TECHNISCHE UNIVERSITÄT CHEMNITZ· - ZWICKAU FAKULTÄT FÜR INFORMATIK

PROFESSUR SYSTEMPROGRAMMIERUNG UND BETRIEBSSYSTEME

Diplomarbeit

Ein Pascal-Compiler für die Betriebsysteme OS/2 Warp, Windows 95 und Windows NT - Design und Implementierung

eingereicht von:	Rene Nürnberger
geboren am:	26. Juni 1969 in Karl-Marx-Stadt
Betreuender Hochschullehrer:	Prof. Dr. W. Kalfa
Betreuer:	Dipl.-Inf S. Graupner
Eingescannt und Überarbeitet	Ing. Wolfgang Draxler

Chemnitz, den 13. Februar 1997

1 Inhaltsverzeichnis

2 Abkürzungsverzeichnis

TS .....T erminalsymbol
NTS ….....Nichttenninalsymbol
BNF ….....Backus Naur Form
EBNF .....Erweiterte Backus Naur Form
ASCÜ . .__ .....American Standard Code for Information Interchange
TOS ….....Top of Stack (oberstes Stackelement)
UNCOL ..... Universal Computer Oriented Language
GAG ….....Gerichteter azyklischer Graph
ATS ….....abstraktes Terminalsymbol
TLS .....Thread Local Storage

3 Kapitel 1 Einleitung

Der Compilerbau wird oft als ,,Konigsdisziplin" der Informatik bezeichnet. In der Tat umfasst das Gebiet des Compilerbaus viele Disziplinen, welche ein akademisch gebildeter Informatiker heutzutage beherrschen muß. Die Ausbildung zum Informatiker soll nicht nur fundierte praktische Kenntnisse, sondern auch Einsichten und Verständnis für die Funktionsweise eines Computers vermitteln. Ein Compiler stellt das Bindeglied zwischen der Hardware eines Rechners und dem Benutzer bzw. der Software her. Compiler sind Werkzeuge zur Übersetzung einer Sprache in eine andere. In der Regel arbeitet jeder Informatiker täglich mit Compilern, ein Verständnis der internen Struktur eines Compilers ist selten notwendig und doch oft hilfreich. Informatiker, welche verstehen, wie ein Compiler arbeitet, sind oft bessere Informatiker. Darüber hinaus ist der Bau eines Compilers geradezu ein Lehrstück in Software-Engineering und modernen Programmiertechniken.

Das Gebiet des Compilerbaus hat eine relativ lange Tradition. Die ersten Compiler entstanden schon in den frühen 50er Jahren. Der Compilerbau ist damit eine der ältesten Informatikdisziplinen überhaupt. Im weiteren Verlauf der Entwicklung wurden leistungsfähige Algorithmen und Softwarewerkzeuge entwickelt, um die überaus komplexen Aufgaben zu erleichtern, die der Bau eines Compilers mit sich bringt. Erste Grundlagenarbeiten zu Verfahren und Techniken eines Übersetzers stammen ebenfalls aus den 50er Jahren (z.B. [Chr56] und [Ers58]) und flossen in die Entwicklung der ersten Assembler (welche auch Compiler sind) sowie der ersten Hochsprachencompiler Algol 60 und Algol 68 ein.
Ein Compiler selbst ist eine überaus große und komplexe Applikation. Beispielsweise betrug der Aufwand zur Implementation des ersten Fortran Compilers laut [Aho90] circa 18 Mannjahre. Aus diesem Grund blieb das Schreiben von Compilern lange einigen wenigen Firmen vorbehalten. Aufgrund der rasanten Entwicklung der Rechentechnik, der systematischen Methoden und der Techniken des Compilerbaus kann ein einfacher Compiler heutzutage auch von Studenten entwickelt werden.

3.1 Aufgabenstellung

In dieser Diplomarbeit soll gezeigt werden, daß ein funktionsfähige moderner Compiler im Rahmen einer Diplomarbeit entworfen und implementiert werden kann. Besonderer Schwerpunkt wird auf die interne Struktur des Compilers gelegt. Neben den Möglichkeiten zur Generierung eines effizienten Zwischencodes werden auch für den Compilerbau notwendige Datenstrukturen diskutiert und die Relevanz von Betriebssystemfunktionen auf den Compilerbau aufgezeigt.

Aufbauend auf diese Untersuchungen soll der Prototyp eines Object-Pascal Compilers für die Betriebssysteme OS/2 Warp, Windows 95 und Windows NT implementiert werden. Außerdem ist eine Laufzeitbibliothek zu entwerfen, welche das Laufzeitsystem beinhaltet und weitere elementare Funktionen bereitstellt. Zusätzlich soll untersucht werden, inwieweit eine Generierung von Standard-Linkformaten sinnvoll und möglich ist.
Bei der Implementation ist ein mögliches späteres Portieren auf andere Plattformen zu berücksichtigen. Dies erfordert eine besonders sorgfältig Planung und Strukturierung des Compilers. Wesentliche Teile des Compilers sollen maschinenunabhangig implementiert werden. Der generierte Zwischencode soll weitgehend unabhängig von der Zielplattform implementiert und optimiert werden.

Voraussetzung für die Implementation eines kompletten Compilers innerhalb einer Diplomarbeit sind grundlegende Kenntnisse der Methoden des Compilerbaus. Aufbauend auf eine ca. 3 jährige Erfahrung auf dem Gebiet der kommerziellen Compilerentwicklung konnte auf eine Vielzahl von Verfahren und Methoden aus vorhandenen Programmierbibliotheken zurückgegriffen werden. Eine vollständige Neuimplementation aller Komponenten hätte den Rahmen dieser Diplomarbeit bei weitem gesprengt. Außerdem werden an gewissen Stellen Einschränkungen und Kompromisse eingegangen, um den zeitlichen Rahmen einer Diplomarbeit einzuhalten.

3.2 Motivation und Ziele

Im Rahmen dieser Diplomarbeit soll ein kompletter 32 Bit Pascal-Compiler für die Intel-Plattform entworfen und implementiert werden. Der zu implementierende Compiler soll den kompletten Sprachumfang der objektorientierten Sprache Object-Pascal beherrschen. Dassei wird vom vorhandenem Design und der Implementierung eines existierenden und im Quelltext vorliegenden Pascal-Compilers ausgegangen [1] Dieser vorhandene Compiler weist eine Reihe von gravierenden Schwachen auf:

- Der Compiler ist schlecht strukturiert. Daraus resultiert eine schlechte Wartbarkeit und eine hohe Fehleranfälligkeit. Zudem erzeugt der Compiler nur Zielcode für OS/2 Systeme. Durch die schlechte Strukturierung des Compilers ist eine Anpassung an andere Zielsysteme schwer möglich. Viele Komponenten des Compilers sind speziell auf OS/2 ausgerichtet. Durch eine klare Strukturierung und Modularisierung soll in der vorliegenden Arbeit auch eine Codeausgabe für Windows 95 und Windows NT realisiert werden.
- In diesem Compiler wird auf die explizite Erzeugung eines Zwischencodes verzichtet. Aus diesem Grunde ist eine Optimierung des Zielcodes nur in sehr begrenztem Umfang möglich. Außerdem erschwert dieser Umstand wesentlich die Portierung des Compilers.
- Der vorliegende Compiler ist relativ langsam. Da der vorhandene Compiler auch professionell in der Praxis eingesetzt wird, wurden von einigen Fachzeitschriften auch Tests dieses Compilers durchgeführt [2] In diesen Tests spielte die mangelnde Compiliergeschwindigkeit immer eine große Rolle. Durch die Verwendung besserer Algorithmen konnte mit Sicherheit eine Geschwindigkeitssteigerung erreicht werden.

Ausgehend von diesem Ist-Zustand ist es notwendig, ein Redesign des vorhandenen Compilers durchzuführen, um folgende Ziele zu erreichen:

- Portierbarkeit[3] des Compilers

1Es handelt sich hierbei um den selbstentwickelten professionellen OS/2 Pascal Compiler "Speed-Pascal", welcher in Fachkreisen größtenteils positive Kritiken erhielt.

2Man vergleiche im Literaturanhang z.E. [057/96], [To5/96], [052/951 und [052/96]

3Unter „Portierung" und „Portierbarkeit" soli im weiteren Verlauf der Arbeit ein Anpassen des Backends an eine neue Zielarchitektur verstanden werden, d.h. eine Codegenerierung für eine andere Rechnerarchitektur oder ein anderes Betriebssystem.

Beim Entwurf und der Implementation des neuen Compilers soll eine Portierung des Compilers auf andere Plattformen berücksichtigt werden. Grundlegende Komponenten des Compilers müssen aus diesem maschinenunabhängig entworfen und implementiert werden.

- Wartbarkeit des Compilers
  Der herzustellende Compiler soll eine einfache Fehlerbehebung und -diagnostik ermöglichen. Dies bedingt eine gute Strukturierung der Compilerkomponenten und eine strenge Modularisierung des Compilers.

- Erzeugung eines effizienten Zielcodes
  Ein wesentlicher Schwachpunkt des vorhandenen Compilers ist die schlechte Qualität des erzeugten Zie1codes. Dies liegt zum einem am Verzicht auf die Erzeugung eines Zwischencodes, zum anderen in der schlechten Strukturierung des Compilers. Für den zu entwerfenden Compiler soll ein geeigneter Zwischencode ausgewählt oder erstellt werden. Auf diesem Zwischencode sollen Optimierungen ausgeführt werden.

- hohe Compilergeschwindigkeit
  Die Compiliergeschwindigkeit ist in der "Welt der PC-Benutzer" ein wesentliches Kriterium für die Beurteilung der Qualität eines Compilers. In vielen Compilertests wird die Compiliergeschwindigkeit explizit ermittelt und bewertet[4] . Aus diesem Grunde sollen spezielle, schnelle Algorithmen zum Einsatz kommen, auch wenn dies teilweise durch eine erhöhte Komplexität erkauft werden muß.

- Vollständigkeit und Korrektheit des Compilers
  Der zu entwerfende Compiler soll alle Sprachkonstrukte der Quellsprache unterstützen. Ausgehend von einer formalen Grammatikdefinition wird der Compiler entworfen und implementiert.

- gute Strukturierung
  Aufbauend auf Untersuchungen zu möglichen Strukturvarianten wird die am besten geeignete Variante ausgewählt und implementiert.

Diese Ziele definieren gleichermaßen die Randbedingungen dieser Diplomarbeit. Bei Entscheidungen, welche das Design oder die Implementation des Compilers betreffen, werden diese Forderungen in besonderer Weise berücksichtigt. Außerdem soll das Redesign möglichst unter Ausnutzung vorhandener Erfahrungen im Compilerbau durchgeführt werden.

Vorhandene Quelltexte und Komponenten des vorhandenen Compilers sollen nach Möglichkeit im neuen Entwurf berücksichtigt und verwendet werden.

Wesentliche Schwerpunkte der Arbeit beim Redesign des Compilers liegen in der klaren Struktur des neuen Compilers und in der Auswahl und Generierung eines Zwischencodes.

3.3 Vorgehensweise

Die vorliegende Arbeit beginnt mit einer kurzen Darstellung der Grundlagen des Compilerbaus sowie den Eigenschaften von Quell- und Zielsprache. Diese Überlegungen fuhren in die Thematik des Compilerbaus ein. Anschießend wird der allgemeine Aufbau eines Compilers diskutiert und verschiedene grundsätzliche Ansätze zur Realisierung der Struktur eines Compilers untersucht. Grundlegende

4Man Vergleiche z.B. [To5/96] fur den hier zugrundeliegenden Compiler "Speed-Pascal" und den Pascal Compiler "Virtual-Pascal" oder [T04/93] fur C++ Compiler

Komponenten eines Compilers werden anhand ihrer Mechanismen und Funktionsweisen erlüutert und in das Gesamtkonzept eines Compilers eingeordnet. Die für die Implementierung ausgewählte Variante wird im Detail erläutert. Danach wird gezeigt, wie die Realisierung des Zwischencodes erfolgen kann. Dassei wird insbesondere auf Vor- und Nachteile verschiedener Realisierungsvarianten eingegangen. Die Schnittstellen zwischen den verschiedenen Komponenten des Compilers werden diskutiert und ausgewählt. Die Untersuchung der konkreten Implementierung erfolgt dann anhand der diskutierten Konzepte und Mechanismen. Dassei wird auch auf die Realisierung der Laufzeitbibliothek und die Möglichkeit der Erzeugung von Standard-Linkformaten hingewiesen. Auf den Einfluß der Zielsystemarchitektur wird bei Bedarf eingegangen und die Auswirkungen auf die beschriebenen Konzepte erläutert. abschließend erfolgt die Diskussion von Erweiterungen bzw. Verbesserungen am hier vorgestellten Konzept und die mögliche Vorgehensweise beim Portieren des Compilers.
Die im Rahmen dieser Diplomarbeit vorgenommenen Untersuchungen werden auf einem INTEL Pentium 133 mit 32 Mb Hauptspeicher unter folgenden Betriebssystemen durchgeführt:

1. OS/2 Warp V3 und OS/2 Warp V4, die Programmentwicklung erfolgt mit Hilfe Von Borland C++ für OS/2, Version 2.0.
2. Windows 95 und Windows NT, die Programmentwicklung erfolgt mit Hilfe Von Borland C++ für Win32, Version 4.5.

Quelltextbeispielen bzw. der Diskussion von Programmierfragmenten der Compiler-Implementation liegt ANSI C bzw. C++ zugrunde. Auf Verwendung von compilerspezifischen Funktionen wird soweit als möglich verzichtet, um eine leichte Migration des Compilers zu ermöglichen.

4 Kapitel 2
Grundlagen des Compilerbaus

Da für das Verständnis des inneren Aufbaus eines Compilers bestimmte Grundkenntnisse wichtig sind, wird in den nächsten Abschnitten eine kurze Einführung in die theoretischen Grundlagen des Compilerbaus und der formalen Sprachen gegeben. Die meisten der nachfolgend beschriebenen Techniken und Konzepte wurden bereits in den 50er und 60er Jahren entwickelt und sind heute weitestgehend erforschte Gebiete. Da viele Wissensgebiete in einschlägiger Literatur detailliert dargestellt sind, werden hier nur diejenigen Konzepte erläutert, welche zum weiteren Verständnis der Arbeit notwendig sind. Anregungen zur weiteren Vertiefung finden sich im Literaturanhang.

4.1 Sprache und Syntax

Jede Sprache besitzt eine konkrete Struktur, Programmiersprachen bilden da keine Ausnahme. Die Struktur einer Sprache wird durch ihre Grammatik oder Syntax bestimmt. Dies soll am Beispiel der deutschen Sprache verdeutlicht werden. Ein korrekter deutscher Satz kann mit einem Subjekt, gefolgt von einem Prädikat beginnen. Die Sprachstruktur einer Sprache wird durch Regeln festgelegt. So konnte eine Regel zur Bildung deutscher Sätze lauten:

Satz ::= Subjekt Prädikat.

Die Bezeichner "Satz", "Subjekt" und ,,Prädikat" heißen Nichtterminalsymbole der Sprache. Nichtterminalsymbole haben stellvertretenden Charakter und beziehen sich auf syntaktische Klassen, welche durch andere Symbole substituiert werden können. Das Symbol ,,::=" bedeutet soviel wie ,,kann substituiert werden durch". Das obige Beispiel besagt also, daß das Nichtterminal "Satz" substituiert werden kann durch das Nichtterminal "Subjekt", gefolgt vom Nichtterminal ,,Prädikat". Damit die Definition der Sprache korrekt ist, muß jedes Nichtterminal genau einmal auf der linken Seite einer Regel auftauchen. Solche Regeln werden Produktionsregeln genannt. Die obige Regel kann also erweitert werden zur Regelmenge:

Satz ::= Subjekt Prädikat.
Subjekt ::= 'Er' | 'Sie'.
Prädikat ::= 'liest' | 'schreibt'.

Die Bezeichner in Hochkommata werden Terminalsymbole der Sprache genannt. Terminalsymbole haben elementaren Charakter und können nicht weiter substituiert werden. Die Menge von Terminalsymbolen einer Sprache wird auch als Alphabet der Sprache bezeichnet. Terminalsymbole und Nichtterminalsymbole bilden zusammen das Vokabular der Sprache. Im obigen Beispiel sind ,,Er", "Sie", "liest" und "schreibt" Terminalsymbole der Sprache. Ein Satz der Sprache ist syntaktisch korrekt, wenn ausgehend von einem Startsymbol durch Anwendung der Produktionsregeln der Satz in Terminalsymbole zerlegt werden kann.

Das Symbol ,|" besagt, daß für die Produktion mehrere mögliche Ableitungen existieren und hat die Wirkung von "oder". Die Produktionsrege! <Subjekt = 'Er' | 'Sie'> besagt also, daß das Niehtterminal "Subjekt" entweder mit dem Terminal ,,Er" oder dem Terminal "Sie" substituiert werden kann. 1m obigen Beispiel ist "Satz" das Startsymbol der Grammatik. Es lassen sieh genau 4 mögliche Sätze aus der obigen Grammatik erzeugen:

Er liest
Er schreibt
Sie liest
Sie schreibt

Eine Grammatik ist ein Tupel G = (V,∑,R,s)[5], wobei

- V eine endliche Menge von Symbolen beschreibt (Vokabular der Sprache). Diese Symbole können Terminalsymbole (im folgenden wird hierfür die Abkürzung TS verwendet) oder nicht-terminale Symbole (Abkürzung NTS) sein.
- ∑eine endliche Menge von sprachbildenden Zeichen oder Zeichenreihen (Terminalsymbole) beschreibt (Alphabet der Sprache). Es gilt immer: |∑| < |V|. Die Menge der NTS ergibt sieh aus der Mengendifferenz von V und ∑
- R eine Menge von Produktionsregeln beschreibt. Die linke Seite einer Produktionsregel muß mindestens ein NTS enthalten. Die rechte Seite einer Produktionsregel kann beliebige Zeichenreihen aus dem Vokabular der Sprache oder das Symbol ϵ für eine leere Zeichenreihe enthalten. Produktionsregeln werden oft auch als *syntaktische Regeln* bezeichnet. Zu jedem NTS gehört eine syntaktisehe Regel.
- s bezeichnet das Startsymbol der Grammatik. Ausgehend von diesem Symbol wird ein Satz der Grammatik abgeleitet. Das Startsymbol einer Grammatik ist immer ein NTS.

Für die Ableitung eines gegebenen Satzes aus der Grammatik existieren grundsätzlich 2 verschiedene Möglichkeiten:

1. Linksableitung:
   Regeln der Grammatik werden zuerst auf das in einer Regel am weitesten links stehende NTS angewendet.
2. Rechtsableitung
   Regeln der Grammatik werden zuerst auf das in einer Regel am weitesten rechts stehende NTS angewendet.

In der Praxis existieren zu einer Regel der Grammatik manchmal mehrere Varianten für Links- oder Rechtsableitungen. Auf diese Problematik wird im Abschnitt "Der Parser" naher eingegangen.

4.2 Die Chromsky Hierarchie von Grammatiken

Die Untersuchung von Grammatiken wurde in den 50er Jahren besonders von N. Chromsky vorangetrieben [Chr56]. Nach ihm ist eine Klassifizierung der möglichen Grammatiken benannt, die sogenannte Chromsky Hierarchie von Grammatiken.
Mit ∑* wird im weiteren die Menge aller Zeichenreihen bezeichnet, welche sich aus dem Alphabet ∑ bilden lassen. V* bezeichnet die Menge aller Zeichenreihen, die sich aus dem Vokabular V bilden lassen. Eine formale Sprache L über einem Alphabet ∑ist immer eine Teilmenge von ∑*. Die Bildungsvorschrift von L aus ∑* wird durch die Grammatik der formalen Sprache L festgelegt. Mit I wird im weiteren ein Satz der formalen Sprache bezeichnet.

---

5 Für die Darstellung einer Grammatik existieren in der Literatur verschiedene Varianten. In einigen Compilerbuchem wird das Vokabular der Sprache nicht mit in die Grarnmatikdefinition aufgenommen und statt dessen die Menge der Terminalsymbole bzw. Nichtterminalsymbole explizit aufgeführt. In dieser Arbeit wird die Darstellungsweise verwendet, welche an der TU-Chemnitz im Rahmen der Compilerbau- Vorlesung gelehrt wurde.

Wenn $x_0, x_1, x_2, ..., x_n \in V^*$, dann wird die Folge x_0 --> x_1, --> x_2 --> ... --> x_n als Ableitungsfolge von x_n aus x_0 bezeichnet, wobei die Übergange von x_i zu x_{i+1} durch Anwendung der Regel ri aus der Regelmenge R definiert sind. Eine Regel r_i wird notiert mit $r_i=(u_i, v_i)$. Die Symbole ui sind hierbei immer NTS, V_i sind beliebige Zeichenreihen aus dem Vokabular der Sprache. Diese Darstellung ist gleichbedeutend mit obiger Darstellungsweise $u_i ::= v_i$

Es existieren nach Chromsky 4 Klassen von Grammatiken:

1. Satzgliederungsgrammatik (Typ 0 Grammatik)
   Alle Regeln ri=(u_i, v_i) einer solchen Grammatik haben die Eigenschaft

   ui = x_1zx_2 und x_1, x_2, v ∈ V* und z ∈ V \ ∑

   Das bedeutet, daß die linke Seite einer Regel beliebige Symbole enthalten kann, mindestens jedoch I NTS. Die rechte Seite einer Regel kann durch eine beliebige Zeichenfolge aus dem Vokabular V* beschrieben werden.

   Einer solche Grammatik kann durch einen Turingautomaten berechnet werden. Es kann jedoch nicht allgemein entschieden werden, ob jeder mögliche gegebene Satz aus der Grammatik erzeugt werden kann. Es können Sätze existieren für die dies nicht entscheidbar ist. Aus diesem Grund ist dieser Grammatiktyp für den Compilerbau nur von theoretischer Bedeutung.

   Ein Beispiel einer solchen Grammatik ist G=(V, ∑, R, s) mit

   V = {S,A,B,'x' ,'y' ,'z')
   ∑={'x','y','z')
   s=S
   R={
   r1=('y'S'x', A'x''x'),
   r2={A,'y'S'x''y'B),
   r3=('z'B'x',''x''y')
   }

2. Kontextabhängig . Grammatik (Typ 1 Grammatik)

   Alle Regeln r_i = {u_i, v_i) einer solchen Grammatik haben die Eigenschaft

   u_i = x_1zx_2 und v_i=x_1wx_2 und x_1, x_2, ∈ V* und z ∈ V \ ∑und w ∈ V*

   Dieser Grammatiktyp entspricht einer Typ 0 Grammatik mit der Nebenbedingung, daß die Bedingung length{u) <= length{v) (Längenmonotonie) gilt.

   Für eine solche Grammatik kann nicht allgemein entschieden werden, ob ein gegebener Satz aus der Grammatik erzeugt werden kann. Die einzige Möglichkeit besteht im Abzahlen und Aufschreiben aller Möglichkeiten. Aus diesem Grund ist dieser Grammatiktyp für den Compilerbau ebenfalls nur von geringer Bedeutung.

3. Kontextfreie (Grammatik Typ 2 Grammatik)

   Alle Regeln r_i = (u_i, v_i) einer solchen Grammatik haben die Eigenschaft u_i ∈ V \ ∑und v_i ∈ V*. Dies

bedeutet, daß auf der linken Seite einer Regel immer genau ein Nichtterminal steht. Die rechte Seite einer Regel kann aus beliebigen Zeichenfolgen aus dem Vokabular V* bestehen. Eine Grammatik dieses Typs kann durch einen Kellerautomaten berechnet werden.

Ein Beispiel einer solchen Grammatik ist G=(V, ∑ R, s) mit

V = {expr, ziffer, '(', ')', '+', '*', '0' .. '9'}
∑= {'(', ')', '+', '*', '0' .. '9'}
R = {
r_l =(expr,expr'+' expr),
r_2={expr,expr' *, expr),
r_3={expr,'('expr')'),
r_4={expr,ziffer),
r_5={ziffer,'0'),
r_6={ziffer,'1'),
r_7=(ziffer,'2'),
r_8={ziffer,'3'),
r_9={ziffer,'4'),
r_{l0}={ziffer,'5'),
r_{ll}={ziffer,'6'),
r_{12}=(ziffer,'T),
r_{13}={ziffer,'8'),
r_{14}={ziffer,'9')
}

4. Reguläre Grammatik (Typ 3 Grammatik)

Alle Regeln $r_i = (u_i,.v_i)$ einer solchen Grammatik haben die Eigenschaft

$u_i \in V \setminus \sum$und (v_i = zw oder v_i = w) und z ∈ V \ ∑und w ∈ ∑* (Ünkslinear)
oder
$u_i \in V \setminus \sum$und (v_i = wz oder v_i = w) und z ∈ V \ ∑und w ∈ ∑* (rechtslinear)

Eine Grammatik dieses Typs kann durch einen endlichen Automaten berechnet werden. Die linke Seite einer Regel dieses Grammatiktyps ist immer ein NTS. Auf der rechten Seite einer Regel dürfen NTS nur am weitesten links (linkslineare Grammatik) oder am weitesten rechts (rechtslineare Grammatik) stehen. Dieser Grammatiktyp ist für den Compilerbau wichtig. da sich für jede Typ 3 Grammatik durch eine endliche Berechnung die Gültigkeit eines gegebenen Satzes effizient bestimmen läßt. Leider sind praktisch angewandte Hochsprachen wie Pascal meist keine regulären Sprachen. Typ-3-Grammatiken kommen hier jedoch bei der Erkennung von Grundsymbolen einer Sprache (siehe Abschnitt 2.4) zur Anwendung.

Ein Beispiel für eine linkslineare, reguläre Grammatik ist G=(V, ∑ R, s) mit

V = {z_1, z_2, z_3, z_4, z_5, z_6 ,z_7, ziffer, 'e', '+', '-', '.'}
∑= {'e', '+', '-', '.'}
s=z_4
R={
r_l=(z_4,z_3 zjffer),
r_2=(z_3,z_3 ziffer),

r_3=(z_3,z_2 '+'),
r_4=(z_3,z_2 '-'),
r_5=(z_2,z_l 'e'),
r_6=(z_4,z_1),
r_7=(z_l,z_l ziffer),
r_8=(z_1,ziffer),
r_9=(z_l,z_5 '.'),
r_{10}=(z_5, z_6 ziffer),
r_{ll}=(z_6,ziffer),
r_{12}=(z_6,z_7 '+'),
r_{13}=(z_6,z_7 '-'),
r_{14}=(z_7,ϵ)
}

Diese Grammatik beschreibt reele Zahlen mit oder ohne Angabe des Exponenten- Gültige Sätze der Sprache sind z.B. 0.5, aber auch $10.565e^{+10}$. Das NTS »ziffer« soll eine Zahl zwischen 0 und 9 beschreiben.
Da sich reguläre Grammatiken durch endlichen Automaten berechnen lassen, kann für jede reguläre Grammatik ein solcher Automat erstellt werden. Eine durch eine reguläre Grammatik beschriebene Sprache heßt reguläre Sprache. Der endliche Automat für die obige Beispielgrammatik mit dem Startzustand 0 ist in Abbildung 1 dargestellt. (Fehlzustände sind weggelassen, Endzustände sind durch einen Doppelkreis gekennzeichnet).

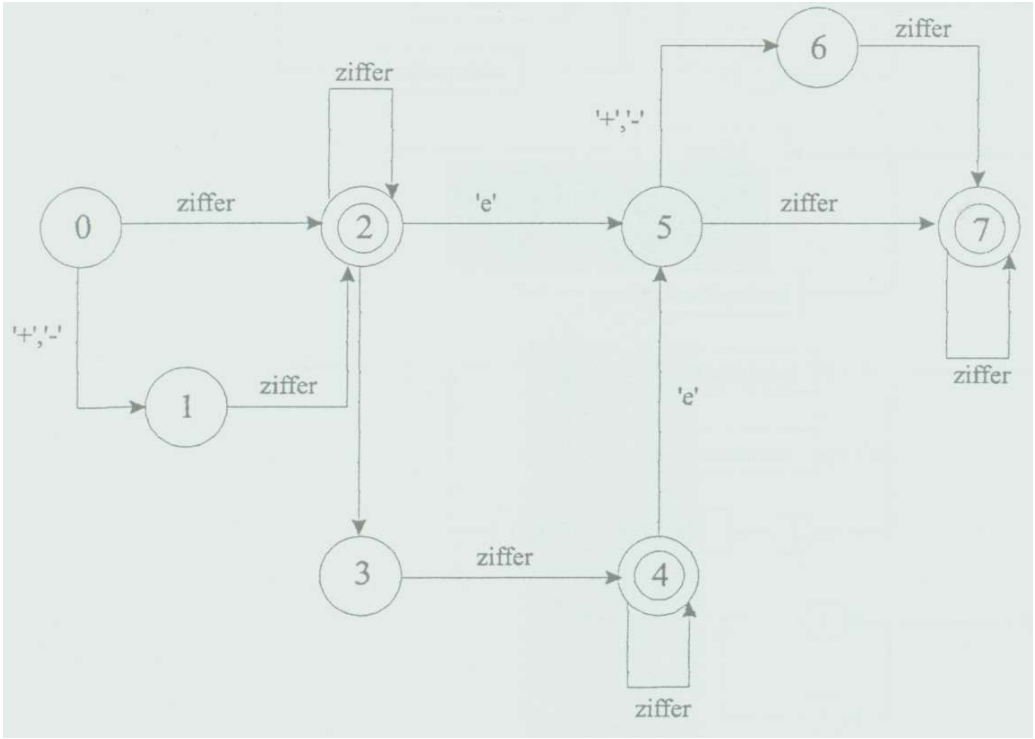


Abbildung 1: Der endliche Automat für die Beispielgrammatik

4.3 Darstellungsmöglichkeiten für Grammatiken

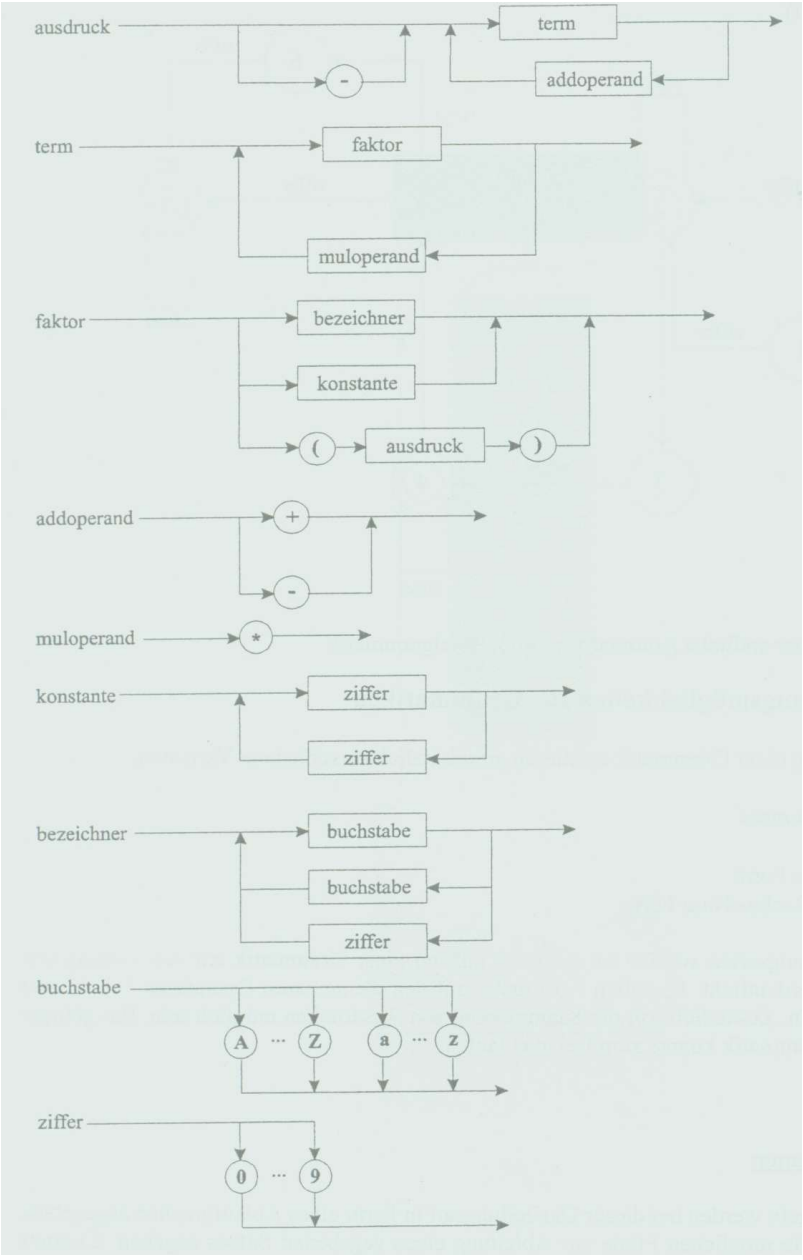
Zur Darstellung einer Grammatik existieren grundsätzlich verschiedene Varianten.

- Syntaxdiagramme
- Tupelform
- Backus Naur Form
- Erweiterte Backus Naur Form

Diese Darstellungsarten werden im weiteren anhand einer Grammatik zur Auswertung von Ausdrücken verdeutlicht. Es sollen Ausdrücke mit den elementaren Operatoren '+', '-' und '*' zulässig sein. Zusätzlich soll die Klammerung von Ausdrücken möglich sein. Ein gütiger Satz dieser Grammatik konnte zum Beispiel lauten:

i+j+(i*j)

1. Syntaxdiagramm
   Produktionsregeln werden bei dieser Darstellungsart in Form eines Ablaufgraphen angegeben, wobei Pfeile die möglichen Pfade zur Ableitung eines gegebenen Satzes angeben. Existiert kein solcher Pfad, so ist die Eingabe syntaktisch falsch. Terminalsymbole werden von einem Kreis bzw. einer Ellipse umgeben dargestellt, NTS innerhalb eines Rechtecks. Startsymbol der Grammatik ist das Nichtterminal 'ausdruck'.



2. Tupelform
   Eine Grammatik in Tupelform hat die Gestalt G = (V, ∑, R, s), wobei die Symbole V, ∑, R und s wie in Abschnitt 2.1 definiert sind.

   Die Grammatik zur Auswertung von Ausdrücken lautet also $G_{Ausdrücke}$ = (V, ∑, R, s) mit

   V= {ausdruck,term,faktor,addoperand,muloperand,konstante,bezeichner,buchstabe,ziffer, hilf1 hilf2 hilf3 'A'.. 'Z', 'a' .. 'z', '0' .. '9', '(', ')', '+', '-', '*'}

   ∑= {'A' .. 'Z', 'a' .. 'z', '0', '9', '(', ')', '+', '-', '*'}
   s=ausdruck
   R={
   r_1=(ausdruck,'-' ,hilf1),
   r_2=(ausdruck, hilf1),
   r_3=(hilf1, term),
   r_4=(hilf1, term addoperand hilf1),
   r_5=(term, faktor),
   r_6=(term, faktor muloperand term),
   r_7=(faktor, bezeichner),
   r_8=(faktor, konstante),
   r_9=(faktor,'(' ausdruck 'n,
   r_{10}=(addoperand, '+'),
   r_{11}=(addoperand,'-'),
   r_{12}=(muloperand, '.'),
   r_{13}=(konstante,ziffer hilf2),
   r_{14}=(hilf2,ziffer hilf2),
   r_{15}=(hilf2,E),
   r_{16}=(bezeichner,buchstabe hüD),
   r_{17}=(hilD,buchstabe hüf3),
   r_{18}=(hilf3,ziffer hüD),
   $r1_9$=(hilf3, ϵ),
   r_{20}=(buchstabe,' A' .. 'l'),
   r_{21}=(buchstabe,'a' .. 'z'),
   r_{22}=(ziffer,'0' .. '9')
   }

   Zur Darstellung der Grammatik wurden einige NTS als Hilfszustände neu eingeführt.

3. Backus Naur Form (BNF)

   Diese Darstellungsform wurde 1960 zur Beschreibung der Programmiersprache AIgo160, einem Vorlaufer von Pascal, von J. Backus und P.Naur eingefuhrt [Nau60]. Zur Beschreibung der Syntax wird eine Metasprache mit einigen Metasymbolen eingeführt. Metasymbole sind:

   '::=', bedeutet soviel wie 'kann substituiert werden durch',
   · '|', Alternative (oder), zur Substitution des NTS gibt es mehrere Möglichkeiten,
   · 'ϕ', Das NTS kann auch durch die leere Zeichenfolge abgeleitet werden.

   Auf der linken Seite einer Regel steht immer ein NTS. Terminalsymbole werden in Hochkommata

eingeschlossen. Den Abschluss einer Regel bildet ein Punkt. Die Grammatik in BNF lautet also:

ausdruck	::=	'-' hilf1 | hilf1.
hilfl	::=	term | term addoperand hilf1.
term	::=	faktor | faktor muloperand term.
faktor	::=	bezeichner | konstante | '(' ausdruck ')'.
addoperand	::=	'+' | '-'.
muloperand	::=	'*'
konstante	::=	ziffer hilf2.
hilf2	::=	ziffer hilf2 | ϕ
bezeichner	::=	buchstabe hif3.
hilD	::=	buchstabe hilf3 | ziffer hilf3 | . ϕ
buchstabe	::=	'A' | 'B' | ... | 'Z' | 'a' | 'b' |···| ż'.
ziffer	::=	'0' | '1' | ... | '9'.

4. Erweiterte Backus_N_a_ur_Form (EBNF)

Die Backus Naur Form hat den entscheidenden Nachteil, dass man gezwungen ist, rekursive Definitionen zur Darstellung von Wiederholungen und Optionalität zu verwenden. In [Wir77] wird deshalb eine erweiterte Backus Naur Form vorgeschlagen, welche die BNF um Metasymbole zur Anzeige von Wiederholungen und Optionalität erweitert:

'{' Zeichenfolge '}', die Zeichenfolge kann wiederholt werden oder nicht auftreten. Der Ausdruck X :: = XY | ϕ in EBNF ist also gleichbedeutend mit X ::= XY | ϕ in BNF

'[' Zeichenfolge ']', die Zeichenfolge ist optional und kann einmal oder nicht auftreten. Der Ausdruck X ::= [Y] in EBNF ist also gleichbedeutend mit X ::= Y | . ϕ in BNF.

Die Grammatik in EBNF lautet:

ausdruck	::=	['-'] term {addoperand term}
term	::=	faktor {muloperand faktor}
faktor	::=	bezeichner | konstante | '(' ausdruck ')'
addoperand	::=	'+' | '-'
muloperand	::=	'*'
konstante	::=	ziffer {ziffer}
bezeichner	::=	buchstabe {buchstabe | ziffer}
buchstabe	::=	'A' | 'B' | … | 'Z' | 'a' | 'b' | …·| 'z'
ziffer	::=	'0' | '1' | … | '9'

Im Weiteren werden in dieser Arbeit für die Darstellung von Grammatiken die erweiterte Backus Naur Form oder Syntaxdiagramme verwendet. Falls sich eine Grammatik durch eine einzige EBNF Regel ohne Rekursion ausdrücken lässt, so ist die Grammatik regulär. Der Ausdruck auf der rechten Seite der Regel heißt dann *regulärer Ausdruck*. Die obige

Grammatik ist aufgrund der Zentralrekursion in der Regel faktor ::= '(' ausdruck ')' nicht regulär.

4.4 Eigenschaften der Quellsprache

Grundlage für die Konstruktion eines Compilers ist die vollständige Definition der Quellsprache des Compilers. Falls ein Compiler für eine noch nicht existierende Quellsprache - zum Beispiel für eine spezialisierte Anwendersprache - erzeugt werden soll, so ist zunächst die Syntax dieser Sprache genau zu definieren. Die Syntax einer Sprache sollte den Übersetzer definieren und nicht umgekehrt. Falls die zu übersetzende Sprache bereits existiert, so liegt meist eine komplette Syntaxbeschreibung vor.

In dieser Arbeit wird als Quellsprache eine Weiterentwicklung von Pascal - Object Pascal - -verwendet. Die Sprache Object-Pascal wurde aufgrund ihrer starken Verwandtschaft zu Standard-Pascal und Turbo-Pascal, sowie ihrer neuen mächtigen Möglichkeiten wie

- strukturierte Ausnahmebehandlung,
- Klassen und Metaklassen,
- Runtime Type Information,
- Unit Konzept und
- Objektorientierung in 2 Varianten (Objektmodell von Borland-Pascal oder Delphi)

gewählt. Objekt-Pascal ist eine relativ junge Sprache. Sie wurde Anfang der 90er Jahre von der Firma Borland aus den existierenden Turbo- und Borland-Pascal Versionen entwickelt. Obwohl Object-Pascal vollständig abwartskompatibel zu Turbo- bzw. Borland-Pascal und eingeschränkt auch zu Standard-Pascal ist, wurden von Borland einige mächtige Konzepte eingeührt, welche Object-Pascal zu einer modernen objektorientierten Programmiersprache machen.

4.4.1 Grundsymbole der Quellsprache

Der Eingabetext eines Compilers besteht aus einer Folge von Zeichen eines Zeichensatzes. In dieser Arbeit wird durchgängig der ASCÜ Zeichensatz verwendet. Dieser Zeichensatz besteht aus einer Menge von 256 möglichen Zeichen. Teilzeichenreihen werden meist zu Grundsymbolen (Token) zusammengefasst. Bestimmte Zeichenreihen haben in vielen Programmiersprachen eine feste Bedeutung und können nicht umdefiniert werden. Sie werden Schlüsselwörter der Sprache genannt. Schlüsselwörter einer Programmiersprache werden im folgenden **fett** und in **GROSSBUCHSTABEN** dargestellt. In Object Pascal sind zum Beispiel "**BEGIN**" und „**END**" Schlüsselwörter.

Grammatiken für die Erkennung von Schlüsselwörtem sind meist reguläre (Typ-3) Grammatiken. Somit kann dieser Teil der Analyse durch einen endlichen Automaten beschrieben werden. Ein solcher endlicher Automat wird zum Beispiel durch den Scannergenerator LEX erzeugt. Reale Programmiersprachen, welche im allgemeinen nicht regulär sind, werden zweistufig definiert [Wir95]. Die Syntax der Sprache (I. Stufe) beruht auf abstral.1en Terminalsymbolen[6]. Dieser Teil der Analyse wird im folgenden *Syntaxanalyse* genannt. Programme zur Syntaxanalyse heißen *Parser*. Die zweite Stufe setzt die abstrakten Terminalsymbole aus konkreten Terminalsymbolen aus dem Alphabet der Sprache zusammen. Dieser Teil wird als *lexikalische Analyse* bezeichnet. Programme zur lexikalischen Analyse werden *Scanner* genannt. Tabelle 1 verdeutlicht diesen Sachverhalt.

Prozeß	Eingabe	Komponente	Kategorie
lexikalische Analyse	Zeichen	Scanner	regulär (Typ 3)

6 Andere Bezeichnungen hierfür sind Token und Morpheme

syntaktische Analyse	abstrakte Terninalsymbole	Parser	kontextfrei (Typ 2)

Tabelle 1 :Die zweistufige Definition von Programmiersprachen

Der Begriff der konkreten und abstrakten Terminalsymbole wurde durch Niklaus Wirth (z.B. in [Wir95] S. 14) geprägt. Konkrete Terminalsymbole sind alle in der Grammatik zulässigen Termninalsymbole (z.B. Buchstaben oder Ziffern). Abstrakte Terminalsymbole (Token) werden durch den Scanner aus konkreten Terminalsymbolen zusammengesetzt. Abstrakte Terminalsymbole sind:

1. In der Sprachdefinition zulässige Buchstaben des Zeichensatzes (z.B. a, b, A, B)
2. Operatorsymbole (z.B. +, -, /, *)
3. Spezialsymbole (z.B. :=, <»
4. Trennsymbole (z.B. ; , .)
5. Die Ziffern 0 .. 9
6. Schlüsselwörter der Sprache (z.B. '**IF**', '**BEGIN**')

Die Schreibweise von Schlüsselwörtern, Bezeichnern und Spezialsymbolen einer Sprache wird durch die Definition der Sprache festgelegt. In Object-Pascal wird nicht zwischen Groß- und Kleinschreibung unterschieden, aus diesem Grund sind die Schreibweisen „Begin" und „**BEGIN**", aber auch „BeGin" für das Schlüsselwort „BEGIN" lexikalisch (von der Schreibweise her) und semantisch (von der Wirkung her) identisch. Des weiteren wird die Abgrenzung der Grundsymbole untereinander durch Leerzeichen, Tabulatoren oder Zeilenwechsel festgelegt. Diese Definitionen bestimmen die Arbeitsweise der lexikalischen Analyse.

4.4.2 Syntaktische Struktur der Quellsprache

Es ist sinnvoll, möglichst alle Eigenschaften der Quellsprache vor dem Entwurf der lexikalischen und syntaktischen Analyse des Compilers exakt zu definieren. Ein Compiler muss, ausgehend von dieser formalen Definition, einen Eingabetext verarbeiten. Dassei sind spezielle Eigenschaften oder Mehrdeutigkeiten der Sprache zu berücksichtigen. Eine vollständige Definition der Syntax von Object-Pascal findet sich im Anhang. Diese Definition wurde anhand der Originaldokumentation von Borland [Bor96] und weiterer Dokumente erstellt und beschreibt den formalen Aufbau eines Object-Pascal Programms. Zur Darstellung der Syntax wurde die oben beschriebene Metasprache EBNF gewählt, da Grammatikbeschreibungen in dieser Darstellungsart besonders kompakt und leicht lesbar sind.

Die beschriebene Object-Pascal Grammatik ist nicht regulär, da zum Beispiel die Zentralrekursionen in der Definition des NTS "ausdruck" oder die Rekursionen bei verschachtelten Blocken („BEGIN" ... "END") nicht beseitigt werden können. Somit kann die Grammatik nicht zu einer linkslinearen bzw. rechtslinearen Grammatik transformiert werden. Durch die stark rekursive Struktur von Hochsprachen wie C oder Pascal kann es nicht überraschen, dass sich die meisten Programmiersprachen nicht durch reguläre Grammatiken darstellen lassen. Für den Compilerbau stellt dies jedoch kein wesentliches Problem dar, zudem kann durch die Zweiteilung der Definition der Syntax (siehe Abschnitt 2.4.1) diese Problematik etwas entschärft werden.

4.4.3 Mehrdeutigkeiten der Quellsprache

Objekt-PASCAL ist, wie viele Hochsprachen, eine mehrdeutige Sprache. Ein bekanntes Beispiel ist das Problem des "dangling else":

anwelsung ::= if-anw
| …
if-anw ::= '**IF**' ausdruck '**THEN**' anweisung ['**ELSE**' anweisung].

Für den Ausdruck "**IF** a=1 **THEN IF** b=l **THEN** c:=l **ELSE** c:=2" existieren zwei mögliche Ableitungen wie Abbildung 3 zeigt.

Bild auf Seite 20 einfügen

Abbildung 3: Das Problem des "dangling" (hängenden) else

Alle Hochsprachen, welche dieses Konstrukt enthalten, unterstützen standardmäßig die erste Variante, d.h. ordnen das "ELSE" dem letzten noch freien ,,IF" zu.

Eine Lösung dieses Problems besteht in der Umdefinition der Grammatik mit Hilfe neuer NTS wie in [Aho90] beschrieben:

anwelsung ::= matched_anw
| unmatched_anw.
if-anw ::= '**IF**' ausdruck '**THEN**' matched_anw ['**ELSE**' matched_anw].
|…
unmatched_anw ::= '**IF**' ausdruck '**THEN**' anweisung
| '**IF**' ausdruck '**THEN**' matched_anw '**ELSE**' unmatched_anw.

In obiger Grammatik muß eine Anweisung zwischen einem "**THEN**" und einem ,,**ELSE**" 'geschlossen' sein (matched_anw), das bedeutet, sie darf nicht mit einem freien "**THEN**", gefolgt von einer Anweisung enden. Geschlossene Anweisungen sind alle ,,**IF-THEN**" Anweisungen, welche nur geschlossene Anweisungen enthalten oder alle anderen Anweisungen außer ,,**IF-THEN**".

4.4.4 Kontextabhängigkeiten der Quellsprache[7]

Reale Programmiersprachen sind im eigentlichen Sinne nicht kontextfrei. Die formale Definition der Syntax definiert gewissermaßen nur eine Obermenge aller möglichen korrekten Programme. Programmiersprachen sind aber in der Regel durch zusätzliche Regeln (Semantik) gekennzeichnet. Diese Regeln bestimmen, ob ein formal syntaktisch korrektes Programm auch sinnvoll ist. Zum Beispiel hat in Pascal jeder Ausdruck einen bestimmten Typ (*Typisierung*), viele Operationen sind nur für Ausdrücke gleichen Typs zulässig. Dieser Analyseteil wird als *semantische Analyse* bezeichnet und ist oft in die syntaktische Analyse eingebettet. Kontextabhängigkeiten von Grammatiken können durch attributierte Grammatiken dargestellt werden. Dassei wird einzelnen Produktionsregeln eine Attributregel hinzugefugt, welche den Kontext der Regel definiert.

