1366	1393	2007	2580
	458	488	519	521	562
	602	603	611	790	819
	822	828	829	831	891
name	2075	2091	2124	2132	2138
	2144	231			
neos	1306	1309			
next	110	2001	2031	2054	624
	664	698	771	775	
noquote	789	842			
num()	2221	2240	2244	1052	279
numid()	1082	280			

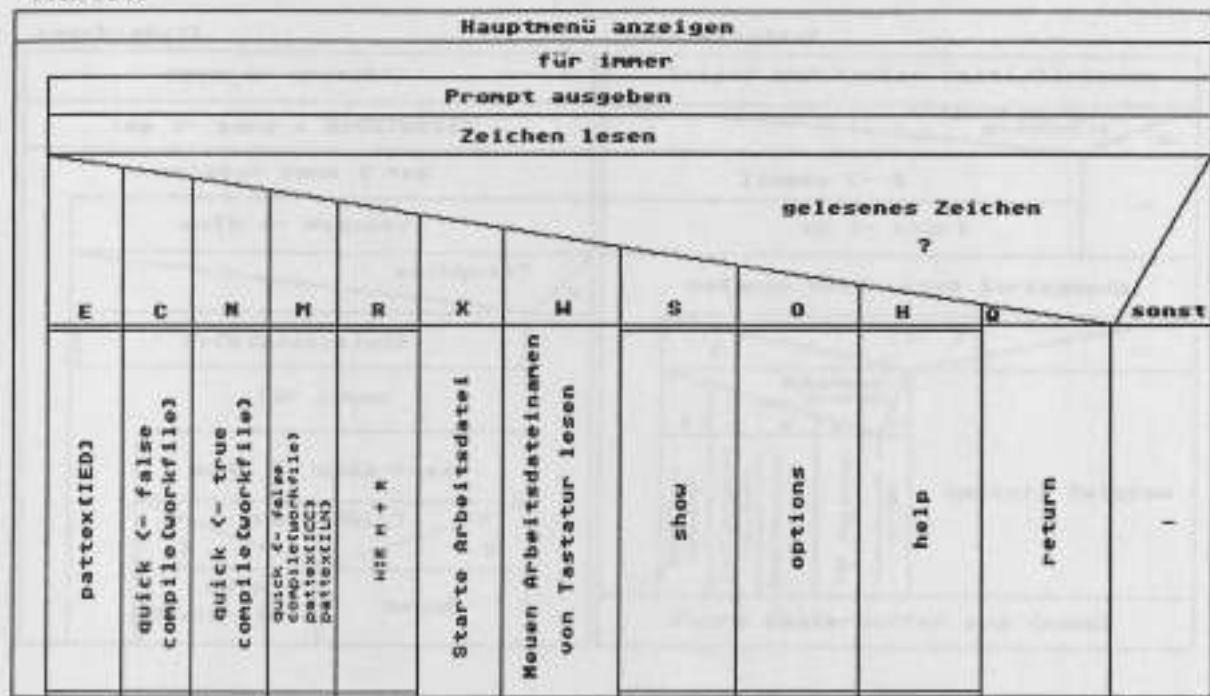
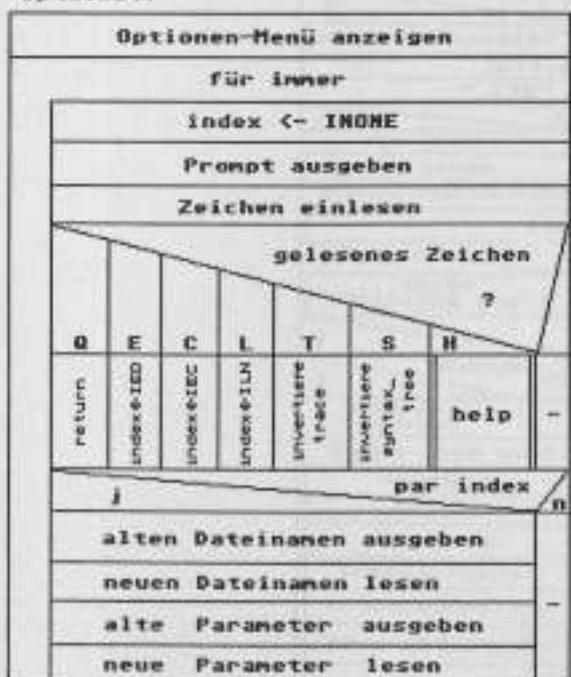
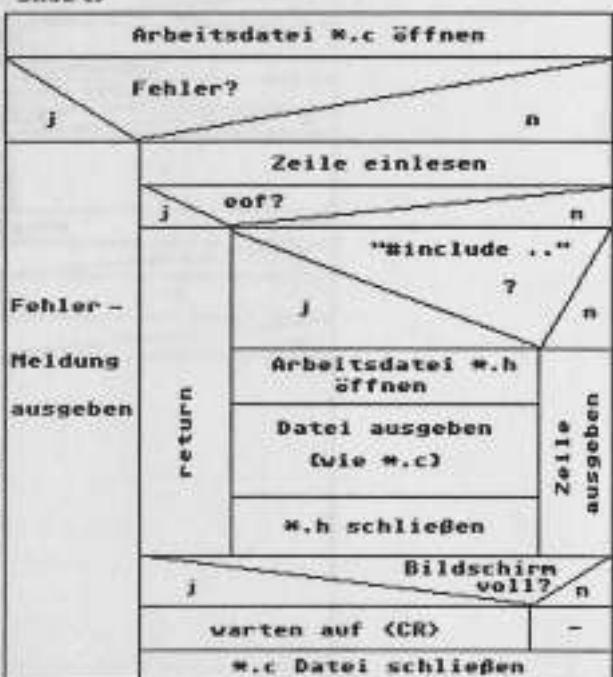
ocdec1()	1991	2174	341		
off	198	2330	2335	2551	2552
on	197	2331	2336	2551	2552
onechar	2253	2255	2256		
op()	277	909			
options()	2345	347			
opts	193	2334			
opts1	203	2330			
opts2	195	2335			
outsyms()	2166	270	646		

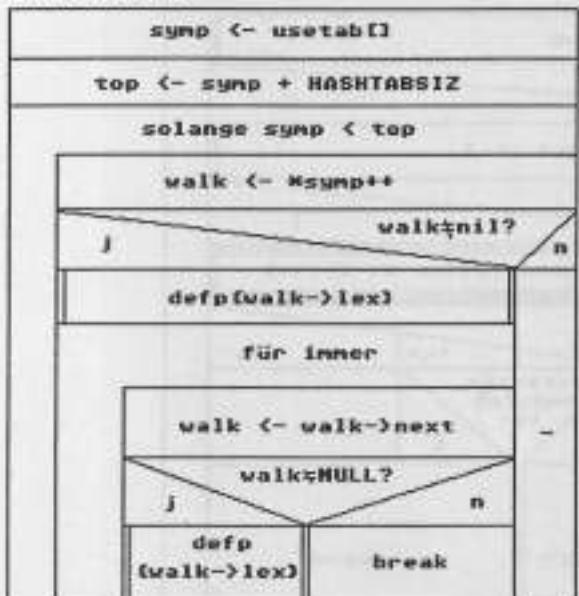
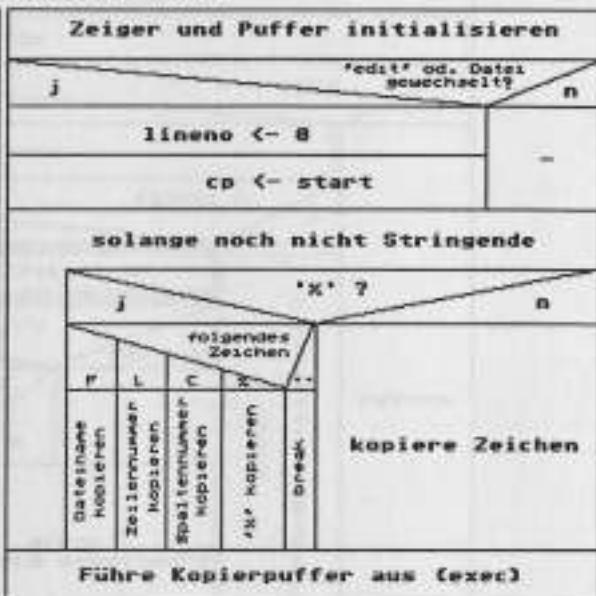
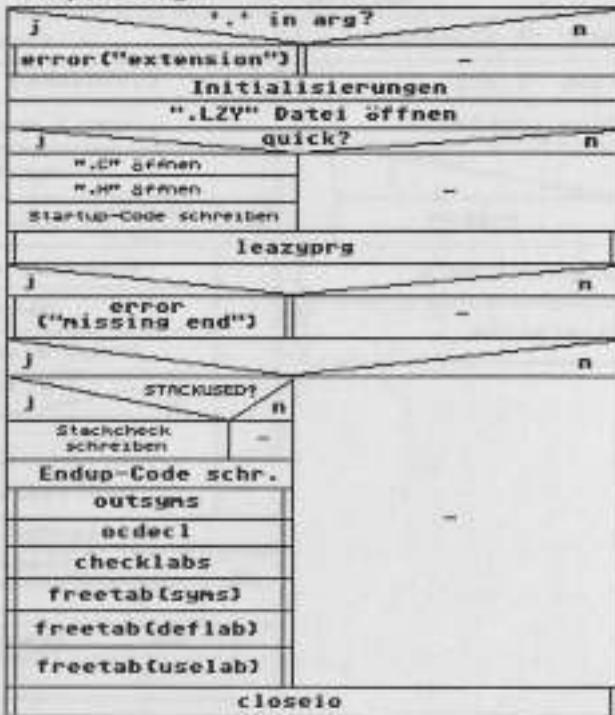
p	2088	2113	235	454	601
	617	619	624	625	630
	657	659	661	664	665
	666	670	692	694	696
	697	698	734	737	740
	764	767	769	770	771
	775	794	809	812	828
param	182	2222			
paramptr	2222	2232	2237	2240	2243
	2256	2258			
parname	171	2334	2385		
pathname	2197	2204	261		
patptr	2222	2232	2235	2238	2241
	2245				
patt	178	2222	2334	2388	2389
pattex()	2219	2413	2425	2437	2447
	2448	2453	345		
perror()	2097	2260			
pext	2091	2092	2123	2131	2147
	234				
prgid	120				
printf()	1961	1971	2080	2106	2183
	2272	2276	2289	2300	2322
	2324	2330	2334	2335	2353
	2384	2387	2404	2405	2410
	2528	2543	456	488	590
	601	613	651	661	666
	809				
procenv	2430	2443	245	2458	471
prompt	200	2353	2410		
protocol()	1449	1507	1541	1582	1608
	1634	1660	1693	1724	1751
	1779	1811	1854	1882	1908
	1924	500			
ps	2117	2138	2144		
putc()	522				
putchar()	1274	2580			
puts()	2171	2203	2294	2581	505
	570	631	634	673	797
q	692	697	699	790	821
	840	842	843	845	
qexpr()	1567	1838	1840	319	
qexprlist()	383	320			
qfactor()	317				
qid()	1527	1529	1565	1836	311
qidlist	374	395	405	423	431
qidlist()	312				
qnum()	313				
qnumid()	314				
qop()	310				
qrelation()	1480	1679	316		
qterm()	318				
quick	152	2115	2154	2419	2432
	2445	2454	2552	2574	560
	585	590	720	808	