7 Es ist der Unterschied zwischen diesem Kontextbegriff und den Begriff einer kontextfreien Sprache zu beachten

4.5 Eigenschaften der Zielsprache - am Beispiel der Maschinensprache

Die genaue Kenntnis der Eigenschaften der Zielmaschine bzw. der Zielsprache sind die Voraussetzung für die Erzeugung eines unter der Zielmaschine abarbeitungsfähigen Zielprogramms. Diese Definitionen bestimmen im wesentlichen den Aufbau des Codeerzeugungsteils des Compilers .

- Registersatz
  Wie viele Register stehen auf der Zielmaschine zur Verfugung?
  Sind bestimmte Register für spezielle Anwendungsfälle reserviert?
  Sind bestimmte Operationen nur mit bestimmten Registern möglich ?
  Welche Möglichkeiten ergeben sich aus der Anzahl der verfügbaren Register für die Codeoptimierung (Verwendung von Registervariablen)?

- Instruktionsformat
  0-Adreß Maschinen: beide Operanden und das Ergebnis werden in einem Keller gespeichert.
  l-Adreß Maschinen: ein Operand und das Ergebnis stehen in einem speziellen Register (Akkumulator).
  2-Adreß Maschinen: ein Operand nimmt nach der Operation das Ergebnis auf
  3-Adreß Maschinen: Operanden und Ergebnis werden unabhängig voneinander gespeichert.

- Adressierungsarten:
  Welche Adressierungsarten existieren für die Zielmaschine?
  Können eventuell mehrere Operationen für einen Speicherzugriff zusammengefasst werden (z.B. Indexberechnung für Felder)?

- · Speicherstruktur
  Ist der Speicher sequentiell organisiert?
  Maximaler Umfang des verfügbaren Hauptspeichers?
  Existieren Segmente oder Einschränkungen beim Zugriff auf den Hauptspeicher?
  Realisierung des Laufzeitstacks auf der Zielmaschine?
  Auswirkungen der Ausrichtung (Alignment) von Datenobjekten?

- Parallelverarbeitung
  Unterstützt die zugrunde liegende Architektur das parallele Abarbeiten von Codeteilen (Pipelining, instruction caching)?
  Kann durch günstige Anordnung der Befehle (instruction scheduling) ein Performancegewinn erreicht werden? - Für RISC Rechner ist die Nutzung dieser Parallelverarbeitung die entscheidende Grundlage!

5 Kapitel 3
Struktur und Bausteine eines Compilers - grundsätzliche Varianten

Ein Compiler ist ein Werkzeug, das ein in einer *Quellsprache* formuliertes *Quellprogramm* in ein in einer Zielsprache formuliertes *Zielprogramm* übersetzt. Quellsprache und *Zielsprache* können in einer für den Menschen lesbaren Form oder in einer maschinenlesbaren Binärform abgelegt sein. Meist übersetzt ein Compiler ein Quellprogramm, welches in einer Programmiersprache formuliert ist, in Maschinencode

eines Prozessors. Es existieren jedoch auch Compiler, die einen zu interpretierenden[8] Code (P-Code) erzeugen, und Transpiler, welche die Obersetzung von einer Programmiersprache in eine andere bewältigen. Die schematische Arbeitsweise eines Compilers ist in Abbildung 4 dargestellt.

Quellprogramm - Compiler Zielprogramm

Abbildung 4: Schematische Arbeitsweise eines Compilers

Ein Compiler muss alle gültigen Sprachkonstrukte der Quellsprache akzeptieren und den Nutzer über Fehler im Quellprogramm informieren. Formal gesehen, sollte ein Compiler natürlich möglichst effizienten Code erzeugen. Wichtiger ist jedoch noch, dass korrekter Code erzeugt wird. Voraussetzung dafür ist, dass der Compiler selbst und das Quellprogramm korrekt sind. Eine Applikation ist immer nur so verlässlich, wie der Compiler, welcher die Applikation kompiliert hat. Deshalb muss ein Compiler alle legalen Sprachkonstrukte der Quellsprache in gleichbedeutende Sätze der Zielsprache umsetzen. Dafür benötigt der Compilerbauer fundierte Kenntnisse der Definition und Syntax von Quell- und Zielsprache.

In dieser Diplomarbeit wird die Übersetzung einer höheren Programmiersprache als Quellsprache in eine Maschinensprache als Zielsprache untersucht. Die Sprachdefinition der höheren Programmiersprache legt hierbei fest, welche Quellsprachenprogramme korrekt sind und spezifiziert zusammen mit den semantischen Eigenschaften der Quellsprache, welche Ergebnisse die Ausführung des Programms in der Maschinensprache des Prozessors mit bestimmten Eingabedaten liefert. Die Eigenschaften der zu erzeugenden Maschinensprache legen die konkrete Formulierung des im Quellprogramm gegebenen Algorithmus in der Zielsprache fest. Die Darstellungen in der Quellsprache und in der Zielsprache eines Compilers beschreiben also ein und denselben Algorithmus. Abbildung 5 enthält die Darstellung eines solchen Compilers.

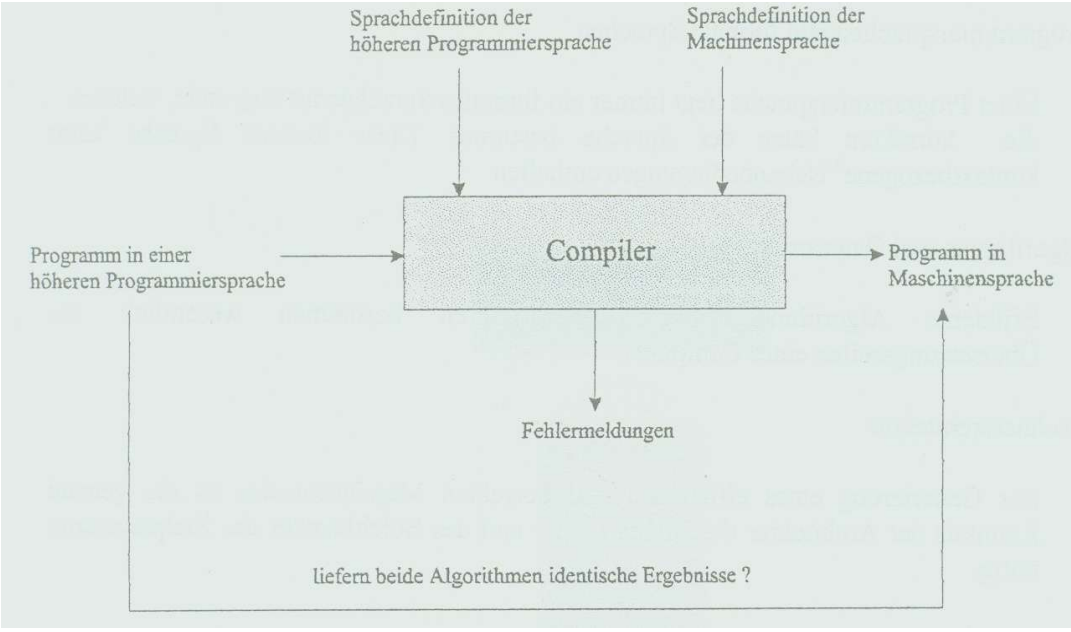


Abbildung 5: Die Übersetzung einer höheren Programmiersprache in Maschinensprache

8 Unter einem interpretierten Code soll im folgenden nicht die Interpretation des Maschinencodes durch eine CPU verstanden werden, sondern die Interpretation einer Interpretersprache (wie etwa BASIC) oder eines P-Codes (wie etwa Java) durch einen Software-Interpreter im Sinne einer abstrakten Maschine.

Sprachdefinitionen bestehen im Allgemeinen aus Regeln, welche die Menge der korrekten Quellprogramme eindeutig festlegen. Ein Compiler darf nur korrekte Quellprogramme akzeptieren und muss alle fehlerbehafteten Programme abweisen. Außerdem muss für jedes korrekte Quellprogramm ein ebenso korrektes Zielprogramm mit einem äquivalenten Algorithmus erzeugt werden. Treten während der Übersetzung Fehler im Quellprogramm auf, so soll der Compiler diese melden. Ist das Programm fehlerfrei in der Syntax der Quellsprache formuliert, so ist es *übersetzbar*. Ist es darüber hinaus frei von logischen und Programmierfehlern, so ist es *ausführbar* und liefert die gewünschten Ergebnisse. In der Regel kann ein Compiler zur *Übersetzungszeit* nur prüfen, ob ein Programm Übersetzbar ist. Der logische und semantische Test des Programms erfolgt meist zur *Laufzeit* mit Hilfe von speziellen Werkzeugen (Debugger). Darüber hinaus kann ein Compiler, in begrenztem Umfang, mögliche logische Fehler aufspüren und dem Benutzer melden. Beispiele hierfür sind die zahlreichen Warnungsmeldungen gängiger C-Compiler. Modeme Compiler implementieren meist zusätzlich eine *Fehlerstabilisierung*. welche es erlaubt, die Analyse fehlerhafter Programme fortzusetzen und so eventuell weitere Fehler zu entdecken. Die Erzeugung von korrekten Zielprogrammen aus fehlerhaften Quellprogrammen ist jedoch schwer oder nicht möglich.

Aus den obigen Ausführungen geht hervor, dass die Qualität und Übersetzungsgeschwindigkeit eines Compilers im wesentlichen durch die Verwendung effektiver, korrekter Algorithmen bestimmt wird. Zum Bau eines Übersetzers sind eine Reihe von Teilaufgaben zu lösen., welche durch geeignete Schnittstellen nach den Regeln des Software-Engineerings zur Modularisierung zu einer Gesamtlosung kombiniert werden. Dassei kommt den Strukturen und den Schnittstellen der einzelnen Compilerkomponenten ein hoher Stellenwert zu. Compilerbau erfordert (unter anderem) umfassende Kenntnisse der folgenden Informatikdiszilinen:

- Programmiersprachen und formale Sprachen
  Einer Programmiersprache liegt immer ein formales Sprachgerüst zugrunde, welches die korrekten Sätze der Sprache bestimmt. Diese formale Sprache kann kontextbezogene[9] Nebenbedingungen enthalten.

- Algorithmen und Datenstrukturen
  Effiziente Algorithmen und Datenstrukturen bestimmen wesentlich die Übersetzungszeiten eines Compilers.

- Rechnerarchitektur
  Zur Generierung eines effizienten und korrekten Maschinencodes ist die genaue Kenntnis der Architektur der Zielmaschine und des Befehlssatzes des Zielprozessors nötig.

- Software Engineering
  Ein Compiler sollte streng nach den Regeln der Modularisierung und des Information-Hiding konzipiert sein. Die enorme Komplexität eines Compilers sollte in viele sinnvolle Teilaufgaben zerlegt werden. Der Test der Korrektheit der Teilprogramme kann dann getrennt erfolgen.

    - Betriebssysteme
      Zur Erstellung einer Laufzeitbibliothek und systemnaher Routinen sind genaue Kenntnisse des

9 9 Unter kontextbezogenen Nebenbedingungen sollen hier Regeln zu verstehen sein, welche in der Grammatik nicht definiert sind (z.B. Sichtbarkeitsregeln und Typisierung). Der Begriff des Kontexts ist hier in Bezug auf den Begriff der kontextfreien Sprachen etwas zweideutig. Da in der Literatur zwischen diesen beiden Kontextbegriffen kein Unterschied gemacht wird, wird der Kontextbegriff hier für beide Arten gleichermassen benutzt.

Zielbetriebssystems und der zugrunde liegenden Architektur unbedingt erforderlich.

Diese Prinzipien sind für jeden Informatiker relevant. In ihrer Gesamtheit kombiniert, ergeben sie einen flexiblen, wartungssicheren und fehlerfreien Compiler.

Die Implementierung eines Compilers ist im wesentlichen eine Software-Engineering Aufgabe. Für eine solch komplexe Applikation, wie es ein Compiler ist, ist es nicht nur sinnvoll, sondern auch unbedingt notwendig, den Prozeß der Übersetzung einer Quellsprache in eine Zielsprache in mehrere Teilaufgaben zu zerlegen. Dies ergibt sich aus folgenden Anforderungen, welche ein guter Compiler erfüllen sollte:

- Ein Compiler sollte testbar sein, das bedeutet, die Korrektheit der verwendeten Algorithmen sollte nachgewiesen werden können. Dies kann durch einen speziellen Satz von Testprogrammen erfolgen. Wird der Compiler in mehrere Teilprogramme zerlegt, so kann jedes einzelne Teilprogramm durch simulierte Eingaben getrennt getestet werden.
  Eine Änderung eines Teilprogramms soll andere Teilprogramme nicht beeinflussen, dafür sind geeignete Schnittstellen zwischen den Teilprogrammen zu entwerfen. Dies erleichtert auch die Fehlersuche im Compiler und beeinflusst im wesentlichen die Wartbarkeit des Compilers.

- Ein Compiler sollte effizient sein. Dies bedingt die Verwendung von spezialisierten schnellen Algorithmen und eine sorgfaltige Planung der Compilerstruktur. Einzelne Teilalgorithmen sollten durch eventuell bessere Algorithmen ersetzt werden können, ohne die restlichen Komponenten zu beeinflussen.

- Ein Compiler sollte portierbar sein. Durch die Aufspaltung von Teilaufgaben eines Compilers in separate Module wird eine Widerverwendbarkeit dieser Module erreicht. Nur so kann ein Compiler mit vertretbarem Aufwand, zum Beispiel durch Abänderung des Codegeneratormoduls, auf eine andere Zie1p1attform portiert werden.

Darüber hinaus erleichtert die Unterteilung in Teilaufgaben die Strukturierung des Compilers und erhöht die Zuverlässigkeit durch bessere Übersichtlichkeit der einzelnen Komponenten. In diesem Kapitel werden grundsätzliche Möglichkeiten zur Strukturierung eines Compilers untersucht. Dafür werden zunächst die Bausteine eines Compilers klassifiziert und in das Gesamtkonzept der Übersetzung eingeordnet. Anhand dieser Bausteine wird untersucht, wie sie im Rahmen eines Compilers sinnvoll kombiniert werden können. Dassei wird auch auf verschiedene Realisierungsvarianten der einzelnen Compilerkomponenten eingegangen und auf spezifische Vor- und Nachteile hingewiesen.

5.1 Bausteine eines Compilers

Ausgehend von den Aufgaben eines Compilers lässt sich der Prozeß der Übersetzung von der Quellsprache zur Zielsprache in mehrere Teilaufgaben zerlegen. Diese Teilaufgaben beschreiben hierbei logische Schritte bei der Transformation des Quellprogramms zum Zielprogramm.

1. Lexikalische Analyse (Scanner)
   Lesen der Eingabe und Zusammenfassen von Zeichenfolgen zu abstrakten Terminalsymbolen (z.B. Schlüsselwörte, Spezialsymbole). Überlesen von redundanten Quelltextinformationen wie Kommentare und Leerzeichen.

2. Syntaktische Analyse (Parser)
   Überprüfen der syntaktischen Korrektheit des vom Scanner gelieferten Symbolstroms. Aufbau eines

Parserbaumes als interne Repäsentation des Quellprogramms. Reagieren auf Syntaxverstöße durch geeignete Fehlerbehandlung, eventuelle Attributierung des Parserbaumes.

3. Semantische Ana1yse
   Überprüfen des vom Parser gelieferten Parserbaumes auf semantische Korrektheit, wie Typkompatibilität und Sichtbarkeitsregeln (Scope rules).

4. Zwischencodegenerierung (Zwischencodegenerator)
   Erzeugung eines Zwischencodes aus dem Parserbaum. Optimierung des Zwischencodes. Vorbereitungen zur Generierung des Maschinencodes.

5. Codeerzeugung (Codegenerator)
   Erzeugung des Zielcodes aus dem Zwischencode durch geeignete Codeauswahl. Eventuell Nachoptimierung des Maschinencodes. Optional kann ein Linken von eventuell eingebundenen externen Modulen erfolgen (ansonsten ist ein separater Linkerlauf erforderlich um eine ausführbare Datei zu erhalten, wenn das Linken nicht dynamisch zur Laufzeit erfolgt). Ausgabe der Objektdatei bzw. der ausführbaren Datei.

Diese *aufgabenorientierte Compilerstruktur* ist in Abbildung 6 schematisch dargestellt.

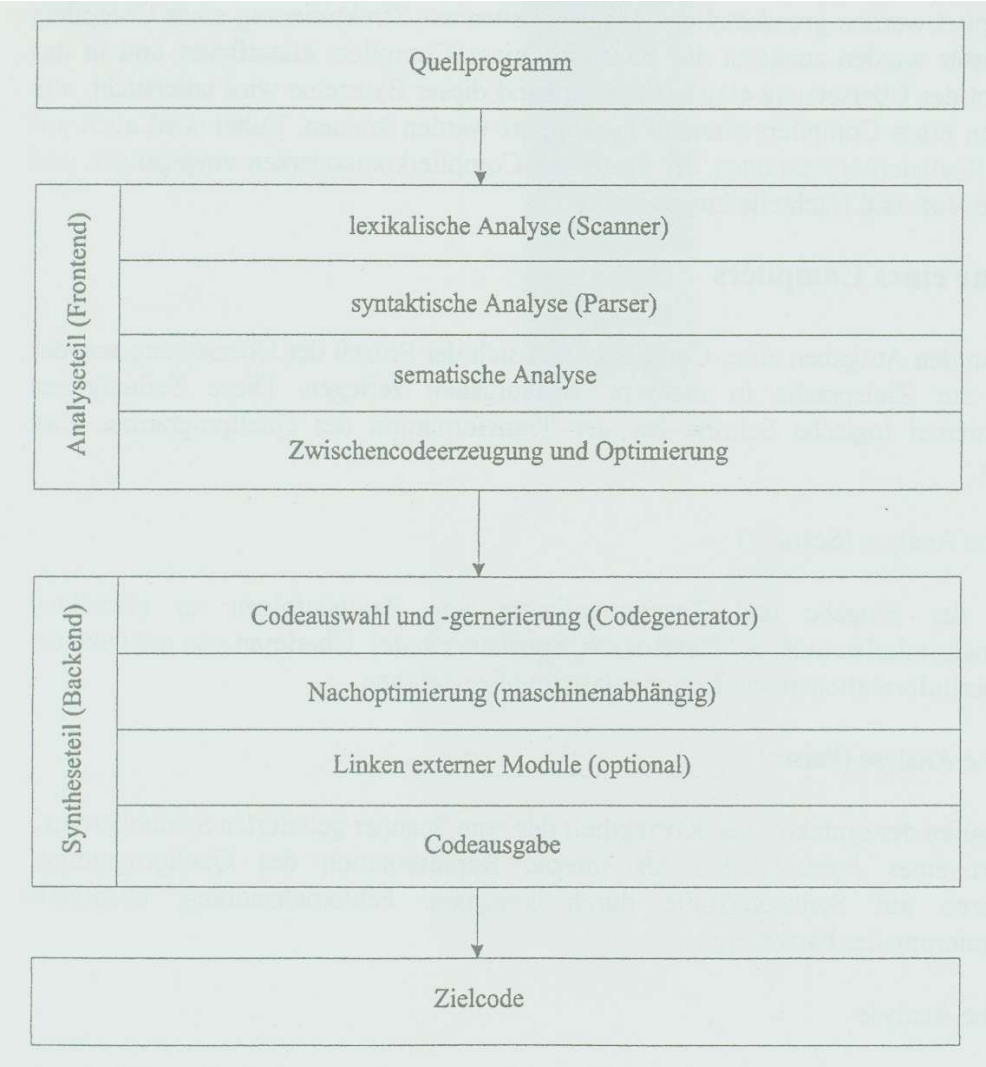


Abbildung 6: Die Frontend-Backend Compilerstruktur

Die lexikalische, syntaktische und semantische Analyse sowie die Zwischencodeerzeugung werden zusammengefasst als *Analyseteil* des Compilers bezeichnet. Die Codeerzeugung bildet den *Syntheseteil*. Der Analyseteil wird oft auch als *Frontend*, der Syntheseteil als *Backend* des Compilers bezeichnet. Schnittstelle zwischen Frontend und Backend bildet der Zwischencode, welcher in einer geeigneten Zwischensprache formuliert ist.

Generell sollte versucht werden, das Frontend des Compilers möglichst maschinenunabhängig zu halten. Bei der Portierung des Compilers muss dann lediglich das Backend angepasst werden. Im Fall des hier zu implementierenden Compilers muss lediglich die Codeausgabe an die verschiedenen Zielbetriebssysteme adaptiert werden, da die zugrunde liegende Prozessorarchitektur (INTEL) bei den zu unterstützenden Betriebssystemen identisch ist.

5.2 Grundsätzliche Realisierungsmöglichkeiten

Ausgehend von der Frontend-Backend Struktur gibt es verschiedene Möglichkeiten, einen Compiler zu strukturieren. So können einzelne Komponenten entfallen oder neue Komponenten eingebunden werden. Einfache Compiler verzichten sogar ganz auf diese Struktur und erzeugen den Zielcode in einem einzigen Quelltextdurchlauf Naturgemäß haben diese Compiler nur einen begrenzten Spielraum zur Codeoptimierung. Im Gegensatz dazu existieren hochoptimierende Compiler mit entsprechend exzellenter Codequalität zum Preis einer extrem hohen Komplexität und eventueller Einbußen bei der Compiliergeschwindigkeit. Auf die verschiedenen Varianten wird in den folgenden Unterabschnitten eingegangen.

Generell gelten folgende Regeln:

- Ein Weglassen des Scanners ist nur für Quellsprachen mit sehr begrenztem Umfang bzw. geringer Komplexität möglich. Der Parser übernimmt in diesem Fall die Rolle des Scanners mit.

- Ein Weglassen des Parsers ist nicht sinnvoll, da dieser erst einen Compiler ausmacht.

- Ein Weglassen des Zwischencodes und direktes Erzeugen des Zielcodes bedingt oft eine unzureichende Codequalität, da Optimierungen unmöglich oder schwer zu realisieren sind. Außerdem wird der Parser insgesamt komplexer und schwieriger wartbar. Eine Portierung ist dann schlecht möglich.

- Statt direkten Zielcode in Binarform zu erzeugen, kann zum Beispiel ein Assemblerquelltext erzeugt werden, welcher daraufhin von einem Assembler in den Zielcode der Maschine übersetzt wird. Dies bedeutet jedoch einen gewissen Geschwindigkeitsverlust, da nach jeder Compilierung ein Assembler- und Linkerlauf erforderlich ist. Auf der anderen Seite kann ein solcher Compiler natürlich die auf dem Zielsystem verfügbaren Werkzeuge (Assembler und Linker) nutzen. In einigen gcc Implementierungen wird dieser Weg beschritten. In der Literatur wird besonders in [Wir95] von einer solchen Compilerstruktur aus Geschwindigkeitsgründen abgeraten.

Die Wahl, welche Variante für einen zu implementierenden Compiler die günstigste ist, hangt vom Einsatzzweck des Compiler, der Mächtigkeit der Quellgrammatik und den zu realisierenden Optimierungen ab. So bieten einfache Compiler oft eine hohe Compiliergeschwindigkeit, der erzeugte Code ist jedoch meist nur befriedigend oder mangelhaft. Außerdem sollte die Wartbarkeit und Zuverlässigkeit des Compilers mit in Betracht gezogen werden. Generell nehmen natürlich auch die Anforderungen des Nutzers an einen Compiler zu. So waren zum Beispiel die ersten Turbo-Pascal

Compiler von der Dateigröße her bedeutend kleiner[10] als die heutigen Versionen, boten aber weniger Komfort.

5.2.1 Mehrpasscompiler

Diese Compiler sind dadurch gekennzeichnet, dass sie den Zielcode in mehreren Durchlaufen (Pässe) erzeugen. Jeder Pass erzeugt eine Zwischendarstellung, welche vom nächsten Pass als Eingabe akzeptiert wird. Die Anzahl der Durchläufe wird im Wesentlichen durch den vorhandenen Hauptspeicher (bzw. Plattenspeicher bei externer Ablage der Zwischenergebnisse) und den Speicherbedarf für die einzelnen Durchläufe bestimmt. Die ersten Compiler hatten aufgrund der geringen Leistungsfähigkeit der damaligen Computersysteme eine sehr hohe Anzahl von Durchläufen. So hatten laut (Wir95] die PL/I Compiler in den 60er und 70er Jahren bis zu 70 Pässe. Die Zwischendarstellungen der einzelnen Pässe wurden auf externen Speichern abgelegt, was eine extrem geringe Compiliergeschwindigkeit dieser Systeme impliziert. Aus diesem Grund sind heutzutage Compiler mit mehr als 4 Pässen selten.

5.2.2 Ein-Pass-Compiler

Durch Zusammenfassen von Aufgaben der verschiedenen Durchläufe kann eine Reduzierung der benötigten Passes bis zum Extremfall von einem einzigen Pass erreicht werden. Ein-Pass Compiler verzichten auf die explizite Erzeugung eines Zwischencodes, statt dessen wird der Zielcode direkt bei Erkennen eines Konstrukts der Quellsprache erzeugt. Meist wird nicht einmal ein Parserbaum erstellt. Die Aufrufe für die Generierung der Zielsprache finden sich dann direkt in den Prozeduren des Syntaxanalysators. In [Wirth95] wird ein solcher Compiler für die Sprache Oberon-0, einer Untermenge von Oberon, beschrieben. Der Autor räumt jedoch selbst die Nachteile dieser Compilerstruktur ein. Optimierungsmöglichkeiten sind bei einer solchen Compilerstruktur sehr begrenzt, da der Zielcode praktisch "on-the-fly" erzeugt wird. Hat ein solcher Compiler ein Konstrukt der Quellsprache als syntaktisch korrekt erkannt, so wird ohne Kenntnis des darauf folgenden Quelltextes und der daraus resultierenden Optimierungsmöglichkeiten der Zielcode erzeugt. Nachträgliche Änderungen am Zielcode (Back patching) sind nur schwer und mit unvertretbar hohem Aufwand möglich. Zudem ist ein solcher Compiler schwer zu warten und enthält unter Umständen viel redundanten Code. Eine Portierung eines solchen Compilers auf andere Zielplattformen ist nur schwer zu realisieren, da der gesamte Compiler stark maschinenabhängig wird. Einsatzgebiete für eine solche Compilerstruktur sind Compiler, welche eine maschinenunabhängige Zielsprache erzeugen, wie zum Beispiel „Pretty-Printer" (Textformatierer).

5.2.3 Zwei-Pass-Compiler

Diese Compiler implementieren die Frontend-Backend Struktur. Im ersten Pass (Frontend) wird ein (meist maschinenunabhängiger) Zwischencode erzeugt, welcher im zweiten Pass vom Backend in den Zielcode transformiert wird. Neben der besseren Übersichtlichkeit und Wartbarkeit des Compilers bringt diese Struktur weitere unschätzbare Vorteile:

- Das Frontend kann getrennt vom Backend entworfen und implementiert werden. Wird die Zwischensprache bereits beim Entwurf des Compilers festgelegt, oder eine existierende Zwischensprache verwendet, so stellt die Zwischensprache die Schnittstelle zwischen Frontend und Backend dar. Änderungen am Frontend beeinflussen dann das Backend in keiner Weise. Dadurch kann ein hohes Maß an Modularisierung und Flexibilität erreicht werden.

10 Turbo-Pascal 3.0 hatte gerade mal eine Größe von etwa 40KB und enthilt neben dem Compiler auch noch einen Editor und einen Debugger!

- Zusätzlich können unter Beibehaltung des Frontends, durch einfache Anpassung des Backends an andere Architekturen, verschiedene Plattformen unterstützt werden. Auch eine Anpassung des Frontends an eine andere Quellsprache, unter Beibehaltung des Backends, ist denkbar. Mit Hilfe einer genügend mächtigen Zwischensprache können so mit minimalem Aufwand eine Vielzahl von Programmiersprachen und grundsätzlich verschiedene Rechner- und Prozessortypen unterstützt werden. Zur Unterstützung von x Sprachen und y Prozessortypen sind dann statt x*y Compiler nur x Frontends und y Backends nötig. Die Unterstützung von verschiedenen Betriebssystemarchitekturen auf dem gleichen Rechnertyp erfolgt dann einfach durch Anpassung des vom jeweiligen Betriebssystem geforderten Programmdateiformats (Dateierweiterung .EXE unter DOS,OS/2 und Windows). Ein prominentes Beispiel für diese Vorgehensweise bei der Portierung von Compilern ist der Delphi-Compiler der Firma Borland, welcher ein Object-Pascal Frontend mit einem C-Backend verbindet. Das Resultat ist ein hochoptimierter Code unter BeibehaItung der Vorteile von Pascal, wie leichte Erlernbarkeit und Typsicherheit.

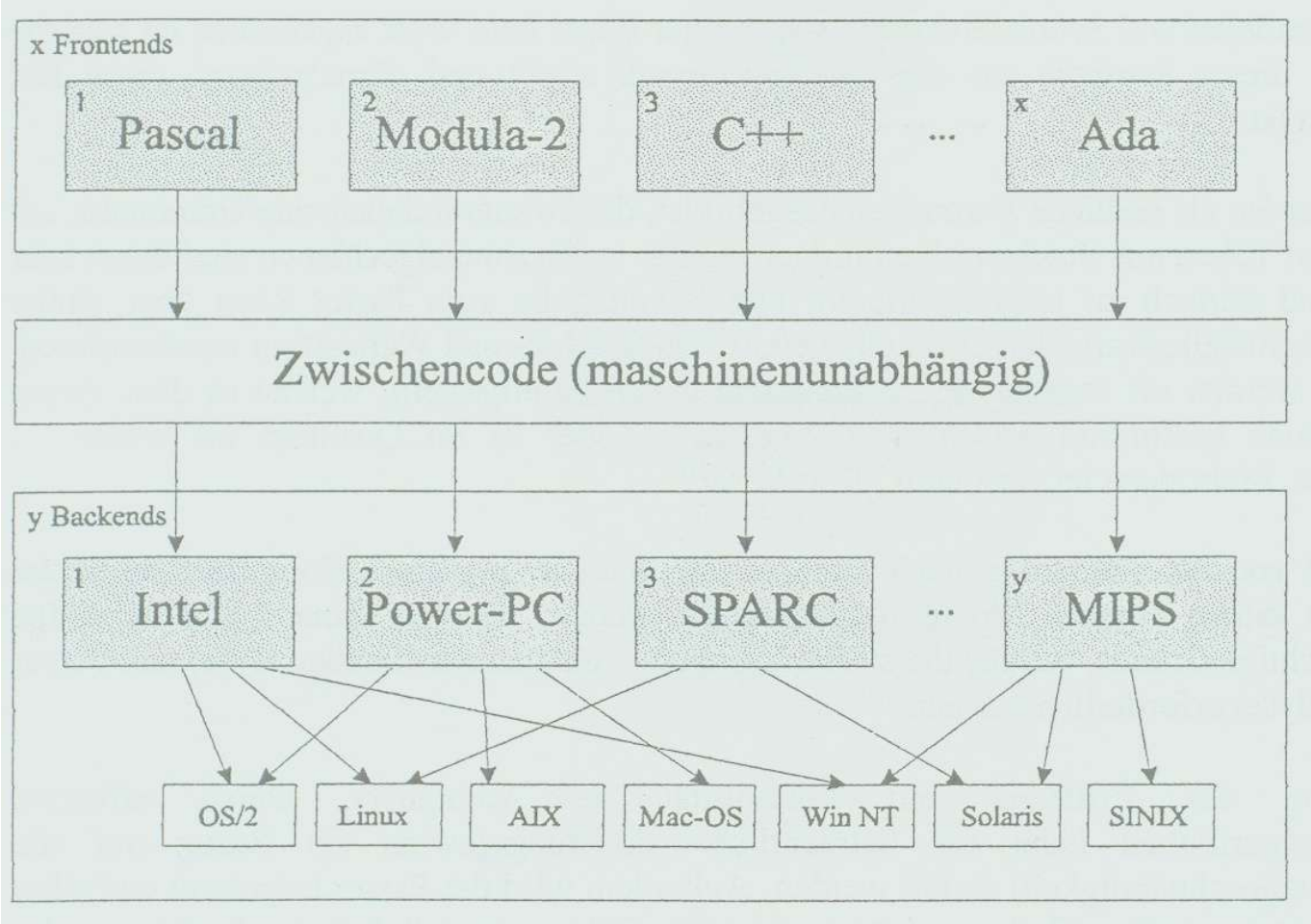


Abbildung 7: Nutzung der Frontend-Backend Struktur zur Portierung von Compilern

Aufgrund der eindeutigen Vorzüge der Frontend-Backend Compilerstruktur und der Forderung nach Portabilität in der Aufgabenstellung, wird diese Strukturvariante zur Implementierung des hier vorgestellten Compilers ausgewählt.

5.3 Der Scanner

Zentrale Aufgabe des Scanners ist es, den Parser mit den benötigten Eingabedaten zu versorgen. Außerdem übernimmt der Scanner meist andere wichtige Aufgaben, um den Parser zu entlasten, wie das Überlesen von Kommentaren und Leerzeichen. Grundsätzlich ist der Scanner der einzige Teil eines Compilers, welcher direkten Zugriff auf den Quelltext hat. Der Scanner liest die einzelnen Zeichen des Eingabestromes (meist von einer externen Quelldatei) und komprimiert diese Eingabe zu einem *Symbolstrom*. Dassei werden aus mehreren Zeichen bestehende Zeichenfolgen zu einem abstrakten Terminalsymbol (Token) umgewandelt. Zusätzlich zum Symbolstrom stellt der Scanner einen so genannten *Wertestrom* bereit, welcher nahere Auskunft über die Art des abstrakten Terminalsymbols gibt. Der Symbolstrom ist Ausgangspunkt für die darauf folgende Syntaxanalyse, der Wertestrom stellt zusätzliche Informationen über ein Symbol, wie zum Beispiel den Typ einer Variablen oder den Wert einer Konstanten, bereit. Für das Quelltextfragment

x := y + 26

könnte ein Pascal-Scanner (vereinfacht) folgende abstrakte Terminalsymbole und Werte generieren:

Symbol	Bezeichner	Zuweisung	Bezeichner	Plus	Zahl
Wert	‚x'	-	‚y'	-	26

Operatorsymbolen und Schlüsselwörtern wird in der Regel kein Wert zugeordnet, da sich die Bedeutung dieser Symbole aus der Quellgrammatik ergibt und oftmals durch diese fest vorgegeben ist.

Scanner werden als endliche Automaten ausgebildet, da die zugrunde liegende Grammatik zur Bildung von Token aus Zeichenfolgen immer regulär ist. Scanneralgorithmen sind daher sehr effektiv und einfach zu implementieren. Die Schnittstelle zum Parser kann über einige wenige Schnittstellenvariable erfolgen, welche den Symbol- und Wertestrom repräsentieren. Zusätzlich werden oft so genannte *Lookahead-Zeichen* bereitgestellt, welche es dem Parser erlauben, eine bestimmte Anzahl von Zeichen (1 oder 2) im Quelltext im voraus zu untersuchen. Mehr dazu im Abschnitt „Der Parser".

Ausgehend von der Forderung nach strenger Modularisierung eines Compilers sollte der Scanner in einem eigenen Modul implementiert werden. Darüber hinaus gibt es einige weitere wichtige Gründe, welche die strikte Trennung von Scanner (Lexikanalyse) und Parser (Syntaxanalyse) erforderlich machen:

- Erhöhung der Effizienz und Portabilität des Compilers. Durch effektive Scanneralgorithmen kann ein beträchtlicher Leistungsgewinn (in Bezug auf die Compiliergeschwindigkeit) erzielt werden. Außerdem wird der Parser insgesamt einfacher und leichter wartbar. Zudem ist es in einigen Fällen, speziell bei der Portierung des Compilers, sinnvoll, den Parser von eventuell gerätespezifischen Zeichensätzen abzugrenzen.

- Der Parser ist einfacher zu implementieren, da durch den Parser keine redundanten Informationen, wie Leerzeichen oder Kommentare verarbeitet werden müssen und der Aufbau von abstrakten Terminalsymbolen aus Einzelzeichen durch den Scanner erfolgt.

5.3.1 Implementationsmöglichkeiten für Scanner

Zur Implementation von Scannern gibt es grundsätzlich 2 Varianten:

a) Verwendung eines Scanner-Generators (z.B. LEX)

Diese Art der Scannerimplementation erlaubt eine besonders schnelle und fehlerfreie Generierung von Scannern. Ausgehend von einer Definition der Grammatik wird von einem Scannergenerator ein Scanner erstellt. Dassei sind die Schnittstellen zum Parser meist durch den Scannergenerator fest vorgegeben. Die erzeugten Scanner sind, eine korrekte Definition der Quellgrammatik vorausgesetzt, immer korrekt und sehr leicht zu warten. Darüber hinaus gibt es jedoch einige gravierende Nachteile dieser Methode. So ist der Compilerbauer beim Portieren des Scanners immer vom Vorhandensein des Scannergenerators auf der jeweiligen Zielplattform abhängig. Außerdem erlauben Scannergeneratoren oftmals keinen Zugriff auf die internen Puffer des Eingabestromes, so dass eine Optimierung durch zusätzliche Puffer schwer möglich ist [Ah090]. Die vom Scannergenerator bereitgestellten Routinen zum Zugriff auf Symbol·und Wertestrom sind fest vorgegeben und können nicht abgeändert werden. Diese Nachteile, welche oft zu gravierenden Geschwindigkeitseinbußen führen, machen Scannergeneratoren oft unattraktiv für "große" Compiler. In [Aho90] wird daher auch auf die Bedeutung der Variante b) bei der Generierung von schnellen Scannern hingewiesen.

b) Schreiben eines Scanners "von Hand"

Diese Methode ist wesentlich schwieriger zu implementieren als a) und oft auch fehleranfälliger und schwerer wartbar. Trotzdem werden auch Scanner für professionelle Compiler nach diesem Schema entworfen. Da der Scanner die Quelltextdatei Zeichen für Zeichen einliest und somit ein nicht unerheblicher Teil der Compilierzeit in Prozeduren des Scanners verbraucht wird, bestimmt die Effizienz eines Scanners im wesentlichen die Geschwindigkeit eines Compilers. Von besonderer Bedeutung ist hierbei die Pufferung der Eingabe, um Zugriffe auf langsame externe Speichermedien zu minimieren. Effiziente Algorithmen zur Implementation von Scannerpuffem sind in [Aho90] ausführlich beschrieben. Außerdem können bei einem von Hand geschriebenen Scanner, auf die Quellsprache spezialisierte Algorithmen zum Einsatz kommen. Dadurch ist diese Variante der Implementierung von Scannern oft bedeutend schneller als a).[11]

Zum Entwurf des Scanners des in dieser Arbeit implementierten Compilers wird Variante b) gewählt. Die Gründe liegen zum einen in der besseren Effizienz dieser Methode, zum anderen im Vorhandensein von bestimmten Scannerprozeduren und Techniken im existierenden Compiler, welche im vorliegenden Compiler zum Einsatz kommen sollen und somit nicht neu implementiert werden müssen. Weitere Beispiele für handgeschriebene Scanner finden sich im Compiler "lee" dessen Implementierung in [Fra95] für die Programmiersprache "C" beschrieben ist, sowie in [Job92] für den Pascal-Compiler ,, PASC".
Die Rolle eines Scanners im Compilerprozeß ist in Abbildung 8 schematisch dargestellt. Datenströme sind als durchgehende Linien eingetragen, die Darstellung einer Steuerung erfolgt durch gestrichelte Linien. Aus der Darstellung ergeben sich auch die Abhängigkeiten zwischen dem Scanner einerseits und dem Parser andererseits. Eingabe für den Scanner ist die Quelldatei, Ausgabe des Scanners der Symbol- und Wertestrom, welche der Parser zur Syntaxanalyse benutzt.

Bild auf Seite 33

Abbildung 8: Die Rolle des Scanners im Compilerprozeß

11 Laut [Aho90] kann die Effizienz noch einmal durch Verwendung der Assemblersprache als Implementationssprache des Scanners gesteigert werden, was jedoch eine weitere Erhöhung der Komplexität und Fehleranfälligkeit des Scanners bedingt. Außerdem ist diese Aussage beim heutigen Stand der Optimierungsstrategien moderner Compiler wohl etwas gewagt.

5.3.2 Fehlerbehandlung im Scanner

Für den Scanner sind nur zwei mögliche Arten von Fehlern relevant:

- Erreichen des Endes der Quelldatei bevor der Parser seine Arbeit erledigt hat (zum Beispiel eine fehlende schließende Kommentarklammer)
- Erkennen eines Zeilenwechsels innerhalb einer Stringkonstante

In beiden Fällen wird der Analyseprozeß in den meisten Compilern in der Regel abgebrochen, da eine weitere sinnvolle Syntaxanalyse unmöglich wird. Der Scanner gibt eine aussagekräftige Fehlermeldung aus und beendet den Compilerlauf

5.4 Symboltabellenvenvaltung

Scanner und Parser benutzen beide die so genannten *Symboltabelle*. Die Symboltabelle speichert alle Daten, welche im Verlauf der Compilierung für den Compiler interessant sind. Im einzelnen sind dies:

- Namen von deklarierten Variablen und deren zugehörige Typen,
- Namen von deklarierten oder vordefinierten Typen,
- Namen von vereinbarten Konstanten zusammen mit deren Werten und Typen.
- Namen von vereinbarten Prozeduren und Funktionen und ihre formalen Parameter,
- Gültigkeitsbereiche von Bezeichnern.

In der Symboltabelle können darüber hinaus weitere Daten verwaltet werden. Jedesmal wenn der Compiler im Quelltext auf einen Namen trifft wird in der Symboltabelle der entsprechende Eintrag gesucht. Existiert kein solcher Eintrag, wird der Name (bei Deklarationen) in die Symboltabelle eingetragen. Die Informationen, welche zu den einzelnen Namen gespeichert werden, können variieren. So sind bei Eintragen für Typen andere Informationen relevant als zum Beispiel bei einem Eintrag für eine Variable.

Betrachtet wird folgende Pascal Deklaration:

```
TYPE Recordl=RECORD
                a,b : Integer;
          END;

VAR r:Record1;
```

Bei der Syntaxanalyse von "**TYPE**" entscheidet der Parser, den Namen ,,Record1" als neuen Typbezeichner in die Symboltabelle einzutragen. Existiert schon ein Eintrag mit diesem Namen, so wird ein Fehler ausgegeben, da der Name doppelt belegt wäre, und bei späteren Zugriffen nicht mehr entscheidbar ist, welcher Bezeichner zu benutzen ist. Die Komponenten "a" und "b" des Records werden

zur späteren Identifikation ebenfalls in die Symboltabelle eingetragen. Beim Parsen der Variablendeklaration wird der Name ,,r" in die Symboltabelle eingetragen. Der Compiler sucht nach dem Namen ,,Recordl" und findet ihn in der Symboltabelle als Typbezeichner wieder. Somit ist die Deklaration korrekt und der Eintrag für die Variable "r" bekommt den Typ ,,Recordl". Beim späteren Zugriff auf die Variable dient der Typbezeichner zur Festlegung der gültigen Operationen mit der Variablen. So könnte über die Qualifikation "r.a:=I" auf die Komponente "a" des Records zugegriffen werden.

Die Symboltabelle speichert also *Kontextinformationen*[12] über Datenobjekte. Obwohl die meisten Programmiersprachen auf einer kontextfreien Grammatik beruhen, sind viele Hochsprachen doch stark kontextabhängig. Dies ergibt sich aus dem in den meisten prozeduralen Programmiersprachen vorhandenen *Typkonzept* und den *Sichtbarkeitsregeln* (Scope-rules). Jedem Bezeichner ist in der Regel ein Typ zugeordnet. Außerdem kann der Bezeichner auch nur innerhalb eines bestimmten Quellcodeabschnittes sichtbar sein, zum Beispiel eine lokale Variable innerhalb des Anweisungsteiles einer Prozedur (Sichtbarkeitsbereich von Bezeichnern). Das strenge Typkonzept von Pascal verbietet bestimmte Operationen mit inkompatiblen Typen. So sind Programme, welche im formalen Sinne der kontextfreien Grammatik genügen, aufgrund der Verletzung von Kontextregeln doch keine korrekten Programme der Quellsprache. Ein weiteres Beispiel wäre die Verwendung eines Bezeichners vor seiner Deklaration, was in Pascal strikt verboten ist.

Für die Implementation einer Symboltabelle gibt es verschiedene Möglichkeiten. Eine einfache Lösung wäre eine baumartige Darstellung, welche alle Bezeichner aufnimmt. Deklarierte Bezeichner (hier ,, Record1" und ,,r") werden durch eine einfach verkettete Liste aufgenommen. Durch Querverweise auf andere Definitionen entsteht eine baumartige Struktur. Das Suchen eines Bezeichners erfolgt durch sequentielles Durchlaufen der Liste. Für das obige Beispiel könnte folgende Datenstruktur entstehen:

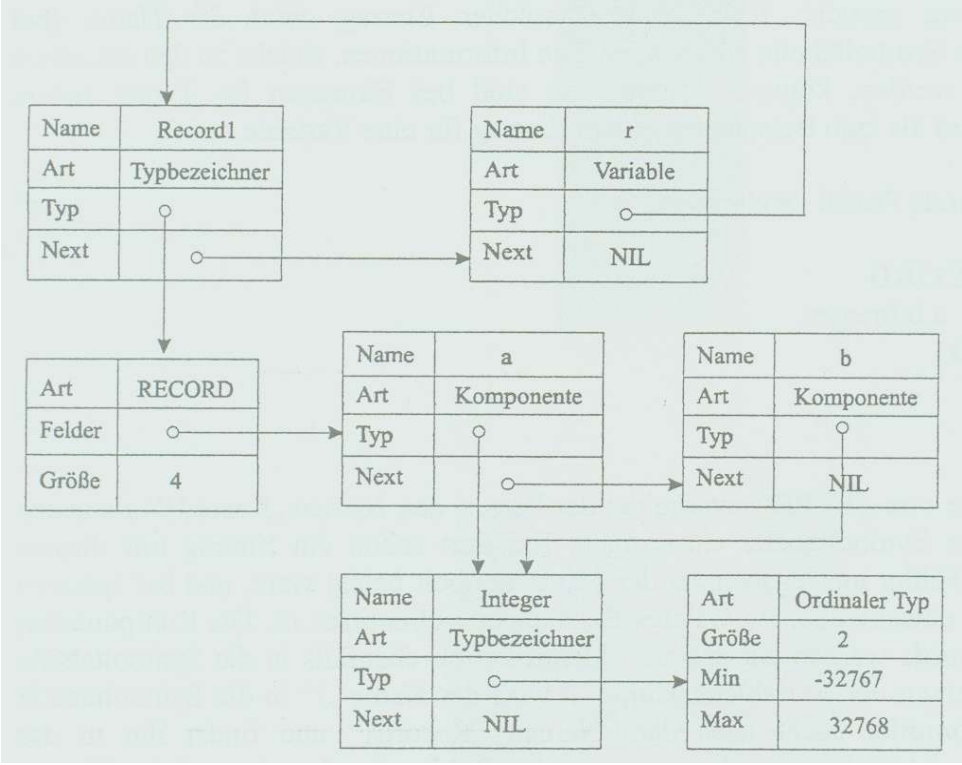


Abbildung 9: Eine mögliche Symboltabellendarstellung des Beispiels

12 Es besteht wiederum ein Unterschied zwischen dem Kontextbegriff innerhalb einer Grammatik und den Kontextinformationen (zum Beispiel Typinformationen) von Datenobjekten in der Symboltabelle

Schon aus diesem einfachen Beispiel ist ersichtlich, dass die Symboltabellenverwaltung überaus komplexe Aufgaben zu bewältigen hat. Die Komplexität der Darstellung von Informationen wachst mit den Möglichkeiten der Quellsprache. So wird für eine Sprache mit einigen wenigen fest vordefinierten Typen eine andere Darstellung zu bevorzugen sein als für eine Sprache, welche auch die Definition von neuen Datentypen zulässt. Generell ist die Darstellung von Bezeichnern als verkettete Liste nur sinnvoll, wenn abzusehen ist, dass die Symboltabelle nur wenige Einträge aufnehmen soll oder wenn die Suchzeiten für den Compiler nur eine untergeordnete Rolle spielen. Meist werden (unter Umständen auch mehrere) *Hashverfahren* eingesetzt, um den Prozess des Suchens in der Symboltabelle zu beschleunigen. Eine andere Möglichkeit wäre der Einsatz von sortierten Suchbäumen. Bei der Implementation der Symboltabelle für den vorliegenden Compilerentwurf wird auf die relevanten Verfahren und Datenstrukturen noch näher eingegangen.

In gewisser Weise nimmt die Symboltabelle auch schon Bezug auf die gewählte Zielarchitektur. So wird zum Beispiel auch die Größe von Datentypen hier festgehalten und eine sequentielle Anordnung der Speicherzellen (Für Records und Felder) angenommen. Da in Pascal ein Integer mit 2 Byte definiert ist (Borland Pascal definiert den Standardtyp ,,LongInt" mit 4 Byte), stellt dies kein wesentliches Problem dar. In anderen Sprachen, wie etwa "C", ist die Größe des Datentypes "Integer" von der Zielplattform abhängig (2 Byte für 16 Bit Maschinen und 4 Byte für 32 Bit Maschinen). Zudem soll im vorliegenden Compiler die Standardgröße von Integer durch eine Compilerdirektive gesteuert werden können, um eine Portierung von Programmen aus der 16-Bit Welt zu vereinfachen.