r	715	717	719		
relation()	1121	1454	1665	282	
repenv	2081	2098	2261	2406	246
ret	185	2276			
retcode	2197	2207	2224	2258	2402
	261				
s	1010	1393	2022	2025	2028
	2033	570	732	734	891
scanf()	1426				
setbuf()	2105	2129	2134		
setjmp()	2406	2430	2443	2458	
show()	2285	348			
size_t	2296	254	2555	2556	2557
	534				
sizeof()	254	774			
sline	2113	2272	2295	2296	2303
	232				
sname	2075	2093	2109	2124	238
	464				
stackused	148	1754	1782	2009	2157
stage	1468	1490	150	1670	1681
	1697	1704	1709	1712	1728
	1731	1737	1738	1827	1842
	1861	1870	1887	1895	2150
	2152	519			
start	2088	2229	2244	237	2407
	457	459	816		
str	500	505	531	534	536
	542				
strcat()	2033	2034	2205	2206	2237
	2240	2243	2256	2287	2298
	542	543			
strchr()	2075				
strcmp()	2028	769			
strcpy()	2075	2092	2123	2131	2147
	2204	2287	2298	589	778
strlen()	534	776			
strncmp()	2296	536	717	819	
strrchr()	2091	2138	822		
swapused	149	1510	2012		
symbol	108	110			
symbol_t	112	1993	1994	2025	2046
	2047	250	271	274	609
	610	617	648	649	657
	685	687	688	692	761
	764	773			
symfmt	601	616	656	661	666
symp	1993	1994	1996	1997	2046
	2047	2049	2050	609	610
	613	614	617	648	649
	654	657	687	688	690
	692				
syms	1993	2119	2177	250	254
	609	648			
syntax_tree	157	2330	2372	2372	502
system()	2207				

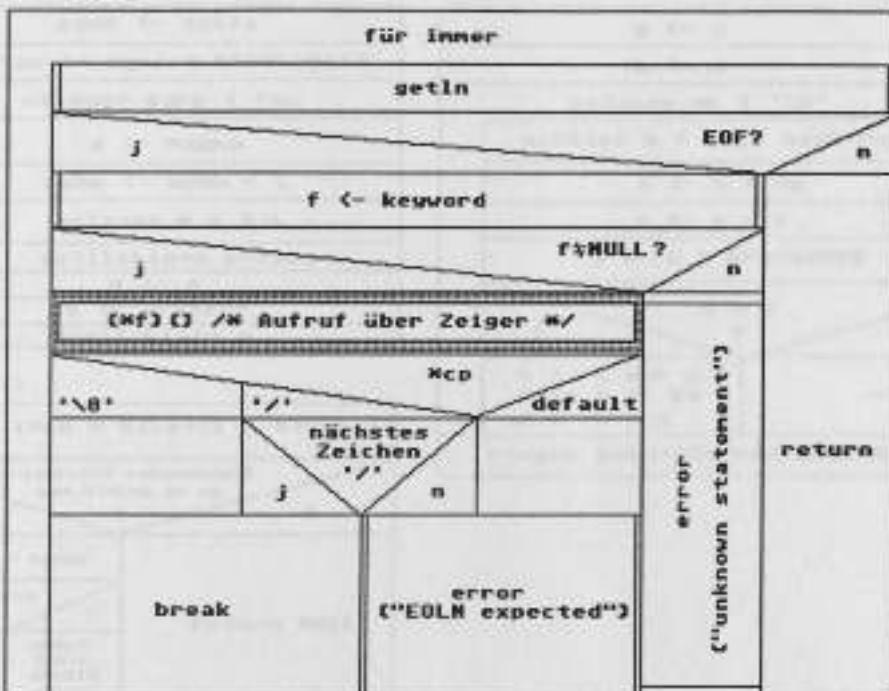
t	522	813	842		
tab()	264	486	504		
tabentry	358	581	714		
table	365	581	685	687	714
	761	764	773	775	776
tabsize	2119	2120	2121	254	
term()	1193	284			
tida	2071	2108			
time()	2071				
time_t	2071				
tm	2072	2183			
to	97				
top	1994	1996	2047	2049	610
	614	649	654	688	690
toupper()	2235	2354	2411	2570	
tp	581	584	585	585	588
	589	714	715	717	720
trace	151	1912	1928	2335	2367
	2551	2572	824		
u	790	812	813	814	816
	819	821	822		
usage()	2526	2576	2579	2584	350
uselab	2046	2121	2179	252	
v	790	821			
valid	153	2294	2421	2426	2440
	2449	2454			
walk	1993	1997	1999	2001	2002
	2025	2027	2028	2031	2046
	2050	2052	2054	2055	
workfile	181	2237	2287	2298	2324
	2405	2420	2433	2446	
wprotect	147	2126	557	558	

**error(msg)****closeio()****outsyms()****main(argc, argv)**

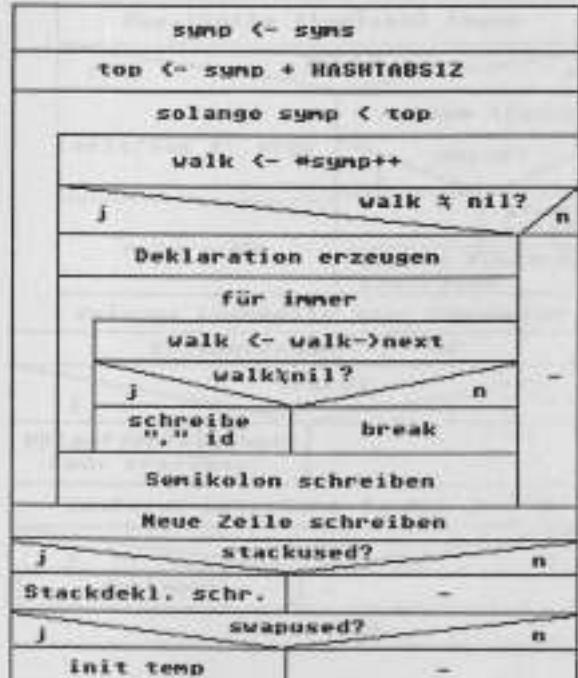
**ishell()****options()****show()**

**checklabs()****pattex(index)****compile(arg)**

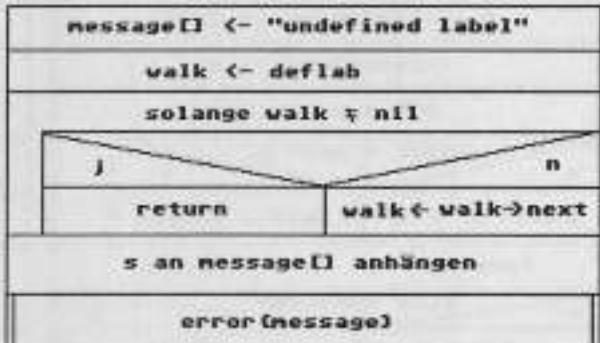
### leazyprg()