5.5 Der Parser

Im Analyseprozess eines Compilers prüft der Parser, ob die Folge von Symbolen, welche vom Scanner geliefert wird, durch die Quellgrammatik erzeugt werden kann. In den meisten Compilern steuert der Parser darüber hinaus den gesamten Übersetzungsvorgang, das heißt, er fordert bei Bedarf neue Symbole vom Scanner an. An geeigneter Stelle ruft der Parser Routinen des Zwischencodegenerators zur Erzeugung des Zwischencodes auf Außerdem wird in vielen Parsern auch die semantische Analyse direkt in die Prozeduren zur Analyse der verschiedenen syntaktischen Kategorien eingebunden.
Für die Implementation eines Parsers existieren eine Reihe von (meist tabellengesteuerter) Verfahren, von denen die wichtigsten in Abbildung 10 dargestellt sind. Einige dieser Verfahren werden in den folgenden Abschnitten kurz vorgestellt.

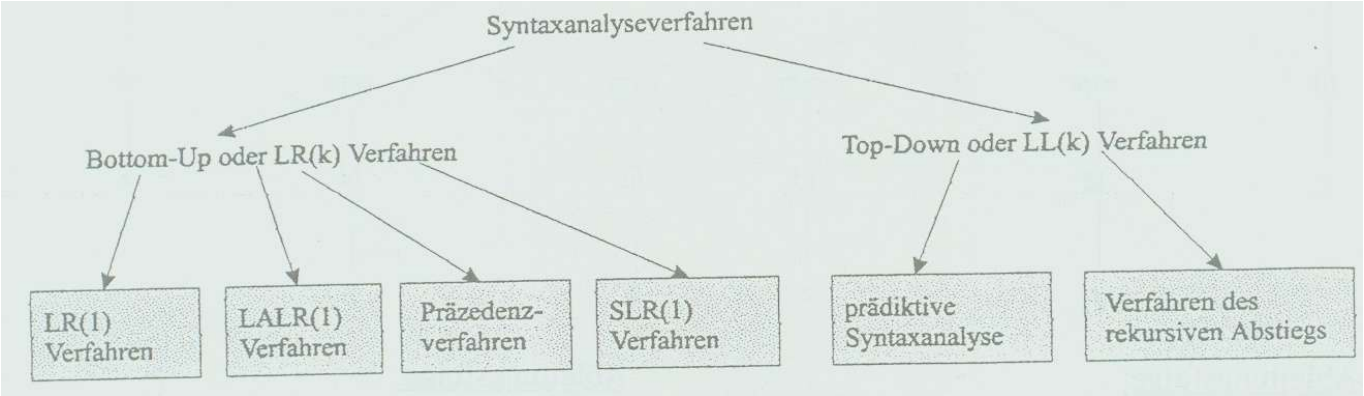


Abbildung 10: Die gebräuchlichsten Syntaxanalyseverfahren

LR(k) und LL(k) sind Syntaxanalyseverfahren, welche mit k Symbolen *Lookahead* auskommen. Ein Lookahead Symbol ist ein Symbol des Symbolstroms, welches der Parser ausgehend vom aktuellen

Symbol "im Voraus" betrachten kann, um die weiteren Analyseschritte entscheiden zu können. Lookahead Symbole sind sinnvollerweise meist auf 1 oder 2 beschränkt. Die gebräuchlichsten Parser sind somit LR(l) und LL(l) Parser.

- LR(k): Lesen der Eingabe von Links nach rechts mit rechtsseitiger Ableitungsfolge der Eingabe und Vorgriff auf k Symbole .

- LL(k): Lesen der Eingabe von Links nach rechts mit linksseitiger Ableitungsfolge der Eingabe und Vorgriff auf k Symbole.

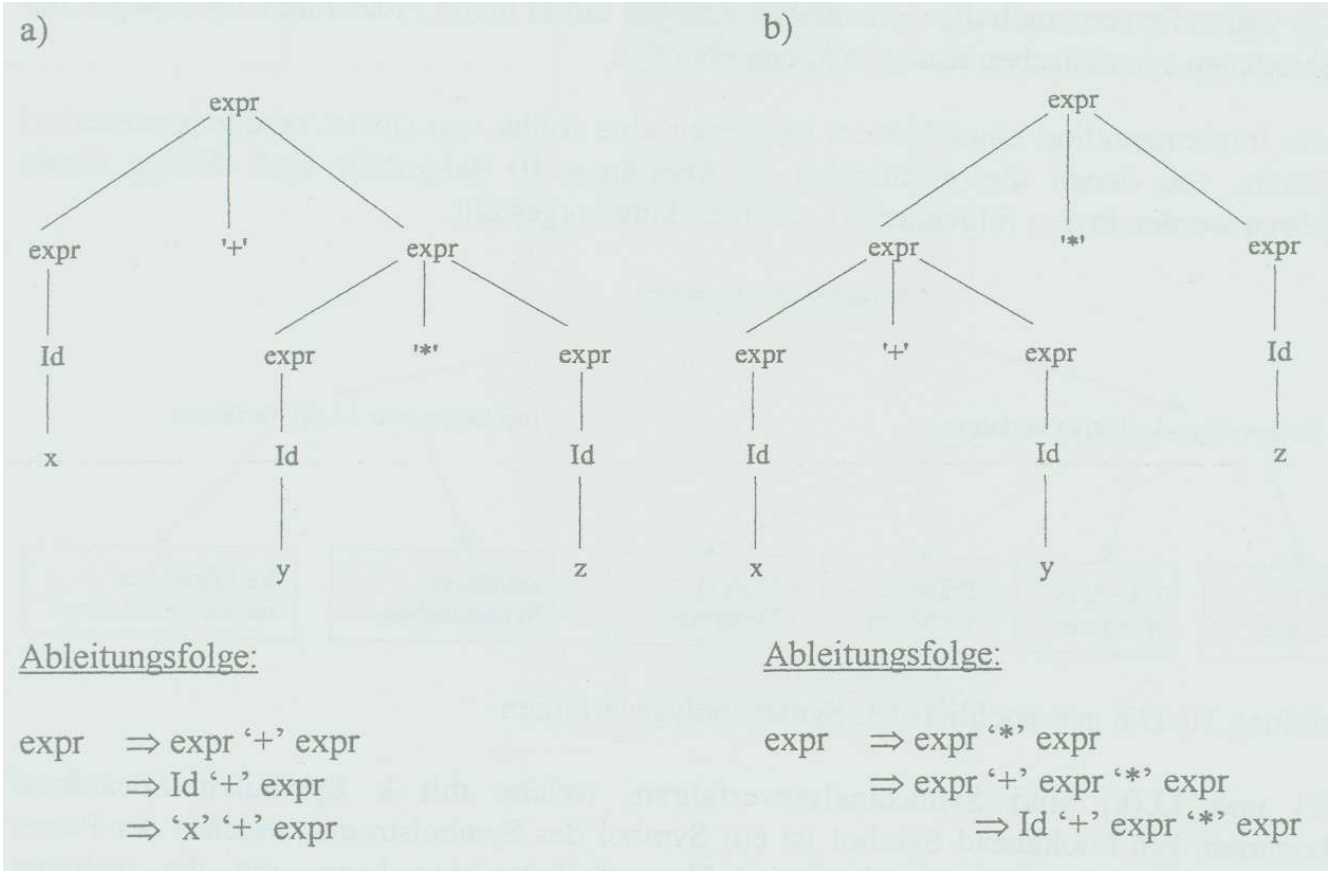
Der Parser sollte (zumindest prinzipiell) einen so genannten *Parserbaum* erstellen - bei rekursiv-descent Parsern (Verfahren des rekursiven Abstiegs) ergibt sich der Parserbaum implizit aus den Aufrufen von rekursiven Prozeduren. Der Parserbaum repräsentiert die interne Darstellung des Eingabestromes, enthalt jedoch keinerlei redundante Informationen mehr. Zur Verdeutlichung dieses Sachverhaltes wird folgender kleiner Ausschnitt aus einer Grammatik betrachtet:

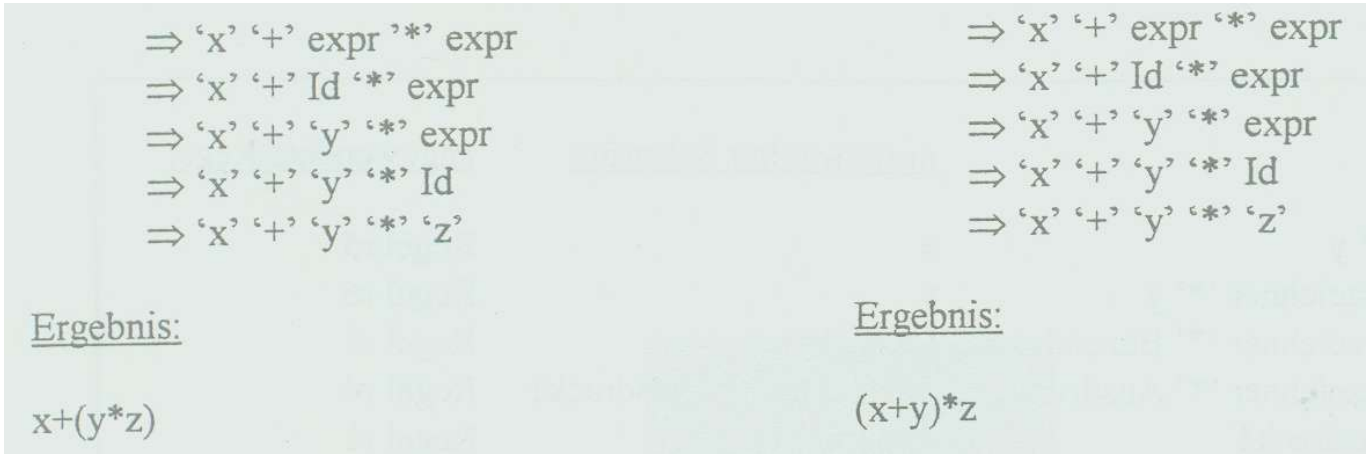
```
expr    ::=     expr '+' expr
            |   expr '*' expr
            |   Id.
```

Das NTS „Id" soll hierbei in Pascal erlaubte Bezeichner verkörpern. Das Startsymbol der Grammatik soll "expr" sein. Für den Quelltextausschnitt

x+y*z

existieren nun zwei, in der folgenden Abbildung als a) und b) dargestellte, mögliche Parserbäume, da es zwei mögliche Linksableitungen für den Ausdruck gibt:





Die in der Mathematik und in nahezu allen Programmiersprachen vorgegebene Priorität von Operatoren würde den Parserbaum a) bevorzugen. Es liegt in der Verantwortung des Parsers, diesem Sachverhalt Rechnung zu tragen, wenn nicht bereits in der Quellgrammatik diese Mehrdeutigkeiten ausgeschaltet werden können. So ließe sich durch Modifikation der Grammatik zu

```
expr    ::=     term { '+' term }
term    ::=     Id { '*' Id }
```

das Problem beseitigen, das heißt, es gäbe nur noch eine mögliche Linksableitung für das NTS "expr" und zwar die korrekte Variante x+(y*z).

5.5.1 Bottom-Up Syntaxanalyseverfahren (LR(k) Verfahren)

Kennzeichnend für diese Art der Syntaxanalyseverfahren ist, dass versucht wird, aus dem aktuellen Eingabestring einen Parserbaum zu konstruieren. Es wird also der Eingabestring auf das Startsymbol der Grammatik reduziert, indem in verschiedenen Schritten am weitesten links stehende Substrings in dem Eingabestring gesucht werden, welche der rechten Seite einer Regel entsprechen. Der Substring wird dann durch die linke Seite dieser Regel ersetzt. Effektiv wird damit eine Rechtsableitungen des Satzes in umgekehrter Reihenfolge (Postfixnotation) erzeugt. Ergibt sich nach wiederholtem Ausführen dieses Algorithmus das Startsymbol der Grammatik, so ist der Eingabestring ein korrekter Satz der Sprache. Betrachtet wird hierzu folgende Beispielgrammatik:

G = (V, ∑, R, s) mit

```
V={Ausdruck, Ausdruckl, Bezeichner, '+' , '*' , 'x', 'y'}
∑={'+', '*', 'x', 'y'}
s=Ausdruck
R={
        rl =(Ausdruck, AusdruckI),
        r2=(Ausdruck,Ausdruckl '+' Ausdruck),
        r3=(Ausdruck I ,Bezeichner),
        r4=(AusdruckI,Bezeichner ,*, Ausdruckl),
        r5=(Bezeichner, 'x'),
        r6=(Bezeichner,'y')
}
```

Der Satz 'x*y' wird nun im Bottom-Up Verfahren wie folgt reduziert:

	Eingabe	ausgewählter Substring	angewendete Regel
	x * y	x	Regel r5
⇒	Bezeichner '*' y	y	Regel r6
⇒	Bezeichner '*' Bezeichner	Bezeichner	Regel r3
⇒	Bezeichner '*' Ausdruck1	Bezeichner '*' Ausdruck1	Regel r4
⇒	Ausdruck1	Ausdruck1	Regel r1
⇒	Ausdruck		

dies entspricht der Rechtsableitung:

Ausdruck --> Ausdruckl
--> Bezeichner ,*, Ausdruckl
--> Bezeichner ,*, Bezeichner
--> Bezeichner ,*, y
--> x '*' y

Da sich also aus x '*' y das Startsymbol der Grammatik reduzieren lässt, ist der Satz syntaktisch korrekt. Offensichtlich ist jedoch die Auswahl der Regeln problematisch. So könnte der Eingabestring ,, Bezeichner * y" auch zu ,,Ausdruckl * y" reduziert werden, da der Substring ,,Bezeichner" auch auf der rechten Seite der Regel r3 vorkommt. Dies ergäbe folgende (nicht korrekte) Reduktion des Satzes:

	Eingabe	ausgewählter Substring	angewendete Regel
	x * y	x	Regel r5
⇒	Bezeichner '*' y	Bezeichner	Regel r3
⇒	Ausdruck1 '*' y	Ausdruck1	Regel r1
⇒	Ausdruck '*' y	y	Regel r6
⇒	Ausdruck '*' Bezeichner	Bezeichner	Regel r3
⇒	Ausdruck '*' Ausdruck1	Ausdruck1	Regel r1
⇒	Ausdruck '*' Ausdruck		

Es ist also ohne zusätzliche Nebenbedingungen nicht möglich, einen Satz eindeutig zu reduzieren, bzw. immer die richtige Regel zur Reduktion eines Substrings zu finden. Zur Lösung dieses Problems schaut der Bottom-Up Syntaxanalysator um ein (oder mehrere) Symbol(e) voraus (Lookahead), um die korrekte Regel zu identifizieren. Vor Ersetzung eines Substrings S durch eine linke Seite L einer Regel testet der Parser das auf S folgende Symbol. Existiert eine Ableitung, in der auf L dieses Symbol folgen kann, so ist L die korrekte Substitution für S. Ansonsten wird L verworfen und eine andere Regel mit S auf der rechten Seite ausgewählt. Existiert keine weitere solche Regel und kein weiterer Substring auf den eine Regel zutrifft, so ist die Eingabe unzulässig. Im obigen Beispiel würde der Parser zunächst versuchen, den Substring "Bezeichner" mit Hilfe der Regel r3 zu ,,Ausdruckl" reduzieren. Da dies mit obiger Bedingung nicht möglich ist, wird die Regel r6 mit dem neuen Substring "y" ausgewählt und "y" zu ,,

Bezeichner" reduziert.

Das oben beschriebene allgemeine Verfahren der Bottom-Up Syntaxanalyse wird auch als Shift-Reduce Syntaxanalyse bezeichnet. Typischerweise sind diese Syntaxanalysatoren *Kellerautomaten*, das bedeutet, als Eingabepuffer wird ein Stack verwendet. Zu Anfang ist der Stack leer.

5.5.2 Top-Down Syntaxanalyseverfahren (LL(k) Verfahren)

Im Gegensatz zum Bottom-Up Verfahren, wird beim Top-Down Syntaxanalyseverfahren versucht, ausgehend von einem Startsymbol, den Eingabestrom als Satz der Sprache zu erkennen. Im folgenden wird hierfür die Methode des rekursiven Abstiegs (recursive descent) erläutert, welche im vorliegenden Compiler zu Einsatz kommen soll. Darüber hinaus existieren weitere (tabellengesteuerte) Verfahren. Auf diese wird im Literaturanhang verwiesen.

Betrachtet wird noch einmal die Beispielgrammatik G = (V, ∑, R, s) mit

```
V={Ausdruck,Ausdruckl ,Bezeichner, '+' ,'*', 'x' ,'y'}
∑={'+', '*', 'x', 'y'}
s=Ausdruck
R={
        rl =(Ausdruck,Ausdruck1),
        r2=(Ausdruck,Ausdruck1 '+' Ausdruck),
        r3=( Ausdruck1 ,Bezeichner),
        r4=(Ausdruck1,Bezeichner ,*, Ausdruck1),
        r5=(Bezeichner, 'x'),
        r6=(Bezeichner,'y')
}
```

Kennzeichnend für das Verfahren des rekursiven Abstiegs ist, dass für jedes NTS ein Erkennungsalgorithmus in Form einer Prozedur existiert, welche die Korrektheit der Eingabe für dieses NTS prüft. Da Hochsprachen wie "C" oder ,,Pascal" den rekursiven Aufruf von Prozeduren erlauben, ist hiermit auch die Analyse von rekursiven Grammatiken möglich. Jedes Auftreten eines NTS in der Grammatik wird in den Aufruf der zugehörigen Prozedur überfuhrt. Der Analyseprozess beginnt mit dem Aufruf der Prozedur für das Startsymbol der Grammatik. In der Beispielgrammatik existieren genau 3 NTS. Zur Vereinfachung wird angenommen, dass der Scanner die folgenden Symbole für den Symbolstrom liefert:

Sym_Plus	Aktuelles Zeichen ist ein '+',
Sym_Mul	Aktuelles Zeichen ist ein '*',
Sym_Bez	Aktuelles Zeichen ist ein 'x' oder 'y',
Sym_EOF	Das Ende der Eingabe ist erreicht.

Außerdem soll ein globales Symbol "Sym" existieren, welches den Code des aktuellen Symbols enthält. Durch den Aufruf der Scanner-Prozedur "Next" sol1 der Scanner das nächste Symbol in der Variablen "Sym" ablegen. Eine Prozedur ,,Error" soll einen Fehler anzeigen und die Analyse beenden. Daraus ergibt sich folgendes (vereinfachtes) Programm zur Analyse der gegebenen Grammatik:

```
//Analyse
#include parser.h      // Prozedur Prototypen und Definitionen

void Bezeichner()      // Prozedur für das NTS Bezeichner
{
  if (Sym != S_Bez) Error();
  Next();              // nächstes Symbol lesen
};

void Ausdruckl()       // Prozedur für das NTS Ausdruckl
{
  Bezeichner();        // Aufruf der Prozedur für das NTS Bezeichner
  if (Sym == Sym_Mul)
  {
    Next();            // nächstes Symbol holen
    Ausdruckl();       // rekursiver Aufruf für das NTS Ausdruckl
  }
};

void Ausdruck()        // Prozedur für das NTS Ausdruck
{
  Ausdruckl();         // Aufruf der Prozedur für das NTS Ausdruckl
  if (Sym == Sym_Plus)
  {
    Next();            // nächstes Symbol holen
    Ausdruck();        // rekursiver Aufruf für das NTS Ausdruck
  }
};

void Analyse()
{
  Next();              // Erstes Symbol holen
  Ausdruck();          // Aufruf der Prozedur für das Startsymbol
```

Aus dem Beispiel ist ersichtlich, dass recursive descent Parser mit relativ geringem Aufwand zu implementieren sind. Der Vorteil dieser Methode besteht darin, dass keinerlei Analysetabellen benötigt werden. Der Laufzeitstack wird als Kellerspeicher des Ableitungsweges benutzt. Die Reihenfolge der aufgerufenen Prozeduren baut damit implizit einen Parserbaum auf, das heißt, der bisherige Analyseweg ist auf dem Laufzeitstack des Compilers abgelegt.

5.5.3 Implementationsmöglichkeiten für Parser

LR(k) und LL(k) Syntaxanalysatoren beruhen meist auf tabellengesteuerten Verfahren. Dazu wird eine LL(k) oder LR(k) Syntaxanalysetabelle anhand der Grammatik erstellt. Die Erstellung derartiger Tabellen ist für kleine Grammatiken noch praktikabel, der Aufwand und die Fehleranfälligkeit wachsen jedoch mit zunehmender Komplexität der Grammatik. Zudem muss bei jeder Änderung der Grammatik eine komplett neue Analysetabelle erstellt werden. Auf die Erstellung derartiger Tabellen wird in dieser Arbeit nicht eingegangen, es wird auf die im Anhang aufgelistete Literatur verwiesen.

Zur Implementation von Parsern gibt es grundsätzlich 2 Varianten:

a) Verwendung eines Parsergenerators (z.B. YACC)
   Es existiert eine ganze Reihe so genannter Parsergeneratoren, welche das Aufstellen der LL(k) und LR(k) Analysetabellen automatisieren. Eingabe eines solchen Generators ist eine Definition der Quellgrammatik. Ausgabe ist meist ein Quelltext in einer Hochsprache. So erzeugt der Parsergenerator YACC aus einer in einer Metasprache angegebenen Syntaxdefinition einen C-Quelltext. Zusätzlich kann der Nutzer beliebige C-Anweisungen in die Definition der Grammatik einfugen, um Aktionen beim Erkennen einer syntaktischen Kategorie ausfuhren zu können.

b) Schreiben eines Parsers "von Hand"
   Diese Methode ist schwieriger zu implementieren als a), hat jedoch den Vorteil der vollständigen Kontrolle des Analyseprozesses. Diese Vorgehensweise ist in der Regel nur für Parser nach der Methode des rekursiven Abstiegs praktikabel, da die Erstellung von LL(k) und LR(k) Analysetabellen manuell nicht empfehlenswert ist.

Zum Entwurf des Parsers des in dieser Arbeit implementierten Compilers wird Variante b) gewählt. Die Gründe liegen im wesentlichen in der Erfahrung des Autors beim Erzeugen derartiger Parser und dem Vorhandensein von Routinen für recursive descent Parser in der Programmierbibliothek des Autors und im vorhandenen Compilerdesign. Die Implementierung mittels eines Parsergenerators wie YACC ist sicherlich eleganter, setzt aber zum einen das Vorhandensein von YACC auf den Zielbetriebssystemen voraus und erlaubt dem Compilerbauer zum anderen weit weniger Einflussnahme auf den intemen Ablauf der Analyse. Weitere Beispiele für handgeschriebene Parser finden sich im Compiler "lcc" dessen Implementierung in [Fra95] für die Programmiersprache "C" beschrieben ist, sowie in [Job92] für den Pascal Compiler ,,PASC".

Die Rolle eines Parsers im Compilerprozess ist in Abbildung 12 schematisch dargestellt. Aus der Darstellung ergeben sich auch die Abhängigkeiten zwischen dem Parser einerseits und dem Scanner sowie Codegenerator andererseits. Eingabe für den Parser ist der vom Scanner gelieferte Symbol- und Wertestrom, Ausgabe des Parsers der Parserbaum, aus welchem der Zwischencode erzeugt wird.

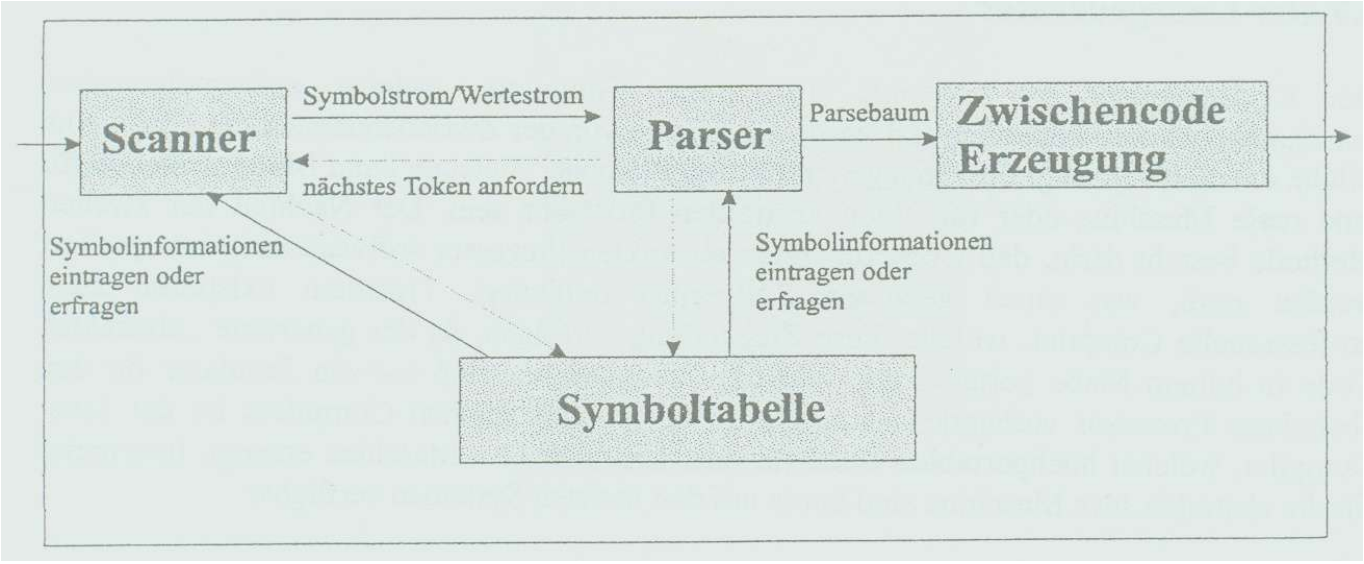


Abbildung 12: Die Rolle des Parsers im Compilerprozess

5.5.4 Fehlerbehandlung im Parser

Der Parser sollte grundsätzlich in der Lage sein, auch syntaktisch falsche Programme bis zum Ende analysieren zu können, um eventuell weitere Fehler zu finden. In der Praxis ist dieses Vorhaben jedoch ein sehr schwieriges Unterfangen. So kann ein einziger Fehler (zum Beispiel eine falsch geschriebene oder nicht deklarierte Variable) eine Fülle von Folgefehlern nach sich ziehen.

5.6 Zwischencodeerzeugung

Im Analyseprozess eines Compilers erzeugt der Zwischencodegenerator aus dem vom Parser gelieferten Parserbaum eine geeignete Zwischendarstellung. Im Vergleich zur direkten Erzeugung des Zielcodes hat eine Zwischendarstellung folgende Vorteile:

1. Die Portierung des Compilers auf eine andere Zielsprache wird erst durch einen Zwischencode mit vertretbarem Aufwand möglich. Beim Anpassen des Compilers bleibt das Frontend unverändert, nur das Backend wird modifiziert.

2. Es können maschinenunabhängige Optimierungen über dem Zwischencode durchgeführt werden.

3. Die Wartbarkeit und Übersichtlichkeit des Compilers verbessern sich.

Ein Zwischencode wird im allgemeinen durch eine exakt definierte Zwischensprache beschrieben. Zur Darstellung der Zwischensprache gibt es grundsätzlich die Möglichkeiten der graphischen Darstellung (z.B. in Form von gerichteten azyklischen Graphen) oder die Darstellung als Code für einen abstrakten Prozessor (z.B. Drei-Adress-Code). Im folgenden werden die Begriffe „Zwischencode" oder „ Zwischensprache" für beide Darstellungsformen gleichermassen verwendet. Da der Begriff des "abstrakten Prozessors" relativ umfassend ist, sind natürlich auch abstrakte Prozessoren denkbar, welche Graphen als Eingabe akzeptieren. Für eine detaillierte Einführung in die Zwischencodeerzeugung (speziell für Drei-Adress-Code) wird auf [Aho90] verwiesen.

5.7 Der Codegenerator

Der Codegenerator ist derjenige Teil eines Compilers, welcher notwendigerweise maschinenabhängig implementiert wird. Ausgehend von der Zwischendarstellung - falls eine solche existiert - erzeugt der Codegenerator den Zielcode. Zielcode kann Maschinencode für eine reale Maschine oder für einen abstrakten Prozessor sein. Der Nachteil der zweiten Methode besteht darin, dass Code für einen abstrakten Prozessor softwareseitig interpretiert werden muss, was einen gewissen Zeitverlust impliziert. Trotzdem existieren auch professionelle Compiler, welche diese Zielstellung verfolgen, da der generierte "abstrakte" Code in hohem Masse portabel ist - auf der Zielmaschine muss nur ein Emulator für den abstrakten Prozessor vorhanden sein. Ein Beispiel eines solchen Compilers ist der Java-Compiler, welcher hochportablen Code für eine abstrakte Java Maschine erzeugt. Interpreter für die abstrakte Java Maschine sind heute auf den meisten Systemen verfügbar.

Ein Kriterium für einen guten Codegenerator ist natürlich, dass er stets korrekten Code erzeugt. Außerdem soll der erzeugte Zielcode von hoher Qualität, das heißt gut optimiert sein. Laut [Aho90] ist die Erzeugung von optimalem Code jedoch ein mathematisch nicht entscheidbares Problem. Ein Codegenerator kann nur versuchen, möglichst guten Code zu erzeugen, ein Optimum wird jedoch nicht erreichbar sein.

Wie für Scanner und Parser existieren auch für Codegeneratoren entsprechende Werkzeuge, we1che aus einer abstrakten Definition einen Codegenerator erzeugen. Diese Werkzeuge stehen aber in der Regel nur unter UNIX zur Verfügung und sollen im vorliegenden Compiler nicht zum Einsatz kommen.

6 Kapitel 4
Entwurf des Frontends

In den vorangegangenen Kapiteln ist deutlich geworden, dass die Entscheidungen, welche beim Entwurf eines Compilers zu treffen sind, nicht immer unproblematisch sind. Jedes Verfahren hat seine spezifischen Vor- und Nachteile und beeinflusst in der Regel weitere Entwurfsentscheidungen. Für den zu implementierenden Compiler steht die Einfachheit der zu wühlenden Algorithmen und Datenstrukturen im Vordergrund. In vielen Fällen sind sicherlich bessere Algorithmen verfügbar, wie zum Beispiel im Bereich der Codeoptimierung. Eine detaillierte Untersuchung hatte den Rahmen dieser Diplomarbeit bei weitem gesprengt. Ein weiteres Kriterium ist die Wiederverwendbarkeit vorhandener Algorithmen und Unterprogramme. Ein vollständiger Neuentwurf wäre im Rahmen einer halbjährigen Diplomarbeit nicht möglich gewesen. Zusätzlich zu den in der Aufgabenstellung enthaltenen Forderungen sollen für den Entwurf des Compilers weitere Randbedingungen gelten:

- Der Compiler soll, wenn möglich, vorhandene Bibliotheken und Quelltexte des vorhandenen Compilers nutzen. Es waren Module zum Linken der Objektdatei und der Codeausgabe sowie Scannerroutinen und Parseralgorithmen vorhanden. Diese sollen nach Möglichkeit genutzt werden., urn eine zeitaufwendige Neuimplementation zu vermeiden. Bei der Einbindung dieser Algorithmen ist auf strenge Modularisierung zu achten.

- Der Compiler soll auf Compiliergeschwindigkeit ausgerichtet werden. Dies wird durch spezialisierte Algorithmen und einen internen Linker erreicht. Optimierungen werden auf einige wenige wirkungsvolle Verfahren beschränkt.

- Die Laufzeitbibliothek wird teilweise aus vorhandenen Quelltexten erzeugt. Eine komplette Neuimplementierung ist aus zeitlichen Gründen nicht möglich.

- Als Implementierungssprache des Compilers wird C++[13] verwendete, da für alle Zielbetriebssysteme ein C++-Compiler zur Verfügung stand. Außerdem erzeugen C/C++ Compiler im Vergleich zu anderen Compilem einen besonders gut optimierten Code.

6.1 Eigenschaften der Quellsprache

Als Quellsprache wird laut Aufgabenstellung die Sprache Object-Pascal gewählt. Object-Pascal ist eine objektorientierte Weiterentwicklung der Sprache Turbo-Pascal. Wie andere moderne Hochsprachen zeichnet sich Object-Pascal durch Sprachkonstrukte zur Modularisierung (Unit Konzept) und Konstrukte zur Bildung von abstrakten Datentypen aus. Damit steht Object-Pascal anderen Sprachen, wie MODULA 2 oder C++, nicht nach. Für den Compilerbauer ergeben sich aus der Grammatik der Quellsprache jedoch einige Komplikationen:

- Object-Pascal erlaubt im Gegensatz zu C oder C++ die Schachtelung von Prozeduren in beliebiger Tiefe. Dies stellt besondere Anforderungen an die Symboltabelle und die Anwendung von Gültigkeitsregeln.

- Object-Pascal besitzt ein strenges Typkonzept. Operationen mit inkompatiblen Typen sind in der

13 Allerdings ohne objektorientierte Sprachkonstrukte

Regel nicht zulässig. Zudem ist die uneingeschränkte Bildung neuer Typen aus vorhandenen Typen möglich. Dies bedingt ein Typsystem, welches die Einhaltung gewisser Typverträglichkeitsregeln überprüft.

- Object-Pascal ist eine objektortientierte Sprache. Zusätzlich zum "alten" Objektmodell von Turbo-Pascal wurde ein neues, wesentlich erweitertes Objektmodell eingeführt. Dieses System lehnt sich stark an C++ an (obgleich es keine Mehrfachvererbung zulässt). Für den Compilerbauer bedeutet dies zusätzlichen Aufwand, da die Probleme der Überprüfung der Gültigkeitsregeln und der Typvertraglichkeit durch objektorientierte Sprachkonstrukte weiter verschärft werden.

- Voraussetzung für den Entwurf des Compilers ist das Vorhandensein einer Definition der Quellsprache. Da eine solche Definition in kompakter Form nicht vorhanden war, wird eine EBNF Darstellung der Grammatik aus vorhandenen Dokumenten, wie zum Beispiel [Bor96J, erstellt. Diese Darstellung der Grammatik ist die Grundlage für die weiteren Entwurfsschritte und ist im Anhang enthalten. Die Semantik der einzelnen Sprachkonstrukte wird ebenfalls aus vorhandenen Dokumentationen und Büchern entnommen.

6.2 Modularisierung der Compileraufgaben

Als Ausgangspunkt für den Entwurf des Compilers ist es zunächst sinnvoll, eine grobe Aufteilung der voraussichtlich benötigten Compilerkomponenten vorzunehmen. Ausgehend von der Frontend-Backend Compilerstruktur werden die einzelnen Compileraufgaben in separaten Modulen untergebracht. Dies ermöglicht auch die spätere Anpassung des Compilers an weitere Betriebssysteme bzw. Zielmaschinen. Die für den zu realisierenden Compiler vorgesehene Modulaufteilung ist in der folgenden Abbildung dargestellt:

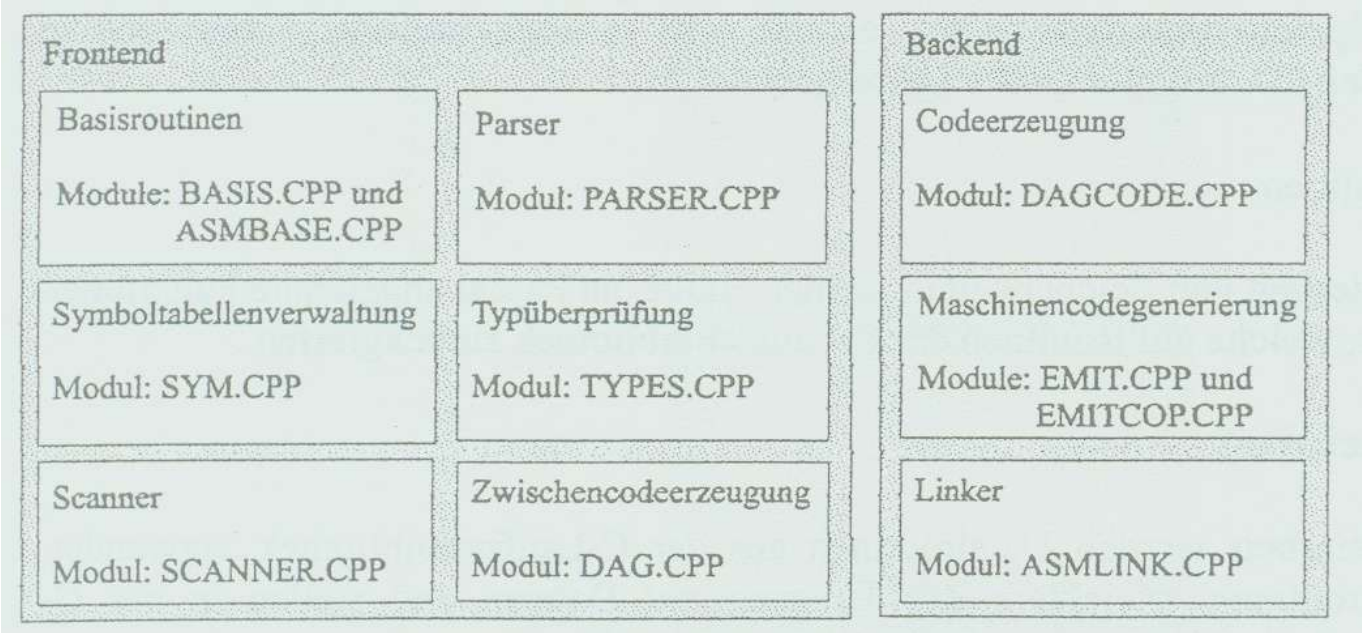


Abbildung 13: Ein Entwurf für eine mögliche Modularisierung des Compilers

Zielmaschinenabhängig sind nur die Komponenten des Backends. Das Frontend wird völlig maschinenunabhängig implementiert. Der Linker wird aus Geschwindigkeitsgründen mit in das Backend übernommen, obwohl es durchaus möglich gewesen wäre, einen externen Linker zu verwenden. Die

Gründe dafür werden im Kapitel ,,Entwurf des Backends" noch genauer betrachtet.

Weitere Überlegungen sollten sicherlich die Aufteilung der Compileraufgaben in einzelne Prozesse betreffen. Es wäre denkbar, für den Scanner, Parser und Codegenerator jeweils eigene Prozesse zu benutzen, welche parallel ablaufen und über geeignete Schnittstellen miteinander kommunizieren. Aufgrund des zusätzlichen Overheads für die Synchronisierung der einzelnen Prozesse darf jedoch die Effizienz einer solchen Maßnahme auf den zu unterstützenden Betriebssystemen (in der Regel nur 1 Prozessor vorhanden oder unterstützt) zumindest angezweifelt werden. Zudem würde dies den gesamten Compiler nicht unwesentlich komplizierter machen. Im zu implementierenden Compiler wird deshalb auf diese Möglichkeit verzichtet und statt dessen ein einzelner Prozess für den gesamten Compiler verwendet.

6.3 Entwurf der Basismodule

Die Basismodule sollen die darüber liegenden Teile des Compilers von den Unterschieden der zugrunde liegenden Zielbetriebssysteme abgrenzen, sowie gemeinsam benutzte Routinen zur Verfügung stellen:

- Speicherverwaltung
  Es wird ein globaler Applikationsheap verwaltet, welcher bei Bedarf auch komplett (das bedeutet ohne explizites Freigeben jeder einzelnen Speicheranforderung) freigegeben werden kann. Die Verwaltung dieses Heaps erfolgt über das jeweilige Betriebssystem. Die Allokation von Speicherobjekten auf diesem Heap soll über die Funktion "GetMem" erfolgen, die Freigabe über die Funktion ,,FreeMem". Die Allokation eines neuen Heaps (unter Freigabe aller Speicherobjekte auf dem bisherigen Heap) soll durch die Funktion ,,NewCompilerHeap" möglich sein. Dies ermöglicht die komplette Freigabe aller vom Compiler angeforderten Speicherressourcen ohne explizites, zeitaufwendiges Freigeben jedes einzelnen allokierten Speicherobjektes. Diese Technik wird in abgewandelter Form auch in [Fra95] für den C-Compiler ,,lee" angewendet.

- Zeichenkettenroutinen
  Zur Manipulierung von Zeichenketten werden einige, an Pascal angelehnte Funktionen bereitgestellt, welche auf Routinen der C-Laufzeitbibliothek zurückgreifen.

- Routinen zur Dateiarbeit
  Für die Dateiarbeit werden Dateiroutinen aus der C-Laufzeitbibliothek verwendet. Einige Hilfsroutinen erleichtern den Umgang mit Dateien und stellen in der C-Laufzeitbibliothek nicht vorhandene Routinen zur Verfügung.

- Fehlerausgabe
  Für die Ausgabe von Fehlermeldungen werden Konstanten für jede Fehlermeldung bereitgestellt. Außerdem soll für jeden Fehler die Angabe eines zusätzlichen Zeichenketten-Parameters möglich sein, welcher den Fehler naher beschreibt. Die Auslösung eines fatalen Fehlers soll über die Funktion ,,*Error*" erfolgen. Der Compilerlauf wird daraufhin abgebrochen und benötigte Ressourcen freigegeben. Der Aufruf der Funktion "*ParserError*" soll ebenfalls einen Fehlertext ausgeben, der Compilerlauf kann jedoch fortgesetzt werden. Durch Aufruf der Funktion "*Warning*" wird eine Warnungsmeldung ausgegeben. Compilerwarnungen sollen selektiv ein- und ausgeschaltet werden können.

6.4 Entwurf des Scanners

Aufgabe des Scanners ist es, abstrakte Terminalsymbole aus konkreten Terminalsymbolen zu erzeugen (siehe Kapitel 3). Scanner beruhen auf regulären Sprachen (Typ-3 Grammatiken). Das zugrunde liegende Prinzip ist also das Prinzip eines endlichen Automaten. Scanner erkennen Symbole in Zeichenfolgen, ausgehend von zugrunde liegenden Regeln. Die Anzahl und die Zusammensetzung der abstrakten Terminalsymbole (Token) werden durch die Quellgrammatik bestimmt. Hier spielen speziell die in der Quellsprache vorhandenen Schlüsselworte und Operatoren sowie die Bildungsregeln für Bezeichner eine Rolle.

Ausgehend von den Überlegungen in Kapitel 3, soll der Scanner "von Hand" erstellt werden, das heißt ohne Benutzung eines Scannergenerators. Zusammenfassend spielen folgende Überlegungen für diese Entscheidung eine Rolle:

- Es bestand die Möglichkeit der Nutzung von vorhandenen Routinen und Erfahrungen in der Konstruktion von Scannern. Zum großen Teil können Routinen (mit wenigen Änderungen) aus dem vorhandenen Compiler übernommen werden. Im speziellen werden effiziente, auf die Quellsprache abgestimmte, Algorithmen zum Erkennen von abstrakten Terminalsymbolen und zur Pufferung der Eingabe verwendet. Diese Techniken stammen zum Teil aus anderen Publikationen (z.B. [Aho90] und [Fra95]) und haben sich in der Praxis bewährt.

- Aus den Randbedingungen der Diplomarbeit ergibt sich die Forderung nach der Priorität der Compilergeschwindigkeit. Laut [Fra95] verbringt ein Compiler bis zu 50% der Compilierzeit in Routinen des Lexikanalysators (Scanners). Eine Beschleunigung dieser Komponente bestimmt somit in großem Maße die gesamte Geschwindigkeit der Compilierung. In [Aho90] wird die Ansicht vertreten, dass von Hand erstellte Scanner oftmals schneller sind als solche, welche von einem Scannergenerator erzeugt wurden. Diese Auffassung erscheint einleuchtend, da ein Scannergenerator prinzipiell die speziellen Eigenschaften der Quellsprache nicht ausnutzen kann. Scannergeneratoren decken eine ganze Reihe von Einsatzgebieten ab, somit kommen allgemeiner Algorithmen zum Einsatz.

- Ein Scannergenerator (wie etwa LEX) stand auf den Zielbetriebssystemen nicht zur Verfügung. prinzipiell wäre es möglich gewesen, den Scanner auf einem andere System (etwa UNIX oder DOS) generieren zu lassen und den erzeugten C-Quelltext anzupassen. Dies hätte jedoch einen nicht unerheblichen Mehraufwand mit sich gebracht.

- Ausgehend von den Forderungen nach strenger Modularisierung und Strukturierung wird der Scanner als Unterroutine des Parsers entworfen. Darüber hinaus gibt es einige weitere wichtige Punkte, welche die Trennung von Scanner und Parser sinnvoll erscheinen lassen [Ah090]:

    1. Der Entwurf des Compilers wird insgesamt vereinfacht. Durch die Verlagerung von Teilaufgaben der Analyse in den Scanner ergibt sich eine wesentliche Entlastung des Parsers. Zum Beispiel werden Leerzeichen und Kommentare bereits vom Scanner entfernt. Außerdem erfolgt die Bildung von abstrakten Terminalsymbolen bereits im Scanner. Die Routinen des Parsers werden hiermit insgesamt einfacher und leichter wartbar.

    2. Die Effizienz (Compiliergeschwindigkeit) des Compilers wird verbessert. Da der Scanner in einem eigenen Modul untergebracht ist, können die Algorithmen des Scanners effizienter und spezifischer implementiert werden. Auch eine spätere Neuimplementation des Scanners über einen Scannergenerator wird möglich.

3. Die Portierung des Compilers wird vereinfacht (oft erst dadurch möglich!). Gerätespezifische Eigenschaften und Besonderheiten des Eingabealphabetes werden auf den Scanner beschränkt. Spezielle Zeichen (wie etwa „^") sind nur für den Scanner von Bedeutung.

Die Routinen des Scanners sollen im Modul "SCANNER.CPP" untergebracht werden. Funktionsprototypen und Deklarationen sind im Modul "SCANNER.H" enthalten. Der Symbolstrom für den Parser (siehe Kapitel 3) soll über eine globale Variable „*Sym*" bereitgestellt werden, welche die Codierung des aktuellen abstrakten Terminalsymbols enthalten soll. Der Wertestrom wird über eine Variable „*yylval*" bereitgestellt. Die Anforderung eines neuen Symbols soll über die Funktion „*GetTok*" erfolgen. Scanner und Parser bilden somit ein Erzeuger-Verbraucher Paar. Der Scanner speichert immer genau ein Symbol (und einen evtl. dazugehörigen Wert). Sobald der Parser das Symbol verarbeitet hat, fordert er über „*GetTok*" ein neues Symbol beim Scanner an. Das aktuelle Zeichen der Eingabe soll in einer Variablen „*ch*" abgelegt sein. Die Bereitstellung eines Lookahead Zeichens für den Parser erfolgt über die Variable „*chN*" (nächstes Zeichen der Eingabe). Die Zeichenketterepräsentation des aktuellen Symbols soll in der Variablen „*yytext*" verfügbar sein. Da Pascal nicht zwischen Groß- und Kleinschreibung unterscheidet, erscheinen die Zeichen des Eingabestromes hier in Großbuchstaben umgewandelt. Die interne Repräsentation von Bezeichnern erfolgt ebenfalls in Großbuchstaben.

6.4.1 Die Bildung von abstrakten Terminalsymbolen im Scanner

Die Art und Anzahl der benötigten abstrakten Terminalsymbole kann systematisch aus der Grammatikbeschreibung im Anhang A abgeleitet werden. Schlüsselwörtern, Operatoren und Spezialsymbolen wird kein Wert in „*yylval*" zugewiesen, die Bedeutung dieser Symbole ergibt sich aus der Grammatikbeschreibung und der Semantik der Sprache. Die komplette Aufstellung der vom Scanner gelieferten abstrakten Terminalsysmbole findet sich im Anahang B.

Der Scanner soll für die in Tabelle 2 dargestellten Zeichenkettenarten Symbole zurückliefern. Für jede Klasse von möglichen Zeichenketten ist in der Tabelle ein Beispiel angegeben. Der Wert der verschiedenen Variablen für die Beispielzeichenkette ist jeweils In Klammem angegeben.

Beschreibung	Beispiel	Inhalt der Variablen "Sym"	Inhalt der Variablen „yylval"	Inhalt der Variablen „yytext"
Schlüsselwörter von Pascal	Begin	Identifikation des Schlüsselwortes (S_BEGIN)	Nicht benutzt	. .BEGIN"
Operatoren von Pascal	<=	Identifikation des Operators (S_LE)	Nicht benutzt	·. <="
Spezialsymbole von Pascal	[	Identifikation des Spezialsymbols (S_LBRAC)	Nicht benutzt	„[„
Konstanten	123	S_Zahl, S_FloatZahl oder S_STRING_Text (S_Zahl)	Wert der Konstanten (123)	„123"
Bezeichner	Integer	Identifikation des Bezeichners (S_TypBez)	Eintrag des Bezeichners in der Symboltabelle (Verneis auf den Basistyp „Integer“	"INTEGER"

Tabelle 2: Beispiele für die vom Scanner gelieferten Symbole und Werte

Zum Erkennen der abstrakten Terminalsymbole aus der Zeichenfolge des Eingabestromes werden in der Entwurfsphase des Scanners so genannte *Übergangsdiagramme* [Ah090] erstellt. Diese Übergangsdiagramme stellen *endliche Automaten* dar und beschreiben die Aktionen des Scanners bei der Bildung der abstrakten Terminalsymbole. Die Übergangsdiagramme müssen deterministisch sein, d.h. für jeden Zustand existiert nur maximal ein Folgezustand für jedes mögliche Zeichen der Eingabe. Da die lexikalische Struktur von Pascal relativ simpel ist, ist die Konstruktion derartiger Übergangsdiagramme von Hand noch praktikabel. Eine Automatisierung kann durch LEX erreicht werden, dieser Scannergenerator soll aber aus den oben aufgeführten Gründen nicht zum Einsatz kommen. Das zu verwendende Übergangsdiagramm ergibt sich aus dem aktuellen Zeichen der Eingabe. Da die Erstellung derartiger Diagramme trivial (wenn auch zeitaufwendig) ist, soll die Vorgehensweise hier nur an zwei Beispielen demonstriert werden. Die Implementation des Scanners kann dann systematisch aus den Übergangsdiagrammen abgeleitet werden. Diese Technik wird in ähnlicher Form in [Kas90] sowie in [Fra95] für Scanner verwendet.

Die nachfolgenden Übergangsdiagramme s

- Startzustand ist der Zustand 0. Dieser Zustand wird immer dann initialisiert, wenn der Parser die Funktion „*GetTok*“ aufruft, das heißt ein neues Symbol anfordert.

- Ausgehend von einem beliebigen Zustand liest der Scanner das nächste Zeichen der Eingabe. Dieses Zeichen bestimmt den Folgezustand. Für jeden Zustand existiert ein eindeutiger Folgezustand. Beim Lesen eines Zeichens aus dem Eingabestrom wird die aktuelle Position im Eingabestrom um eins erhöht.

- Die für den jeweiligen Zustandsübergang notwendigen Bedingungen (aktuelle(s) Eingabezeichen) werden an den Übergangskanten angegeben.

- In einigen Zustanden ist es notwendig, das aktuelle Zeichen der Eingabe zu verwerfen. Dies erfolgt durch Rücksetzen der Position im Eingabestrom um ein Zeichen. Das verworfene Zeichen wird dann vom Scanner beim erneuten Aufruf von „*GetTok*" erneut gelesen. Solche Zustände werden durch einen ,,*" gekennzeichnet [Aho90].

- Endzustande des Automaten werden durch einen Doppelkreis gekennzeichnet. An einem Endzustand liefert der Scanner das erkannte abstrakte Terminalsymbol an den Parser und übergibt die Kontrolle zurück an den Parser. Ein erneuter Aufruf von „*GetTok*" beginnt wieder im Zustand 0.

- Ein Fehlerzustand wird erreicht, wenn das gelesene Zeichen kein gültiges Zeichen des Pascal-Eingabealphabets (etwa ,,!") ist oder kein Folgezustand für den aktuellen Zustand und das aktuelle Zeichen existiert. Der Fehlerzustand kann nur von Zustand 0 erreicht werden, da alle anderen Zustände eventuell ungültige Zeichen verwerfen und das bis dahin erkannte Symbol zurückliefern. Ein Fehlerzustand wird durch ein ,,F" gekennzeichnet.

Zur Verdeutlichung der Vorgehensweise wird im folgenden ein Ausschnitt aus dem Übergangsdiagramm des Scanners zum Erkennen der abstrakten Terminalsymbole, welche sich aus dem aktuellen Zeichen ,, <“ ergeben., angegeben (siehe auch [Aho90]). Jeder Aufruf des Scanners beginnt wieder bei Zustand 0. Endzustände liefern den Wert des erkannten Symbols zurück.

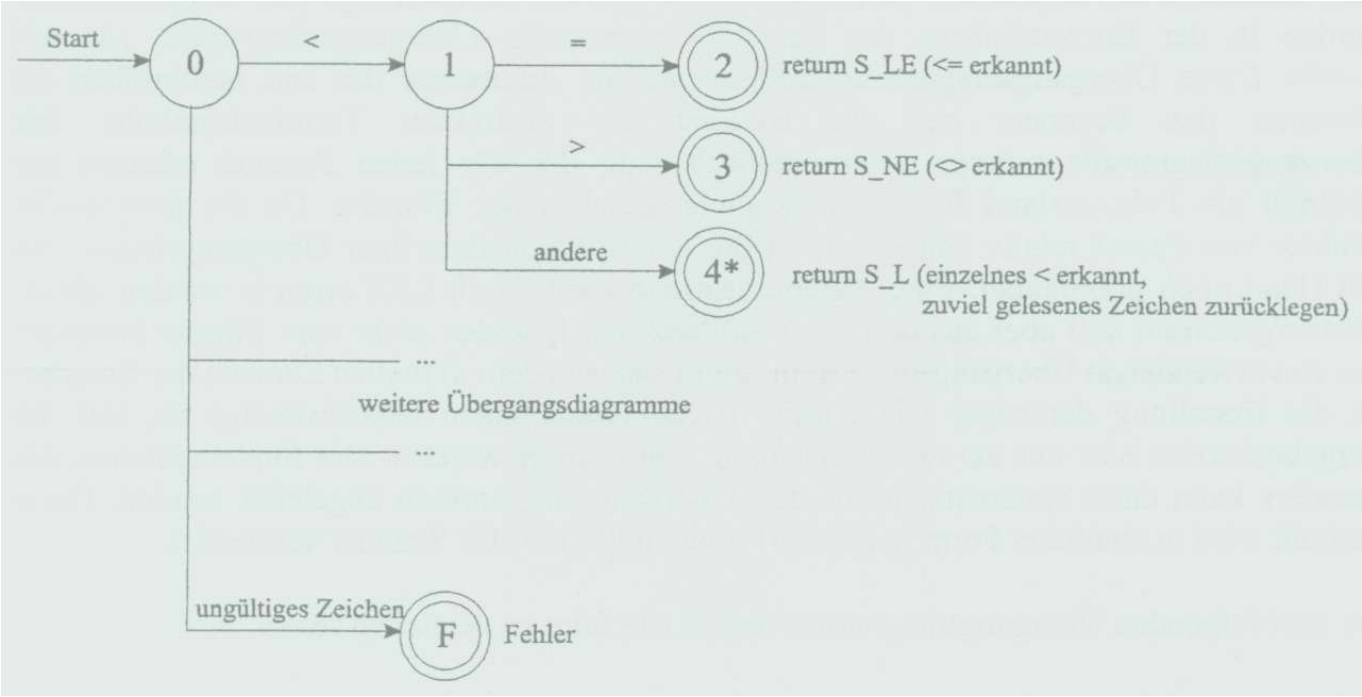


Abbildung 14: Ausschnitt aus dem Übergangsdiagramm für das Zeichen „<“

Mit "weiteren Übergangsdiagrammen" sind hier die Zustünde für die restlichen gültigen Zeichen des Eingabealphabets gemeint. Die einzelnen Übergangsdiagramme werden also zu einem Diagramm zusammengefasst. Das obige Übergangsdiagramm berücksichtigt alle in Pascal erlaubten abstrakten Terminalsymbole, welche sich ergeben können, wenn das Zeichen „<" vom Scanner gelesen wird. Der Scanner prüft zunächst das auf „<" folgende Zeichen und entscheidet daraufhin, ob dafür ein gültiger Folgezustand (obige Zustande 2 und 3) existiert. Gibt es keinen derartigen Zustand, so wird das gelesene Zeichen zurückgelegt und das bis dahin akzeptierte Symbol zurückgeliefert (obiger Zustand 4). Ob die Zurückgelieferten Symbole in ihrer Reihenfolge syntaktisch korrekt sind, wird ausschließlich vom Parser entschieden.

6.4.2 Erkennen von Schlüsselwörtern

Ein weiteres Problem ist das Erkennen von Schlüsselwörter und das Zurückliefern des entsprechenden Symbols. Die Schlüsselwörter von Object-Pascal bestehen aussehließ1ich aus Buchstaben. Zwischen Groß- und Kleinschreibung wird nicht unterschieden.

a) In [Ah090J wird vorgeschlagen, Schlüsselwörter als spezielle Bezeichner in die Symboltabelle einzutragen. Die Suche nach einem Schlüsselwort erfolgt dann über die Routinen der Symboltabellenverwaltung. Der Vorteil dieser Methode liegt in der besseren Übersichtlichkeit und Handhabbarkeit. Zudem können neue Schlüsselwörter ohne viel Aufwand hinzugefugt werden. Allerdings müssen gewisse Gesehwindigkeitseinbußen hingenommen werden, da das Suchen in der Symboltabelle meist über eine Hashfunktion erfolgt, und die Berechnung des Hashwertes eine nicht unerhebliche Zeit in Anspruch nimmt.

b) Eine weitere Möglichkeit zum Erkennen von Schlüsselwörtern ist das Anlegen einer statischen Tabelle und das Suchen eines Schlüsselwortes über eine geeignete Hashfunktion. Dieses Verfahren wird zum Beispiel in einigen "gcc" Implementierungen verwendet und ist von der

Effizienz her besser zu bewerten als die obige Variante, da eine auf die Schlüsselwörter spezialisierte Hashfunktion zum Einsatz kommen kann.

c) Für den hier zu implementierenden Compiler soll eine dritte Variante benutzt werden. Das hier vorgestellte Verfahren wird zum Beispiel auch in [Fra05] für den C-Compiler „Icc“ benutzt. Die Autoren betonen die Geschwindigkeitsvorteile dieser Methode gegenüber den obigen Verfahren. Die Idee ist, dass Schlüsselwörter wie jedes andere abstrakte Terminalsymbol behandelt werden. Dadurch entfällt das eventuelle Bilden eines Hashwertes zum schnellen Auffinden des Schlüsselwortes. Statt dessen werden einige wenige Zeichenvergleiche durchgeführt, so dass ein Schlüsselwort mit relativ geringem Aufwand identifiziert werden kann. Der Nachteil dieser Methode besteht in der stark steigenden Anzahl der Zustände des Übergangsdiagrammes und der damit verbundenen wachsenden Komplexität der Graphen (und damit des Scanners). Außerdem muss bei jedem Hinzufügen eines Schlüsslwortes das Übergangsdiagramm und der Scanner modifiziert werden. Im Fall des hier zu implementierenden Compilers standen die Schlüsselwörter der Sprache von Anfang an fest. Aus diesem Grund ist dies kein gravierender Nachteil. es überwiegen bei weitem die Geschwindigkeitsvorteile. Laut Randbedingungen ist dies ein Kriterium für den Entwurf des Compilers. Im Fall des hier implementierten Compilers konnte die Compiliergeschwindigkeit durch diese Methode gegenüber Methode a) um ca. 5-10% gesteigert werden!

Zur Verdeutlichung dieses Verfahrens soll im folgenden das Übergangsdiagramm für den Buchstaben "W" betrachtet werden, welches auch die Object-Pascal Schlüsselwörter "WHILE" und "WITH" erkennt.

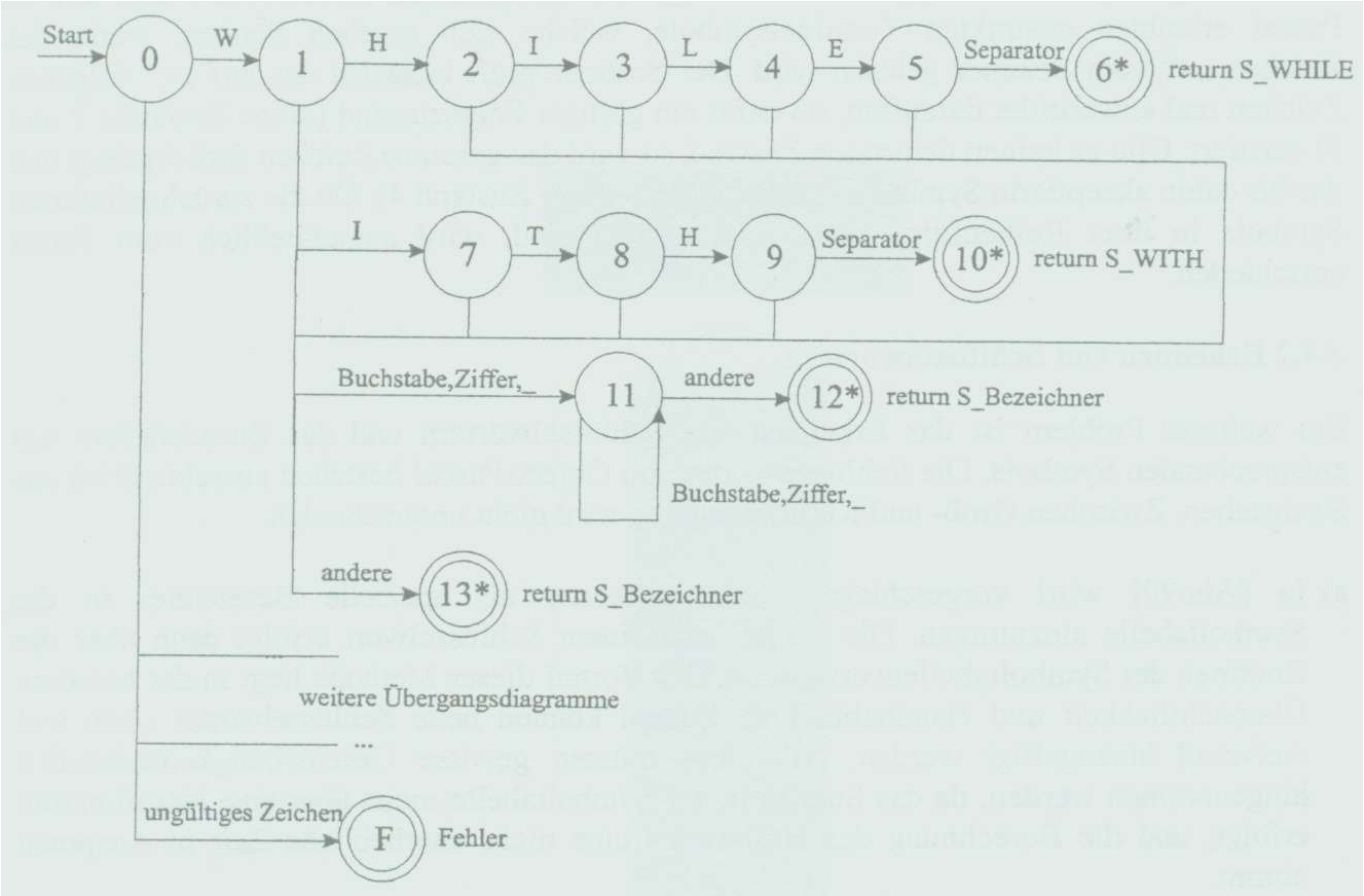


Abbildung 15: Das Übergangsdiagramm für den Buchstaben "W"

Unter einem Separator sollen in diesem Diagramm alle Zeichen verstanden werden, welche nicht in

einem Pascal-Bezeichner vorkommen können, d.h. alle Zeichen außer Buchstaben, Ziffern und dem Unterstrich. Zu beachten ist, dass die Zeichen der Eingabe vom Scanner bereits in Großbuchstaben umgewandelt werden (Pascal ist im Gegensatz zu "C" nicht casesensitiv!). Das Symbol S_Bezeiehner wird nicht direkt an den Parser weitergegeben. Stattdessen sucht der Scanner die entsprechende Zeichenkette in der Symboltabelle und liefert das dort gespeicherte Symbol und einen Verweis auf den Eintrag zurück. Existiert für den aktuellen Eingabestring in der Symboltabelle kein entsprechender Eintrag, so wird das abstrakte Terminalsymbol S_Name (bisher undefinierter Bezeichner) zurückgeliefert.

Auf den ersten Blick erscheint dieses Diagramm recht komplex. Es ist jedoch relativ einfach, aus einem solchen Diagramm systematisch einen entsprechenden Programmcode zu entwickeln (siehe Abschnitt ,, Implementation des Scanners"). Dabei werden die Zustände des Automaten durch Programmstellen repräsentiert. Die Darstellung der Übergänge erfolgt durch Verzweigungen und Schleifen [Kas90]. Eine Automatisierung dieses Vorganges wird Zum Beispiel durch den Scannergenerator ,,LEX" erreicht.

6.4.3 Pufferung der Eingabe

Für einen schnellen Scanner ist es von besonderer Bedeutung, die Zeichen des Eingabestromes in irgendeiner Form zu puffern. Da der Zugriff auf ein externes Speichermedium extrem langsam ist (der Scanner liest den Eingabestrom Zeichenweise!), bietet es sich an, den gesamten Eingabestrom[14] im Speicher zu halten. Die Markierung der aktuellen Position im Eingabestrom erfolgt dann einfach über einen Offset zum Start des allokierten Speicherbereiches. Die zu unterstützenden Betriebssysteme beinhalten ein sehr effektives Speichermanagement, so dass diese Methode auch bei großen Quelldateien oder knappen Hauptspeicher noch effizient ist. In [Aho90] wird darüber hinaus Techniken der zweigeteilten Puffer beschrieben, welche besonders für Systeme mit begrenztem Hauptspeicher interessant sind. Die hier zu unterstützenden Systeme stellen ausreichend virtuellen Hauptspeicher (RAM plus Auslagerungsspeicher) zur Verfugung. Beim Start des Compilers wird der gesamte Quelltext in einen internen Speicherbereich übertragen und die Quelldatei daraufhin geschlossen. Dies hat auch den Vorteil, dass die Quelldatei, sobald sie geschlossen ist, von anderen Prozessen im System modifiziert werden kann (etwa um während der Compilierung den Quelltext zu ändern), ohne den aktuellen Compilerlauf zu beeinflussen.

6.4.4 Behandlung van Kommentaren und Leerzeichen

Kommentare und Leerzeichen sind aus Sicht des Parsers redundant und werden deshalb bereits vom Scanner überlesen. Leerzeichen spielen darüber hinaus als Trennsymbole eine Rolle. Das Erkennen von Leerzeichen und Kommentaren erfolgt nach demselben Muster wie die Erkennung von abstrakten Terminalsymbolen, d.h. über die Erstellung eines Übergangsdiagramms. Der Scanner liefert jedoch nach Erkennung eines solchen Konstruktes kein Symbol beim Parser ab, sondern springt zum Startzustand 0 zurück, um das nächste für den Parser relevante Symbol zu erkennen.

6.5 Entwurf der Symboltabelle

Die Symboltabelle stellt die zentrale Datensammelstelle eines Compilers dar. Alle während der Compilierung anfallenden Informationen werden zum Zweck der späteren Verwertung hier gespeichert.

14 Hiermit sind Quellprogramme, gemeint, die heutzutage komplett in den Hauptspeicher passen.

Scanner und Parser machen beide intensiven Gebrauch von der Symboltabelle, um Bezeichner einzutragen bzw. zu finden. Deshalb ist der Entwurf der Symboltabelle mit besonderer Sorgfalt vorzunehmen. Deklarationen für die Symboltabelle sollen in der Datei „SYM.H“ abgelegt werden. Die Implementation erfolgt im Modul „SYM.CPP“

6.5.1 Sichtbarkeit von Bezeichnern

Zusätzlich zur Speicherung von Informationen hat die Symboltabellenverwaltung noch eine weitere wichtige Funktion. Sie verwaltet die Sichtbarkeit von Bezeichnern. In Pascal gilt Regel, dass (abgesehen von einigen Ausnahmen) jeder Bezeichner vor seiner ersten Benutzung deklariert werden muss. Diese Deklaration schließt die Angabe von zusätzlichen Informationen (z.B. eines Typs) für den Bezeichner ein. Pascal unterscheidet zwei Arten von Bezeichnern:

1. Globale Bezeichner
   Diese Bezeichner sind in allen Prozeduren und Funktionen des Programms sichtbar. Globale Bezeichner können durch lokale Bezeichner überlagert werden, d.h., sie sind dann während der Sichtbarkeit des lokalen Bezeichners nicht sichtbar (werden von diesen überdeckt).

2. Lokale Bezeichner
   Diese Bezeichner sind nur innerhalb der umgebenden Prozedur oder Funktion sichtbar. Da Object-Pascal die Schachtelung von Prozeduren zulässt, können auch lokale Bezeichner überlagert werden.

Bezeichner werden also auf Prozedurebene[15] verwaltet, wobei man das Hauptprogramm als eine alles umschließende Prozedur[16] verstehen konnte. Die Symboltabellenverwaltung muss diesen Regeln Rechnung tragen. Dafür wird für das Hauptprogramm und für jede verwendete Prozedur/Funktion ein Namensraum (Scope) verwaltet, welcher die dort sichtbaren Bezeichner aufnimmt. Der Eintrag der Bezeichner erfolgt durch den Parser oder den Scanner. Für jede Prozedur/Funktion wird beim Erkennen ihrer Deklaration ein neuer Namensraum angelegt. Namensraum werden als Stack verwaltet, wobei der sich an der Spitze des Stacks (TOS - Top of Stack) befindliche Namensraum zuerst durchsucht wird, dann der darunterliegende usw. Beim Verlassen des Anweisungsteiles einer Prozedur/Funktion wird das oberste Stackelement vom Stack genommen und das darunterliegende wird zum TOS. Dies sichert die korrekte Verwaltung von hierarchisch verschachtelten Namensraurnen. Das Suchen eines Bezeichners beginnt stets bei dem durch TOS bezeichneten Namensraum.

Für das Anlegen eines neuen Namensraumes soll die Funktion „*DictNew*" bereitgestellt werden. Die Modifikation des Stacks soll über die Funktionen „*PushLevel“* und „*PopLevel*" erfolgen. Der Stack wird als verkettete Liste implementiert. Die Variable „*scopeStack*" soll auf den TOS verweisen.

6.5.2 Das Unitkonzept von Object-Pascal

Zusätzlich verschärft werden die Regeln zur Sichtbarkeit von Bezeichnern durch die in Objekt-Pascal gegebene Möglichkeit der Nutzung von Units. Units sind separat übersetzbare Einheiten, welche, ähnlich zu MODULA-2, Bezeichner importieren und exportieren können. Units gestatten die Entwicklung von abstrakten Datentypen. Dabei können Teile der Implementation vom Nutzer verborgen werden.

15 Hiermit sind sowohl Prozeduren und Funktionen, als auch Objektmethoden gemeint.

16 Allerdings nicht im strengen Sinne der Definition von Prozeduren. Diese Vereinfachung wird hier nur aus Gründen der besseren Verständlichkeit gemacht.

Bezeichner, welches außerhalb der Unit sichtbar sein sollen, werden im „Interface Teil“ der Unit deklariert. Private Bezeichner sind nur innerhalb der Unit sichtbar und werden im „Implementation Teil“ der Unit angegeben. Eine Unit kann nur in ihrer Gesamtheit (d.h. mit allen öffentlichen Bezeichner) importiert werden. Generell gilt, dass die von einer Unit exportierten Bezeichner immer globale Bezeichner sind. Die folgende Abbildung stellt eine typische Nutzung von zwei Units dar:

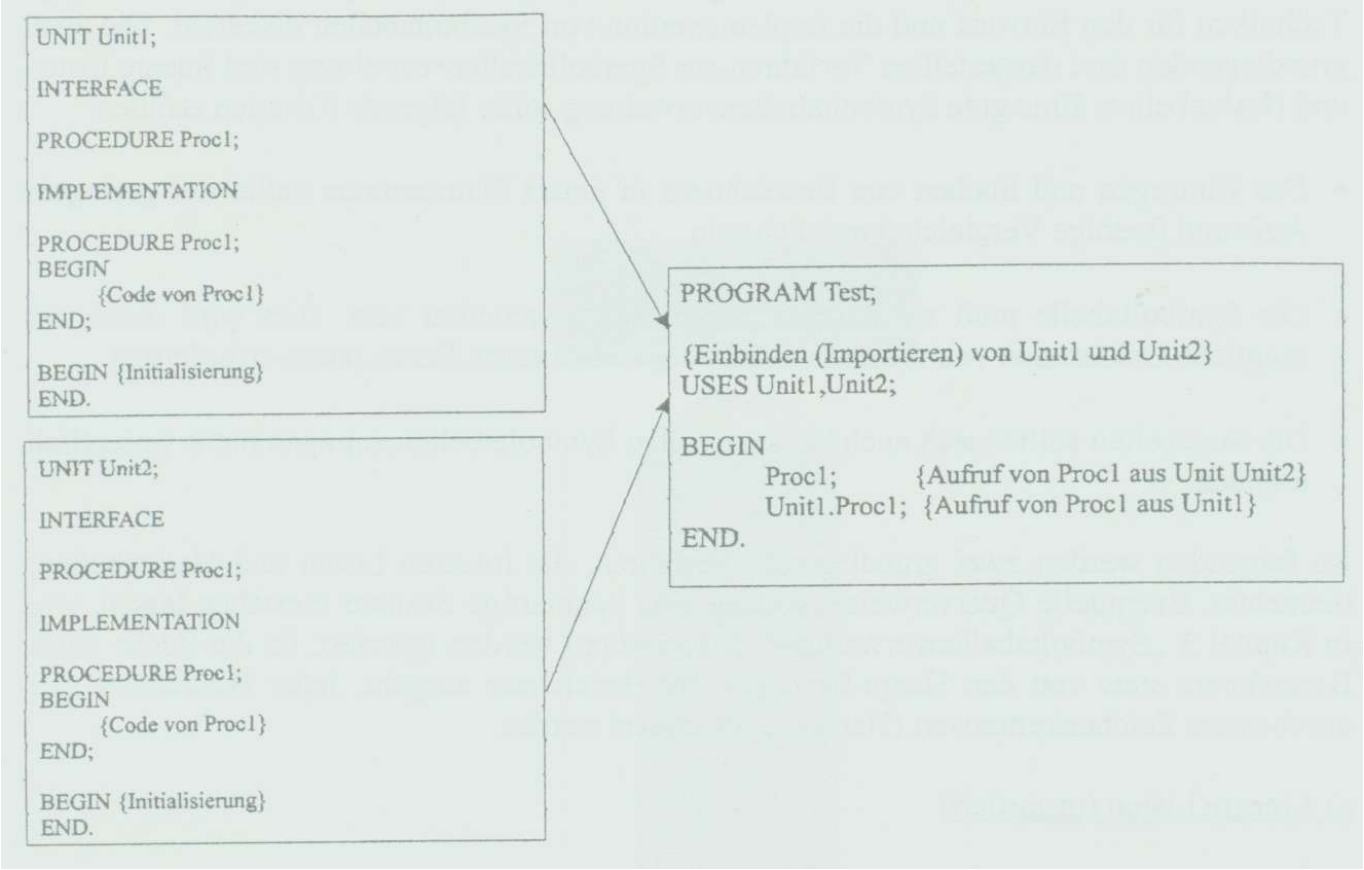


Abbildung 16: Das Unit-Konzept von Object-Pascal

Das Hauptprogramm (Test) importiert die Units "Unit1" und "Unit2" und macht damit deren öffentliche Bezeichner im Hauptprogramm sichtbar. Entgegen der Regel, dass Bezeichner in Pascal innerhalb eines Namensraumes nicht doppelt vorkommen dürfen, können globale Bezeichner durch das Einbinden von Units überschrieben werden. Im Beispiel überschreibt das Einbinden von "Unit2" den in "Unit1" exportierten globalen Bezeichner ,,Proc1". Der Aufruf von ,,Procl" aus "Unit1" ist nur noch über den sogenannten qua1ifizierten Import "Unit1.Procl" möglich. Aus dem Unit-Konzept von Object-Pascal ergeben sich für die Symboltabellenverwaltung folgende Konsequenzen:

1. Für jede importierte Unit muss ein eigener Namensraum angelegt werden.

2. Die Reihenfolge, in welcher die Namensraume für Units auf dem Stack abgelegt werden, wird durch die Reihenfolge ihres Imports bestimmt.

3. Zum Zweck des qualifizierten Imports muss die Symboltabelle in der Lage sein, Bezeichner aus einem angegebenen Modul aufzufinden.

Aufgrund der schon diskutierten Trennung der Namensraume und der Ablage auf einen Stack fügen sich diese Forderungen relativ nahtlos in das besprochene Konzept ein.

6.5.3 Der Aufbau der Namensräume

Die effiziente Verwaltung von Bezeichnern ist ein wesentliches Kriterium für die Compilergeschwindigkeit. Aus vorhandenen Erfahrungen zeigt sich, dass eine ungenügend strukturierte oder ineffiziente Symboltabellenverwaltung die nachfolgenden Teile eines Compilers erheblich komplizierter und langsamer machen kann. In [Aho90] werden einige Techniken für den Entwurf und die Implementation von Symboltabellen diskutiert. Die zwei grundlegenden dort dargestellten Verfahren zur Symboltabellenverwaltung sind lineare Listen und Hashtabellen. Eine gute Symboltabellenverwaltung sollte folgende Kriterien erfüllen:

- Das Eintragen und Suchen von Bezeichnern in einem Namensbaum sollte mit geringem Aufwand (wenige Vergeleiche) möglich sein

- Die Symboltabelle muss zur Laufzeit dynamisch erweiterbar sein. Dies wird durch die mögliche Deklaration von theoretisch beliebige vielen neuen Bezeichnern erforderlich.

- Die Suchzeiten sollten sich auch bei sehr vielen Symboltabelleneinträgen nicht wesentlich erhöhen.

Im folgenden werden zwei grundlegende Verfahren, die linearen Listen und Hashtabellen, betrachtet. Eventuelle Querverweise (weiche eine baumartige Struktur entstehen lassen, wie in Kapitel 3 "Symboltabellenverwaltung" beschrieben) werden ignoriert, da die Suche eines Bezeichners stets von den Haupt-Eintragen für Bezeichner ausgeht. Jeder Bezeichner soll durch einen Zeichenkettenwert (Name) repräsentiert werden.

a) Lineare Listen (unsortiert)
   Dieses Verfahren ist relativ leicht zu implementieren. Die Symboltabelleneinträge werden linear durch Referenzen miteinander verkettet. Altemativ können die Eintrage auch in einem Feld (Array) abgelegt sein, wie in [Aho90] beschrieben.

   Zum Auffinden eines Bezeichners sind bei dieser Struktur bei n Eintragen im Durchschnitt n/2 Vergleiche durchzuführen. Die Kosten für das Auffinden eines Bezeichners sind somit proportional zu n. Im ungünstigsten Fall (der zu findende Bezeichner steht immer am Ende der Liste) sind die Kosten gleich n.

   Die Kosten zum Eintragen eines Bezeichners sind ebenfalls proportional zu n, da Pascal Bezeichner mit derselben Identifikation in einem Namensraum nicht erlaubt. Somit muss zunächst die gesamte Liste durchsucht werden, um sicherzustellen, dass der Bezeichner nicht bereits enthalten ist.

   Die maximalen Kosten für das Einfügen von n Namen und das Auffinden von e Bezeichnern für diese Variante sind also n(n+e) [Aho90].

b) Hashtabellen

   Eine weiter wesentliche effektivere Methode besteht in der Verwendung eines Hashalgorithmus. Dabei wird für jeden Bezeichner ein (nicht notwendigerweise eindeutiger) Hashwert gebildet. Das Auffinden des Bezeichners kann dann unter Verwendung des Hashwertes wesentlich schneller erfolgen. Für den hier zu implementierenden Compiler soll das in [Aho90] beschriebene

Verfahren des „*offenen Hashings*“ verwendet werden. Kennzeichnend für dieses Verfahren ist, dass es keine Begrenzung für die Anzahl der möglichen Symboltabelleneintrage gibt. Das zugrunde liegende Verfahren beruht auf drei Ideen [Ah090]:

1.) Es wird eine HashtabeIle bereitgestellt. Diese Tabelle besteht aus einem Feld (Array) mit m Einträgen. Die Größe dieser Tabelle ist fest und hängt von der Anzahl der voraussichtlich zu erwarteten Symboltabelleneintrage und dem verfügbaren Hauptspeicher ab.

2.) Jeder der m Eintrage der HashtabeIle identifiziert eine einfach verkettete Liste.

3.) Zum Auffinden eines Bezeichners wird ein Hashwert h gebildet, welcher den Index in der HashtabeIle darstellt. Die Hashfunktion hash(s) liefert also für die Zeichenkette (den Namen des Bezeichners) s einen Hashwert 0 <= h <= m-1. Durch Suchen in der durch h indizierten Liste kann ein Bezeichner in der Regel durch wenige Vergleiche gefunden werden.

Der maximale Aufwand zum Einfugen von n Bezeichnern und das Auffinden von e Bezeichnern reduziert sich bei diesem Verfahren zu n(n+e)/m. Der Wert von m kann beliebig hoch gewählt werden. Diese Methode ist also generell effektiver zu bewerten als a) [Ah090]. Da der Speicherbedarf für die Verwaltung mit steigendem m wächst, ist m so zu wählen, dass ein günstiges Verhältnis zwischen Zeit- und Platzbedarf entsteht. Die Hashfunktion ist so zu wählen, dass möglichst wenige Bezeichner den gleichen Hashwert erhalten.

c) Andere Verfahren

Grundsätzlich sind zur Verwaltung der Symboltabelle noch andere Verfahren möglich, beispielsweise die Nutzung von binären Suchbäumen. In der Literatur wird auf diese Verfahren nicht eingegangen. Eine wesentliche Effizienzsteigerung gegenüber einem geschickt gewähltem Hashverfahren ist nicht zu erwarten.

Da genügend Erfahrungen in der Nutzung von Hashfunktionen für die Symboltabellenverwaltung zur Verfugung standen und dieses Verfahren auch in der Literatur favorisiert wird, soll es für den zu entwerfenden Compiler verwendet werden.

6.5.4 Vordefinierte Bezeichner

In Object-Pascal existieren eine ganze Reihe vordefinierter Bezeichner. Diese unterteilen sich in Standardtypen (z.B. Integer), Konstanten (z.B. True) und Funktionen (z.B. Ord). Vordefinierte Bezeichner werden in der Initialisierungsphase in den Namensraum des Hauptprogrammes eingetragen und sind von da ab sichtbar. Ein Überschreiben dieser Bezeichner ist zwar möglich, sollte aber aus Portabilitäts- und Übersichtlichkeitsgründen vermieden werden. Eine komplette Aufstellung aller vordefinierten Bezeichner findet sich in Anhang C.

6.6 Entwurf des Parsers

Aufgabe des Parsers ist es, den vom Scanner gelieferten Symbolstrom auf syntaktische Korrektheit hin zu überprüfen. Darüber hinaus soll der gesamte Übersetzungsvorgang vom Parser gesteuert werden. Der Parser wird nach der Methode des rekursiven Abstiegs entworfen. Jedes NTS der Grammatik erhalt also eine Parserprozedur zugeordnet, wie in Kapitel 2 beschrieben. Der Start des Parsers erfolgt durch den

Aufruf der Prozedur für das NTS, welches Startsymbol der Grammatik ist. Der Parser

- fordert bei Bedarf über die Funktion "*GetTok*" neue Symbole vom Scanner an,
- entscheidet die syntaktische Korrektheit des vom Scanner gelieferten Symbolstromes und löst entweder einen fatalen Fehler („*Error*") oder eine Warnung ("*Warning*") aus,
- überprüft die semantische Korrektheit des Programms. Dazu werden Typinformationen über Datenobjekte zum Zweck des späteren Vergleichs festgehalten,
- baut einen Zwischencodebaum durch Aufruf von Routinen des Zwischencodegenerators auf,
- ruft an geeigneter Stelle Routinen des Codegenerators auf, um den Zielcode zu erzeugen.

Die Routinen des Parsers können zum großen Teil aus dem vorhandenen Compiler übernommen werden. Deshalb wird für die Implementation des Parsers kein Parsergenerator verwendet, da sich die vorhandenen Routinen in der Praxis bewährt haben. Zusätzliche Erweiterungen sind mit wenig Aufwand möglich. Bei der Übertragung der Routinen bzw. beim Hinzufügen von Prozeduren zum Parser wird strikt auf die Übereinstimmung mit der im Anhang enthaltenen Object-Pascal Grammatik geachtet.

6.7 Fazit

In diesem Kapitel wurden mögliche Varianten für den Entwurf des Frontends diskutiert und ausgewählt. Es ist deutlich geworden, dass es zu allen wichtigen Komponenten des Frontends mehrere alternative Lösungsvarianten gibt. Die Auswahl der Lösungsvarianten gründet sich auf die Randbedingungen der Diplomarbeit wie

a) Maximale Ausnutzung vorhandener Compilerkomponenten und Erfahrungen

b) Forderung nach Compilergeschwindigkeit

c) Einfachheit und Effizienz der verwendeten Algorithmen

Obwohl sich die Generierung der Komponenten des Frontends durch Scanner- und Parsergeneratoren (LEX und YACC) weitgehend automatisieren lässt, wird bei dem zu entwerfenden Compiler auf diese Möglichkeiten verzichtet. Dies lässt sich zum großen Teil auf den obigen Punkt a) und zum Teil auch b) zurückführen. Es erscheint generell zweckmäßiger, vorhandene Erfahrungen, Algorithmen und Unterprogramme so weit als möglich wieder zu verwenden. Dies beeinflusst insbesondere den Entwurf des Scanners und des Parsers. Eine Umstellung auf die erwähnten Generatoren hätte einen nicht unerheblichen Teil der zu Verfügung stehenden Zeit in Anspruch genommen. Außerdem ist die Wiederverwendung vorhandene und erprobter Algorithmen eine generelle Forderung der Informatik. Die vorhandenen Algorithmen und Datenstrukturen werden in sinnvoller Weise in das neue Compilerkonzept eingearbeitet. Diese Vorgehensweise verkürzt zudem die Testphase einzelner Komponenten, da sich z.B. die Parserprozeduren zum Teil schon seit mehreren Jahren in der Praxis bewährt haben[17].

17 Die erste Version von Speed-Pascal ist seit Ende 1994 verfügbar

7 Kapitel 5
Entwurf der Zwischensprache

Die Zwischensprache stellt eine definierte Schnittstelle zwischen dem Frontend und dem Backend des Compilers her. Eine gut durchdachte Zwischensprache vereinfacht die Implementation beider Teile und kann die Portierung eines Compilers wesentlich erleichtern. Der Entwurf der Zwischensprache ist also ein wichtiges Element des gesamten Compilerentwurfes. Beim Entwurf der Zwischensprache sind weitere Aspekte zu berücksichtigen, we1che in den meisten bekannten Zwischensprachen realisiert werden:

- Auflösung der Schachtelung von Prozeduren
  Verschachtelte Prozeduren werden vom Parser aufgelöst. Die Darstellung dieser Konstrukte in der Zwischensprache wird linearisiert, d.h. für die Zwischensprache existieren keine verschachtelten Prozeduren.

- Auflösung von komplizierten arithmetischen Ausdrücken
  Komplizierte arithmetische Ausdrücke werden von der Zwischensprache linearisiert und eine eventuelle Klammerung von Ausdrücken aufgelöst. Die ermöglicht das leichte Umgruppieren von Teilausdrücken zum Zweck der Optimierung und macht die Synthese einfacher.

- Elimination von Ausdrucksredundanzen
  Die Elimination von gemeinsamen Teilausdrücken kann bereits beim Erzeugen der Zwischensprache erfolgen. Dabei werden bestimmte (maschinenunabhängige) Optimierungen bereits in das Frontend des Compilers verlagert. Dies vereinfacht die Codegenerierung wesentlich.

Der Entwurf einer geeigneten Zwischensprache ist fast zwingend notwendig, wenn der Compiler auch auf andere Plattformen portiert[18] werden soll. Nur durch eine Zwischensprache kann die notwendige Unabhängigkeit von Frontend und Backend des Compilers erreicht werden.

7.1 Das Abstraktionsniveau einer Zwischensprache

Je nach Einsatzzweck kann für eine Zwischensprache ein Abstraktionsniveau[19] gewählt werden. Es wird hierbei zwischen maschinen- und sprachspezifischen Zwischensprachen unterschieden. Darüber hinaus hat es in den 50er Jahren (vergleiche [Str58]) Bestrebungen gegeben, eine universelle Zwischensprache (UNCOL - Universal Computer Oriented Language) zu definieren. Ziel war es, die Definition der Zwischensprache so universell wie möglich zu halten und damit eine möglichst große Anzahl von Quellsprachen und Zielmaschinen zu unterstützen. Das Vorhandensein einer UNCOL hätte es ermöglicht, die Portierung von Compilern mit extrem geringem Aufwand zu ermöglichen, da für jede Zielmaschine nur ein einziger Codegenerator nötig wäre, welcher die von den verschiedenen Compilern erzeugten Zwischendarstellung für den jeweiligen Prozessor umsetzt. Voraussetzung dafür ist die völlige Quell- und Zielsprachenunabhängigkeit der UNCOL. Aufgrund der Vielzahl der unterschiedlichen

18 Unter Portierung soll im folgenden das Anpassen des Backends an eine neue Hardwarearchitektur oder ein neues Betriebsystem gemeint sein.

19 Hiermit soll im folgenden die Quellsprachenabhängigkeit der Zwischensprache gemeint sein. Steigendes Abstraktionsniveau bedeutet von der Quellsprache.

Programmiersprachenkonzepte und der unterschiedlichen Maschinenarchitekturen ist das Konzept der UNCOL gescheitert (IEuI88]), obwohl in der Vergangenheit Versuche unternommen wurden, eine solche UNCOL zu realisieren (z.B. die Zwischensprache JANUS in [Bron]). Ein weiterer Grund für das Scheitern des UNCOL Konzeptes ist wohl, dass die verschiedenen Compilerhersteller nicht gewillt oder nicht fähig waren, sich auf solch ein einheitliches Konzept zu einigen. Statt dessen entstanden "Insellösungen" einiger Firmen, welche sich hausintern eine UNCOL entwarfen. Prominentes Beispiel ist die Firma Borland, welche eine einheitliche Zwischensprache für ihre Pascal (Delphi) und C-Compiler verwendet. Dies erlaubt es dem Pascal Compiler, das Backend des C-Compilers zu nutzen und damit eine von C-Compilem gewöhnte exzellente Codequalität zu erreichen.

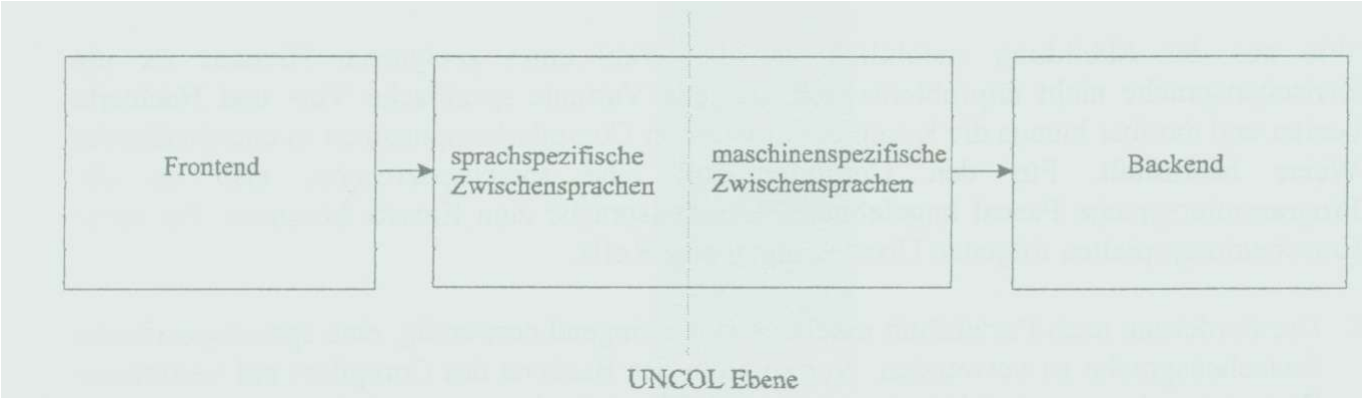


Abbildung 17: Abstraktionsniveaus einer Zwischensprache

Zwischensprachen, welche in der Abbildung links der UNCOL Ebene liegen, enthalten keinerlei maschinenabhängige Elemente, dafür in wachsendem Maße sprachspezifische Konstrukte. Zwischensprachen für Niveaus rechts der UNCOL Ebene sind in wachsendem Maße maschinenabhängig, enthalten jedoch keinerlei sprachspezifische Elemente mehr. Im Idealfall (UNCOL Ebene!) enthält die Zwischensprache weder sprach- noch maschinenabhängige Elemente. Das Niveau einer Zwischensprache bestimmt im wesentlichen den Aufwand zur Portierung eines Compilers auf eine andere Zielmaschine. Ist das Niveau zu niedrig (zu nah an der Maschine), so erschwert dies die Portierung des Compilers auf eine völlig andere Rechnerarchtiktur oder macht sie generell unmöglich. Für ein Höchstmaß an Portabilität ist also eine sprachspezifische Zwischensprache zu wählen. Darüber hinaus existiert natürlich die Möglichkeit, mehrere Zwischensprachen zu verwenden, welche ineinander transformiert werden. Dies impliziert aber zusätzlichen Aufwand und Fehlerquellen. Die Vor- und Nachteile der verschiedenen Möglichkeiten einer Zwischensprache sind in der folgenden Tabelle dargestellt (das Maß der Sprach- und Maschinenabhängigkeit kann sich je nach Anwendungsfall als Vor- oder Nachteil erweisen):

sprachspezifische Zwischensprachen	maschinenspezifische Zwischensprachen
+/- hohes Maß an Sprachabhängigkeit	+/- hohes Maß an Maschinenabhängigkeit
+ einfache Zwischencodegenerierung	+ einfache Codegenerierung
+ gute Portabilität auf andere Maschinen	+ gute Portabilität auf andere Quellsprachen
+ Sprachabhängige Optimierungen auf Zwischensprachenebene möglich	+ Maschinenabhängige Optimierungen auf Zwischensprachenebene möglich
- Nur für eine spezielle Programmiersprache entworfen und nutzbar	- schlechte Portabilität auf andere Maschinen
- komplexere Codegenerierung	- komplexere Zwischencodeerzeugung
- Maschinenabhängige Optimierungen auf Zwischensprachenebene nicht möglich	- Sprachabhängige Optimierungen auf Zwischensprachenebene nicht möglich
	- Eventuell höherer Speicherbedarf für interne Darstellung

Tabelle 3: Die Vor- und Nachteile der verschiedenen Niveaus von Zwischensprachenen

Wie aus der Abbildung ersichtlich, ist die Wahl eines geeigneten Niveaus für die Zwischensprache nicht unproblematisch, da jede Variante spezifische Vor- und Nachteile besitzt und darüber hinaus die Komplexität weiterer Compilerkomponenten in entscheidender Weise beeinflusst. Für den Compiler soll eine sprachspezifische, also an die Programmiersprache Pascal angelehnte, Zwischensprache zum Einsatz kommen. Für diese Entscheidung spielten folgende Überlegungen eine Rolle:

1. Die Forderung nach Portabilität macht es fast zwingend notwendig, eine sprachspezifische Zwischensprache zu verwenden. Nur so kann das Backend des Compilers auf vertretbare Weise an eine neue Architektur angepasst werden. Für die Zwischensprache ist die Zielarchitektur im Fall einer sprachspezifischen Zwischensprache irrelevant.

2. Die Generierung und Optimierung einer sprachspezifischen Zwischensprache erscheint generell einfacher möglich zu sein. Da sich eine solche Sprache stark an die Quellsprache anlehnt, kennen viele Konstrukte der Quellsprache 1:1 in die Zwischensprache umgesetzt werden.

3. Eine sprachspezifische Zwischensprache ist eine viel abstraktere Darstellung des Quellprogramms, als dies eine maschinenabhängige Zwischensprache ist. Bei Bedarf kann die sprachspezifische Zwischensprache auch in eine maschinenabhängige Darstellung transformiert werden. Eine sprachspezifische Zwischensprache bietet also mehr Freiheit in Bezug auf spätere Erweiterungen. So wäre eine Transformation einer sprachspezifischen Zwischensprache in eine, speziell auf eine Zielmaschine abgestimmte Darstellung mit gleichzeitiger Optimierung für diese Zielmaschine möglich. Für diese Vorgehensweise müsste der Codegenerator lediglich um eine zusätzliche Komponente erweitert werden, welche zusätzliche Transformationen und Optimierungen ausführt. das gesamte Frontend und die übrigen Komponenten des Compilers blieben von dieser Maßnahme unbeeinflusst.

7.2 Der Aufbau der Zwischensprache

Für den Entwurf des Compilers sol1 eine Pascal-spezifische Zwischensprache verwendet werden. Intern wird die Zwischensprache als Graph G=(K,B) verwaltet, wobei:

- K eine Menge von Knoten des Graphen ist (Knotenmenge)

- B eine Menge von Bogen (Kanten) ist, welche die Knoten verbinden. Diese Kanten sind graphische Darstellungen der Relationen zwischen den Knoten. Es gilt: $B \subseteq K \times K$.

Die Darstellung der Zwischensprache soll im folgenden durch Syntaxbäume und gerichtete azyklische Graphen (GAG) erfolgen. Syntaxbäume beschreiben hierbei den hierarchischen Aufbau des Quellprogramms. Sie sind analog zu einem Parserbaum aufgebaut, enthalten jedoch keinerlei für die Zwischendarstellung redundante Informationen (wie zum Beispiel Deklarationen) mehr. Ein GAG entsteht aus einem Syntaxbaum durch das Zusammenfassen von gemeinsamen Teilausdrücken. Syntaxbäume sind binare Bäume, von jedem Knoten gehen maximal 2 Bogen (Kanten) ab. Der Zwischencodegenerator transformiert diesen Syntaxbaum in die optimierte Darstellung als GAG. Die Darstellungen als Syntaxbaum und als GAG für den Pascal-Ausdruck "x:= x*y + x*y" sind in Abbildung 18 enthalten.