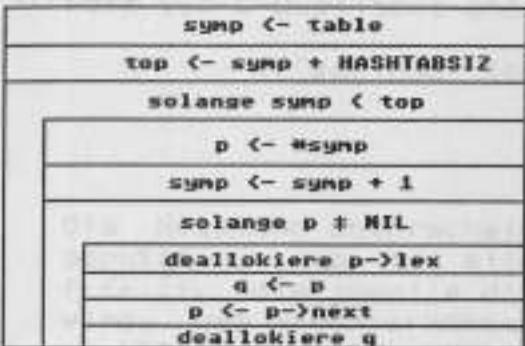
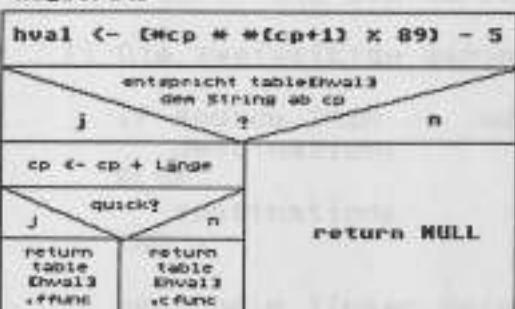
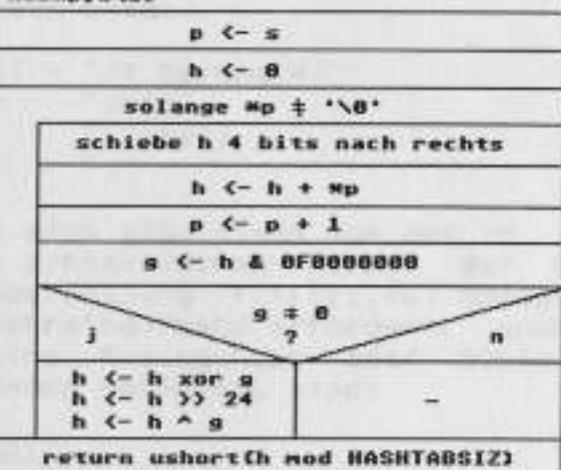
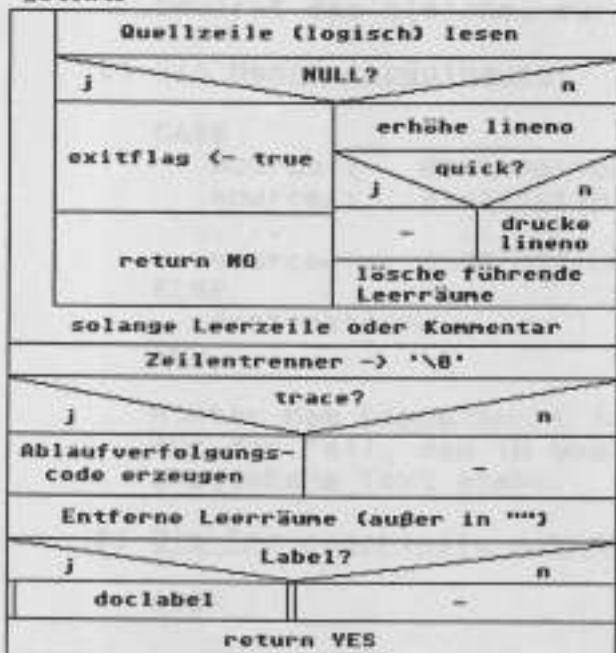
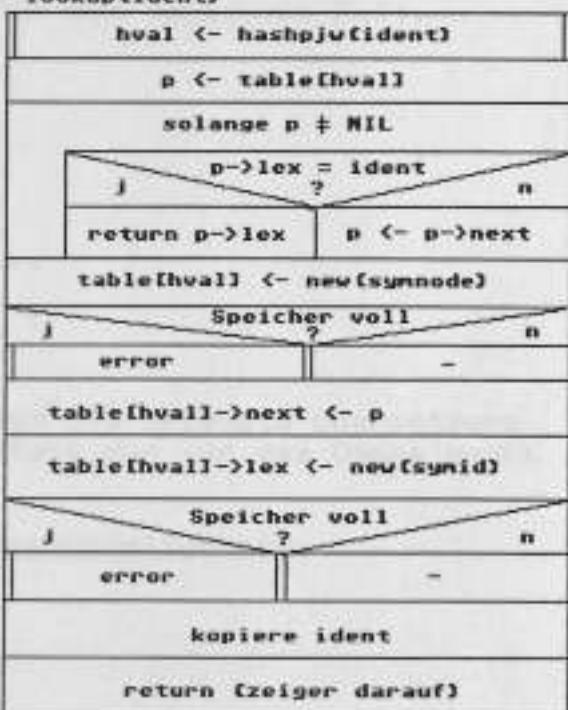