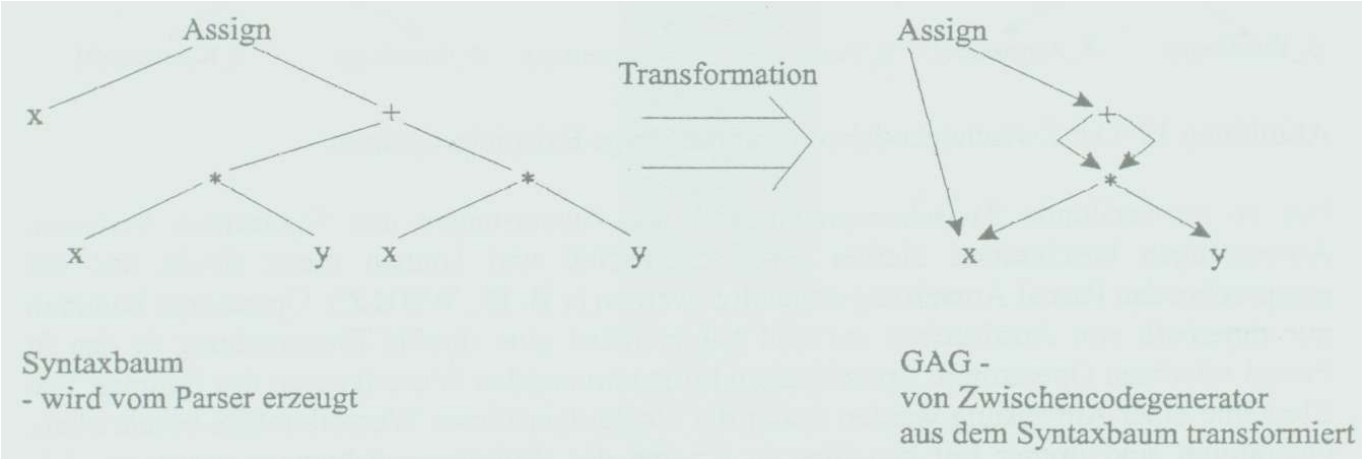


Abbildung 18: Syntaxbaum und GAG für den Pascal-Ausdruck "x:=x·y+x·y"

In der obigen Abbildung für den GAG hat das Blatt x zwei Vorgänger, weil der Ausdruck x jeweils auf der linken und rechten Seite der Zuweisung vorkommt. Außerdem wird der gemeinsame Teilausdruck "x * y" erkannt und von beiden Teilausdrücken genutzt.

Die Darstellung als Baum hat folgende Vortei1e:

- Die hierarchische Struktur der Anweisungen eines Programms bleibt erhalten.
- Der Zwischencodebaum kann sehr leicht vom Parser erzeugt werden, da die Darstellungsweise einem Parserbaum ähnelt.
- Es können leicht andere Formen der Zwischendarstellung (etwa 3-Adreß Code) aus dem Baum erzeugt werden, um die Optimierungsmöglichkeiten zu verbessern

Jede Anweisung des Quellprogramms wird in einen GAG umgesetzt. Ein solcher GAG wird im folgenden auch als *Grundblock* bezeichnet. Eine Folge von Anweisungen wird durch die gerichtete Verbindung von mehreren Grundblöcken erreicht. Der Kontrollfluss des Programms ergibt sich aus der Richtung der Verbindung der einzelnen Grundblöcke. Ein solcher aus einzelnen Grundblöcken zusammengesetzter Baum wird im folgenden als *Zwischencodebaum* bezeichnet

Für das Programm

```
PROGRAM Test;
VAR x,y,z : Integer;
BEGIN
        x:=I;
        y:=2;
        z:=3;
END.
```

entsteht folgender Baum:

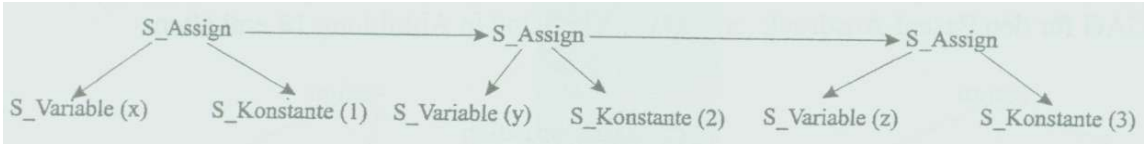


Abbildung 19: Der Zwischencodebaum für das obige Beispielprogramm

Die zu verwendende Zwischensprache soll aus Anweisungen und Operatoren bestehen. Anweisungen beschreiben hierbei einen Steuerfluss und können meist direkt aus der entsprechenden Pascal Anweisung abgeleitet werden (z.B. IF, WHILE). Operatoren kommen nur innerhalb von Ausdrücken vor und haben meist eine direkte Entsprechung in den in Pascal erlaubten Operatoren. Anweisungen bilden immer den Wurzelknoten des Baumes. Die Elemente einer Anweisung werden durch die Nachfolger dieses Wurzelknotens beschrieben. Operatoren sind immer mit den inneren Knoten des Zwischencodebaumes assoziiert. Die Nachfolger dieser Knoten beschreiben die Operanden des Operators. Die Blätter des Baumes werden durch Bezeichner (Variablen oder Konstanten) beschrieben. Diese Blätter können nicht weiter substituiert werden, sind also elementare Elemente der Zwischendarstellung. Die entworfene Zwischensprache besitzt 23 Anweisungen und 63 Operatoren. Die Anweisungen und Operatoren der Zwischensprache sind in Anhang D enthalten. Aufgrund der Gegebenheiten der Zielmaschine und der Erfordernisse der Sprache Pascal enthält die Zwischensprache einige Besonderheiten:

1. Bei arithmetischen Operationen wird zwischen Operationen mit ganzzahligen Werten und Operationen mit Fließkommawerten unterschieden.

   Auf vielen Plattformen werden Fließkommaoperationen durch einen numerischen Koprozessor mit eigenem Befehls- und Registersatz ausgeführt. Da die Zwischendarstellung nur noch rudimentäre Typinformationen beinhaltet, wird die Unterscheidung zwischen Fest- und Fließkommaoperationen bereits beim Aufbau des Zwischencodebaumes getroffen. Dies vereinfacht die Synthese.

2. Bei Operationen mit ganzzahligen Werten wird zwischen vorzeichenbehafteten und vorzeichenlosen Operationen unterschieden.

   Die meisten Architekturen unterscheiden bei bestimmten Operationen (z.B. Division und Multiplikation) zwischen vorzeichenbehafteten und vorzeichenlosen Operationen. Diese Unterscheidung wird ebenfalls bereits beim Aufbau des Zwischencodebaumes getroffen. Für Architekturen, welche diese Operationen nicht unterscheiden, können die beiden Varianten zusammengelegt werden. Diese Unterscheidung wäre nicht zwingend notwendig, vereinfacht aber die Synthese.

3. Operationen mit booleschen Werten werden gesondert behandelt.

   Pascal unterscheidet zwischen kompletter Auswertung eines booleschen Ausdruck (complete boolean evaluation) und ,,Kurzschlussauswertung" (short circuit boolean evaluation). Die Wahl des Modus wird durch Compilerdirektiven gesteuert. Bei einer kompletten Auswertung eines booleschen Ausdrucks wird der gesamte Ausdruck ausgewertet, unabhängig davon, ob der Wert des Gesamtausdrucks bis dahin schon feststeht. Bei einer ,,Kurzschlussauswertung" wird die Berechnung des Ausdrucks abgebrochen, sobald der Resultatwert feststeht. So wird im Fall des Ausdrucks "a AND b" für a=FALSE und b=TRUE nur der Ausdruck a ausgewertet, da der Wert von b am Wert des Ergebnisses (FALSE) nichts mehr ändem kann.

Die gewählte Zwischensprache ist also eine Pascal-spezifische Zwischensprache. Zur Vereinfachung der Codesynthese wurden einige spezielle Elemente in die Zwischensprache integriert.

7.3 Andere Formen der Zwischendarstellung

Aus Syntaxbäumen und GAGs kann leicht eine andere Form der Zwischendarstellung abgeleitet werden, der so genannte 3-Adreß Code. Der 3-Adreß Code ist eine linearisierte Darstellung eines Syntaxbaumes oder eines GAG. 3-Adreß Code ist eine stärker maschinenabhängige Form der Zwischendarstellung als ein GAG, da hier bereits mit Registern gearbeitet werden kann. 3-Adreß Code ist stark verwandt mit Assembler-Code. Der 3-Adreß Code ist eine Folge von Anweisungen der allgemeinen Form:

x := y op z [Ah090]

wobei x, y und z Bezeichner, Konstanten oder vom Compiler generierte temporäre Werte oder Register sind. Für das Beispiel aus Abbildung 18 ergibt sich folgende Darstellung in 3-Adreß Code:

r1 := x * y	r1 := x * y
r2 := x * y	r2 := x * y
r3 := r1 + r2	x := r2
x := 3	
3-Adreß-Code für den Syntaxbaum	3-Adreß-Code für den GAG

Auf die Erzeugung von 3-Adreß Code aus GAGs wird im zu implementierenden Compiler verzichtet. Obwohl diese Form der Darstellung wesentlich bessere Optimierungen zulässt, als dies bei einem GAG möglich ist (siehe [Aho90]), hatte die Umsetzung dieser Prinzipien und der sich daraus ergebenden Optimierungsmöglichkeiten den Rahmen dieser Diplomarbeit gesprengt. Prinzipiell ist ein solches Vorgehen durch eine weitere Transformation der gewählten Zwischensprache jedoch möglich.

7.4 Zwischencodeoperatoren

Viele Operatoren der Quellsprache (etwa eine Addition oder Multiplikation) werden direkt in Operatoren der Zwischensprache umgesetzt. Operatoren verknüpfen *Ausdrücke*. Ein Ausdruck liefert immer einen entsprechenden Wert zurück. Im folgenden werden zwei Arten von Ausdrücken unterschieden:

1. Als *LValue*[20] wird jeder Ausdruck bezeichnet, welcher in Pascal auf der linken Seite einer Zuweisung erlaubt ist. Beispiele hierfür sind "x", aber auch "x[10]" und "x^.Feld[y]".

2. Als *RValue* wird jeder Ausdruck bezeichnet, welcher in Pascal auf der rechten Seite einer Zuweisung erlaubt ist. Ein LValue ist also eine spezielle Form eines RValue .

Alle Operatoren der Zwischensprache erwarten Operanden für die Operation. Ein Operand kann ein LValue oder ein RValue sein. Operanden können wiederum hierarchisch aus Operatoren und Ausdrücken aufgebaut sein. Für eine Operation, welche einen LValue erwartet, ist kein RValue als Eingabe zulässig. Der umgekehrte Fall ist jedoch, mit Bezug auf die obige Definition, sehr wohl korrekt. Die Syntax des Zwischencodebaumes ist immer korrekt, da der Baum intern vom Zwischencodegenerator aufgebaut wird.

20 In C-Lehrbüchern werden die Begriffe „LValue“ und „RValue“ etwas anders definiert. Für diese Arbeit gelten die Definition wie angegeben.

7.4.1 Darstellung von Variablen und Konstanten

Variablen und Konstanten werden auf besondere Weise im Zwischencodebaum dargestellt. Die folgenden Zwischencodeoperatoren verkörpern Variablen und Konstanten:

Operator	Bedeutung
S CStringVariable	linker Teilbaum ist ein Verweis auf eine Variable vom Typ "CString"
S FloatVariable	linker Teilbaum ist ein Verweis auf eine Fließkommavariable
S_FloatZahl	linker Teilbaum ist ein Verweis auf eine Fließkommakonstante
S_Konstante	linker Teilbaum ist ein Verweis auf eine Konstante
S_StringVarible	linker Teilbaum ist ein Verweis auf eine Variable vom Typ „String“
S_STRING_Text	linker Teilbaum ist ein Verweis auf eine Zeichenkettekonstante
S_Variable	linker Teilbaum ist ein Verweis auf eine Variable

Tabelle 4: Die Zwischencodeoperatoren für Variablen und Konstanten

Die Zwischensprache unterscheidet also die Typen:

- numerisch reell,
- Zeichenketten (String oder CString[21]),
- andere Typen.

Diese Entscheidung ist nicht zwingend notwendig, vereinfacht aber die Codegenerierung. Jeder Ausdruck erhält ein Attribute mit dem entsprechenden Typcode. Dureh die obige Unterscheidung kann der Codegenerator leicht entscheiden, ob für eine Operation spezieller Code (etwa der Aufruf einer Bibliotheksfunktion für Zeichenketten) erfolgen soll, oder ob eine implizite Typumwandlung nötig ist.

Variablen werden im Zwischencodebaum als eine Referenz auf den entsprechenden Eintrag in der Symboltabelle gespeichert. Da die Symboltabelle für jede deklarierte Variable nur einen einzigen Eintrag enthält, sind auch mehrfache Referenzen auf ein und denselben Symboltabelleneintrag möglich. Diese Tatsache wird noch eine wichtige Rolle bei der Elimination vom gemeinsamen Teilausdrücken spielen. Operatoren für Variablen und Konstanten können lediglich an den Blättern eines GAG auftreten, sie sind also immer elementarer Natur. Betrachtet wird im folgenden der GAG für den Ausdruck „(a-b)+c". Dabei sollen sämtliche Variablen vom Typ "Integer" sein.

Bild Seite 68.

Abbildung 20: GAG für den Ausdruck „(a-b)+c" mit Darstellung der Referenzen auf die Symboltabelle

Innere Knoten und die Wurzel dieses GAG sind Operatoren, außer Variablen und Konstanten. Alle Blätter des GAG sind Verweise auf Eintrage in der Symboltabelle. Weitere Informationen über das Datenobjekt (etwa die relative Lage einer lokalen Variablen auf dem Laufzeitstack oder die Größe der Variablen) können vom Codegenerator aus der Symboltabelle entnommen werden. Diese Art der Speicherung verringert in entscheidendem Maße den Speicherbedarf für die interne Darstellung der GAGs, da für Variablen lediglich eine Referenz (bei 32 Bit Maschinen 32 Bits) auf einen

21 Ein CString ist eine an C angelehnter nullterminierter String. Dieser Datentyp wurde neu eingeführt um die Portierung von in C geschriebene Programme zu vereinfachen.

Symboltabelleneintrag gespeichert wird.

Konstanten werden im Zwischencodebaum als eine Referenz auf einen Speicherbereich gespeichert, welcher den Wert der Konstanten enthält. für den Ausdruck "s+'abc'" ergibt sich folgender GAG, falls "s" eine Variable vom Typ "String" ist.

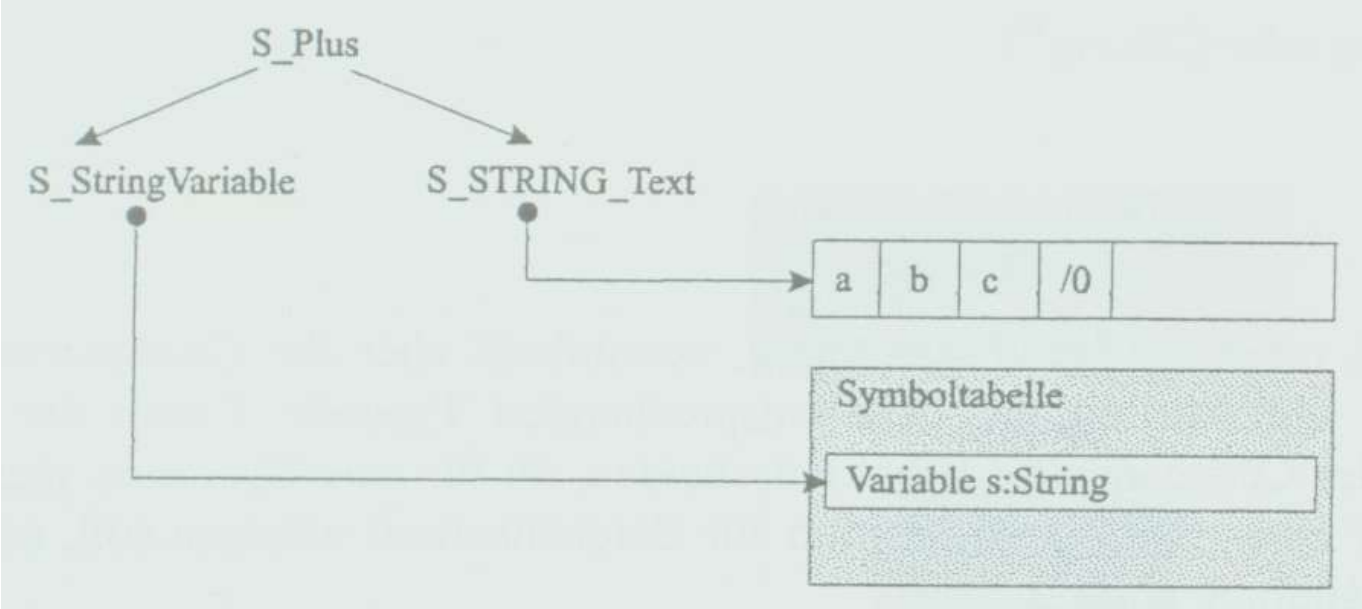


Abbildung 21: GAG für den Ausdruck "s+'abc'" mit Darstellung der Referenzen auf die Symboltabelle und die Stringkonstante (nullterminiert)

7.4.2 Arithmetische Operatoren

Arithmetische Operatoren sind zunächst die vier Grundrechenarten Addition, Subtraktion, Division und Multiplikation. Da viele Rechnerarchitekturen für diese Operationen zwischen vorzeichenbehafteten und vorzeichenlosen ganzzahligen Operanden unterscheiden, werden diese Operationen bereits in der Zwischensprache unterteilt. Dies wäre nicht zwingend notwendig gewesen, entlastet jedoch den Codegenerator, da Typinformationen über Datenobjekte nicht gespeichert und weitergegeben werden müssen. Stattdessen wird nur zwischen den oben aufgeführten Typen unterschieden. Außerdem wird zur Vereinfachung der Codesynthese zwischen Operationen mit reellen und ganzen Zahlen unterschieden. Die Operanden der Operation befinden sich jeweils im linken und rechten Teilbaum des Operatorknotens. Mit "signed" wird eine Operation mit vorzeichenbehafteten Operanden bezeichnet, "unsigned" kennzeichnet eine Operation mit vorzeichenlosen Operanden.

Operator	Bedeutung
S_Plus	ganzzahlige Addition (signed und unsigned)
S_Minus	ganzzahlige Subtraktion (signed und unsigned)
S_IDiv	ganzzahlige Division (signed)
S_UDiv	ganzzahlige Division (unsigned)
S_ITimes	ganzzahlige Multiplikation (signed)
S_UTimes	ganzzahlige Multiplikation (unsigned)
S_FloatPlus	reelle Addition
S_FloatMinus	reelle Subtraktion
S_Divide	reelle Division

S_FloatTimes	reelle Multiplikation

Tabelle 5: Die Zwischencodeoperatoren für arithmetische Operationen

7.4.3 Funktionen

Funktionen werden ebenfalls als Operatoren dargestellt, da sie einen Wert (RValue) zurückliefern. Object-Pascal unterscheidet normale Funktionen, Standardfunktionen, Funktionen welche Objektmethoden sind und Funktionen, welche Klassenmethoden sind.

Operator	Bedeutung
S_ClassFunktion	Aufruf einer KIassenfunktion
S_Funktion	Aufruf einer Funktion
S_ObjFunk'tion	Aufruf einer Objektmethode welche einer Funktion ist
S_StdFunc	Aufruf einer Standardfunk-tion

Tabelle 6: Die Zwischencodeoperatoren für Funktionen

Der GAG für einen Funktionsaufruf enthält in seinem linken Teilbaum einen Verweis auf die Definition der Funktion. Diese Informationen nutzt der Codegenerator zum Beispiel, um zu bestimmen, welche Aufrufart (pascal oder C) zu nutzen ist oder um das Modul, in welchem die Funktion definiert ist, zu ermitteln. Der rechte Teilbaum eines GAGs für einen Funktionsaufruf enthält eine verkettete Liste von GAGs, welche die aktuellen Parameter der Funktion darstellen. Variablenparameter werden mit dem Symbol "S_VAR" bezeichnet, Werteparameter mit dem Symbol "S_CONST'. Das Ende der Parameterliste ist erreicht, wenn der rechte Teilbaum den Wert NIL enthält. Betrachtet wird folgender Funktionsdeklaration:
FUNCTION FuncI (a:Integer; **VAR** b:Integer):Integer;

Für den Aufruf ,,Funcl(1,x)", mit x deklariert als Integer, wird folgender GAG erzeugt:

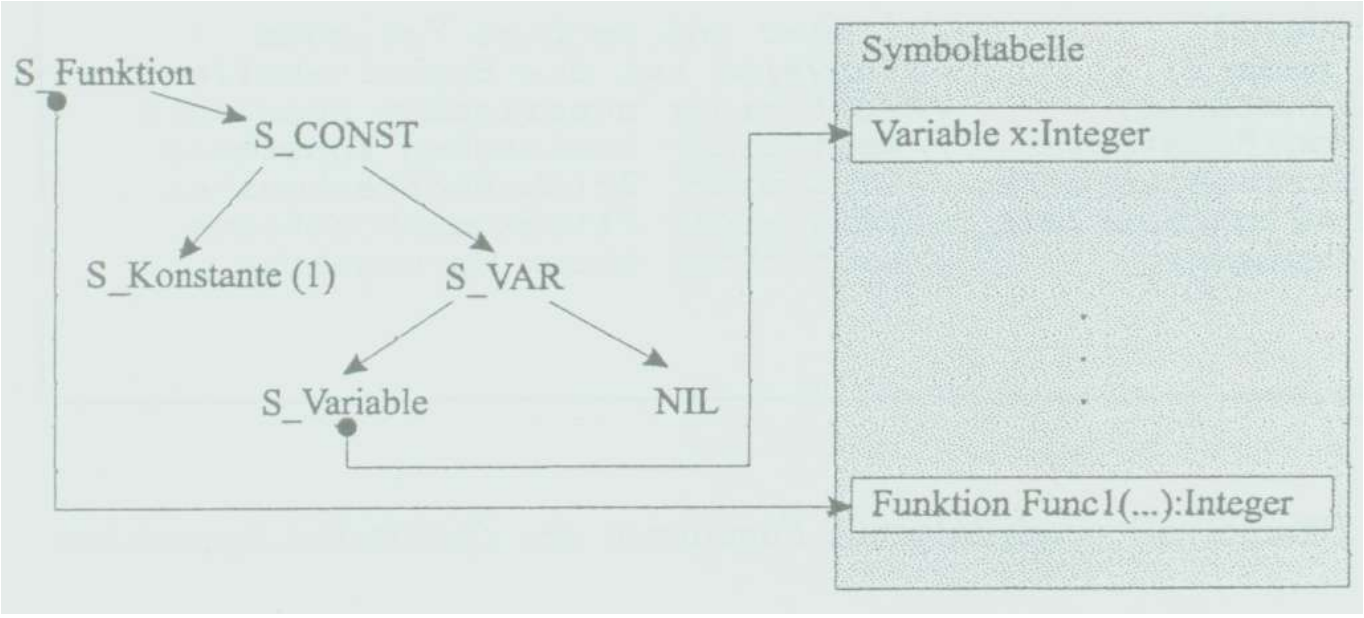


Abbildung 22: Der GAG für den Funktionsaufruf ,,Funcl(l,x)"

Der Rückgabewert der Funktion ist ein RValue vom Typ Integer.

7.5 Zwischensprachenanweisungen

Viele Anweisungen der Quellsprache (etwa eine Zuweisung oder eine Zahlschleife) werden direkt in Anweisungen der Zwischensprache umgesetzt. Zwischensprachenanweisungen haben Steuerungscharakter ähnlich einer Anweisung in Pascal. Zwischensprachenanweisungen werden sequentiell vom Codegenerator in Zielcode umgesetzt, ihre Reihenfolge beschreibt also den Kontrollfluss des Quell- und Zielprogramms. Ein Zwischencodebaum besteht somit aus einer Folge von Zwischensprachenanweisungen. Diese Anweisungen können ihrerseits als Parameter beliebig tief verschachtelte Operatorbäume enthalten. Eine komplette Aufstellung der Anweisungen der Zwischensprache findet sich im Anhang D.

7.6 Entwurf der Zwischencodegenerierung und Optimierung

Zum Aufbau des Zwischencodebaumes werden durch den Parser zunächst Syntaxbäume erstellt. Durch Elimination von gemeinsamen Teilausdrücken generiert der Zwischencodegenerator aus diesen Syntaxbäumen GAGs.

7.6.1 Aufbau des Syntaxbaumes

Syntaxbäume sollen intern von einer Datenstruktur "T_Tree" beschrieben werden. Der Parser benutzt diese Datenstruktur für die hierarchische Analyse des Quellprogramms und zur Konstantenauswertung. Die wichtigsten Felder dieser Datenstruktur sind im folgenden Quelltextausschnitt dargestellt:

```
typedef T_Tree *P_Tree;
typedef struct
  {
     T_Symbol Op;              //Zwischencodeoperator/-anweisung
     P_Typ TypeDesc;           //Verweis auf den Typ des Baumes
     union
       {
          P_Tree Kids[2];          //linker und rechter Teilbaum
          P_Objekt Node[2];        //Verweise auf die Symboltabelle
          longint Konstante;       //Wert einer numerischen Konstante
          boolean BoollKonstante;  //Wert einer booleschen Konstante
          PString StringKonstante; //Wert einer Zeichenkettenkonstante
          extended *FloatKonstane; //Wert einer Fließkonstante
          longint JmpLabel;        //Bezeichner einer Sprungmarke
        } U
  }
} T_Tree;
```

Das Erzeugen dieser Bäume soll durch folgende Funktionen des Zwischencodegenerators realisiert werden:

Funktion	Bedeutung
P_Tree_NewDAGLabel(longint Label);	Anlegen einer Sprungmarke
P_Tree_NewJump(longint Label);	Anleger Sprunganweisung
P_Tree_NewStatemebt(T_Sympol Op, P_Tree Kid1, P_Tree Kid2);	Anlegen einer Anweisung
P_Tree_NewTreeConstant(longint Value);	Anlegen einer Konstante
P_Tree_NewTreeFloatConstant(const extended Value);	Anlegen einer Fließkommakonst.
P_Tree_NewTreeOp(T_Sympol Op, P_Tree Kid1, P_Tree Kid2);	Anlegen einer Operation
P_Tree_NewTreeStringConstant(const string Value)	Anlegen einer Stringkonstante
P_Tree_NewTreeVariable(P_Objekt Node);	Anlegen einer Variable

Tabelle 7: Die Funktionen des Zwischencodegenerators zum Aufbau von Syntaxbäumen

Alle Funktionen liefern einen Zeiger auf eine Datenstruktur vom Typ „T_Tree“ zurück. Der Aufbau des Syntaxbaumes erfolgt nach zusätzlichen Semantikregeln, welche jeder Grammatikregel beigegeben werden. Dies soll an einem Ausschnitt aus der Object-Pascal Grammatik verdeutlicht werden. Hierbei steht NTS „A“ für einen Ausdruck und das NTS „Z“ für eine Zuweisung. Der NTS „V“ verkörpert eine Variable. „Bezeichner“ soll eine Referenz auf einen in der Symboltabelle definierten Bezeichner darstellen. Mit jedem NTS wird ein Wert Typ „P_Tree“ verknüpft. Dies ist in der Tabelle durch „A.Tree“, „V.Tree“ oder „Z.Tree“ dargestellt.

Grammatikregel	Semantikregel zum Aufbau des Syntaxbaunes
Z ::= V ‚:=’ A.	Z.Tree = NewStatement(S_Assign, V.Tree, A.Tree)
A ::= A_1 ‚+’ $A^{2.}$	A.Tree = NewTreeOP(S_Plus, A_1.Tree, A_2.Tree)
A ::= ‚.’ $A_{1.}$	A.Tree = NewTreeOP(S_Negate, A_1.Tree, NIL)
V ::= Bezeichner.	V.Tree = NewTreeVariable(Bezeichner)

Tabelle 8: Beispiele für Semantikregeln zum Erzeugen von Syntaxbäumen.

7.6.2 Erkennen von gemeinsamen Teilausdrücken.

Hat der Parser einen Anweisung vollständig analysiert, so ruft er die Funktion „ListNodes“ auf. Diese Funktion verbindet den übergebenen Baum mit dem vorhandenen Zwischencodebaum und erkennt gemeinsame Teilausdrücke. Dabei wird der durch die Datenstruktur „T_Tree“ beschriebene Systaxbaum in einen GAG transformiert. Die entsprechende Datenstruktur soll „T_DagNode“ lauten. Die wichtigsten Felder dieser Datenstruktur sind im folgenden Quelltextausschnitt dargestellt:

„Listnodes“ durchläuft den übergebenen Zwischencodebaum rekursiv. Gemeinsame Teilausdrücke werden dabei nach folgenden Kriterien erkannt:

Die beiden Syntaxbäume haben identische Operatorsymbole (Feld "Op").

Die beiden Syntaxbäume haben identische linke und rechte Teilbäume bzw. identische Werte fur Konstanten oder Variablen.

Erkennt der Zwischencodegenerator einen solchen gemeinsamen Teilausdruck, so wird die existierende Struktur vom Typ "T_DagNode" benutzt und das Feld "UseCount" um eins erhöht.

```
typedef T_DagNode *P_DagNode;
typedef struct
  {
     T_Symbol Op;                    //Zwischencodeoperator/-anweisung
     P_DagNode Link;                 //Verbindung zum nächsten DAG
     T_Register Register;            //Register für den DAG
     byte UseCount;                  //Anzahl gemeinsamer Teilausdrucke
     byte RegNeed;                   //Registerbedarf für den DAG
     P_DagNode Kids[2];              //linker und rechter Teilbaum
     boolean Available;              //zeigt Gültigkeit für Optimierung an
     union
       {
          P_Objekt Node;             //Verweis auf die Symboltabelle
          longint Konstante;         //Wert einer numerischen Konstante
          boolean BoolKonstante;     //Wert einer booleschen Konstante
          PString StringKonstante;   //Wert einer Zeichenkettenkonstante
          extended *FloatKonstante;  //Wert einer FlieBkommakonstante
          longint JmpLabel;          //Bezeichner einer Sprungmarke
       } U;
  } T_DagNode;
```

Für Syntaxbäume und GAGs gelten folgende Regeln:

- Der Wurzelknoten des Baumes ist immer eine Anweisung der Zwischensprache.
- Blätter des Baumes sind immer Variablen oder Konstanten. Die Darstellung dieser Werte erfolgt in den Feldern ,,Node", ,,Konstante", ,,BoolKonstante", "StringKonstante" und ,,FloatKonstante" der "T_Tree" oder "T_DagNode" Datentypen, wie im Abschnitt 5.4.1 beschrieben.
- Alle anderen Knoten des Baumes werden durch Operatoren der Zwischensprache beschrieben. Die

Operanden für die Operatoren sind im Feld „Kids“ des „T_Tree“ oder „T_DagNode“ Datentypes gespeichert

Als Beispiel wird die Anweisung „e:=(a+b)*(a+b)" betrachtet. Diese Zuweisung enthält den gemeinsamen Teilausdruck „a+b", welcher von beiden Faktoren des Multiplikationsoperators genutzt wird. Der Parser erzeugt zunächst den in Abbildung 23 dargestellten Syntaxbaum. Die Bezeichnungen an den Knoten und Blättern des Baumes entsprechen jeweils dem Wert des Feldes "Op" in der Datenstruktur "T_Tree". Alle Variablen sollen vom Typ "Integer" sein.

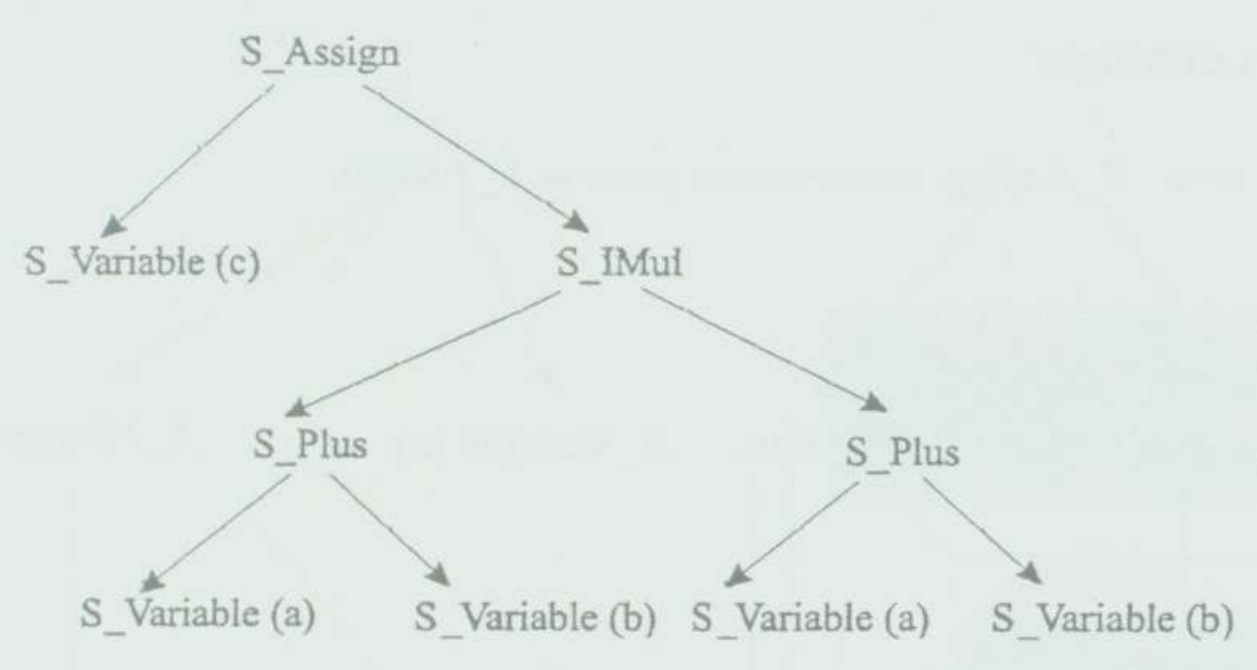


Abbildung 23: Syntaxbaum für die Anweisung "c:=(a+b)*(a+b)"

Aus diesem Syntaxbaum wird durch den Aufruf der Funktion „ListNodes" ein GAG erzeugt. Dafür wird der Baum preorder durchlaufen. Jeder Unterausdruck wird in einem Puffer zwischengespeichert. Bei der Auswertung eines Unterausdruckes wird zunächst Überprüft, ob der entsprechende Ausdruck schon in einem anderen Ausdruck als Teilausdruck vorkommt. Ist dies der Fall, so werden beide Unterausdrücke durch einen gemeinsamen Ausdruck dargestellt - dem gemeinsamen Teilausdruck. Dieses Verfahren setzt sich rekursiv bis zur Wurzel des Baumes fort. Es entsteht der in Abbildung 24 dargestellte GAG.

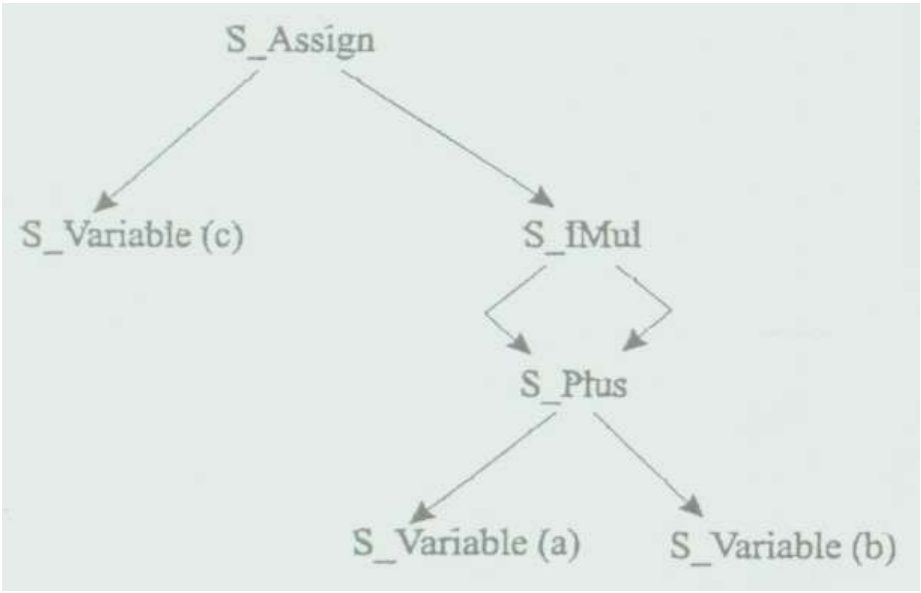


Abbildung 24: GAG für die Anweisung "c:=(a+b)*(a+b)"

Das Erkennen von gemeinsamen Teilausdrücken ist also eine sehr effektive Art der Optimierung. Im obigen Beispiel ist der gemeinsame Teilausdruck natürlich trivial. Es gibt aber durchaus viele Situationen in denen gemeinsame Teilausdrücke nicht sofort erkennbar sind, und eine Optimierung eine beträchtliche Effizienzsteigerung des generierten Codes bewirkt. Das Erkennen von gemeinsamen Teilausdrücken kann außerdem auch über Anweisungsgrenzen hinaus erfolgen. Für das Programm

```
PROGRAM Test;
VAR x,y,z:Integer;
BEGIN
        x:=I;
        y:=x+x;
        z:=y*(x+x)
END.
```

würde folgender Zwischencodebaum erzeugt:

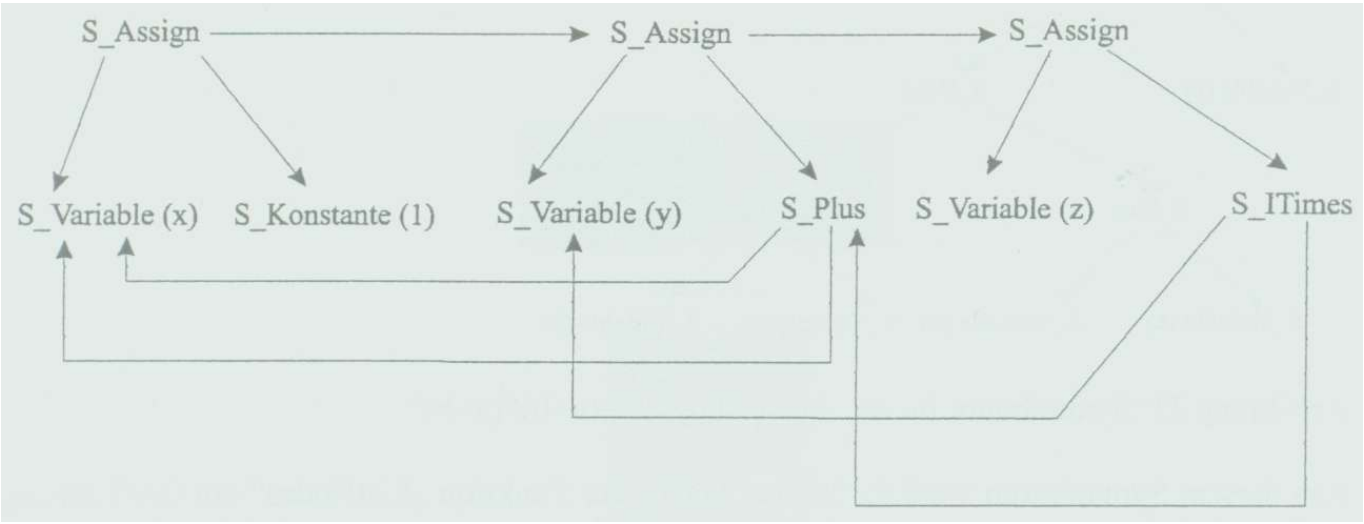


Abbildung 25: Der Zwischencodebaum fur das obige Programm

8 Kapitel 6
Entwurf des Backends

Das Backend eines Compilers generiert den Zielcode aus dem Zwischencodebaum und gibt den Zielcode in eine Datei aus. Im vorliegenden Compiler wurde aus Geschwindigkeitsgründen zusätzlich der Linker in das Baekend übernommen. Wesentliche Komponenten des Baekends können vom vorhandenen Compiler übernommen werden. Dies betrifft speziell den Assembler und den Linker. Der Codegenerator und die Codeausgabe sind die einzigen Teile des Compilers, welche auf die Zielbetriebssysteme Bezug nehmen.

8.1 Eigenschaften der Zielmaschine

Als Zielmaschine werden Prozessoren der INTEL 80X86 Reihe verwendet. Da der hier zu implementierende Compiler ein 32-Bit Compiler sein soll, sind die erzeugten Programme prinzipiell aber erst auf einem INTEL 80386 oder höher ausführbar. Da die zu unterstützenden Betriebssysteme mindestens einen INTEL 80386 voraussetzen, wird der Maschinencode auf diesen Prozessor ausgerichtet. Optimierungen für den INTEL 80486 oder Pentium sind prinzipiell möglich, werden jedoch nicht angewandt, um die Komplexität des Codegenerators gering zu halten.

INTEL 80X86 Prozessoren sind Prozessoren nach dem CISC Prinzip, das bedeutet, es existiert eine Vielzahl von Befehlen und Adressierungsarten. Für den Compilerbauer bedeutet dies eine gewisse Vereinfachung des Codegenerators, da komplexe Operationen (wie zum Beispiel die Berechnung eines Array-Indizes) nicht selten mit einem einzigen Maschinenbefehl codiert werden können. Darüber hinaus existieren jedoch eine Reihe von Schwierigkeiten, welche die Codegenerierung allgemein und die Registerverwaltung im speziellen relativ komplex machen:

- Es existiert nur ein Registersatz mit 8 allgemeinen Registern (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP). Die Register ESP und EBP sind zudem für den Zugriff auf den Laufzeitstack reserviert - ESP speichert den aktuellen Stackpointer und EBP den Base Pointer für das aktuelle Unterprogramm - und können für allgemeine Operationen nicht genutzt werden.

- Für Operationen mit Byte- und Wortoperanden k6nnen die Register EAX, EBX, ECX und EDX noch einmal unterteilt werden (AL .. DL und AH .. DH sowie AX .. DX).

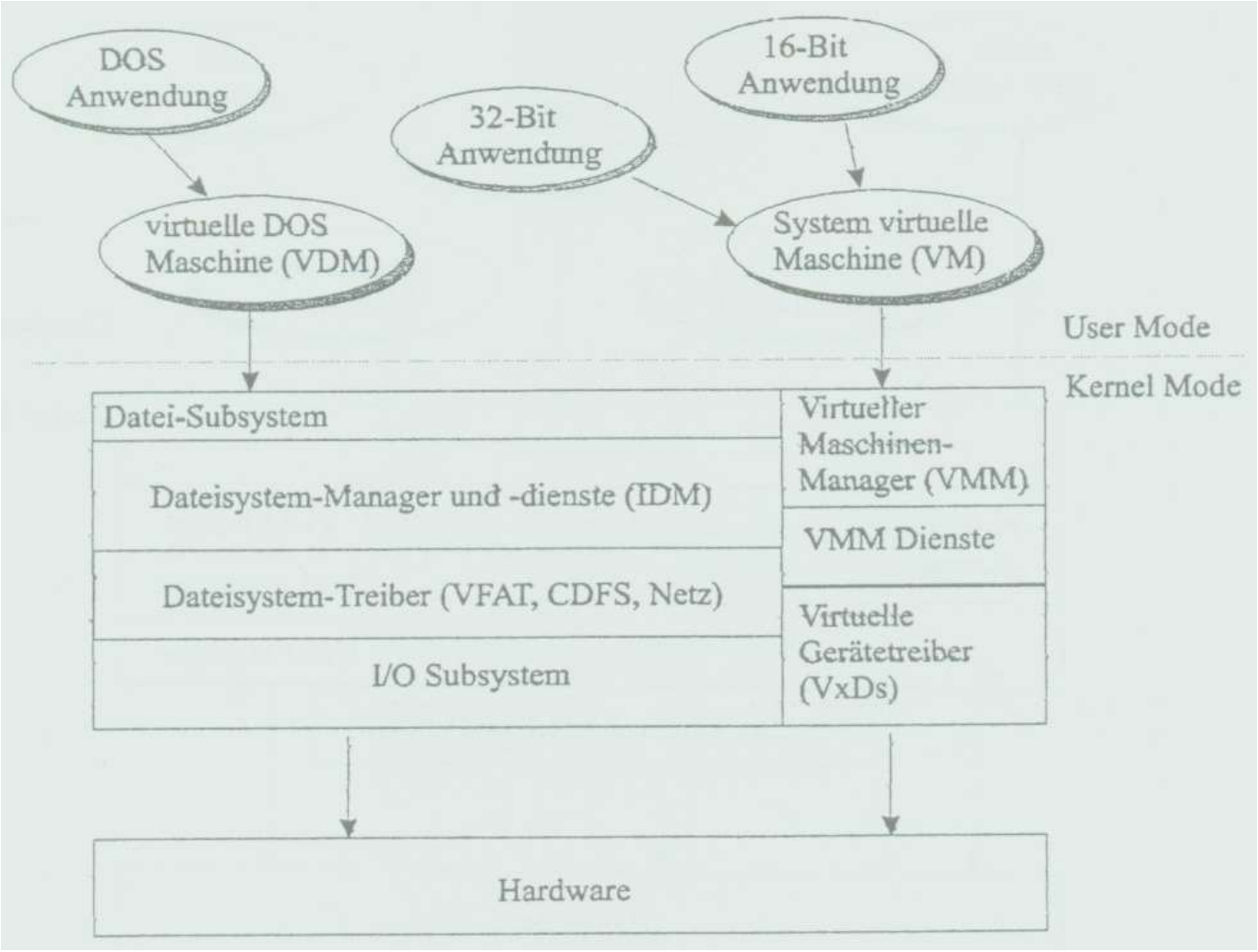
- Intel-Prozessoren bieten die Möglichkeit, durch eine günstige Anordnung der Befehle eine Effizienzsteigerung zu erreichen (Pipelining). Um den Codegenerator einfach zu halten, wird auf die Anwendung dieser Regeln verzichtet.

Voraussetzung für die Erzeugung eines abarbeitungsfähigen Maschinencodes ist eine genaue Dokumentation der einzelnen Befehlscodierungen der Zielmaschine. Als Literatur wird hierfür zum Beispiel [Thi92] verwendet. Da der zu implementierende Compiler direkten Maschinencode erzeugen soll, ist ein Assembler zu entwerfen, welcher vom Codegenerator an geeigneten Stellen aufgerufen wird. In der Vorversion des Compilers wurde ein solcher Assembler bereits implementiert. Die Quelltexte des alten Compilers werden also in Bezug auf den Assembler weitestgehend übernommen.