### ocdecl()



### defp(s)



**freetab()****keyword()****hashpjw(s)****getin()****lookup(ident)**

## Die formale Spezifikation des Codegenerators

Die folgende Spezifikation des Codegenerators beschreibt formal das Verhalten des Compilers, indem zu jedem Nonterminal n der Grammatik die dadurch erzeugte Übersetzung T(n) in Form von C-Quelltext angegeben wird:

```
T("springe" label) = /* Sprung */
                      "goto"
                      T(label)
                      ";"
```

Die Notation unterscheidet sich also etwas von der in [23] benutzten, wo zu allen syntaktischen Formen der Form  $f_1 f_2 \dots f_n$  die jeweils die Übersetzung  $T(f_1 f_2 \dots f_n)$  angegeben wird, was einen erhöhten Schreibaufwand erfordert, unübersichtlicher ist, aber keine Bedingungen oder Schleifen erfordert, wie sie im folgenden definiert sind:

### 1) Die zweiseitige Bedingung:

```
IF source THEN      oder    IF NOT source THEN
  destination1           destination1
ELSE                   ELSE
  destination2           destination2
END                    END
```

wenn beim linken Beispiel im Quellcode *source* folgt, so wird *destination<sub>1</sub>* generiert, sonst *destination<sub>2</sub>*. Rechts bewirkt das gleiche, nur für den negativen Fall.

### 2) Die Mehrfachbedingung:

```
CASE
  source1: destination1
  source2: destination2
  ...
  sourcen-1: destinationn-1
ELSE
  destinationn
END
```

Hinter dem Doppelpunkt folgt die erzeugte Übersetzung für den Fall, daß im Quelltext der vor dem Doppelpunkt angegebene Text steht.

### 3) Die Endlosschleife mit Abbruchmöglichkeit:

```
LOOP
  ...
  BREAK
  ...
END
```

Bis zum Auftauchen eines BREAK wird in einer Schleife immer wieder die angegebene Codefolge erzeugt. Dieses Konstrukt wird meist für Argumentlisten benutzt.

Kursiv gerdruckt sind Aufrufe von Funktionen, wie z.B. *gen()*, die Code für übergebene Literale, Bezeichner etc. zu erzeugen vermag, oder *lookup()*, das den Zugriff auf die Symboltabelle regelt, sowie interne Aktionen des Compilers, die in geschweifte Klammern eingeschlossen sind (" { ... } " ).

Die Beschreibung wurde aus der Implementierung zugrundeliegenden Grammatik erstellt, die mehr Produktionen als die Grammatik aus der Sprachdefinition enthält, um die Übersetzung zu vereinfachen, und aus der soweit möglich jegliche Rekursion entfernt wurde. Doch nun zur eigentlichen Beschreibung des LEAZY-Codegenerators:

```
T(S)      =  "#include <stdio.h>"  
           "#include <stdlib.h>"  
           "#include <"  
           gen(filename(header))  
           ">"  
           "void main(void) {"  
           LOOP { stage = 0 }  
           T(statement)  
           IF EOF THEN  
               BREAK  
           END  
           END  
           IF { stackused } THEN  
               "if(SP<S!;SP>&(S[8192])) {"  
               "puts(\"stack error\");"  
               "exit(1);"  
               "}"  
           END  
           "exit(0);"  
           "}"  
           outsyms()  
  
T(statement) = CASE  
    "cr":          T(cr)  
    "quit":        T(quit)  
    "request":     T(request)  
    "print":       T(print)  
    "set":         T(set)  
    "if":          T(if)  
    "continue":    T(continue)  
    "for":         T(for)  
    "push":        T(push)  
    "pop":         T(pop)  
    "exit":        T(exit)  
    "stop":        T(quit)  
    "loop":        T(loop)  
    "while":       T(while)  
    "trace":       T(trace)  
    "untrace":     T(untrace)  
    "inc":         T(inc)  
    "dec":         T(dec)  
    "clr":         T(clr)  
    "else":        T(else)  
    "swap":        T(swap)  
    "end":         T(end)  
END
```

```

T(label)      =   "L"
                  T(id)
                  ":"  

T(op)         =   CASE
                  "=":   "=="  

                  "#":   "!="  

                  ">":   ">"  

                  "<":   "<"  

                  ">=":  ">="  

                  "<=":  "<="
                  END  

T(id)         =   " "
                  gen(lookup(id))  

T(num)        =   gen(strtol(num))  

T(numid)      =   CASE
                  "0": "1": "2": "3": "4": "5":  

                  "6": "7": "8": "9": "0": "-": T(num)
                  ELSE
                  T(id)
                  END  

T(relation)   =   T(expr)
                  T(op)
                  T(expr)  

T(factor)     =   IF "(" THEN
                  "("
                  T(expr)
                  ")"
                  ELSE
                  T(numid)
                  END  

T(term)        =   T(factor)
                  LOOP
                  CASE
                  "*": "*"
                  "/": "/"
                  "%": "%"
                  ELSE
                  BREAK
                  END
                  T(factor)
                  END  

T(expr)        =   T(term)
                  LOOP
                  CASE
                  "+": "+"
                  "-": "-"
                  ELSE
                  BREAK
                  END
                  T(term)
                  END

```

```

T(cr)      =      "putchar('\\n');"

T(print)   =      LOOP
                  IF "" THEN
                      "printf(\"%s\", "
                      gen(strlit)
                      ");"
                  ELSE
                      "printf(\"%ld\", "
                      T(expr)
                      ");"
                  END
                  IF NOT ",," THEN
                      BREAK
                  END
                  END
                  END

T(quit)    =      "exit(0);"

T(continue) =      "goto L"
                    gen(lookup(id))
                    ";" 

T(request) =      LOOP
                  "scanf(\"%ld\", &"
                  T(id)
                  ");"
                  IF NOT ",," THEN
                      BREAK
                  END
                  END
                  END

T(swap)    =      "_temp="           { swapused = YES }
                    temp1 = T(id)
                    ":" 
                    gen(temp1)
                    "="
                    temp2 = T(id)
                    ":" 
                    gen(temp2)
                    "=_temp;" 

T(set)     =      LOOP
                  T(id)
                  "="
                  T(expr)
                  ":" 
                  IF NOT ",," THEN
                      BREAK
                  END
                  END
                  END

T(clr)     =      LOOP
                  T(id)
                  "=0L;"
                  IF NOT ",," THEN
                      BREAK
                  END
                  END
                  END

```

```
T(inc)      =  LOOP
                T(id)
                "++;" 
                IF NOT "," THEN
                    BREAK
                END
            END

T(dec)      =  LOOP
                T(id)
                "--;" 
                IF NOT "," THEN
                    BREAK
                END
            END

T(while)    =  "while("
                T(relation)
                "){"      { stage++ }

T(else)     =  T(end)
                "else{"   { stage++ }

T(if)       =  "if("
                T(relation)
                ")"
                IF "continue" THEN
                    T(continue)
                ELSE
                    "{"      { stage++ }
                END

T(end)      =  "}"    { --stage }
```

```
T(push)      =    LOOP
                  "*SP++="      { stackused = YES }
                  T(expr)
                  ","
                  IF NOT "," THEN
                      BREAK
                  END
              END

T(pop)       =    LOOP
                  T(id)
                  "=**-SP;"
                  IF NOT "," THEN
                      BREAK
                  END
              END

T(for)        =    "for("
                  temp = T(id)
                  "="
                  T(expr)
                  ";"
                  gen(temp)
                  "<="
                  T(expr)
                  ";"
                  gen(temp)
                  "++){"   { stage++ }

T(loop)       =    "for(;;){"
                  { stage++ }

T(exit)       =    "break;"

T(trace)      =    { trace = YES }

T(untrace)    =    { trace = NO }
```