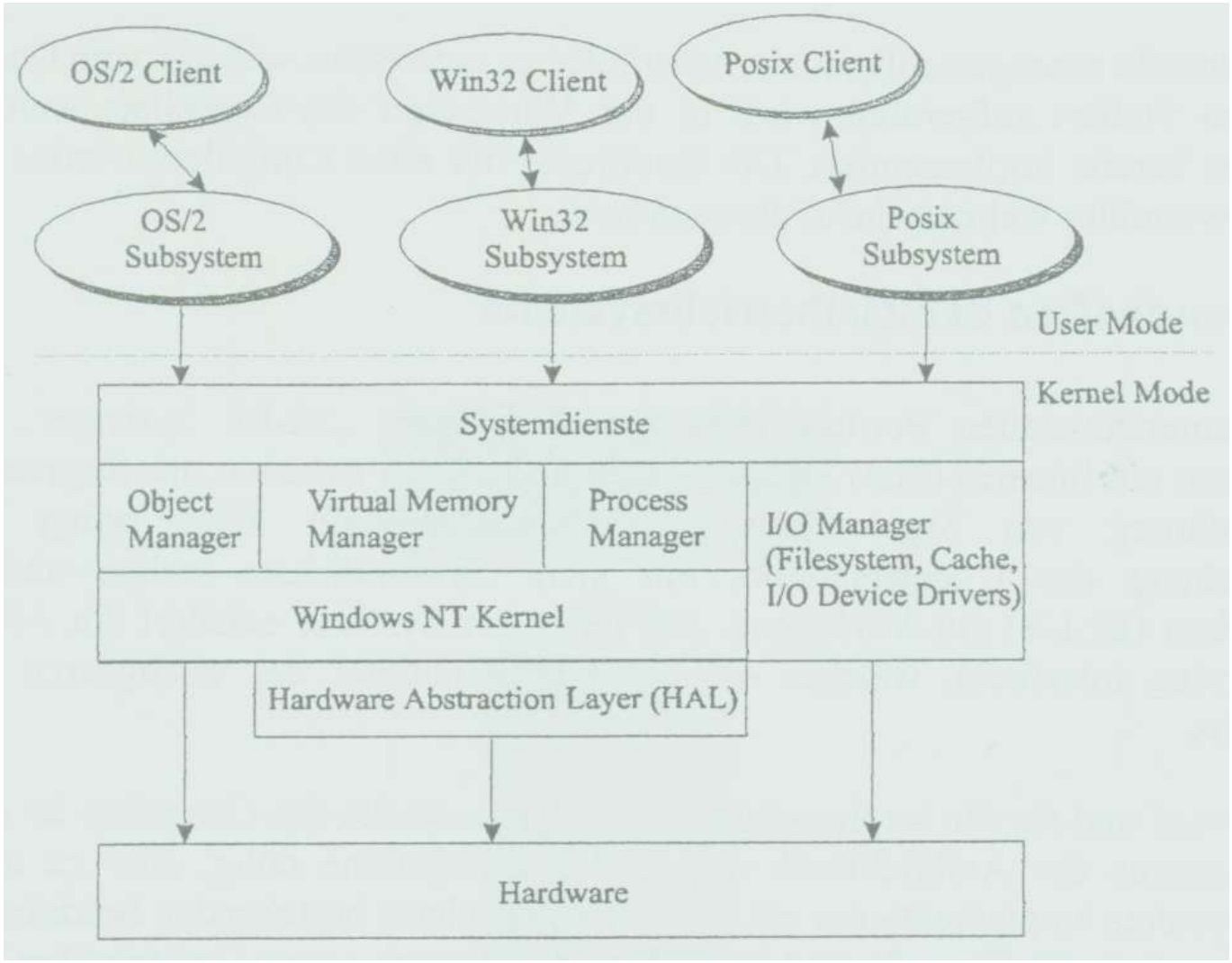
8.2 Eigenschaften der Zielbetriebssysteme

Die zu unterstützenden Betriebssysteme sind allesamt „32-Bit Systeme". Alle Systeme unterstützen ein lineares (Flat) Hauptspeichermodell, ein Arbeiten mit Segmentregistern und die Beachtung von Segmentgrenzen entfallen deshalb. Dies bringt eine gewisse Vereinfachung des Codegenerators mit sich. Systemdienste stehen über dynamische Bibliotheken (DLL's) zur Verfügung. Für jedes der Systeme existiert ein API (Application Programming Interface), welches die Art und Parameter der verfügbaren Systemdienste spezifiziert.

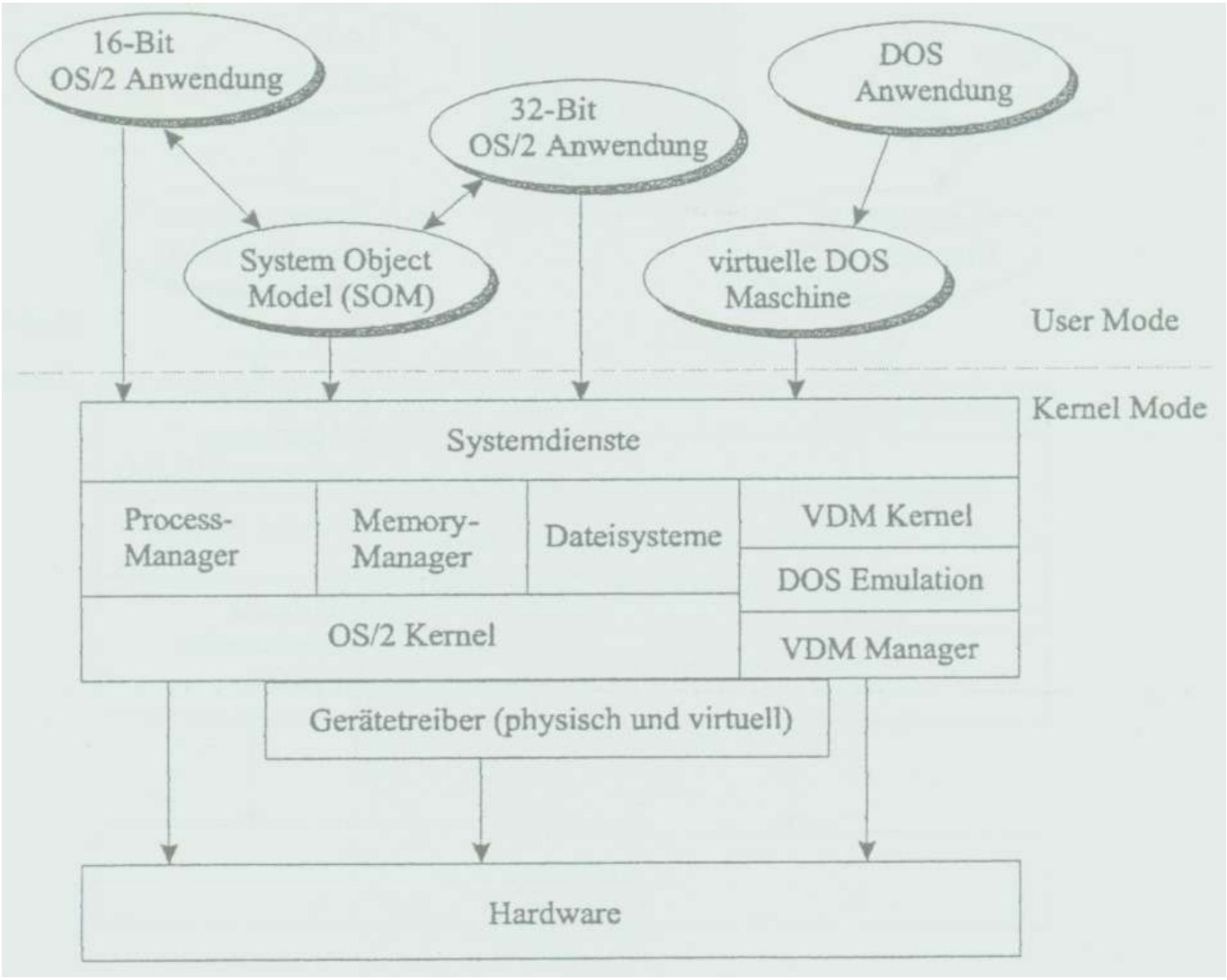
Zum Entwurf und für die Implementation der Syntheseteile des Compilers ist zumindest eine grobe Kenntnis der Architekturen der Zielbetriebssysteme nötig. Alle zu unterstützenden Betriebssysteme implementieren ein aus zwei Schichten bestehendes Betriebssystemmodell. In der Kern-Schicht (Kernel) sind Betriebssystemdienste sowie Gerätetreiber enthalten. Der Zugriff auf Betriebssystemdienste erfolgt ausschließlich über definierte Funktionen. Hardwareseitig werden diese 2 Schichten für Intel-Prozessoren direkt auf das Schutzkonzept des Prozessors abgebildet. Dabei erhält die Kern-Schicht die höchste Privilegebene (maximale Abschirmung vor Anwendungen). Die vereinfachte Architektur der Zielbetriebssysteme ist in den folgenden Abbildungen dargestellt:



Windows 95: Viele Betriebssystemdienste arbeiten in der User-Schicht und sind damit relativ ungeschützt. 32-Bit Applikationen arbeiten in einem eigenen Adressraum, l6-Bit Applikationen teilen sich einen Adressraum.



Windows NT: Betriebssystemdienste sind vollständig von Applikationen isoliert und arbeiten in der Kemel-Schicht. Ein Aufruf eines Dienstes erfolgt über Call-Gates, dies bedeutet maximale Abschirmung des Betriebssystemkerns, impliziert jedoch zeitaufwendige Privilegwechsel. Applikationen laufen in der User-Schicht. 32 Bit und 16 Bit Applikationen haben jeweils eigene Adressräume.



OS/2 Warp: Das Basissystem ist vollständig von Applikationen isoliert, einige Systemdienste arbeiten jedoch in der User-Schicht und sind deshalb weniger geschützt. 32 Bit und 16 Bit Applikationen haben jewei1s eigene Adressräume.

Für den Compilerbauer ergeben sich hieraus folgende Konsequenzen:

- Da alle Systeme im geschützten (protected) Modus des Prozessors arbeiten, sind alle 32 Bit Applikationen vollständig von einander isoliert. Jeder Prozess besitzt seinen eigenen Adressraum. Verletzungen des Schutzkonzeptes werden vom Prozessor erkannt und vom Betriebssystem bearbeitet.

- Betriebssystemdienste stehen ausschließlich über dynamische Bibliotheken zur Verfügung. Der Compiler muss Code zum Einbinden und Erzeugen von DLL's bereitstellen. Das Laufzeitsystem stellt dem Nutzer eine definierte Schnittstelle zum Betriebssystem bereit und nutzt die Dienste des Systems.

- Direkte Zugriffe auf die Hardware des Rechners sind nicht zulässig. Alle Operationen müssen mit Hilfe von Systemdiensten erfolgen.

8.3 Entwurf des Codegenerators

Aufgabe des Codegenerators ist es, aus dem vom Zwischencodegenerator erzeugten Zwischencodebaum den Zielcode zu erzeugen. Der Codegenerator ist aus diesem Grunde stark von der Zielmaschine abhängig.

Neben der Erzeugung von Zielcode können im Codegenerator zusatzlich maschinenspezifische Optimierungen durchgeführt werden. Der Codegenerator arbeitet eng mit dem Assembler zusammen, welcher die vom Codegenerator ausgewählten Assembleranweisungen in Maschinencode übersetzt. Die logische Grundstruktur des Codegenerators ist in Abbildung 26 dargestellt.

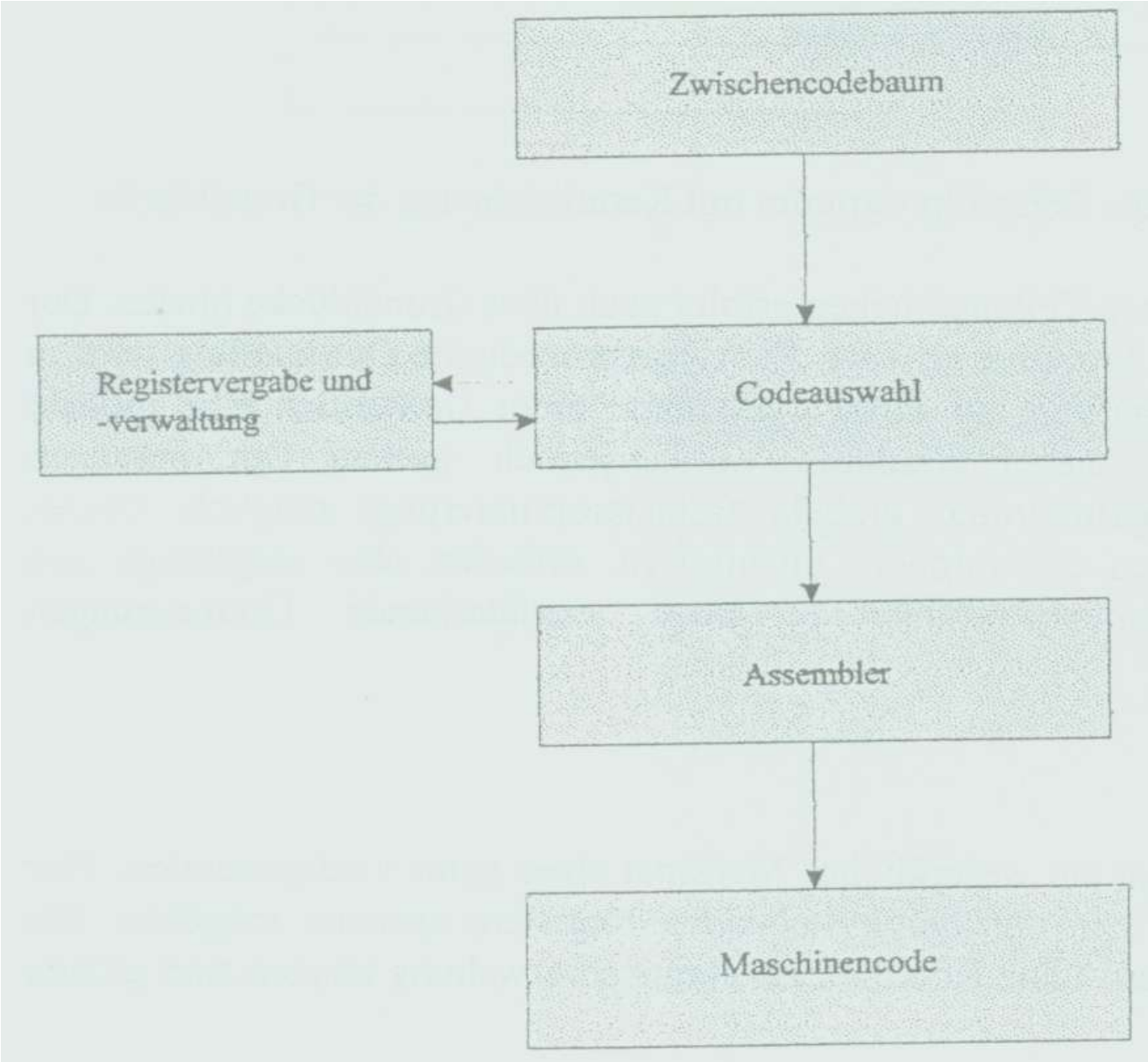


Abbildung 26: Die logeische Grundstruktur des Codegenerators

8.3.1 Erzeugung von Zielcode aus Grundblöcken

Der Zwischencodebaum besteht aus Grundblöcken, welche in ihrer sequentiellen Abfolge die Logik des Programms definieren. Beispielsweise hat das Programm

PROGRAM Test;
VAR x,y:lnteger;
BEGIN

```
    x:=I;
    y:=x+I;
    WriteLn(x+y);
END.
```

drei Grundblöcke - zwei Zuweisungen und einen Prozeduraufruf. Der Zielcode wird für jeden Grundblock getrennt erzeugt. Dabei bleibt die Ausführungsreihenfolge der Grundblöcke erhalten, um die Logik des Programms nicht zu verändem. Der GAG für das obige Programm ist in Abbildung 26 dargestellt.

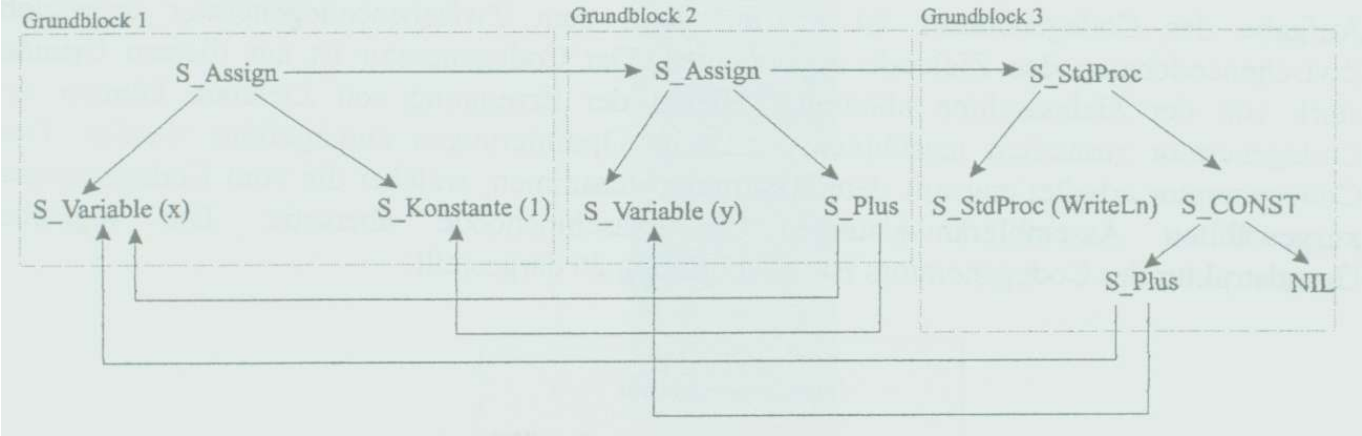


Abbildung 26: GAG fur das obige Beispielprogramm mit Kennzeichnung der Grundb16cke

Die Erkennung von gemeinsamen Teilausdrücken erfolgt auch über Grundblocke hinaus. Der Zielcode wird separat für jede Prozedur erzeugt. Eine interprozedurale Optimierung wird in diesem Compiler nicht durchgeführt, da diese Maßnahme unter Umständen eine globale Datenflussanalyse erforderlich macht. Dadurch waren jedoch weitere Optimierungen (Registervariablen, Schleifenoptimierung, globale Registeroptimierung) möglich. Dieses Vorgehen ist in professionellen C-Compilern anzutreffen, erfordert aber sorgfältige und langwierige Tests, um semantikverändemde oder -verfälschende Optimierungen auszuschließen.

8.3.2 Registervergabe

Eine gute Registerverwaltung ist ein wesentliches Merkmal eines guten Codegenerators. Der Codegenerator sollte mit den zur Verfügung stehenden Registern sparsam umgehen. Ein Registermangel ist zu venneiden[22] Die Routinen zur Registerverwaltung können zum großen Teil aus dem vorhandenen Compiler Übernommen werden. Der Codegenerator verwaltet Register nach den folgenden Kriterien:

- Die Operanden einer Operation werden zum spätest möglichen Zeitpunkt in ein Register geladen, um Registermangel zu vermeiden.

- Funktionsaufrufe innerhalb eines Ausdrucks erfordern eventuell ein Retten der zu diesem Zeitpunkt benötigten Register. Diese Register werden auf dem Laufzeitstack zwischengespeichert.

- Ergebnisse von Ausdrücken, welche gemeinsame Teilausdrücke repräsentieren, werden solange wie möglich in einem Register gehalten. Ist dies aus Registermangel nicht mehr möglich, so muss der gemeinsame Teilausdruck erneut berechnet werden. Dies gilt auch über die Grenzen von

22 U**nter einem Registermangel soll ein Zustand des Codegenerators verstanden werden, in dem keine weiteren** CPU-Register mehr zur Speicherung von Werten zur Verfugung stehen, da alle Register schon belegt sind.

Grundblöcken hinaus.

- Auf die Nutzung von Registervariablen wird aufgrund der begrenzten Registeranzahl der INTEL Prozessoren verzichtet. Dies würde die Anzahl der für gemeinsame Teilausdrücke zur Verfügung stehenden Register stark einschränken und kann eventuell kontraproduktiv wirken. Ausserdem würde diese Massnahme unter Umständen eine prozedurübergreifende Datenflussanalyse erfordern.

Betrachtet wird der Programmausschnitt:

```
PROGRAM Test;

TYPE PRecord=ATRecord;
     TRecord=RECORD
                a.b:Longlnt;
     End;

VAR a:ARRAY[O .. 20.0 .. 20] OF PRecord;

PROCEDURE Swap(i:Longlnt);

VAR Temp: longlnt;

BEGIN
  Temp:=a[i.i]^.a;
  a[i,i]^.a:=a[i.i+l]^.b
  a[i.i+l]^.b:=Temp;
End;
```

Die Prozedur Swap hat drei Grundblöcke und verschiedene gemeinsame Teilausdrücke bei der Berechnung der Offsets der Feldelemente. Der generierte Assemblercode ist in der folgenden Tabelle für die einzelnen Grundblöcke dargestellt:

Grundblock 1 Temp:=a[i,i]^.a	Grundblock 2 a[i,i]^.a:=a[i,i+1]^.b	Grundblock 3 a[i,i+1]^.b:=Temp
MOV EAX,i	MOV ECX;I	MOV EAX,Temp
IMUL EAX,4	ADD ECX,1	MOV [ECX+4],EAX (*)
MOV ECX,i	IMUL ECX,4	
IMUL ECX,84	MOV EDX,i	
LEA ECX,A[ECX+EAX]	IMUL EDX,84	
MOV EAX,[ECX] (*)	LEA EDX,A[EDX,ECX]	
MOV ECX,[EAX] (*)	MOV ECX, [EDX] (*)	
MOV Temp,ECX	MOV EDX,[ECX+4] (*)	
	MOV [EAX],EDX (*)	

Tabelle 9: Der Assemblercode für die einzelnen Grundblöcke des Beispielprogramms

Wie aus der Tabelle ersichtlich ist, werden gemeinsame Teilausdrücke in den Registern EAX und ECX gehalten. Diese gemeinsamen Teilausdrücke sind in der Tabelle mit einem Stern gekennzeichnet. Der generierte Assemblercode ist nicht optimal und könnte weiter verbessert werden. Es ist jedoch klar ersichtlich, dass der generierte Zielcode bedeutend kleiner und schneller ist, als ohne die Erkennung von gemeinsamen Teilausdrücken. Bei der Codeauswahl werden die Ausführungszeiten der verschiedenen

CPU-BefehIe berücksichtigt. So erfolgt zum Beispiel das Laden eines Registers mit dem Wert Null durch eine XOR Operation dieses Registers mit sich selbst. Einige Multiplikations- und Diyisionsbefehle (etwa mit Zweierpotenzen) werden durch bitweises Verschieben realisiert.

8.3.3 Assemblierung

Die Erzeugung von Maschinencode aus dem Zwischencodebaum erfolgt m mehreren Schritten:

1. Durchlaufen des Zwischencodebaumes durch den Codegenerator in einer geeigneten Reihenfolge,

2. Auswahl der für Zwischencodeanweisungen oder -operatoren sinnvollen Assembleranweisungen oder -anweisungsfolgen durch den Codegenerator,

3. Aufruf von Prozeduren des Assemblers zur Generierung des Maschinencodes. Die Parameter dieser Funktionen repräsentieren die vom Codegenerator generierten Darstellungen der Objekte des Zwischencodebaumes auf Assemblerniveau.

Die Prozeduren und Funktionen des Assemblers werden zum großen Teil aus dem vorhandenen Compiler übernommen. Es wird einen einheitliche Schnittstelle von Funktionen zum Zugriff auf den Assembler bereitgestellt. Diese Funktionen werden an geeigneter Stelle vom Codegenerator zur Erzeugung von Maschinencode aufgerufen. Zusätzlich kann der Assembler den erzeugten Code als Assemblerlisting in eine Datei ausgeben. Dadurch kann die Ausgabe des Compilers leicht überprüft werden. Der erzeugte Maschinencode ist zunächst noch verschieblich und kann auch noch Referenzen auf bisher undefinierte Objekte enthalten. Diese Referenzen werden durch den Linker aufgelöst.

Die in der folgenden Tabelle dargestellten Funktionen werden vom Codegenerator zur Assemblierung aufgerufen. Für jede Prozedur ist zusätzlich ein Beispiel angegeben. Die Definitionen sind in der Datei ,, EMIT.H" enthalten.

Funktion	Bedeutung	Beispiel
EmitCALL I	Generierung eines Unterprogrammaufrufes	CALLN32 Test
EmitCALLDLL I	Generierung eines DLL-Aufrufes nach Index	CALLDLL PmWin,320
EmitCALLDLL N	Generierung eines DLL-Aufrufes nach Name	CALLDLL MyDII,'Test'
EmitJMP	Generierung einer Sprungoperation	JMP Label!
EmitLabel	Generierunl! einer Sprungmarke	Labell:
EmitM	Generierung von Code mit einem Speicheroperanden	POP [EDI]
EmitMI	Generierung von Code mit einem Speicher- und einem konstanten Operanden	MOV BYTE PTR [EDI],!
EmitMR	Generierung von Code mit einem Speicher- und einem Regjsterooeranden	MOV [EDij,EAX
EmitRI	Generierung von Code mit einem Register- und einem konstanten Operanden	MOV EAX,IO
EmitRM	Generierung von Code mit einem Register- und einem Soeicheroperanden	MOV EAX,[EDI+ I 0]
EmitRR	Generierung von Code mit zwei Registeroperanden	MOVEAX,EBX
EmitS	Generierung von direktem Code	CLD

Tabelle 10: Die Funktionen zur Assemblierung des Zielcodes

8.3.4 Maschinenabhängige Optimierungen

Zusätzlich zur Generierung des Zielcodes können im Backend maschinenabhängige Optimierungen des Zielcodes durchgeführt werden. In diese Optimierungskategorie fallen Transformationen des Zielcodes, welche nur für eine bestimmte Zielmaschine korrekt sind. Diese Optimierung wird oft auch als *Nachoptimierung* bezeichnet. Meist wird die Nachoptimierung so realisiert, dass ein zusätzlicher Filter zwischen Codeauswahl und Assembler installiert wird. Dieser Filter puffert eine begrenzte Anzahl von Assemblerinstruktionen, meist zwei oder drei. Deshalb wird diese Optimierungsart im englischen mit "peephole optimization" bezeichnet, was im deutschen etwa mit "Guckloch-Optimierung" übersetzt werden könnte. Es wird eine begrenzte Anzahl von Instruktionen, ein "Guckloch", betrachtet, auf denen Optimierungen ausgeführt werden. Für diese Optimierungen bietet sich natürlich ein weit engerer Spielraum an als auf der Ebene der Zwischensprache. Ein typisches Beispiel sind aufeinander folgende redundante Lade- und Speicheroperationen. Der Codegenerator erzeugt auf der Ebene eines Grundblockes relativ gut optimierten Code. An den Grenzen der Grundblöcke werden jedoch eventuell redundante Anweisungen generiert. Für die folgenden zwei Anweisungen ist auf der rechten Seite der Tabelle der Assemblercode angegeben, der ohne Nachoptimierung erzeugt werden würde:

Anweisung	Assemblercode
v:=w+x;	MOY EAX,w ADD EAX,x MOY v,EAX
y:=v+z;	MOY EAX,v (*) ADD EAX,z MOY v,EAX

Tabelle 11: Zwei Pascal Anweisungen mit generiertem Zielcode

Die mit einem Stern gekennzeichnete Assembleranweisung ist redundant und wird vom Nachoptimierer entfernt.

8.4 Entwurf der Codeausgabe

Im Rahmen der Codeausgabe wird das Ergebnis der Compilierung in eine externe Datei ausgegeben. Dabei werden eventuell noch bestehende Vorwärtsreferenzen aufgelöst und der Bezug zu externen Modulen hergestellt. Der Compiler kann drei mögliche Ausgabeformate erzeugen:

1. Eine vorcompilierte Unit (Dateiextension .SPU)

2. Eine direkt ausführbare Datei (Dateiextension .EXE)

3. Eine dynamische Bibliothek (Dateiextension .DLL)

Da die Codeausgabe in dem hier vorgestellten Compiler die letzte Stufe des Compilers ist, beeinflussen die Entwurfsentscheidungen hier andere Komponenten des Compilers nicht mehr in gravierender Weise

8.4.1 Codeausgabe für eine Unit

Die Codeausgabe in eine Unit erfolgt in einem eigenem Format (Extension .SPU). Dieses Format ist nicht kompatibel zu den Standard-Linkformaten (Extension .OBJ) auf den jeweiligen Betriebssystemen. Die Entscheidung für ein proprietäres Format wurde vor allem durch die Widerverwendbarkeit vorhandener Routinen aus dem existierenden Compiler beeinflusst. Darüber hinaus gibt es einige weitere Gründe für diese Entscheidung:

1. Die Unit-Dateien enthalten neben dem verschiebbaren Maschinencode auch Informationen, welche für den Compiler relevant sind. Dies sind zum Beispiel Typdeklarationen oder Funktionsprototypen. Die Standard-Linkformate bieten in der Regel keine Möglichkeit für eine Einbeziehung derartiger Informationen. In C oder C++ wird das Problem durch Header-Dateien gelöst, in Modula 2 durch die Trennung in Definitions- und Implementationsmodule. Turbo-Pascal definiert Units jedoch als eigenständige Einheiten, welche auch ohne Vorhandensein eines Quelltextes (d.h. ohne eine erneute syntaktische Prüfung) genutzt werden können. In Turbo-Pascal haben diese Dateien die Extension. TPU und enthalten al1e relevanten Informationen für die Nutzung der Unit. Dieses Prinzip wurde von Turbo-Pascal übernommen.

2. Die Definitionen der Standard-Linkformate unterscheiden sich für die zu unterstützenden Betriebssysteme. Microsoft definiert für Win32 ein an das Unix COFF (Common Object File Format) angelehntes Format, wahrend IBM für OS/2 ein 0MF (Object Module Format) definiert. Eine Unterstützung beider Formate hatte eine Verdopplung des Aufwandes für die Codeausgabe bedeutet. Außerdem hat sich das vorhandene SPU-Format (Speed-Pascal-Unit) in der Praxis bewährt und brauchte nicht mehr getestet zu werden.

3. Eine Umsetzung der Codeausgabe für andere Betriebsysteme wird erleichtert, da das Unit-Ausgabeformat unverändert weiterverwendet werden kann. Lediglich die Codeausgabe für ausführbare Dateien war zu modifizieren.

Die Erzeugung von Standard-Linkformaten erscheint trotzdem als eine mögliche sinnvolle Erweiterung oder Verbesserung des Compilers, da dies eine Reihe von Vorteilen mit sich bringt:

- Es können Module von Drittanbietern in das Pascal Programm eingebunden werden. Im hier vorgestel1tem Compiler ist dies nur über den Umweg der Definition einer dynamischen Bibliothek möglich.

- Die generierten Units können von anderen Compilern eingebunden werden. Dies erhöht die Widerverwendbarkeit der Module. Im entworfenen Design sind die generierten Units nur durch diesen Compiler nutzbar.

- Es kann der vom jeweiligen Betriebssystem bereitgestellte Linker zur Programmverbindung genutzt werden. für den hier entworfenen Compiler musste statt dessen ein eigener Linker eingesetzt werden[23], welcher aber in der Vorversion des Compilers bereits implementiert wurde und hier genutzt werden konnte.

---

23 Dies kann ein Vor· oder Nachteil sein, da integrierte Linker oft schneller sind als externe. Andererseits sind externe Linker natürlich besser in das System integriert und stehen in der Regel auf jedem System zur Verfügung.

Es ist zu beachten, dass die Erzeugung von Standard-Linkformaten eine Unterteilung der Unit-Datei in zwei getrennte Dateien (eine Objektdatei und eine Definitionsdatei) notwendig macht. Auf diese Möglichkeit wurde in dieser Arbeit aus Zeitgründen verzichtet, obwohl eine Generierung vom Standard-Linkformaten aus obigen Gründen durchaus sinnvoll erscheint. Eine Umstellung würde vor allem die Routinen, welche Units schreiben und einlesen, beeinflussen. Prinzipiell ist dies realisierbar.

8.4.2 Codeausgabe für ein Programm oder eine dynamische Bibliothek

Windows und OS/2 verwenden für ausführbare Dateien zueinander inkompatible Dateiformate. Unter Windows 95 und Windows NT wird das PE-Format (Portable Executable File Format) verwendet, unter OS/2 das LX-Format (Linear Excutable File Format). Beide Formate sind ähnlich aufgebaut. Die Generierung einer Unit oder einer ausführbaren Datei bildet in dem hier vorgestellten Compiler den letzten Schritt des Übersetzungsvorganges. Eine solche Datei kann direkt unter dem jeweiligen Betriebssystem ausgeführt werden. Eine vereinfachte Darstellung des Aufbaus der beiden Formate ist in Abbildung 27 enthalten. Für eine genaue Beschreibung der einzelnen Teile und Sektionen wird auf die entsprechende Microsoft ([Mic94]) und IBM ([IBM94]) Dokumentation im Anhang verwiesen.

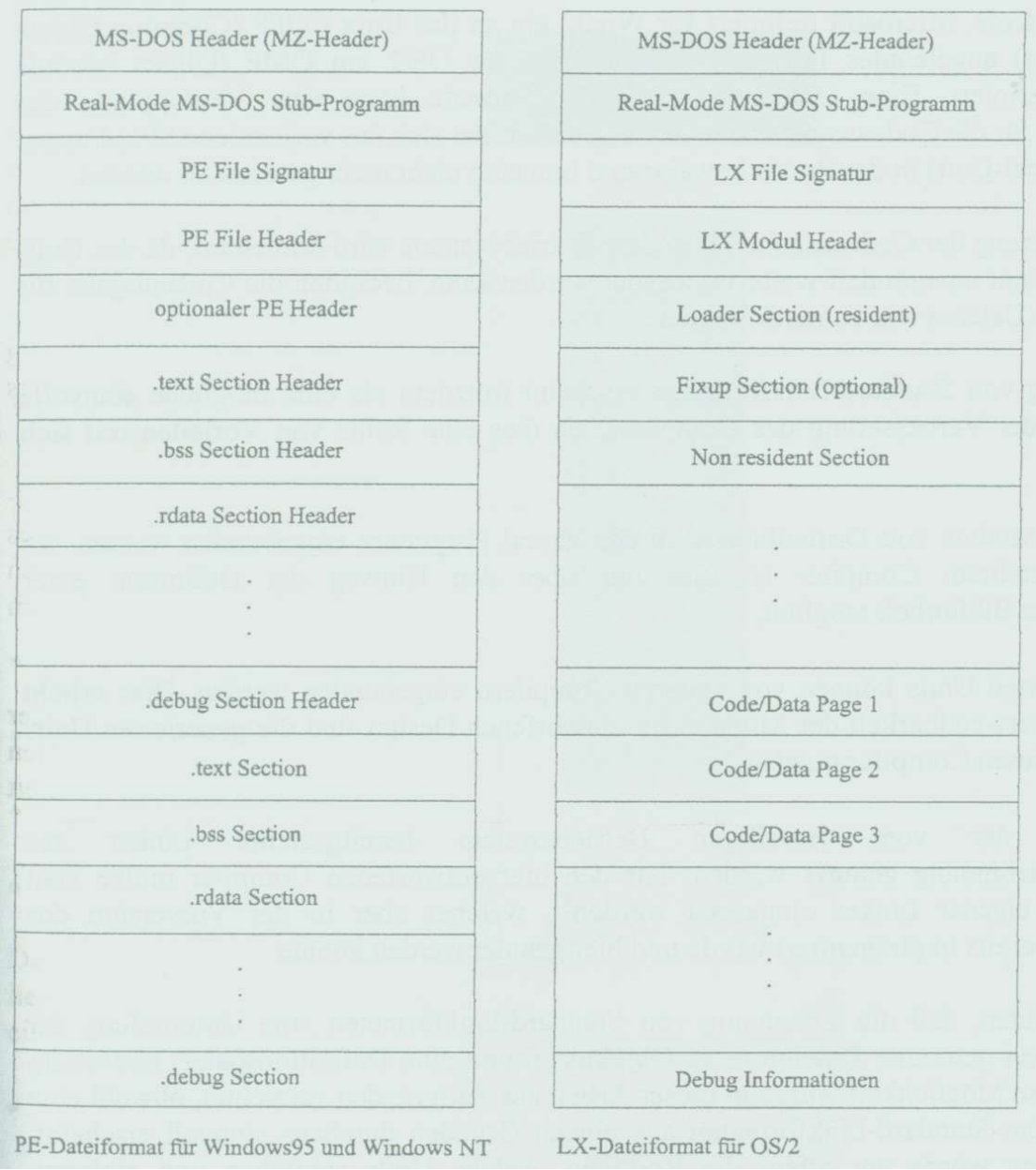


Abbildung 27: Die Dateiformate für ausführbare Dateien unter Win32 und OS/2

Beide Formate beginnen aus Kompatibilitätsgründen zu MS-DOS mit einem Real-Mode Stub. Es wird eine Meldung ausgegeben, falls versucht wird, das Programm unter MS-DOS zu starten (z.B. "Dieses Programm erfordert Microsoft Windows"). Nach dem Stub folgt die Kennung des Dateiformats in Form von 2 Bytes (entweder ,,PE" oder "LX"). Der nachfolgende Header gibt unter anderem Auskunft, ob es sich bei der Datei um eine Programmdatei oder eine dynamische Bibliothek handelt. Der Aufruf von Betriebssystemdiensten erfolgt über dynamische Bibliotheken. Die Schnittstelle zum Betriebssystem erfolgt über Funktionsnamen oder -nummern in diese Bibliotheken. Die Auflösung (Relokation) der Aufrufadressen erfolgt zur Laufzeit des Programms durch den so genannten Lader. Insgesamt ist das PE-Format etwas klarer aufgebaut und erscheint leistungsfähiger. Das IBM LX-Format ist speziell auf Intel Prozessoren ausgerichtet, was sich besonders durch die Anordnung des Codes in Form von festen Seiten (4 KB) in der Datei bemerkbar macht. Aus diesem Grund wird bei OS/2 für den Power PC ein anderes Format (ELF) verwendet.

8.4.3 Linken externer Module

Das Linken externer Module wird mit in das Backend integriert. Dabei werden undefinierte Referenzen auf externe Objekte durch den Linker aufgelost. *A*ls Alternative wäre es durchaus möglich gewesen, einen externen Linker (bei den unterstützten Betriebsystemen gehört ein Linker zum Lieferumfang) zu verwenden. Aus folgenden Gründen wurde auf diese Möglichkeit verzichtet:

1. In der Vorversion des Compilers stand ein Linker zur Verfügung. Durch die Beibehaltung des Unit-Formates der Vorversion konnten auch grosse Teile dieses Linkers wieder verwendet werden. Dieser Linker hat sich in der Praxis bewährt.

2. Das erzeugte Unit-Format ist kein auf den Betriebssystemen gültiges Standard-Linkformat. Eine Umstellung auf einen externen Linker würde die Modifizierung dieses Formates erfordern, da externe Linker immer das Format auf dem jeweiligen Betriebssystem erfordern. Die Gründe für die Verwendung eines eigenen Formates sind im Abschnitt 6.4.1 dargestellt.

3. Ein interner Linker arbeitet generell schneller als ein externer, da sich die vom Linker benötigten Informationen schon im Speicher befinden und nicht neu geladen werden müssen. Aus der allgemeinen Forderung nach Maximierung der Compiliergeschwindigkeit kommt nur ein schneller interner Linker in Frage.

9 Kapitel 7
Implementation des Compilers

In der Implementation des Compilers werden die im Entwurf festgelegten Konzepte umgesetzt. Einige Teile des neuen Compilers werden vom vorhandenen Compiler übernommen. Dies betrifft insbesondere:

- einige Hilfsprozeduren für den Scanner,
- die Prozeduren des Parsers,
- große Teile des Assemblers und des Linkers,
- die Codeausgabe.

Da der vorhandene Compiler in Pascal implementiert war, müssen die vorhandenen Quelltexte nach C++ umgesetzt werden. In die Parserprozeduren muss der Aufbau des Syntaxbaumes, welcher im alten Compiler nicht erzeugt wurde, eingebaut werden. Komplett neu implementiert werden:

- der Scanner,
- die Symboltabellenverwaltung,
- die Zwischencodegenerierung
- die Codegenerierung.

Da der Schwerpunkt der Arbeit im Entwurf der Compilerstruktrur und der Zwischensprache liegt, wird in diesem Kapitel nur auf ausgewählte Implementationsdetails eingegangen. Im einzelnen sind dies:

- Generierung von Scannerprozeduren aus Übergangsdiagrammen,
- Implementation der SymboItabelIenyerwaltung,
- Implementation der Zwischencodegenerierung und Erkennen von gemeinsamen Teilausdrücken.

9.1 Modulstruktur

Als Grundlage für die Modulstruktur wird der Entwurf zur Modularisierung aus Abschnitt 4.2 verwendet. Funktional zusammengehörige Aufgaben werden im selben Modul untergebracht.

Die Modulstru1ctur des Compilers mit Angabe der Abhängigkeiten zwischen den Modulen ist in Abbildung 28 dargestellt.

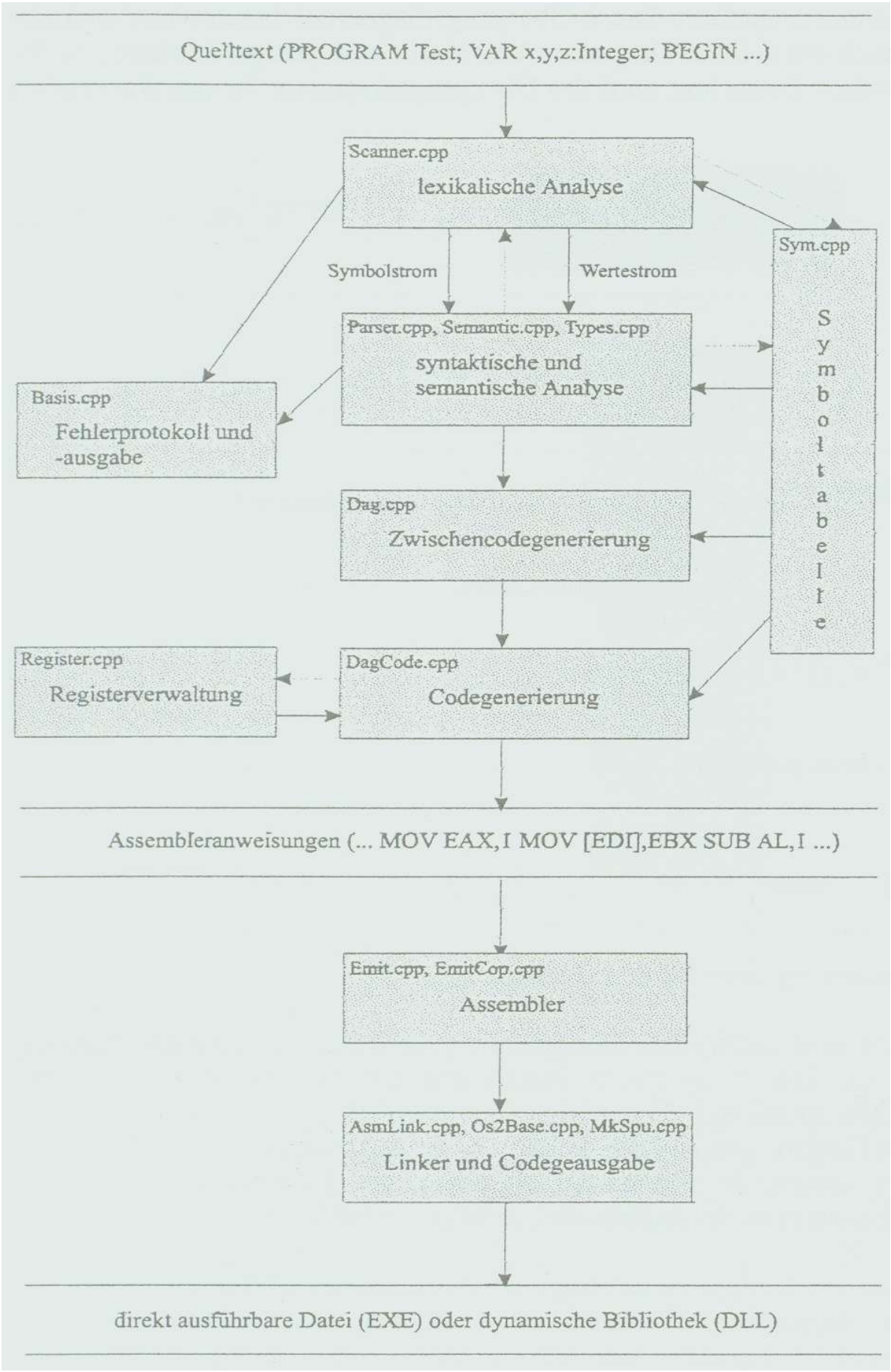


Abbildung 28: Die Modulstruktur des Compiliers mit Angaben der Abhängigkeiten

9.2 Implementation von Scannerprozeduren aus Übergangsdiagrammen

In Abschnitt 4.4. wurden Übergangsdiagramme für die Bildung von abstrakten Terminalsymbolen im Scanner eingeführt. Diese Übergangsdiagramme beschreiben endliche Automaten. Jeder Weg durch ein solches Diagramm beschreibt eine gültige Zeichenreihe für ein abstraktes Terminalsymbol. Betrachtet wird das Übergangsdiagramm für den Buchstaben „W“ aus Abschnitt 4.4.2:

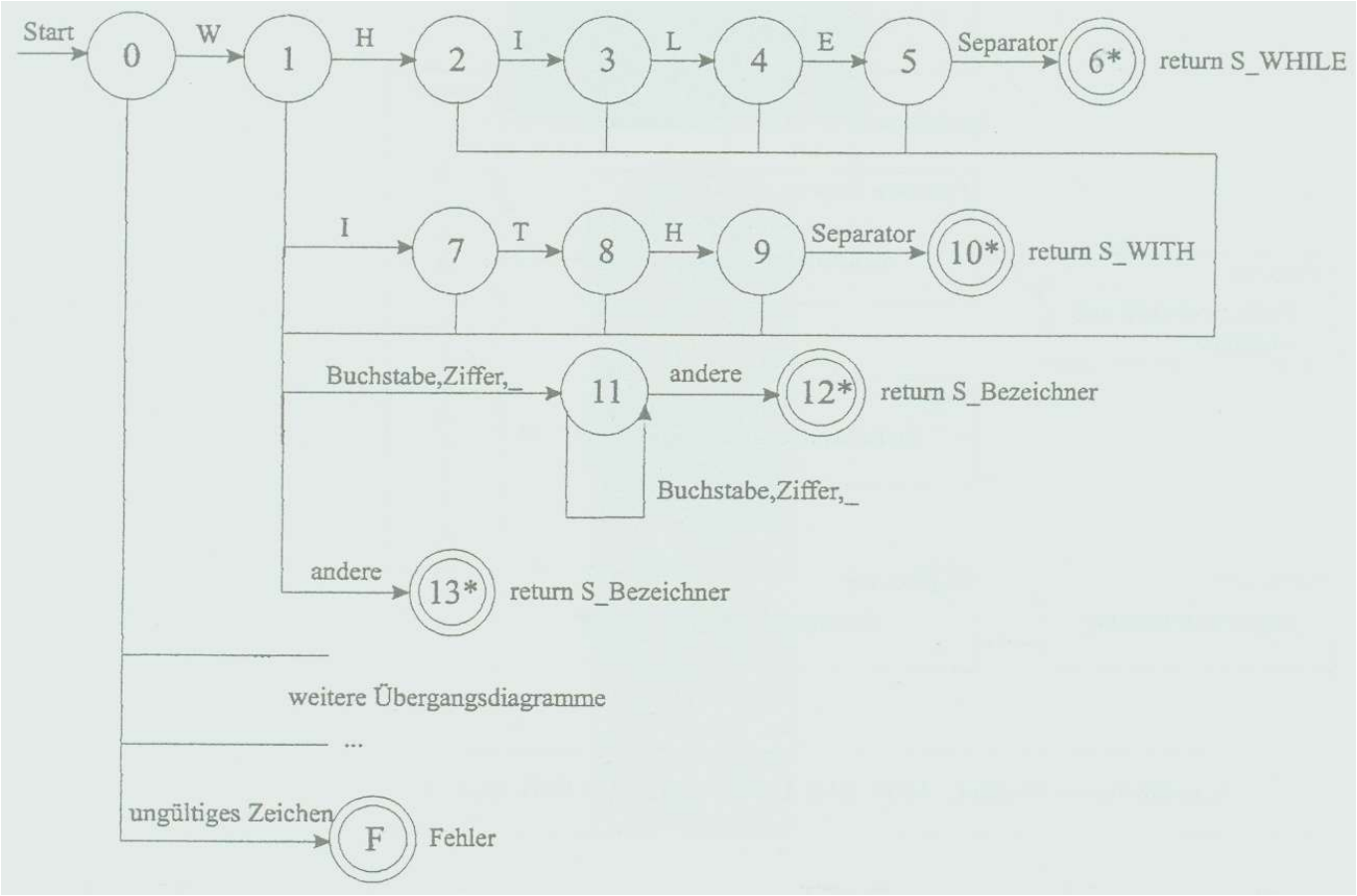


Abbildung 29: Das Übergangsdiagramm für den Buchstaben „W“

Endzustände des Automaten sind durch einen Doppelkreis gekennzeichnet, normale Zustände durch einen einfachen Kreis. Ein „*“ in einem Endzustand gibt an, dass der Scanner das aktuelle Symbol der Eingabe zurücklegt. Ein solcher Automat hält an, wenn vom aktuellen Zustand mit dem aktuellen Eingabesysmbol kein weiterer Übergang möglich ist. Die bis dahin durchlaufene Zeichenfolge entspricht dem erkannten abstrakten Terminalsymbol. Ist der betreffende Zustand kein Endzustand des Automaten, so liegt ein Fehler vor.

Zentraler Teil der lexikalische Ananlyse ist die Implementation dieser endlichen Automaten. Wie in Abschnitt 4.4 dargestellt, werden die endlichen Automaten zu einem Gesamtautomaten kombiniert. Dieser Automat erkennt alle gültigen abstrakten Terminalsymbole der Eingabezeichenfolge und liefert einen entsprechenden Code zurück. Für die Implementation des endlichen Automaten kommen tabellengesteuerte und direkt programmierte Verfahren in Frage. Tebellengesteuerte Automaten werden zum Beispiel vom Scanngenerator LEX erzeugt. Dabei wird die Übergangsfunktion des endlichen Automaten durch eine Matrix beschrieben. Eine weitere Matrix liefert den Rückgabecode (oder eine leere Ausgabe, falls der aktuelle Knoten nicht Endknoten ist) für den aktuellen Knoten. Diese Tabelle können eine beträchtlichen Umfang annehmen, weshalb eine Generierung von Hand nicht sinnvoll

erscheint. Da im Entwurf der Scanner eine Entscheidung zugunsten eines handgeschriebenen Scanners getroffen wurde, kommt die direkte Programmierung des Automaten zum Einsatz. Dieses Verfahren wird in der Literatur auch als das laufzeiteffizientere angesehen, obwohl eine direkte Untersuchung nicht aufgeführt ist.