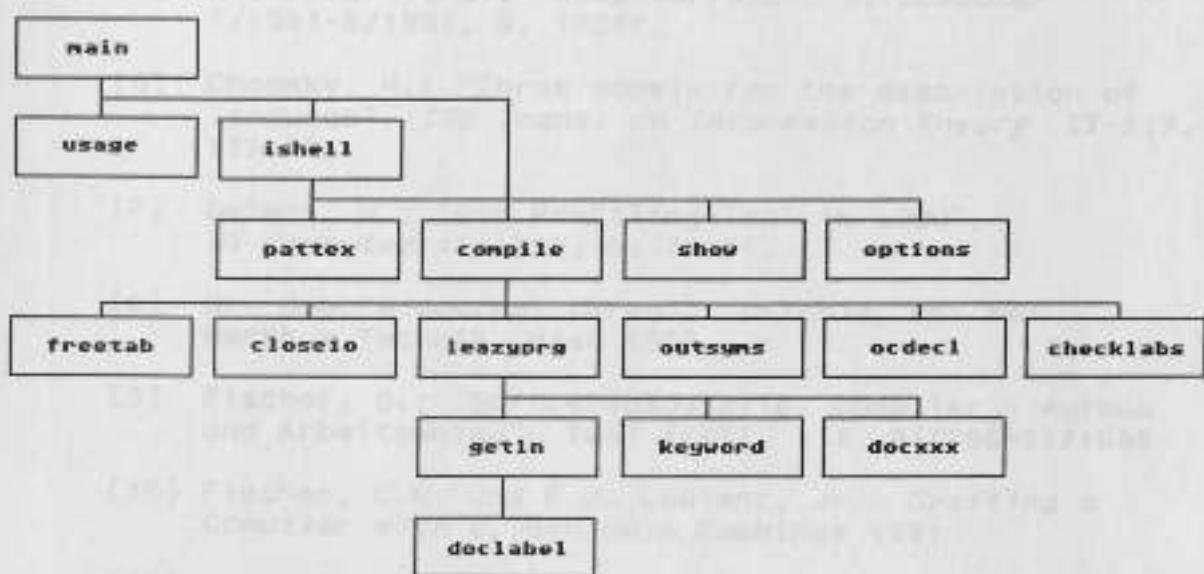


Abb. 8.1) Die funktionale Struktur des Programms

9) BIBLIOGRAPHIE

- [1] Aho, A.V., R. Sethi und J.D. Ullman: *Compilers - Principles, Techniques and Tools*, Addison-Wesley, Reading, Mass., 1986
- [2] Ambrosch, W. und F. Beer: "Die Compilermacher - Teil 2: YACC", c't 2/1991, S. 222ff.
- [3] Bailey, C.B. und R.E. Jones: "Usage and Argument Monitoring of Mathematical Library Routines", *ACM - Transactions on Mathematical Software* 1, 3 (1975), S. 196 ff.
- [4] Brooks, F.P.: *The mythical man-month*, Reading, Mass., 1975
- [5] Chakravarty, M.: "Compiler-Bau", *ST Computer* 1/1991-5/1991, S. 123ff.
- [6] Chomsky, N.: "Three models for the description of language", *IRE Trans. on Information Theory* IT-2:3, 113-124
- [7] Detert, U.: "Ein Profiling-Tool in LOGO", *ST Computer* 11/1988, S. 79 ff.
- [8] Dr. Dobb's Journal (Hrsg.): *C-Tools*, Dt. bei Markt & Technik, Haar 1986
- [9] Fischer, O.: "Softwarebaustelle. Compiler - Aufbau und Arbeitsweise", Teil I-III, c't 9/1985-11/1985
- [10] Fischer, C.N. und R.J. LeBlanc, Jr.: *Crafting a Compiler with C*, Benjamin Cummings 1991
- [11] Hopcroft, J.E.: *Introduction to automata theory, language and computation*, Reading, Mass. 1979
- [12] Johnson, S.C.: "Yacc - yet another compiler compiler", Computing Science Technical Report #32, AT&T Bell Laboratories, Murray Hill, N.J. 1975
- [13] Kernighan, B.W. und D.M. Ritchie: *The C programming language*, Prentice-Hall 1978
- [14] Kernighan, B.W. und D.M. Ritchie: *Programmieren in C*, 2. Auflage: ANSI-C, Hanser, München, Wien 1990
- [15] Knuth, D.E.: *The Art of Computer Programming*, Vol. I, Fundamental Algorithms, Addison-Wesley, Reading, Mass. 1968, 1973
- [16] Kopp, H.: *Compilerbau - Grundlagen, Methoden, Werkzeuge*, Hanser, München, Wien 1988

- [17] Lesk, M.E.: "Lex - A Lexical Analyzer Generator"  
Computing Science Technical Report No. 39,  
Bell Labs, Murray Hill 1975
- [18] Peters, F.E.: *Einführung in mathematische Methoden der Informatik*, BI Wissenschaftsverlag, Mannheim, Zürich 1974
- [19] Seimet, U.: "Assembleroptimierung", *ST Computer* 7/1989, S. 157 ff.
- [20] Wachtmeister, F.: "Programmtransformationen - nicht nur für Datenflußrechner", *TOOLBOX* 5/1989
- [21] Wallwitz, R. und H. Stäudel: "Mit dem Turbo nach C", *TOOLBOX* 9/1989, S. 66 ff.
- [22] Weber, H.: "Erkennen und Handeln - Lexikalische Analyse mit LEX", *iX* 5/1989, S. 80 ff.
- [23] Wirth, N.: *Compilerbau - Eine Einführung*, 4. Aufl., Teubner LAMM, Stuttgart 1977, 1986
- [24] Wirth, N.: *Programming in Modula 2*, 3rd ed., Springer 1985
- [25] Zima, H.: *Compilerbau I - Analyse*, BI Wissenschaftsverlag, Mannheim 1989
- [26] Zima, H.: *Compilerbau II - Synthese und Optimierung*, BI Wissenschaftsverlag, Mannheim 1989