Bei der direkte Programmierung des Scannerautomaten, werden die Übergansdiagramme systematisch in Programmcode umgesetzt [Kas90]. Dabei werden die Zustände des Automaten durch Stellen in Programm repräsentiert und die Übergänge zwischen den Zuständen durch bedingte Verzweigung und Schleifen.

Die zentrale Funktion des Scanners und zugleich die Implementation des endlichen Automaten soll die Funktion „*GetNextToken*“ sein. Diese Funktion soll folgende Aufgaben erfüllen:

1. Überlesen von Leerzeichen und Kommentaren sowie das Lesen des nächsten Zeichens der Eingabe.

2. Zusammenfassen von Einzelzeichen zu Zeichenketten (z.B. Bezeichner oder Zahlen) nach den in Pascal vorgeschriebenen Bildungsregeln. Die aktuelle Zeichenkette soll in der Variablen „*yytext*“ abgelegt sein. Alle Zeichen werden in Großbuchstaben umgewandelt, da Pascal nicht zwischen Groß- und Kleinschreibung unterscheidet.

3. Falls die generierte Zeichenkette ein Bezeichner ist, wird überprüft ob es sich dabei um ein Schlüsselwort handelt. Wird ein Schlüsselwort erkannt, so wird der entsprechende Code zurückgeliefert. Falls die Zeichenkette kein Schlüsselwort ist, so wird sie in der Symboltabelle gesucht. Ist der Bezeichner dort ebenfalls nicht enthalten so ist der Bezeichner undefiniert.

4. Falls die generierte Zeichenkette eine Konstante ist (etwa eine Zahl oder eine Zeichenkettenkonstante), so wird der entsprechende Wert generiert.

Für die folgenden Betrachtungen wird davon ausgegangen, dass sich das aktuelle Zeichen der Eingaben in der Variablen „*ch*“ befindet. Das darauf folgende Zeichen soll in der Variable „*chN*“ enthalten sein. Das Lesen des nächsten Zeichen der Eingabe soll durch eine Funktion „*Input*“ erfolgen. Diese Funktion überträgt den Inhalt von „*chN*“ in „*ch*“ und schreibt das nächste Zeichen in „*chN*“. Eine globales Feld „*ZeichenArt*“ von 256 Elementen soll für jedes Zeichen des ASCII Zeichensatzes eine Code festhalten:

- „*Buchstabe*“, falls es sich bei dem Zeichen um einen Buchstaben handelt,

- „Ziffer“, falls es sich bei dem Zeichen um ein Ziffer handelt,

- „Doppelzeichen“, falls dieses Zeichen eventuell Teil eines aus zwei Zeichen bestehenden Symbols sein kann (etwa das Zeichen „<“, was auch Teil des Operators „<>“ sein kann),

- „Einezelzeichen“, falls das Zeichen nur alleine vorkommen kann (etwa „;“)

- „Hochkomma“, falls das Zeichen ein einfaches Hochkomma ist.

Diese Feld wird bei der Initialisierung des Scanners gefüllt uns ist von da ab konstant. Es vdient zum schnellen Zugriff auf die verschiedenen Zeichenarten. Anhand dieser Informationen wird der Gesamtautomat zunächst in mehrere Teilautomaten zerlegt:

```
switch (ZeichenArt[ch])
{
  case Buchstabe
  {
    // Automaten, welche mit einem Buchstaben beginnen
  }
  break;
  case Ziffer:
  {
    // Automaten, welche mit einer Ziffer beginnen
  }
  break;
  case Doppelzeichen:
  {
    // Automaten für Symbole, welche aus zwei Zeichen
    // zusammengesetzt sein koennen
  }
  break;
  case Einzelzeichen:
  {
    // Automaten, fuer Symbole, welche aus einem Zeichen
    // bestehen
  }
  break;
  case Hochkomma:
  {
    // Automaten fuer Zeichenkettenkonstanten
  }
  break;
  default: Error(Unerwartetes Zeichen);  // Zeichen ungueltig
} // switch
```

Der Automat in Abbildung 29 beginnt mit einem Buchstaben. Es handelt sich also in jedem Fall um einen Bezeichner. Zunächst werden solange Zeichen der Eingabe gelesen, bis das gelesene Zeichen nicht mehr in einem Bezeichner vorkommen kann:

Der obige Codeausschnitt erzeugt einen nullterminierten Bezeicherstring aus den Zeichen der Eingabe. Das folgende Codefragment erkennt aus dieser Zeichenkette die Schlüsselwörter von Pascal. Es ist die direkte Umsetzung des Übergangsdiagramms, aus Platzgründen ist nur der Code für das in Abbildung 29 angegebene Übergangsdiagramm dargestellt. Die der jeweiligen Programmstelle entsprechenden zustände des Automaten in Abbildung 29 sind als Kommentare angegeben.

```
switch (yytext[0]
{
  …
  case 'W':
  {
    if ((yytext[1]== 'H')&&            Zustand 2
       ((yytext[2]== 'I')&&            Zustand 3
       ((yytext[3]== 'L')&&            Zustand 4
       ((yytext[4]== 'E')&&            Zustand 5
       ((yytext[5]==0) return S_WHILE;  Zustand 6

    if ((yytext[1]== 'I')&&            Zustand 7
       ((yytext[2]== 'T')&&            Zustand 8
       ((yytext[3]== 'H')&&            Zustand 9
       ((yytext[5]==0) return S_WHITH;  Zustand 10
  }
  break;
  …
} //switch

// Zustand 12
// Zeichenfolge ist ein Bezeichner, Suchen in der Symboltabelle
LookID(yytext, yylval.o, false);
if (yylval.o!=NIL) return yyval.o->symbol; //Symbol des Bezeichners
                                           //(z.b. S_Funktion)
//Bezeichner ist undefiniert, entweder Fehler ausloesen oder
//S_Bezeichner zurueckliefern.
```

Der Zustand 11 entfällt, da die Zusammensetzung von Bezeichern aus Einzelzeichen bereits erfolgt ist. Der Code für die anderen Übergangsdiagramme wird auf ähnliche Weise erzeugt.

9.3 Implementation der Symboltabellenverwaltung

In der Vorversion des Compilers hat sich gezeigt, dass eine effektive Symboltabellenverwaltung ein wesentliches Geschwindigkeitskriterium für einen Compiler ist. In Abschnitt 4.5 wurde dargelegt, aus welchen Gründen ein Hashverfahren für die Symboltabelle zum Einsatz kommen soll. Dabei sind insbesondere rechenzeitintensive Zeichenkettenoperationen zu vermeiden. In der Symboltabellenverwaltung treten Operationen mit Zeichenketten an verschiedenen Stellen auf:

- Bei der Bildung des Hashwertes muss die gesamte Zeichenkettte durchlaufen werden um eine möglichst gute Verteilung der Hashwerte zu erreichen. Solche Hashfunktionen sind in [Aho90] ausführlich erläutert (z.b. hashpjw).

- Bei der Überprüfung der durch den Hashwert identifizierten Liste von Bezeichnereinträgen müssen die Bezeichnerzeichenketten mit der gegebenen Zeichenkette verglichen werden.

In der zu implementierenden Symboltabellenverwaltung soll hierfür ein effektiver Algorithmus implementiert werden. Dafür sollen speziell die Zeichenkettenoperationen für die Ermittlung des Hashwertes und die Suche eines Bezeichners minimiert werden Jede Zeichenkette soll hierfür nur maximal einmal gespeichert sein. Die Identifikation dieser Zeichenkette erfolgt dann über einen 32-Bit-Wert. Dies hat neben der Einsparung von Speicherplatz den Vorteil, dass beim Suchen eines Bezeichners nur jeweils die Adressen der Zeichenketten (gleich Zeichenketten sind immer auf gleichen Adressen abgelegt!) verglichen werden müssen. Der Hashwert einer Zeichenkette kann auf einfache Weise aus der Adresse abgeleitet werden, da die Adresse einer Zeichenkette immer eindeutig ist. Die Verwaltung von Zeichenketten erfolgt im Modul „STRING.CPP“. In diesem Modul ist eine Funktion „*NewString*“ implementiert, welche Zeichenketten anlegt. ist die übergebene Zeichenkette bereits vorhanden, wird der Zeiger auf den existieren Speicherbereich zurückgeliefert. Das zugrundeliegene Verfahren wurd aus [Fra95] übernommen. Zunächst wird ein Hashwert für die Zeichenkette gebildet. Als Grundlage zur dient ein globales Feld „*scatter*“, welches 256 Zufallszahlen enthält. Jedes mögliche Zeichen der Zeichenkette identifiziert eine solche Zufallszahl. Aus diesen Zufallszahlen wird der Hashwert für die Zeichenkette ermittelt („*str*“ ist hierbei der übergebene String und „*len*“ dessen Länge):

Der resultierende Hashwert dient zur Ermittlung einer globalen Liste von Zeichenketten aus einer globalen Hashtabelle „*Strbuckets*“. In dieser Liste wird die übergebene Zeichenkette zunächst gesucht. Da die zugrunde liegende Hashtabelle relativ gross ist (1024 Einträge) und die Hashwerte gleichmäßig verteilt sind [Fra95], sind hierfür in der Regel nur ein oder zwei Vergleiche durchzuführen. wird die Zeichenkette gefunden, so wird der existierende Zeiger zurückgeliefert, ansonsten wird die Zeichenkette neu angelegt. Dies garantiert, dass jede Zeichenkette nur einmal angelegt wird. Dieser Algorithmus entspricht dem in [Fra95] dargestellten.

Das beschrieben Verfahren vereinfacht die Symboltabllenverwaltung wesentlich. Zunächst ist die Bildung eines Hashwertes für eine übergeben Zeichenkette relativ simpel und erfolgt durch das Makro

```
#define HASHSIZE 256
#define Hash(Name) (unsigned)Name%HASHSIZE
```

Dabei ist „*HASHSIZE*“ die Größe der Hashtabelle für jeden Namensraum der Symboltabelle. Da die Speicheradresse für verschiedene Zeichenketten immer unterschiedlich ist, ergibt sich ein gut verteilter Hashwert. Zu beachten ist, dass die übergebenen Zeichenketten immer durch die Funktion „*NewString*“ angelegt werden müssen. Dies stellt sicher, dass jede Zeichenkette nur einmal abgelegt wird und gleichartige Zeichenketten somit die gleiche Adresse und damit den gleichen Hashwert erhalten. Ein Namensraum selbst wird durch folgende Datenstruktur dargestellt:

```
typedef struct T_Scope_strunct *P_Scope;
typedef struct T_Scope_strunct
{
  P_Objekt Ketten[HASHSIZE];
  P_Objekt Erster, Letzter;
}T_Scope;
```

```
for (longint zaehler=len, end=str; zaehler>0; zaehler--)
  Hashwert=(Hashwert<<1) + scatter[*unsigned char *)end++];
Hashwert= Hashwert%((longint)(sizeof(Strbuckets)/
          sizeof(Strbuckets[0]))
);
```

Der Datentyp P_Objekt verkörpert hierbei einen Eintrag in die Symboltabelle. Diese Datenstruktur enthält unter anderem den Namen des Objektes und seinen Typ. Die Vorgehensweise beim Suchen und Eintragen von Bezeichnern soll hier nur an einem Beispiel verdeutlicht werden. Betrachtet wird die Funktion „*SearchIdentifier*“, welche einen Bezeichner in der Symboltabelle sucht. zu diesem Zweck durchläuft die Funktion alle Namensräume. Der oberste Namensraum ist in der Variablen „*ScopeStack*“ gespeichert. Für jeden Namensraum wird eine Funktion „*ChainNameSearch*“ aufgerufen, welche die Suche in diesem Namensraum übernimmt. Der folgende Quelltextausschnitt die Implementierung der Funktion „*SearchIdentifier*“:

```
void SearchIdentifier(char *Name, P_Objekt * Objekt, boolean Finden)
{
  register P_ScopeStack ScopeStackListe;

  ScopeStackListe=ScopeStack;
  Objekt=NIL;
  while ((ScopeStackListe!=NIL)&&(Objekt==NIL)))
  {
    Objekt=ChainNameSearch(Name,
                        ScopeStackListe->Scope->Ketten[Hash(Name)]);
    ScopeStackListe= ScopeStackListe->next;
  }
  if ((Objekt==NIL)&&(Finden) Error(Undefinierter Bezeichner, Name);
}
```

Die Funktion „*ChainNameSearch*“ wird nun der Bezeichner in dem übergebenen Namensraum gesucht. Dafür werden nur die Adressen der Zeichenketten verglichen, da ausgehend von den obigen Betrachtungen jeder Zeichenkette nur einmal gespeichert ist. Dies ergibt eine wesentliche Geschwindigkeitssteigerung im Gegensatz zum Vergleich von ganzen Zeichenketten.

```
P_Objekt ChainNameSearch(Char *Name,P_Objekt Kette)
{
  while (Kette!=NIL)
  {
    if (Kette->Name==Name) return Kette; //Bezeichner gefunden
    Kette=Kette->Next;
  }
  //Bezeichner nicht gefunden
  return NIL;
}
```

9.4 Implementation der Zwischencodegenerierung

In der Zwischencodegenerierung werden vom Parser generierte Syntaxbaume in GAGs transformiert (siehe Abschnitt 5.6). Dabei werden vom Zwischencodegenerator gemeinsame Teilausdrücke erkannt. Diese Erkennung von gemeinsamen Teilausdrücken kann auch über Anweisungsgrenzen hinaus erfolgen. Dafür müssen alle von Zwischencodegenerator bearbeiteten Ausdrücke zum Zweck des spüteren Vergleiches gespeichert werden. Dies erfolgt in einem global en Feld *„Buckets"*. Diese Datenstruktur ist in der folgenden Abbildung dargestellt:

```
typedef struct T_BucketEntry_struct *P_BucketEntry
typedef struct T_BucketEntry_struct
{
  T_DagNode Node;      // generierter DAG
  P_BucketEntry Next;  // Verbindung zum naechsten Eintrag

} T_BucketEntry;

P_BucketEntry Buckets[32];
```

Für jeden übergebenen Syntaxbaum ruft der Zwischencodegenerator rekursiv die Funktion "GenNode" auf, welche überprüft, ob für den Syntaxbaum schon ein äquivalenter GAG existiert (siehe auch [Fra95]). In einem solchen Fall wird der existierende GAG zurückgeliefert. Der Vergleich erfolgt nach folgenden Kriterien:

1.) Die Operatoren des Syntaxbaumes und des existierenden GAG stimmen überein,

2.) Der Syntaxbaum und der GAG haben identische Teilbäume

3.) Der GAG ist kein L Value, in diesem Fall ist das Feld *„Available"* der *"T_ DagNode"* Datenstruktur auf "TRUE" gesetzt

Im folgenden wird ein Ausschnitt aus der *„ListNodes"* Funktion betrachtet, welche Syntaxbäume in GAGs umwandelt:

```
P_DagNode ListNodes(P_Tree Tree)
{
  P_DagNode result;

  if (Tree==NIL) result=NIL; //Der Baum ist leer
  else
  {
    if (Tree->DagNode!=NIL) //es existiert bereits ein GAG
    {
       result=Tree->DagNode;
    }
    else
    {
      switch (Tree->op) //Teste Operatoren
      {
        case S_Plus: case S_Minus: case S_Times: case S_Divide:
        {
          P DagNode l,r;
          l=listNodes(Tree->U.Kids[O];    //rekursiver Aufruf fuer linken
                                          //Teilbaum
          r=listNodes(Tree->U.Kids[1]);   //rekursiver Aufruf fuer rechten
                                          //Teilbaum
          result=GenNode{Tree-->op, l,r); // Generierung Des DAG
        }
       break
       // …
       //weitere Faelle
       // …
     } // switch
     Tree->DagNode=result
  } // else
  return result;
}
```

„ListNodes" durch1äuft den übergebenen Syntaxbaum rekursiv und ruft zum Erzeugen eines GAG die

Funktion *"GenNode"* auf *"GenNode"* liefert einen existierenden DAG, falls der Syntaxbaum mit einem bereits generierten Syntaxbaum übereinstimmt. Andernfalls erzeugt die Funktion einen neuen GAG, Für den Fall, dass ein neuer GAG erzeugt wird, wird er in die Liste der für die Elimination von gemeinsamen Tei1ausdrücken verfügbaren GAGs aufgenommen. Der folgende Codeausschnitt stellt die Implementation dieses Algorithmus dar:

```
P_DagNode GenNode(T_Symbol DagOp, P_DagNode 1, P_DagNode r)
{
longint HashUert;
P_BucketEntry Entry;
P _DagNode result

 //Hashwert fUr Buckets bestimmen
 HashUert=DagOp&(sizeof (Buckets)/sizeof (T_BucketEntry») -1);
 if (Optimize) //nur wenn Optimierung erwunscht
 {
  //liste der fur diese Operation verfugbaren DAGs Entry=Buckets
  [HashUert);
  //suchen ob DAG schon yon einem anDeren Syntaxbaum erzeugt wurde
  while (Entry!=NIL)
  {
    if ((Entry->Node.Op==DagOp)&&(Entry->Node.Kids[O]==l)&&
        (Entry->Node.Kids[l]==r)&&(Entry->Available))
   {
    result=&(Entry->Node);
    result->UseCount++; //Zahler erhoehen
    return result;
   }
   Entry=Entry->Next;
  } //while
 } //if Optimize
//neuen DAG generieren
GetMem(Entry,sizeof(T_BucketEntry));
Entry->Node.Op=DagOp;
Entry->Node.Kids[O]=l;
Entry->Node.Kids[l]=r;
Entry->Node.UseCount=l;
Entry->Node.Available=TRUE;

//Diesen DAG in die liste eintragen
Entry->Next=Buckets[HashUert];
Buckets(HashUert]=Entry;
return &(Entry->Node);
```

Um eine korrekte Auswertung von gemeinsamen Teilausdrücken zu ermöglichen, wird die Liste der dafür verfügbaren DAGs *(„Buckets")* für jede Prozedur neu initialisiert. Eine ausführliche Darstel1ung dieses Verfahrens findet sich in [Fra95].

10 Kapitel 8
Entwurf und Implementation des Laufzeitsystems

Das Laufzeitsystem stellt elementare Funktionen zum Ablauf eines Programms zur Verfugung. Dazu gehören unter anderem die Speicherverwaltung, die Behandlung von Ausnahmen (Exceptions) und die Unterstützung von mehreren Prozessen bzw. Threads. Das Laufzeitsystem ist zum großen Teil in der Unit "System" enthalten. Diese Unit wird automatisch in jedes Programm eingebunden. Die in der Unit "System" enthaltenen Definitionen und Prozeduren sind für alle mit dem Compiler erzeugten Programme verfügbar. Da das Laufzeitsystem zum großen Teil aus dem vorhandenen Compiler übernommen wurde, werden in diesem Kapitel lediglich besonders wichtige bzw. neu hinzugekommene Teile des Laufzeitsystems dargestellt.

10.1 Speicherverwaltung

Die Betriebssysteme OS/2 Warp, Windows 95 und Windows NT unterstützen ein effektives Speichermanagement. Routinen zur Speicherverwaltung sind direkt über die API Schnittstelle des Betriebssystems aufrufbar. Direkte Zugriffe auf den physischen Hauptspeicher werden von den Betriebssystemen unterbunden. Statt dessen wird der verfügbare Hauptspeicher "virtualisiert". Für Applikationen ist der Adressraum von Adresse 00000000_h bis Adresse $FFFFFFFF_h$ linear durchgängig adressierbar. In Abbildung 23 ist die Aufteilung des Adressraumes eines Prozesses für die zu unterstützenden Betriebssysteme dargestellt.

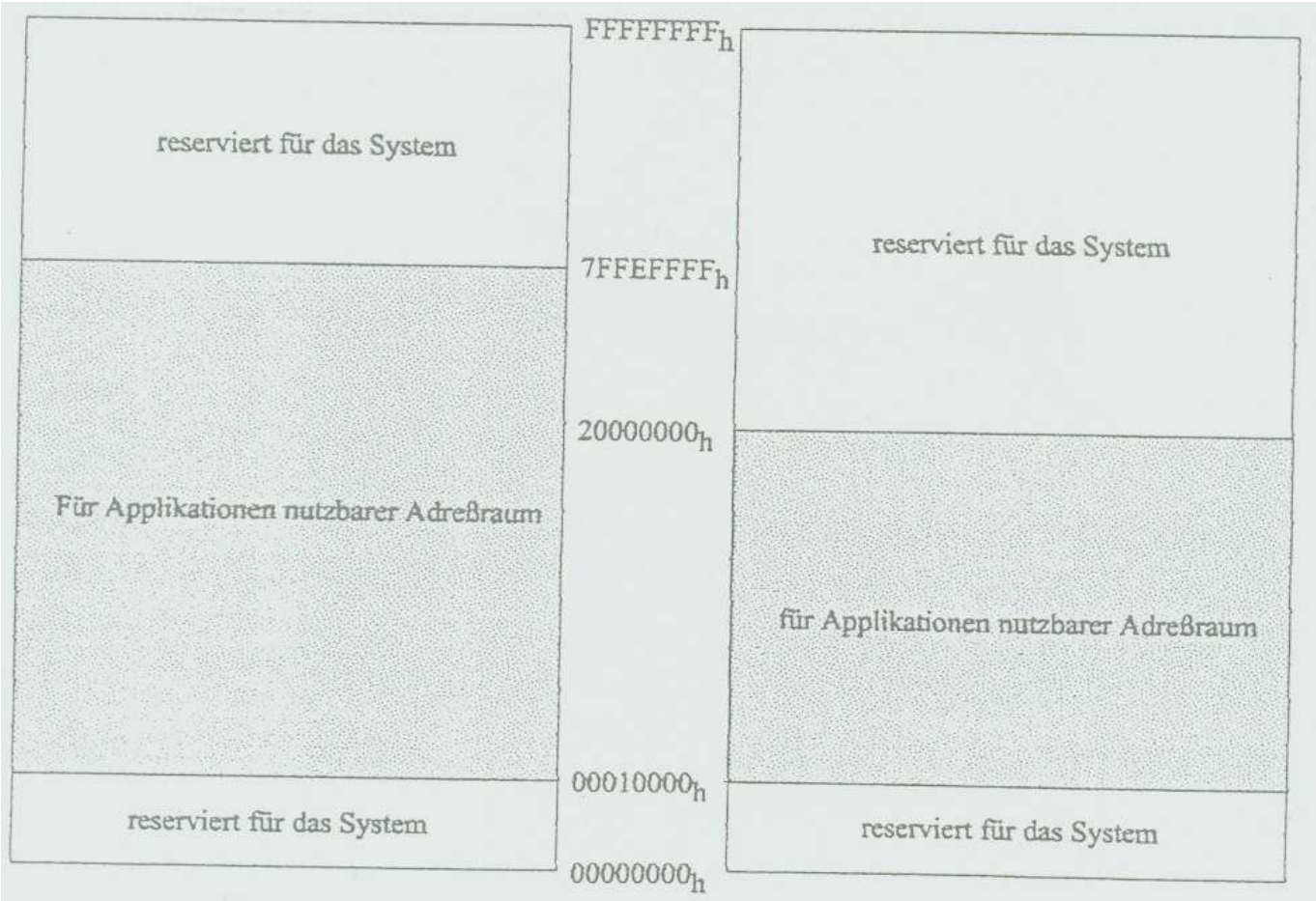


Abbildung 29: Der 4 GB Adressraum eines Prozesses unter Windows 95, Windows NT (links) und OS/2 Warp (rechts)

Aus historischen Gründen ist der für Applikationen nutzbare Adressbereich unter OS/2 Warp auf Adressen unter 512 MB limitiert. Dies garantiert die Lauffähigkeit von Programmen auf OS/2 1.x und 2.x Systemen. Alle unterstützten Betriebssysteme reservieren Teile des Adressraums für das Betriebssystem. Die Zuteilung von Speicher erfolgt über Betriebssystemdienste. Die Speicherverwaltung erfolgt in allen Systemen in mehreren Ebenen. Dieses Prinzip ist in der folgenden Abbildung exemplarisch für die Betriebssysteme Windows95 und Windows NT dargestellt

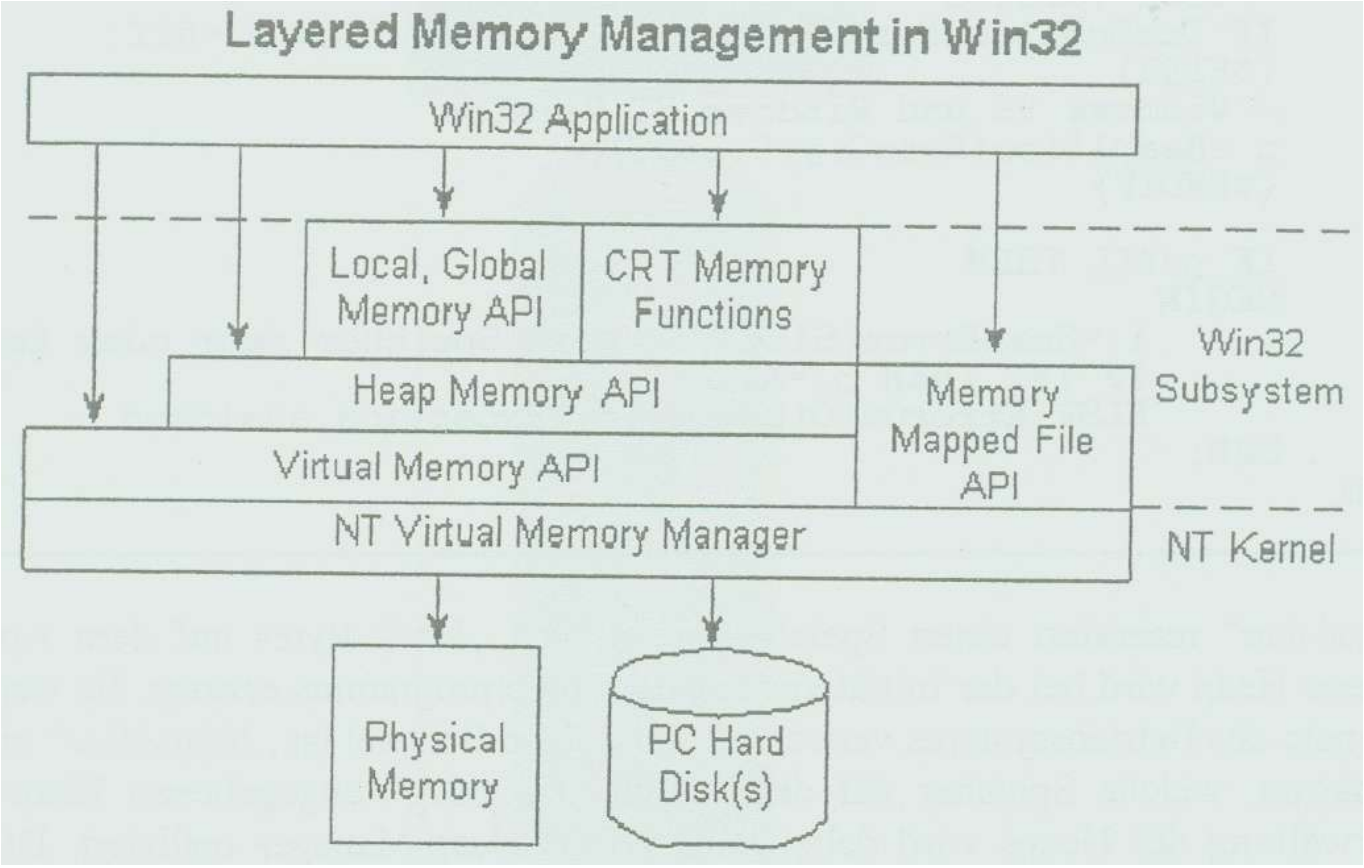


Abbildung 30: Speichermanagement in Windows 95 und Windows NT. (Quelle: [MS94])

Speicher kann vom System jeweils nur für den anfordernden Prozess (private memory) oder für mehrere Prozesse (shared memory) angefordert werden. Das Betriebssystem reserviert entsprechende Bereiche des Prozessadressbereiches für gemeinsam genutzte Daten. Alle zu betrachtenden Systeme unterstützen virtuellen Speicher.

10.1.1Reservierung von Speicher

Die Anforderung von Speicher wird in Pascal durch die Standardfunktionen *„New"* bzw. *"GetMem"* realisiert. Die Funktion *„New"* wird dabei in einen Aufruf von *"GetMem"* umgesetzt:

"New(p);" ist aquivalent zu "GetMem(p,sizeof(p);".

Die Implementation der Prozedur *"GetMem"* ist in der Unit "System" enthalten. „*New"* und *"GetMem"* können von jedem Prozess aus aufgerufen werden, da die System-Unit automatisch in jedes Programm eingebunden wird. Teilweise werden die Funktionen auch für den Programmierer unsichtbar vom Compiler aufgerufen (etwa zur Speicherreservierung für ein durch einen Konstruktor angelegtes dynamisches Objekt). Die Funktion *"GetMem"* ist in der Laufzeitbibliothek folgendermaßen implementiert.

```
FUNCTION GetMem(Var p:POINTER; Size:LongWord);
VAR i:Integer;
BEGIN
  If Size=0 then
    Begin
      p:=NIL
      exit;
    End;
  {$IFDEF OS2}  //bedingte Compilierung
  IF DosSubAllocMem(HeapOrg,p,Size)<>0 then p:NIL;
  {$ELSE}
  P:=HeapAlloc(HeapOrg,0,Size);
  {$ENDIF}
  IF p=NIL THEN
  BEGIN
    i:=HeapError(Size);  //Kein Speicher mehr oder Heapfehler
    IF i=1 THEN p:=NIL
    ELSE ErrorOutOfMemory;  //Exception ausloesen
  END;
END;
```

„*GetMem*“ reservier einen Speicherbereicht von „*Size*“ Bytes auf dem Applikationsheap. Dieser Heap wird bei der Initialisierung des Hauptprogrammes erzeugt. Es werden also direkte Dienste des Betriebsystems verwendet. Im obigen Beispiel ist „*HeapAlloc*“ eine Win32 API-Funktion, welcher Speicher auf dem durch „*HeapOrg*“ angegebenen Heap anfordert. Die Verwaltung des Heaps wird dabei vom Win32 Heap-Manager realisiert. dies erlaubt besonders einfache und fehlerfreie Implementation des Speichermanagements. Die entsprechende OS/2 API-Funktion lautet „*DosSubAllocMem*“.

10.1.2Freigabe von Speicher

Die Freigabe von Speicher wird in Pascal durch die Standardfunktionen „*Dispose*“ bzw. „*FreeMem*“ realisiert. Die Funktion „*Dispose*“ wird dabei in einen Aufruf von „*FreeMem*“ umgesetzt:

„Dispose(p);“ ist äquivalent zu „FreeMem(p, sizeof8p);“.

Die Implementation der Prozedure „*FreeMem*” ist ebenfalls in der Unit „System“ enthalten:

```
PROCEDURE FreeMem(p:POINTER;Size:LongWord);
VAR ok:Boolean;
    i :Integer;
BEGIN
  IF Size=0 THEN exit;
  {$IFDEF OS2}   //bedingte Compilierung fuer OS/2
  ok:=DosSubFreeMem(HeapOrg,p,Size)=0;
  {$ELSE}
  ok:=HeapFree(HeapOrg,0,p);
  {$ENDIF}
  IF not ok THEN
  BEGIN
    i:=HeapError(Size);
    IF i<>1 THEN ErrorInvalidPointer;
  END;
END;
```

„*FreeMem"* gibt einen Speicherbereich von „*Size"* Bytes auf dem Applikationsheap frei. Der Speicher muss zuvor durch die Funktion "GetMem" angefordert worden sein. Die Realisierung der Heapverwaltung erfolgt über die API-Funktionen „*HeapFree*“ (Windows 95 und NT) und „*DosSubFreeMem*“(OS/2).

10.1.3Reservierung und Freigabe von Shared-Memory

Für die Anforderung bzw. Freigabe von gemeinsam benutzten Speicher (Shared-Memory) wurden zwei neue Funktionen in das Laufzeitsystem aufgenommen. Diese Prozeduren existieren in Turbo-Pascal nicht. Shared-Memory ist Speicher, der von mehreren Prozessen gemeinsam genutzt werden kann. Die Realisierung erfolgt wiederum über direkte Aufrufe von Betriebssystemdiensten:

```
//Anfordern yon shared memory
PROCEDURE GetSharedMem(VAR pp:POINTER;size:LongWord);
BEGIN
  {$IFDEF OS2} //bedingte Compilierung
  //Version fuer OS/2
  IF DosAllocSharedMem(pp,NIL,size.O)<>O
    THEN ErrorOutOfMemory;

  {$ELSE}
  //Version fur Windows 95 und Windows NT
  pp:=GlobalAlloc(GMEM_SHARE,size);
  IF pp=NIL THEN ErrorOutOfMemory; //Exception auslosen
  {$ENDIF}
END;

PROCEDURE FreeSharedMem(p:POINTER;size:LongWord);
BEGIN
  {$IFDEF OS2} //bedingte Compilierung
  //Version fuer OS/2
  IF DosFreeMem(p) < >0 THEN ErrorInvalidPointer;
  {$ELSE}
  //Version fur Windows 95 und Windows NT
  IF GlobalFree(p)<>NIL THEN ErrorInvalidPointer; //Exception ausloesen
  {$ENDIF}
END;
```

10.2 Unterstiitzung von Prozessen und Threads

Unterstützung von mehreren Prozessen[24] und optional mehreren Threads pro Prozess, ist ein wesentlicher Vorteil der hier unterstützten Betriebssysteme gegenüber dem Betriebssystem DOS. Es existieren eine ganze Reihe von Systemfunktionen, welche der Verwaltung von Prozessen und Threads dienen. Windows 95, Windows NT und OS/2 implementieren ein dreistufiges Task-Modell - die "Sitzung" (Session), den Prozess und den Thread. Eine Session kann als eine virtuelle Maschine angesehen werden, in welcher ein oder mehrere Prozesse ausgeführt werden können. Jeder Prozess kann wiederum einen oder mehrere Threads starten. Jeder Prozess hat seinen eigenen Prozessadressraum (siehe Abschnitt 7.1). Threads sind eine Art "leichtgewichtige" Prozesse. Sie laufen im Adressraum des erzeugenden Prozesses ab. Dieses Schema ist in Abbildung 31 dargestellt.

24 Unter einem Prozess soll im folgenden verstanden werden, was auf den betrachteten Betriebssystemen als Anwendungsprozess existiert. Anwendungsprozesse können mehrere innere Prozesse in Form von Threads enthalten

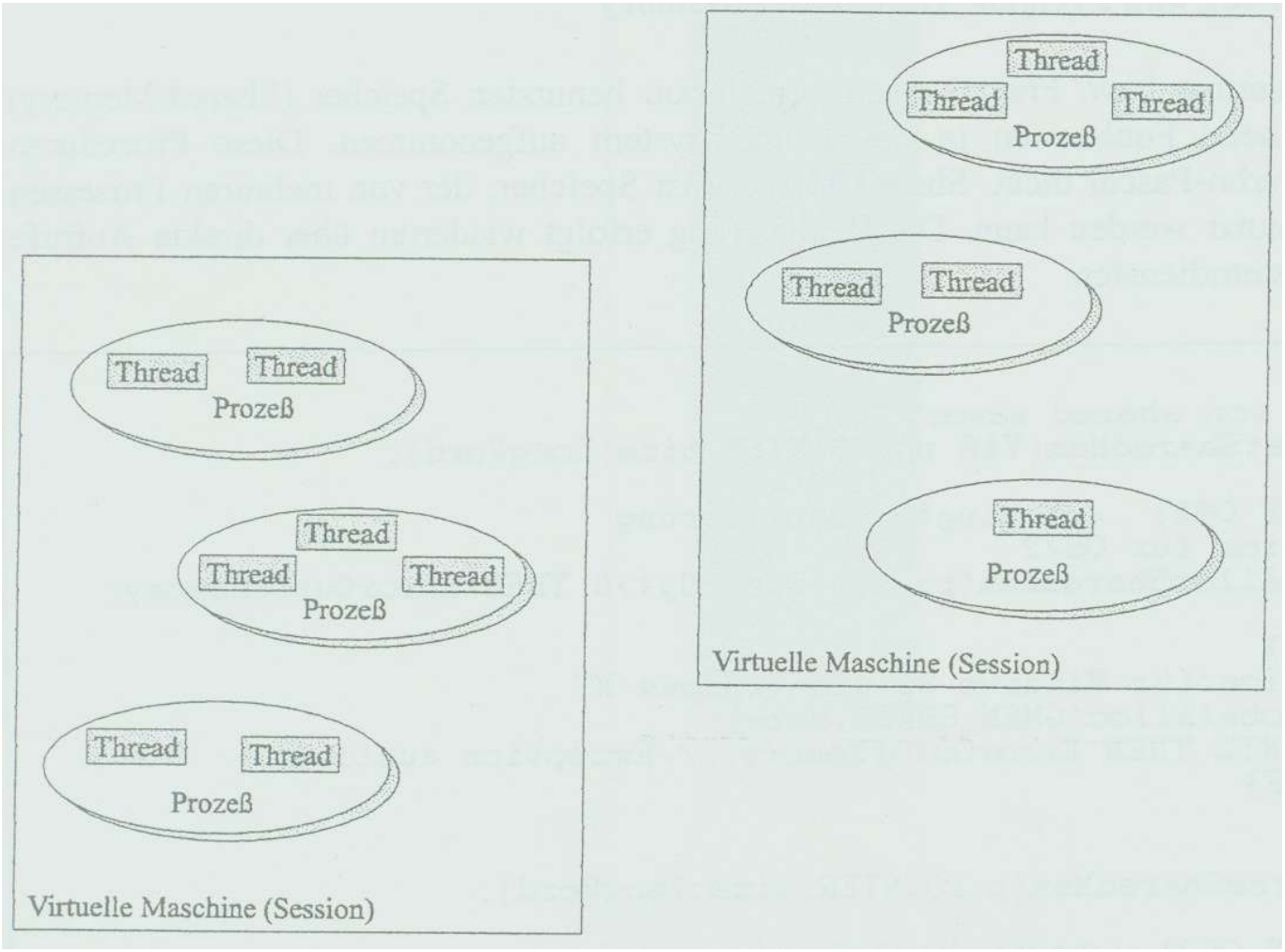


Abbildung 31: Das dreistufige Task-Modell von Windows 95, Windows NT und OS/2

10.2.1Prozesse

Zum Starten und Verwalten von Prozessen existieren in den unterstützten Betriebssystemen verschiedene Betriebssystemdienste. Jeder Prozess enthalt mindestens einen Thread, den so genannten ,, Hauptthread". Dieser Thread wird automatisch vom Betriebssystem erzeugt. Jeder Prozess hat seinen eigenen Prozessadressraum. Die Vergabe der CPU-Zeit zwischen parallel ablaufenden Threads erfolgt dynamisch durch das Betriebssystem über das so genannte Zeitscheibenverfahren. Dabei wird jedem Thread eine Zeitscheibe zugeordnet. Der Thread wird solange ausgeführt, bis seine Zeitscheibe abgelaufen ist oder eine Wartebedingung eintritt. In jedem der beiden Fälle wird die CPU-Zeit an eventuell andere arbeitsbereite Threads abgegeben. Das Erzeugen eines neuen Prozesses erfolgt durch die API-Funktionen *"CreateProcess"* (Windows) und *,,DosCreateProcess"* (OS/2). Die Eintrittspunkte dieser Funktionen in die System-DLLs sind in der Unit "WinBase" (Windows) bzw. ,,BseDos" (OS/2) definiert. Windows Systemdienste werden dabei typischerweise über den Namen importiert, die Zeile

APIENTRY; 'KERNEL32' name 'CreateProcessA'

besagt, dass die API-Funktion "CreateProcessA" in der dynamischen Bibliothek ,,KERNEL32" enthalten ist. Die Direktive ,,APIENTRY" steuert die Aufrufart. In diesem Beispiel erfolgt der Aufruf nach der C-Aufrufkonvention. Für OS/2 erfolgt der Import von API-Funktionen über Indizes in der entsprechenden DLL. Die Zeile

APIENTRY; 'DOSCALLS' index 283

besagt, dass die Funktion ,,DosExecPgm" in der DLL ,,DOSCALLS" den Index 283 hat. Die entsprechenden Relokationen werden vom Lader ausgeführt

Die Bedeutung der einzelnen Parameter kann der einschlägigen Literatur entnommen werden, zum Beispiel [BDE93] und [MS94].

10.2.2Threads

Ein Thread ist ein "leichtgewichtiger" Prozess. Ein Thread besitzt seinen eigenen Stack und ein eigenes Registerset. Alle anderen Betriebsmittel teilt der Thread mit den anderen Threads des erzeugenden Prozesses. Das Prozessorzuteilung der unterstützten Betriebssysteme erfolgt thread-basiert, das bedeutet, dass der Scheduler des Betriebssystems CPU-Zeit an Threads verteilt, nicht an Prozesse. Für Threads existieren verschiedene Prioritätsklassen

- Zeitkritisch (Time Critical)
  Diese Threads werden bevorzugt von System bedient. Sie werden besonders in Rahmen von Echtzeitanwendungen eingesetzt

- Hoch (High)
  Diese Prioritätsklasse wird für Threads eingesetzt, welche gute mittlere Antwortzeiten benötigen, ohne zeitkritisch zu sein

- Normal (Regular)
  Dies ist die Standardvorgabe

- Leer/auf (Idle-Time)
  Diese Prioritätsklasse wird für Threads eingesetzt, welche im Hintergrund laufen.

Zur Generierung und Zerstörung von Threads wurden zwei Funktionen in die Laufzeitbibliothek aufgenommen. Die Funktion zum Erzeugen eines Threads heisst *„Begin Thread'* und ist folgendermassen implementiert:

Für die Erzeugung eines neuen Threads werden die Systemrufe *„DosCreateThread*“ (OS/2) und *"CreateThread*“ (Win95 und NT) genutzt. Die Erzeugung eines Threads benötigt weit weniger Systemressourcen als die Erzeugung eines neuen Prozesses. Threads eines Prozesses können darüber hinaus in idealer Weise über globale Variablen miteinander kommunizieren (eine entsprechende Synchronisation[25] vorausgesetzt). Die Verwendung von globalen Variablen kann jedoch auch zu unerwünschten Nebeneffekten führen. So kann es zum Beispiel zu Problemen kommen, wenn zwei Threads versuchen, gleichzeitig eine globale Variable zu verändern[26]. Der Inhalt der Variablen kann dabei unter Umstanden zerstört oder verfälscht werden. Im folgenden Beispiel kann es zu einer Verfälschung des Wertes der Variablen "x" durch den konkurrierenden Zugriffzweier Threads kommen.

25 Die unterstützten Betriebssysteme bieten hierfür zum Beispiel verschiedene Arten von Semaphoren oder Signale an

26 Genau genommen kommt es schon zu Problemen wenn ein Thread die Variable verändert (schreibt) und ein anderer Thread die Variable liest. Solche Programmstellen werden auch als kritische Abschnitte bezeichnet.

```
VAR x: Integer;
PROCEDURE TestThread;
BEGIN
  x: =x+1;
END;
```

Für den Fall, dass beide Threads die Prozedur "TestThread" aufrufen, ist der Wert der Variablen x nicht mehr definiert. Zur Lösung dieses Problems bietet Object-Pascal die Möglichkeit, thread-lokale globale Variablen anzulegen. Die derart deklarierten globalen Variablen werden dann einmal pro Thread verwaltet. Die entsprechende Deklaration der Variablen "x" im obigen Beispiel müsste dafür lauten:

```
FUNCTION BeginThread(SecurityAttrs:POINTER;StackSize:LongWord;
                     ThreadFunc:TThreadFunc; Parameter: POINTER;
                     Options:LongWord;VAR id:LongWord):LongWord;
VAR Data:PThreadData;
BEGIN
  inc(StackSize,SysTlsSize+4096);
  New(Data);
  DataA.f:=ThreadFunc;
  DataA.p:=Parameter;
  {$IFDEF OS2} //bedingte Compilierung
  //Version fur OS/2
  DosCreateThread(result,@SysThreadProc,Data,Options,StackSize);
  id:=O;
  {$ELSE}
  //Version fur Windows 95 und Windows NT
  result:=CreateThread(SecurityAttrs,StackSize,@SysThreadProc,Data,und
                       Options, id);
  {$ENDIF}
END;
```

```
                     CONST lpCurrentDir:CSTRING;
                     VAR lpStartupInfo:STARTUPINFO;
                     VAR lpInfo:PROCESS_INGFORMATION):BOOL;

                     APIENTRY; 'KERNEL32, name 'CreateProcessA'

//OS/2 Version (Unit BseDos)
FUNCTION DosExecPgm(pObjName:POINTER;cbObjName:LONG;execFlag:ULONG;
                    VAR pArg,pEnv:CSTRING;VAR Res:RESULTCODES):APIRET;

                    APIENTRY; 'DOSCALLS' index 283;
```

Für Thread-lokale Variablen erzeugt der Compiler speziellen Code für Lese- und Schreiboperationen auf die Variable. Dabei wird solchen Variablen ein Offset im Thread-Lokalen Speicherbereich (TLS - Thread Local Storage) des Prozesses zugewiesen. Die Ermittlung der Adresse der Variablen erfolgt dann über die Routine *"GetTisVar"* aus der System-Unit. Diese Aufrufe sind für den Programmierer transparent und werden automatisch vom Compiler generiert. Zum Initialisieren der entsprechenden Datenstrukturen wird in der Implementation der Funktion „BeginThread" zunächst die interne Funktion "SysThreadProc" als Thread-Prozedur angegeben. Diese Funktion fordert den für die Speicherung der TLS-Variablen benötigten Speicher an und ruft danach die eigentliche Thread-Prozedur auf. Zur Speicherung dieser Werte wird der Aufrufstack des Threads benützt:

```
FUNCTION SysThreadProc(Param:PThreadData):LongInt;CDECL;
VAR f :TthreadFunc;
  p;POINTER;
  Data:POINTER;
  Diff : LongTJord;

BEGIN
  f:=ParamA.f;
  p:=ParamA.p;
  Dispose(Param);

//Aufrunden auf Seitengresse
  Diff:=SysTlsSize+4096;
  Diff:=Diff DIV 4096;
  Diff : =Diff*4096;

//Thread-lok. Speicher auf dem Stack belegen und mit 0 fuellen
  ASM
    MOV EDI.ESP
    SUB EDI, 4
    SUB ESP, Diff
    MOV Data,ESP
    //loschen
    MOV ECX, Diff
    SHR ECX, 2
    XOR EAX,EAX STD
    REP STOSD CLD
  End;

  //Thread-lokalen Speicher registrieren
  NewTlsData(GetThreadId-l,Data);

  //Aufruf der Thread-Prozedur
  result: =f(p);

  //Thread beendet sich selbst
  EndThread(0);
End;
```

Die Variable "SysTlsSize" gibt hierbei den für die Speicherung aller Thread-Lokalen Variablen benötigten Speicher an. Dieser Wert wird während der Initialisierung des Moduls gesetzt. Für die spätere Ermittlung der Werte der Variablen muB die Anfangsadresse des auf den Stack angelegten Speicherbereiches während der gesamten Laufzeit des Threads bekannt sein. Windows 95 und Windows NT bieten zu diesem Zweck eigene API-Funktionen an, welche der Verwaltung Yon TLS (Thread Local Storage) dienen. Da OS/2 derartige Möglichkeiten erst ab Version 4.0 bietet, müssen die Werte für OS/2 anders gespeichert werden. Dafür dient ein globales Feld *"l1sData"*, welches über die Identifikationsnummer des Threads (welche immer eindeutig ist und bei 0 fur den Hauptthread beginnt) indiziert wird. Die entsprechende Implementation Yon ,,*Newl1sData"* lautet:

```
PROCEDURE NewTlsData(id:LongWord;Data:Pointer);
Begin
  {$IFDEF OS2} //bedingte Compilierung
  //Version fur OS/2 - Workaround fur nicht existente API
  TlsData[id]:=Data;
  {$ELSE}
  //Version fur Windows 95 und Windows NT - direkter API-Aufruf
  TlsSetValue(TlsIndex,Data);
{$ENDIF}
End;
```

Fur Zugriffe auf TLS-Variablen erzeugt der Compiler speziellen Code. Dabei wird die pro Thread eindeutige Adresse einer solchen Variablen durch einen Aufruf der Prozedur *"GetTlsVar"* ermittelt. Fur das Programm

```
THREADVAR x: Integer;

PROCEDURE TestThread;
BEGIN
  x:=x+l;
End;
```

würde für die Anweisung "x:=x+ I" folgender Assemblercode erzeugt:

```
MOV EAX.OFFSET(TEST.X)
//GetTlsVar liefert die Adresse der TLS-Variablen in EAX CALLN32 SYSTEM.GetTlsVar
MOVSX EAX.WORD PTR [EAX]
ADD EAX.l
PUSH EAX
MOV EAX.OFFSET(TEST.X)
//GetTlsVar liefert die Adresse der TLS-Variablen in EAX CALLN32 SYSTEM.GetTlsVar
POP ECX
MOV [EAX]. CX
```

Die Funktion *"GetTlsVar"* ermittelt die Adresse der TLS-Variablen. Diese Adresse ist für jeden Thread eindeutig. Bei der Ermittlung dieser Adresse muss für OS/2 wiederum ein Kompromiss gewählt werden, da derartige Funktionen erst ab OS/2 V4.0 verfügbar sind. Aus Geschwindigkeitsgründen ist *"GetTisVar"* komplett in Assembler implementiert. Die entsprechende Implementation für *"GetTisVar"* unter OS/2 lautet also:

```
//Version fur OS/2
SYSTEM.GetTlsVar PROC NEAR32 //Register retten
  PUSH EDI
  PUSH EBX

  //Id des aktuellen Threads bestimmen
  MOV EDI, $OC
  MOV EBX,FS:[ED I]
  MOV EBX,[EBX]
  //Offset innerhalb des TLS-Pools
  MOV EAX,[EAX]
  DEC EBX
  MOV EDI,OFFSET(SYSTEM.TlsData)
  LEA EDI,[EDI+EBX*4]
  CMP DWORD PTR [EDI],0
  JNE !TlsOk

  //es handelt sich um den Hauptthread
  MOV EDI,OFFSET(SYSTEM.TlsData)

!TlsOk:
  //Anfangsadresse des TLS addieren
  ADD EAX.[EDI]

  //Register restaurieren
  POP EBX
  POP ED I
  RETN32
SYSTEM.GetTlsVar ENDP
```

Windows 95 und Windows NT ist die Implementation von "GetTlsVar" etwas einfacher, da unter diesen Systemen entsprechende Systemdienste zur Verfügung stehen. Die entsprechende Implementation ist:

```
//Version fur Windows 95 und Windows NT SYSTEM.GetTlsVar PROC NEAR32
//Register retten
  PUSH EBX
  PUSH ECX
  PUSH EDX
  PUSH ESI
  PUSH EDI

  PUSH EAX

  //TLS Startadresse fur aktuellen Thread holen
  PUSH DWORD PTR SYSTEM.TlsIndex
  CALLN32 KERNEL32, 'TlsGetValue'
  CMP EAX, 0
  JNE !TlsOk
  //es handelt sich um den Hauptthread MOV
  EAX,SYSTEM.MainTls
!TlsOk:
  POP EBX
  //addiere TLS Startadresse
  ADD EAX, [EBX]

//Register restaurieren POP EDI
  POP ESI
  POP ED X
  POP ECX
  POP EBX
  RETN32
SYSTEM.GetTlsVar ENDP
```

Zum Zerstören eines Threads dient die Funktion „EndThread" aus der Unit "System". Ein Thread kann sich mit dieser Funktion selbst beenden und einen Ergebniswert zurück liefern:

```
PROCEDURE EndThread(ExitCode:LongInt);
BEGIN
  {$IFDEF OS2} //bedingte Compilierung
  //Version fur OS/2
  DosExit(EXIT_THREAD,ExitCode);
  {$ELSE}
  //Version fur Windows 95 und Windows NT
  ExitThread(ExitCode);
  {$ENDIF}
END;
```

10.2.3Thread-Sicherheit der Laufzeitbibliothek

Beim Entwurf der Laufzeitbibliothek wurden globale Variablen nach Möglichkeit vermieden, um die in Abschnitt 7.2.2 dargestellten Probleme in Verbindung mit Threads zu vermeiden. Grundlegende Funktionen, wie etwa die Anforderung von Speicher, nutzen direkte API-Aufrufe. In der API der entsprechenden Systeme ist die Thread-Sicherheit gewährleistet. Eine weitere Absicherung von

konkurrierenden Threads kann durch Thread-lokale Variablen und die Verwendung von Semaphoren erreicht werden. Dafür stehen jeweils Funktionen der Laufzeitbibliothek bzw. der API des entsprechenden Systems zur Verfügung.

10.3 Ausnahmebehandlung

Als Multitaskingsysteme reagieren die Betriebssysteme Windows 95, Windows NT und OS/2 auf so genannte Ausnahmen (Exceptions). Ausnahmen sind unerwartete Situationen, wie etwa Schutzverletzungen, und können durch Applikationssoftware oder durch das Betriebssystem bzw. die Hardware ausgelost werden. In den meisten Fällen wird der verursachende Prozess beim Erkennen einer Ausnahme vom Betriebssystem beendet. Es ist jedoch möglich, in die Bearbeitung von Ausnahmen einzugreifen und so die Ausnahmebedingung zu lokalisieren und eventuell zu beheben. Object Pascal stellt hierfür die Schlüsselwörter "**TRY**", "**EXCEPT**", „**RAISE**" und „**FINALLY**" zur Verfügung. Ausnahmen werden in allen drei Systemen auf einer per-Thread Basis verwaltet. Dafür wird ein Stack von so genannten Exception-Handler eingerichtet. Exception-Handler sind Codeabschnitte, welche zur Behandlung yon Ausnahmen dienen. Der Exception-Handler an der Spitze dieses Stacks wird beim Auftreten einer Ausnahme als erster aufgerufen. In diesem Handler wird die Exception entweder behandelt oder an den nächsthöheren Handler weitergegeben. 1m untersten Exception-Handler wird der ProzeJ3 beendet und eine Fehlermeldung ausgegeben. In Object-Pascal ist dieses Verhalten völlig transparent. Bei Verwendung der Schlüsselwörter "**TRY**", „**EXCEPT**", „**RAISE**" und "**FINALLY**" wird der entsprechende Code automatisch vom Compiler generiert. Damit dieses Prinzip funktioniert, sind in der Laufzeitbibliothek einige Vorkehrungen zu treffen:

- In der Laufzeitbibliothek muJ3 eine Möglichkeit zur Darstellung einer Ausnahme existieren. Da Object-Pascal eine objektorientierte Sprache ist, wird hierfür eine Objektklasse entworfen, welche Ausnahmen im Sinne eines abstrakten Datentypes kapselt.

- Die Laufzeitbibliothek sorgt für das dynamische Erzeugen und Vernichten von Instanzen der Exception-Klasse. Bei Bedarf soll der Nutzer auch direkten Zugriff auf diese Instanz haben können.

- Die Laufzeitbibliothek registriert einen Basis-Exception-Handler. In diesem Handler soll der Prozess mit einer aussagekräftigen Fehlermeldungen beendet werden, falls kein anderer Handler die Ausnahme behandelt hat.

10.3.1Darstellung von Ausnahmen in der Laufzeitbibliothek

Die Darstellung vn Ausnahmen in der Laufzeitbibliothek erfolgt durch die Basisklasse „*Exception"* in der Unit "System". Vordefinierte und nutzerdefinierte Ausnahmen müssen von dieser Basisklasse abgeleitet sein.

```
TYPE Exception=CLASS
  PRIVATE //private Felder und Methoden
    FMessage:PString;
    FUNCTION GetMessage:STRING;
    PROCEDURE SetMessage(CONST Value:STRING);
  PUBLIC  //oeffentliche Felder und Methoden
    {$IFDEF OS2}  //bedingte Compilierung
    ReportRecord:EXCEPTIONREPORTRECORD;
    {$ELSE}
    ReportRecord:EXCEPTION_RECORD;
    {$ENDIF}
    ExcptNum:LongWord    //Nummer der Ausnahme
    CameFromRTL:Boolean  //Boolean fuer RTL Ausnahme
    Nested:Boolean       //Ausnahme in einer Ausnahme
    ExcptAddr:POINTER    //Adresse der Aushnahme
    {$IFDEF OS2}
    RegistrationRecord:EXCEPTIONREGISTRATIONRECORD;
    ContextRecord:CONTEXTRECORD;
    {$ELSE}
    ContextRecord:CONTEXT
    {$ENDIF}
    //Konstruktor Methode
    CONSTRUCTOR Create(CONST Msg:String);Virtual;
    //Destruktor Methode
    DESTRUCTOR Destroy;OVERRIDE;
  PUBLIC  //oeffentliche Eigenschaften
    PROPERTY Message:STRING read GetMessage write SetMessage;
    PROPERTY MessagePtr:PString read FMessage;
END;
```

Von dieser Basisklasse werden einige vordefinierte Ausnahmen abgeleitet. Diese Ausnahmen werden vom Betriebsystem generiert, wenn das System oder die Hardware eine Ausnahme entdeckt. Instanzen dieser Klassen werden von der Laufzeitbibliothek automatisch erzeugt.

```
TYPE
  EProcessorException=CLASS(SysException);  //Basisklasse fuer CPU Ausnahmen
  EFault = CLASS(EProcessorException);      //Basisklasse fuer Faults
  EGPFault = CLASS(EFault);                 //Schutzverletzung (GPF)
  EStackFault = CLASS(EFault);              //Stackfehler
  EPageFault = CLASS(EFault);               //Pagefehler
  EInvalidOpCode = CLASS(EFault);           //ungueltiger Opcode
  EBreakpoint = CLASS(EProcessorException); //Haltepunkt
  ESingleStep = CLASS(EProcessorException); //Einzelschritt

  EIntError = CLASS(SysException);          //Basisklasse Integer-Ausnahmen
  EDivByZero = CLASS(EIntError);            //Division durch 0
  ERangeError = CLASS(EIntError);           //Bereichsgrenzenfehler
  EIntOverflow = CLASS(EIntError);          //Integer-Ueberlauf

  EMathError = CLASS(SysException);         //Basisklasse FPU Ausnahmen
  EInvalidOp = CLASS(EMathError);           //ungueltiger FPU Opcode
  EZeroDivide = CLASS(EMathError);          //Division durch 0
  EOverflow = CLASS(EMathError);            //FPU Ueberlauf
  EUnderflow = CLASS(EMathError);           //FPU Unterlauf
```

Zusätzlich definiert die Laufzeitbibliothek einige vordefinierte Ausnahmen, welche beim Eintreffen der entsprechenden Bedingung von der Laufzeitbibliothek ausgelöst werden:

```
TYPE
  EOutOfMemory = CLASS(SysException);       //Speichermangel
  EInvalidPointer = CLASS(SysException);    //ungueltiger Zeiger
  EInvalidHeap    = CLASS(SysException);    //ungueltiger Heap

  EInOutError = CLASS(SysException)         //Ein-Ausgabe Fehler
    PUBLIC
      ErrorCode: Integer;
  END;
  EFileNotFound=CLASS(EInOutError);         //Datei nicht gefunden
  EInvalidFileName=CLASS(EInOutError);      //Ungueltiger Dateiname
  ETooManyOpenFiles=CLASS(EInOutError);     //zu viele offene Dateien
  EAccessDenied=CLASS(EInOutError);         //Dateizugriff verweigert
  EEndOfFile=CLASS(EInOutError);            //Dateiende erreicht
  EDiskFull=CLASS(EInOutError);             //Diskette//Platte voll
  EInvalidInput=CLASS(EInOutError);         //ungueltige Eingabe
```

Jede auftretende Ausnahme wird zunächst von der Prozedur *„ExcptHandler"* aus der System-Unit bearbeitet. Diese Prozedur erzeugt die Instanzen der Klasse und verteilt die Ausnahmen auf die einzelnen Handler. Zusammen mit jeder Instanz einer Ausnahme wird eine Zeichenkette gespeichert, welche eine kurze Beschreibung der Ausnahme enthält. Diese Zeichenkette wird zum Beispiel vom Basis-Exception-Handler ausgegeben, wenn kein Handler die Ausnahme bearbeitet hat. In diesem Fall beendet die Laufzeitbibliothek den Prozess zwangsweise. Der folgende Codeausschnitt zeigt einen Teil der Implementation der Prozedur *„ExcptHandler"* für das OS/2 Zielsystem. Die Implementation fur Windows 95 und Windows NT ist ahnlich.

```
BEGIN
  WITH p3 DO
    Registerinfo:= #13#10'at CS:EIP  ='+ToHex(ctx_SegCs )+':'+
                   ToHex(ctx_RegEip);
  IF POINTER(p2.ObjectType)=NIL THEN {Kein Objekt da}
    BEGIN
      CASE p1.ExceptionNum OF  // pruefe Ausnahen
        XCPT_BREAKPOINT:
          p2.ObjectType:=EBreakPoint.Create('Breakpoint exception
                         (EBreakPoint) occured'+RegisterInfo);
        XCPT_ACCESS_VIOLATION:
          p2.ObjectType:=EGPFault.Create('Access violation exception
                         (EGPFault) occured'+RegisterInfo);
        ...  //weitere Ausnahemn
        ELSE  //unbekannte Ausnahme, weiterreichen an das System
              //Don't handle
          BEGIN
            Result:=XCPT_CONTINUE_SEARCH;
            exit;
          END;
      END; //case
    END;
  p2.ObjectType.ReportRecord:=p1;
  p2.ObjectType.RegistrationRecord:=p2;
  p2.ObjectType.ExcptNum:=p1.ExceptionNum;
  p2.ObjectType.ExcptAddr:=POINTER(p3.ctx_RegEIP);
  p2.ObjectType.ContextRecord:=p3;
  longjmp(p2.jmpWorker,LONGWORD(p2.ObjectType));
END;
```

Der Aufruf des eigentlichen Handlers erfolgt innerhalb des Aufrufs der Funktion *"longjmp"*. Diese Funktion arbeitet ähnlich zur C-Bibliotheksfunktion *"longjmp"* und erlaubt einen interprozeduralen Sprung. Die Registrierung der Sprungadresse erfolgt mit der Registrierung des Handlers durch das Schlüsselwort "**TRY**". Dazu wird das folgende Codebeispiel betrachtet:

```
TRY
  p^:=1;
EXCEPT
  Writeln('Fehler beim Zugriff auf p !');
  RAISE; //Exception erneut ausloesen
END;
```

Dafür wird unter OS/2 vereinfacht folgender Assemblercode erzeugt (für Windows ist der generierte Code ähnlich):

```
//TRY
//Exception Handler registrieren
MOV EAX,@SYSTEM.ExcptHandler  //Diese Prozedur wird zuerst aufgerufen
MOV [EBP-40J,EAX
LEA EAX. [EBP-44 ]
PUSH EAX
MOV AL,l
CALLDLL DosCalls, 354         //DosSetExceptionHandler API
ADD ESP,4
MOV DWORD PTR [EBP-36].O
LEA EAX, [EBP-32)
PUSH EAX
CALLN32 SYSTEM.SetJmp         //interprozeduralen Sprung setzen
CMP EAX                       //Exception eingetreten ??
JNE !2
//p^:=l
MOV EAX,Test.P
MOV DWORD PTR [EAX),l
//Exception Handler loeschen, keine Exception aufgetreten
LEA EAX,[EBP-44]
PUSH EAX
MOV AL,l
CALLDLL DosCalls, 355         //DosUnsetExceptionHandler API
ADD ESP,4
JMP !3                        //Exception Handler ueberspringen
//EXCEPT
! 2:                          //Start des eigentlichen Exception-Handlers

PUSH EAX                      //Speichern derException-Instanz
... //Text ausgeben
//RAISE;
CALLN32 SYSTEM.RaiseExceptionAgain //erneutes Ausloeschen der Ausnahme
!3:
... //weiteres Programm
```

Das Schlüsselwort "**RAISE**" bewirkt hierbei den Aufruf des nächsthöheren Handlers. Durch Verschachtelung von "**TRY**" ... "**EXCEPT**" Blöcken ist auch eine hierarchische Ausnahmebehandlung möglich.

11 Kapitel 9
Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurde ein leistungsfähiger 32-Bit Pascal-Compiler entworfen und implementiert. Ausgehend von einem vorhandenen Design und existierenden Quelltexten wurden die Mangel des alten Compilers beseitigt und eine Reihe neuer Möglichkeiten implementiert. Der entstandene Compiler ,,PASC" verfügt über alle Sprachkonstrukte und Eigenschaften der Sprache Object-Pascal inklusive:

- Objektorientierung in 2 Varianten (**OBJECT**, **CLASS**[27],...)
- Strukturierte Ausnahmebehandlung (**TRY**, **EXCEPT** ...)
- Verschiedene AufTufinodeIle (C,Pascal)
- Untersttitzung fur Multithreading **(THREADVAR**,BeginThread ...)

Über die Betrachtung der verschiedenen Möglichkeiten zur grundsätzlichen Strukturierung eines Compilers wurde eine Variante fur die Implementation ausgewählt. Der Entwurf des Compilers und der Zwischensprache erfolgte nach Kriterien der Effizienz und der möglichst großen Wiederverwendbarkeit vorhandener Algorithmen und Quelltexte.

11.1 Was wurde erreicht?

Das vorhandene Quelltextmaterial zur Vorversion des Compilers wurde grundsätzlich überarbeitet, neu strukturiert und erweitert. Dabei wurden wesentliche, im alten Compiler nicht enthaltene, Komponenten neu entworfen und implementiert. Durch eine strikte Trennung yon Syntaxanalyse und Codegenerierung über einen Zwischencode wurde eine Entkopplung der beiden Komponenten erreicht. Dies ermöglichte eine effiziente, sprachunabhängige Optimierung auf Zwischensprachenebene. Durch diese Optimierung wurde die Qualität des vom Compiler erzeugten Maschinencodes erhöht.

Die Vorversion des Compilers war nur auf OS/2 Systemen lauffähig. Durch eine saubere Strukturierung und Modularisierung der Compileraufgaben konnte der Compiler ohne wesentlichen Aufwand auch nach Windows 95 und Windows NT portiert werden. Dabei wurde im wesentlichen nur die Codeausgabe angepasst. Der gesamte Compiler wurde nach der Frontend-Backend Compilerstruktur entworfen und implementiert.

Der Test der Compiliergeschwindigkeit erfolgte durch ein Programm, welches aus circa 16000 Quelltextzeilen besteht. Die Messung wurde unter OS/2 durchgeführt, da der alte Compiler nur auf diesem System lauffähig war. Dabei konnte festgestellt werden, dass sich die Compiliergeschwindigkeit in etwa verdoppelt hat (siehe Tabelle 12). Dies ist insofern interessant, da durch die explizite Erzeugung eines Zwischencodes eigentlich mit erhöhtem Zeitaufwand zu rechnen gewesen ware. Die Wirkung der im neuen Compiler verwendeten effizienteren Algorithmen bewirkt also eine wesentliche Steigerung der Compiliergeschwindigkeit.

---

27 Das neue Objektmodell von Object-Pascal (Schlüsselwort CLASS) lehnt sich stark an C++ an. Aus Kompatibilitätsgründen existiert das alte Objektmodell (Schlüsselwort OBJECT) weiterhin.

Compiler	*Compilierzeit in Sekunden*
alter Compiler: Speed-Pascal 1.5	15 bis 16
neuer Compiler: PASC	8 bis 9

Tabelle 12: Compilierzeiten fur 16000 Zeilen Quelltext mit dem alten und dem neuen Compiler gemessen auf einem Pentium 133 mit 32 ME Hauptspeicher unter OS/2

11.2 Grundsätzliche Überlegungen zur weiteren Verbesserung des Compilers

Grundsätzlich verbesserungswürdig sind die Optimierungsmöglichkeiten. Weitere Optimierungen konnten in etwa eine Speicherung von Variablen in Registem sowie eine interprozedurale Optimierung betreffen. Dafür ist unter Umständen eine globale Datenflussanalyse notwendig. Ausserdem ist zu überlegen, ob es sinnvoll ist, aus dem generierten Zwischencode eine weitere Zwischendarstellung abzuleiten. So wäre eine Optimierung von Schleifen (Loop-unrolling) und eine weitere Verbesserung der Erkennung von gemeinsamen Teilausdrücken wünschenswert.

Als weitere Verbesserung erscheint die Generierung von Standard-Linkformaten der verwendeten Betriebssysteme. Dadurch wäre eine Verwendung von Fremdbibliotheken mit dem Pascal-Compiler möglich. Ausserdem konnte der Linker des jeweiligen Betriebssystems verwendet werden.

12 Anhang A
Die Sprachbeschreibung von Object-Pascal

programm ::= [kopf] block '.'
kopf ::= '**PROGRAMM**' bezeichner [programparams] ';'.
| '**UNIT**' bezeichner ';'
| '**LIBRARY**' bezeichner ';'
unitkopf ::= '**INTERFACE**' [vereinbteil] '**IMPLEMENTATION**'.
programmparams ::= '(' bezeichner {',' bezeichner } ')'.
block ::= { vereinbteil } anwteil.
vereinbteil ::= typevereinb
| vareinb
| constvereinb
| labelvereinb
| procvereinb
| funcvereinb
| methodvereinb
| usesvereinb
| exportsvereinb.
procvereinb ::= prozedurkopf ';' block ';'.
funcvereinb ::= funktionskopf ';' block ';'.
methodvereinb ::= methodkopf ';' bloci ';'.
anwteil ::= verbund-anw.
typevereinb ::= '**TYPE**' typedekl {typedekl}.
typdekl ::= bezeichner '=' typ ';'.
typ ::= einf-typ
| string-typ
| strukt-typ
| pointer-typ
| proc-typ
| variant-typ
| typbezeichner.
einf-typ ::= subrange-typ
| aufz-typ-
| typbezeichner.
subrange-typ ::= konstante '..' konstante.
aufz_typ ::= '(' bezeichnerliste ')'.
string_typ ::= '**STRING**' ['[' konstante ']'].
| '**CSTRING**' ['[' konstante ']'].
strukt-typ ::= ['**PACKED**'] styp.
styp ::= array-typ
| record-typ
| class-typ
| classref-typ
| set-typ
| file-typ
array-typ ::= '**ARRAY**' '[' index-typ { ',' index-typ } ']' '**OF**' typ.

index-typ ::= einf-typ.
record-typ ::= '**RECORD**' [feldliste] [';'] '**END**'.
feldliste ::= fixed-rec [';' variant-rec]
| variant-rec
fixed-rec ::= bezeichnerliste ':' typ { ';' bezeichnerliste ':' typ }.
variant-rec ::= '**CASE**' [bezeichner ':'] tag-typ '**OF**' variant { ';' variant }.
tag-typ ::= typbezeichner.
class-typ ::= classtyp
| objecttyp
classtyp ::= '**CLASS**' [parent] [classdef '**END**'].
objecttyp ::= '**OBJECT**' [parent] [classdef '**END**'].
parent ::= '(' typbezeichner ')'.
classdef ::= { [sichtbarkeit] komponenten ';' }
sichtbarkeit ::= '**PUBLISHED**'
| '**PUBLIC**'
| '**PROTECTED**'
| '**PRIVATE**'
komponenten ::= felddef
| methoddef
| propdef.
felddef ::= bezeichnerliste ':' typ.
methoddef ::= methodkopf [';' methoddirectives] [';' '**ABSTRACT**']'
methoddirectives ::= '**VIRTUAL**' [ganzzahl].
| '**DYNAMIC**'.
| '**MESSAGE**' ganzzahl.
methodkopf ::= constructorkopf
| destructorkopf
| ['**CLASS**'] prozedurekopf
| ['**CLASS**'] funktionskopf.
constructorkopf ::= '**CONSTRUCTOR**' bezeichner [formal-param-list].
formal-param-list ::=
destructorkopf ::= '**DESTRUCTOR**' bezeichner [formal-param-list].
prozedurekopf ::= '**PROZEDURE**' bezeichner [formal-param-list].
unktionskopf ::= '**FUNCTION**' bezeichner [formal-param-list] funcresult.
funcresult ::= ';' typbezeichner.
propdef ::= '**PROPERTY**' bezeichner [prop-interface] prop-spec
prop-interface ::= [prop-params] ':' typbezeichner.
prop-params ::= '[' pram-dekl { ';' param-dekl } ']'.
param-dekl ::= bezeichner ':' typ
prop-spec ::= [read-spec] [write-spec] [stored-spec] [';' '**DEFAULT**'].
read-spec ::= '**READ**' bezeichner.
write-spec ::= '**WRITE**' bezeichner.
stored-spec ::= '**STORED**' bezeichner.
classref-typ ::= '**CLASS**' '**OF**' typbezeichner.
set-typ ::= '**SET**' '**OF**' typ.
file-typ ::= '**FILE**' ['**OF**' typ].
pointer-typ ::= '^' typbezeichner.
proc-typ ::= '**PROCEDURE**' [formal-param-list]
| '**FUNCTION**' [formal-param-list] ':' resulttyp.
resulttyp ::= typbezeichner'

| '**STRING**'
| '**CSTRING**'
| '**FILE**'.

typbezeichner ::= bezeichner.

varvereinb ::= '**VAR**' vardekl {vardekl }
| '**THREADVAR**' vardekl {vardekl}.
vardekl ::= bezeichnerliste ':' typ [absolut] ';'.
absolut ::= '**ABSOLUTE**' varref.

constvereinb ::= '**CONST**' konstantendekl {konstantendekl}.
konstantendekl ::= bezeichner '=' konstante ';'
| bezeichner ':' typ '=' typ-konstante.
typ-konstante ::= konstante
| adress-konstante
| array-konstante
| record-konstante
| proc-konstante.
adress-konstante ::= '@' bezeichner
| '**NIL**'
array-konstante ::= '(' typ-konstante { ',' typ-konstante } ')'.
record-konstante ::= '(' rec-konst-list ')'.
rec-konst-list ::= bezeichner ':' typ-konstante { ';' bezeichner ':' typ-konstante }.
proc-konstante ::= bezeichner
| '**NIL**'
labelvereinb ::= '**LABEL**' label { ',' label }
usesvereinb ::= '**USES**' bezeichner { ',' bezeichner }
importsvereinb ::= '**IMPORTS**' { importslist } '**END**'.
importslist ::= prozedurkopf ';' zeichenkette exp-spec ';'.
::= funktionskopf ';' zeichenkette exp-spec ';'.
exportsvereinb ::= '**EXPORTS**' exportslist { ',' exportslist }
exportslist ::= bezeichner [exp-spec]
exp-spec ::= '**INDEX**' ganzzahl.
| '**NAME**' zeichenkette.

anw-list ::= anweisung { ';' anweisung }.
anweisung ::= [label ';'] einf-anweisung
| [label ';'] strukt-anweisung.
einf-anweisung ::= zuweisung
| prozeduraufruf
| goto-anw.
zuweisung ::= varref ':=' ausdruck.
| funcbezeichner ':=' ausdruck.
funcbezeichner ::= bezeichner.
prozeduraufruf ::= bezeichner [parameterliste]
| varref [parameterliste].
goto-anw. ::= '**GOTO**' label.
strukt-anweisung ::= verbund-anw.
| cond-anw.
| rep-anw.

```
                        |       with-anw.
                        |       except-anw.
verbund-anw  ::=    'BEGIN' anw-list 'END'.
cond-anw     ::=    if-anw
                        |       case-anw.
if-anw       ::=    'IF' ausdruck 'THEN' anweisung [ 'ELSE' anweisung ]
case-anw     ::=    'CASE' ausdruck 'OF' case { ';' case } [ else ] [ ';' ] 'END'.
case    ::=    case-konst ':' anweisung.
case-konst   ::=    konstante [ '..' konstante ] { ';' konstante [ '..' konstante ] }.
else    ::=    'ELSE' anweisung.
rep-anw      ::=    repeat-anw
                        |       while-anw
                        |       for-anw.
repeat-anw   ::=    'REPEAT' anw-list 'UNTIL' ausdruck.
while-anw    ::=    'WHILE' ausdruck 'DO' anweisung.
for-anw      ::=    'FOR' varref ':='  ausdruck todownto 'DO'  anweisung.
todownto     ::=    'TO' ausdruck
                        |       'DOWNTO' ausdruck
with-anw     ::=    'WITH' varref { ',' varref } 'DO' anweisung.
except-anw   ::=    raise-anw
                        |       try-anw.
raise-anw    ::=    'RAISE' [ ausdruck [ 'AT' ausdruck ] ].
try-anw      ::=    'TRY' anw-list 'EXCEPT' excptblock 'END'.
                        |       'TRY' anw-list 'FINALLY' finallyblock 'END'.
excptblock   ::=    anw-list
                        |       handler { ';' handler } [ 'EXCEPT' anw-list ].
handler      ::=    'ON' [ bezeichner ';' ] typbezeichner 'DO' anweisung.
finallyblock ::=    anw-list.

ausdruck     ::=    einf-ausdruck [ vergl-op einf-ausdruck ].
vergl-op     ::=    '<' | '<=' | '>' | '>=' | '=' | '<>' | 'IN' | 'IS'.
einf-ausdruck ::=   term { addoperand term }.
addoperand   ::=    '+' | '-' | 'OR' | 'XOR'.
term    ::=    faktor { muloperand faktor }.
muloperand   ::=    '*' | '/' | 'DIV' | 'MOD' | 'AND' | 'SHL' | 'SHR' | 'AS'.
faktor  ::=    varref
                        |       fak-konstante
                        |       '(' ausdruck ')'
                        |       'NOT' faktor
                        |       sign faktor
                        |       funktionsaufruf
                        |       set-constructor
                        |       ausdr-typcast
                        |       adress-faktor.
fal-konstante  ::=  ganzzahl
                        |       zeichenkette
                        |       konstantenbezeichner
                        |       'NIL'
konstantenbezeichner        ::=     bezeichner.
funktionsaufruf     ::=     bezeichner [ parameterliste ].
```

```
                        |       varref [ parameterliste ].
parameterliste      ::=     '(' parameter{ ',' parameter } ')'.
parameter       ::=     varref
                        |       ausdruck.
set-constructor     ::=     '[' [ members { ',' members } ] ']'.
members         ::=     ausdruck [ '..' ausdruck ].
ausdr-typecast      ::=     typbezeichner '(' ausdruck ')'.
adress-faktor   ::=     '@' varref
                        |       '@' bezeichner.

varef   ::=     bezeichner { qualifier }
                        |       var-typecast { qualifier }
                        |       expression qualifier { qualifier }
qualifier       ::=     index
                        |       feld-qual
                        |       '^'.
index   ::=     '[' expression { ',' expression } ']'
feld-qual       ::=     '.' bezeichner.
var-typecast    ::=     typbezeichner '(' varref ')'.

konstante       ::=     ausdruck.
                        |       'NIL'.
label   ::=     ziffernfolge
                        |       bezeichner.
qualbezeichner      ::=     bezeichner '.' bezeichner { '.' bezeichner }.
bezeichnerliste     ::=     bezeichner { ',' bezeichner }.
bezeichner      ::=     buchstabe { buchstabe | ziffer | '_' }.
                        |       '_' { buchstabe | ziffer | '_' }.
zahl    ::=     [sign] ganzzahl
                        |       [sign] realzahl.
sign    ::=     '+' | '-'.
ganzzahl        ::=     ziffernfolge | '$' hexziffernfolge.
realzahl        ::=     ziffernfolge [ '.' ziffernfolge ] [skalierungsfaktor].
skalierungsfaktor       ::=     'E' [ vorzeichen ] ziffernfolge.
                        |       'e' [ vorzeichen ] ziffernfolge.
hexziffernfolge     ::=     hexziffer {hexziffer}.
ziffernfolge    ::=     ziffer {ziffer}.
zeichenkette    ::=     string { zeichenkette }
                        |       kontrollstring { zeichenkette }.
string  ::=     ''' { stringzeichen } '''.
kontrollstring ::=      '#' ganzzahl { '#' ganzzahl }.
stringzeichen  ::=      chr(0) | chr(1) | ... | chr(12) | chr(14) | chr(15) | ...| chr(38)
                        |       chr(40) | chr(41) | ... | chr(255).
buchstabe       ::=     'A' | 'B' | ... | 'Z' | 'a' | 'b' | ... | 'z'.
hexziffer       ::=     ziffer | 'A' | 'B' | ... | 'F' | 'a' | 'b' | ... | 'f'.
ziffer  ::=     '0' | '1' | ... | '9'.
```

13 Anhang B
Die vom Scanner gelieferten abstrakten Terminalsymbole

Die abstrakten Terminalsymbole, welche vom Scanner geliefert werden, unterteilen sich in Schlüsselwörter, Operatoren, Spezialsymbole, Konstanten und Bezeichner. Die entsprechenden Definitionen sind im Modul SYM.H abgelegt.

13.1 Schlüsselwörter:

Symbol	Schliisselwort
S_ABSOLUTE	Absolute
S_AND	And
S_ARRAY	Array
S_ASM	Asm
S_ASSEMBLER	Assembler
S_BEGIN	Be<rin
S_CASE	Case
S_CLASS	Class
S_CONST	Const
S_CONSTRUCTOR	Constructor
S_CSTRING	CStrin"
S_DESTRUCTOR	Destructor
S_DIY	Diy
S_DO	Do
S_DOWNTO	DownTo
S_ELSE	Else
S_END	End
S_EXCEPT	Excent
S_EXPORTS	Exnorts
S_FILE	File
S_FINALLY	Finally
S_FINALIZATION	Finalization
S_FOR	For
S_FORWARD	Forward
S_FUNCTION	Function
S_GaTO	Goto
S_IF	If
S_IN	In
S_IMPLEMENTATION	Imnlementation

Symbol	Schliisselwort
S_IMPORTS	ImDorts
S_INITIALIZATION	Initialization
S_INTERFACE	Interface
S_LABEL	Label
S_LIBRARY	Librarv
S_MOD	Mod
S_NIL	Nil
S_NOT	Not
S_OBJECT	Object
S_OF	Of
S_OR	Or
S_PACKED	Packed
S_PROCEDURE	Procedure
S_PROPERTY	Property
S_PROGRAM	Program
S_RECORD	Record
S_REPEAT	Repeat
S_SET	Set
S_SHL	Shl
S_SHR	Shr
S_STRING	String
S_THEN	Then
S_THREADVAR	ThreadVar
S_TO	To
S_TRY	Try
S_TYPE	Type
S_UNIT	Unit
S_UNTIL	Until
S_USES	Uses
S_VAR	Var
S_WIDLE	While
S_WITH	With
S_XOR	Xor

13.2 Operatoren

Symbol	Zeichen	Beschreibung
S_AddrOp	@	Adress-Operator
S_Assign	:=	Zuweisungs-Operator
S_Divide	/	Division

Symbol	Zeichen	Beschreibung
S_E	=	Gleich
S_NE	<>	Ungleich
S_G	>	Größer
S_GE	>=	Größer oder gleich
S_L	<	Kleiner
S_LE	<=	Kleiner oder gleich
S_Minus	-	Minus
S_Plus	+	Plus
S_Times	*	Multiplikation

13.3 Spezialsvrnbole

Symbol	Zeichen	Beschreibung
S_Anow	^	Arrow
S_Colon	:	Doppelpunkt
S_Comma	,	Komma
S_LBRAC	[	eckige Klammer auf
S_LPAREN	(	runde Klammer auf
S_Period	,	Bereich
S_Point	.	Punkt
S_RBRAC	]	ckige Klammer zu
S_RPAREN	)	runde Klammer zu
S_Semicolon	;	Semikolon

13.4 Konstanten

Symbol	Beschreibung
S_FloatZahI	Fließkommakonstante
S_STRING_Text	Charakter- oder Zeichenkettenkonstante
S_ZahI	ganzzahIige Konstante

13.5 Bezeichner

Symbol	Beschreibung
S_ClassFunktion	eine Klassenfunktion (Methode) eines Objektes
S_ClassProzedur	eine Klassenprozedur (Methode) eines Objektes

Symbol	Beschreibung
S_Destruktor	ein Destruktor eines Objektes
S_Funktion	eine Pascal Funktion
S_Konstante	eine deklarierte oder vordefinierte Konstante
S_Konstruktor	ein Konstruktor eines Objektes
S_Name	ein bisher nicht deklarierter Bezeichner
S_Name_FORWARD	ein Bezeichner welcher mittels „Forward" deklariert wurde
S_ObjFunktion	eine Funktion (Methode) eines Objektes
S_ObiProzedur	eine Prozedur (Methode) eines Objektes
S_Property	eine Property (Eigenschaft) eines Objektes
S_Prozedur	eine Pascal Prozedur
S_StdProc	eine Pascal Standardprozedur
S_StdFunc	eine Pascal Standardfunktion
S_TybBez	ein deklarierter oder vordefinierter Typbezeichner
S_UnitBez	ein Bezeichner für eine Unit
S_Variable	eine deklarierte Variable

14 Anhang C
Die vordefinierten Bezeichner von Object-Pascal

In den folgenden Tabellen sind alle von Object-Pascal vordefinierten Bezeichner aufgelistet. Einige spezifische Erweiterungen (wie etwa der Datentyp "CString") sind ebenfalls enthalten.

14.1 Standardtypen:

Typ(en)					***Beschreibung***
ShortInt	Integer	LongInt			vorzeichenbehaftete Zahlen
Byte	Word	LongWord			vorzeichenlose Zahlen
Real	Single	Double	Extended	Comp	Fließkommatypen
Boolean	ByteBool	WordBool	LongBool		Wahrheitstypen
String	CString	ShortString	AnsiString		Zeichenkettentypen
File	Text				Datentypen
Pointer					Zeigertyp
Char					Zeichentyp
Variant					Variant-Typ

14.2 Standardkonstanten

Konstante(n)		***Beschreibung***
True	False	Wahrheitswerte wahr und falsch
NIL		Leerer Zeiger
PI		Wert von π (3,1415...)

14.3 Standardprozeduren und -funktionen

Prozedur(en) und oder Funktion(en)				***Beschreibung***
Addr				Adressbestimmung
New	Dispose			Speicherallokierung/-freigabe
Succ	Pred			Vorgänger und Nachfolger
Chr	Ord			Zeichenkonvertierung
Write	WriteLn			Textausgabe
Read	ReadLn			Texteingabe
Sin	Cos	Tan	Cot	Winkelfunktionen

Prozedur(en) und oder Funktion(en)				***Beschreibung***
ArcSin	ArcCos	ArcTan	ArcCot	inverse Winkelfunktionen
Sinh	Cosh	Tanh	Coth	hyperbolisch Winkelfunktionen
Frac	Round	Int	Trunc	Zahlenkonvertierung
Inc	Dec			Inkrementieren/Dekrementieren
Str	Val			Zahlenkonvertierung
Ln	Lg	Lb		Arithmetische Funktionen
Exp	Sqr	Sqrt		Arithmetische Funktionen
Lo	Hi			Zahlenkonvertierung
Low	High			Grenzen von offenen Arrays
Include	Exclude			Mengenoperationen
Odd	Even			boolesche Funktionen
Concat	Length	ToStr	UpCase	Stringfunktionen
SizeOf				Bestimmen der Typgröße
Break	Continue	Exit	Fail	Programmsteuerung

15 Anhang D
Die Anweisungen und Operatoren der Zwischensprache

Die Befehle der Zwischensprache unterteilen sich in Anweisungen und Operatoren. Anweisungen haben Steuerungscharakter, während Operatoren nur innerhalb von Ausdrücken vorkommen. Für jede(n) Anweisung/Operator ist die Bedeutung des vom jeweiligen Knoten im Zwischencodebaum abgehenden linken und rechten Teilbaumes angegeben. Unter einem LValue sind alle Ausdrücke zu verstehen, welche auf der linken Seite einer Zuweisung in Pascal erlaubt sind. Ein RValue repräsentiert einen beliebigen Ausdruck. Operatoren treten nur innerhalb der Auswertung eines LValue's oder eines RValue's auf und liefern immer ein LValue bzw. ein RValue als Resultat. Unter "signed" soll eine Operation mit vorzeichenbehafteten Operanden verstanden werden, "unsigned" bezeichnet eine Operation mit vorzeichenlosen Operanden.

15.1 Anweisungen

Anweisung	***Bedeutung***	***Linker Teilbaum***	***rechter Teilbaum***
S_ASM	Assembler Anweisung	Assembler Befehl(e)	nicht benutzt
S_Assign	Zuweisung	LValue	RValue
S_BoolAssign	Zuweisung an einen Boolean	LValue	RValue
S_CASE	Case-Anweisung	Case-Variable	Case-Typ
S_ClassFunktion	Aufruf einer Klassenfunktion	Verweis auf Definition	Parameter
S_ClassProzedur	Aufruf einer Klassenprozedur	Verweis auf Definition	Parameter
S_Konstruktor	Aufruf einer Konstruktors	Verweis auf Definition	Parameter
S_Destruktor	Aufruf eines Destruktors	Verweis auf Definition	Parameter
S_GOTO	Sprung zu eine Marke	nicht benutzt	nicht benutzt
S_FOR	For-Anweisung	Laufvariable	Abbruchbedingung
S_Funktion	Aufruf einer Funktion	Verweis auf Definition	Parameter
S_IF	If-Anweisung	Bedingun2	nicht benutzt
S_LABEL	Definition einer SDrunQffiarke	nicht benutzt	nicht benutzt
S_ObjFunktion	Aufruf einer Objekt-Funktion	Verweis auf Definition	Parameter
S_ObjProzedur	Aufruf einer Objekt-Prozedur	Verweis auf Definition	Parameter
Parameter	Aufruf einer Prozedur	Verweis auf Definition	Parameter
S_SetAssign	Zuweisung an eine Menge	LValue	RValue
S_StdFune	Auftuf einer Standardfunktion	Verweis auf Definition	Parameter
S_StdProc	Auftuf einer StandardDrozedur	Verweis auf Definition	Parameter
S_TRY	Try-Anweisung	Exception- Variable	nicht benutzt
S_UNTIL	Repeat-Until Anweisung	Bedingung	nicht benutzt
S_VariantAssign	Zuweisung an einen Variant	LValue	RValue
S_WHILE	While-Anweisung	Bedingung	nicht benutzt

15.2 Darstellung von Variablen und Konstanten

Operator	*Bedeutung*	*linker Teilbaum*	*rechter Teilbaum*
S_CStringVariable	Variable vom Tvp CString	Verweis auf Symboltabelle	nicht benutzt
S_FloatVariable	Fließkommavariable	Verweis auf Symboltabelle	nicht benutzt
S_FloatZahl	Fließkommakonstante	Verweis auf Konstante	nicht benutzt
S_Konstante	ganzzahlige Konstante	Verweis auf Konstante	nicht benutzt
S_StringVariable	Variable vom Tvp STRING	Verweis auf Symboltabelle	nicht benutzt
S_STRING_Text	Stringkonstante	Verweis auf Konstante	nicht benutzt
S_Variable	Variable	Verweis auf Symboltabelle	nicht benutzt

15.3 arithmetische Operatoren für Festkommazahlen

Operator	*Bedeutung*	*linker Teilbaum*	*rechter Teilbaum*
S_AND	bitweises UND	RValue (1. Operand)	RValue (2. Operand)
S_IDiv	Division (ganzzahlig, signed)	RValue (1. Operand)	RValue (2. Operand)
S_IMod	Modulus (ganzzahlig, signed)	RValue (1. Operand)	RValue (2. Operand)
S_ITimes	Multiplikation (ganzzahlig, signed)	RValue (1. Operand)	RValue (2. Operand)
S_Minus	Subtraktion (ganzzahlig)	RValue (1. Operand)	RValue (2. Operand)
S_Negate	Vorzeichenwechsel	RValue	nicht benutzt
S_NOT	bitweises Negation	RValue	nicht benutzt
5_OR	bitweises ODER	RValue (1. Operand)	RValue (2. Operand)
S_Plus	Addition (ganzzahlig)	RValue (1. Operand)	RValue (2. Operand)
S_SHL	bitweises Linksverschieben	RValue (1. Operand)	RValue (2. Operand)
S_SHR	bitweises Rechtsverschieben	RValue (1. Operand)	RValue (2. Operand)
S_UDiv	Division (ganzzahlig, unsigned)	RValue (1. Operand)	RValue (2. Operand)
S_UMod	Modulus(ganzzahlig, unsigned)	RValue (1. Operand)	RValue (2. Operand)
S_UTimes	Multiplikation(ganzzahlig, unsigned)	RValue (1. Operand)	RValue (2. Operand)
S_XOR	bitweise XOR	RValue (1. Operand)	RValue (2. Operand)

15.4 arithmetische Operatoren für Fließkommazahlen

Operator	*Bedeutung*	*linker Teilbaum*	*rechter Teilbaum*
S_Divide	Division (Fließkomma)	RValue (1. Operand)	RValue (2. Operand)
S_FloatMinus	Subtraktion (Fließkomma)	RValue (1. Operand)	RValue (2. Operand)
S_FloatPlus	Addition (Fließkomma)	RValue (1. Operand)	RValue (2. Operand)
S_FloatTimes	Multiplikation (Fließkomma)	RValue (1. Operand)	RValue (2. Operand)

15.5 logische Operatoren

Operator	*Bedeutung*	*linker Teilbaum*	*rechter Teilbaum*
S BoolAnd	logisches UND	RValue (1. Operand)	RValue (2. Operand)
S BooiNot	logisches Negation	RValue	nicht benutzt
S BooiOr	logisches ODER	RValue (1. Operand)	RValue (2. Operand)
S BoolXor	logisches XOR	RValue (1. Operand)	RValue (2. Operand)

15.6 Vergleichsoperatoren

Operator	*Bedeutung*	*linker Teilbaum*	*rechter Teilbaum*
S_E	Test ob erster Operand gleich zweiter Operand	RValue (1. Operand)	RValue (2. Operand)
S_G	Test ob erster Operand größer als zweiter Operand (signed)	RValue (1. Operand)	RValue (2. Operand)
S_GE	Test ob erster Operand größer oder gleich als zweiter Operand (signed)	RValue (1. Operand)	RValue (2. Operand)
S_L	Test ob erster Operand kleiner als zweiter Operand (signed)	RValue (1. Operand)	RValue (2. Operand)
S_LE	Test ob erster Operand kleiner oder gleich als zweiter Operand (signed)	RValue (1. Operand)	RValue (2. Operand)
S_NE	Test ob erster Operand ungleich zweiter Operand	RValue (1. Operand)	RValue (2. Operand)
S_IS	IS-Operator	RValue (1. Operand)	RValue (2. Operand)
S_StringCmp	Stringvergleich	RValue	RValue
S_UL	Test ob erster Operand kleiner als zweiter Operand (unsigned)	RValue (1. Operand)	RValue (2. Operand)
S_UG	Test ob erster Operand größer als zweiter Operand (unsigned)	RValue (1. Operand)	RValue (2. Operand)
S_ULE	Test ob erster Operand kleiner oder gleich als zweiter Operand (unsigned)	RValue (1. Operand)	RValue (2. Operand)
S_UGE	Test ob erster Operand größer oder gleich als zweiter Operand (unsigned	RValue (1. Operand)	RValue (2. Operand)

15.7 Operatoren zur Konvertierung von Zahlenformaten

Operator	*Bedeutung*	*linker Teilbaum*	*rechter Teilbaum*
S_IConvert2	Erweitem auf 2 BVtelsigned1	RValue	nicht benutzt
S_IConvert4	Erweitem auf 4 Bvte (signed)	RValue	nicht benutzt
S_Shrink1	Konvertieren auf] BYte	RValue	nicht benutzt
S_Shrink2	Konvertieren auf2 Bvte	RValue	nicht benutzt
S_UConvert2	Erweitem auf2 BYte (unsigned)	RValue	nicht benutzt
S_UConvert4	Erweitem auf 4 B;;te (unsipned)-	RValue	nicht benutzt

15.8 Operatoren für Mengen

Operator	*Bedeutung*	*linker Teilbaum*	*rechter Teilbaum*
S_SetAdd	Mengenaddition	RValue (1. Operand)	RValue (2. Operand)
S_SetCompare	Mengenvergleich	RValue (1. Operand)	RValue (2. Operand)
S_SetMinus	Mengendifferenz	RValue (1. Operand)	RValue (2. Operand)
S_SetMul	Mengendurchschnitt	RValue (1. Operand)	RValue (2. Operand)
S_IN	Test auf Mengenzugehörigkeit	RValue (Menge)	RValue (2. Operand)

15.9 Spezielle Operatoren

Operator	*Bedeutung*	*linker Teilbaum*	*rechter Teilbaum*
S_AddrOo	AdressOperator	LValue	nicht benutzt
S_Arrow	Zeigerreferenz	RValue	nicht benutzt
S_ClassFunktion	Aufruf einer Klassenfunktion	Verweis auf Definition	Parameter
S_Funktion	Aufruf einer Funktion	Verweis auf Definition	Parameter
S_LBRAC	Feldzugriff	LValue	RValue (Index)
S_ObjFunktion	Aufruf einer Objektfunktion	Verweis auf Definition	Parameter
S_Point	Record-Qualifizierung	LValue	Offset
S_StdFunc	Aufruf einer Standardfunktion	Verweis auf Definition	Offset
S_StrAdd	Stringaddition	RValue (1. Operand)	RValue (2 Operand)
S_TypeCast	Explizite Typumwandlung	RValue	resultierender Typ

16 Literaturverzeichnis

[Ah090] Aho, A.v., Sethi, R., Ullmann, J.D., Compilerbau Teill und 2, Addison Wesley 1990.

[And89] Anders, J., Ein MODULA-2 Compiler fur Unix, Dissertation an der TV Karl-Marx-Stadt 1989.

[BDE93] Real-World Programming for OS/2, Blain, D., Delimon, K., English, J., SAMS Publishing 1993.

[Bor96] Borland International, Object-Pascal language guide, Dokumentation zu Borland ,,Delphi" Version 2.0, 1996.

[Bro77] Brown, P.1., Software Portability, Cambridge University Press, 1977.

[Chr56] Chromsky, N., Three models for the description oflanguage, IRE Trans. On Information Theory IT-2:3, 1956.

[Emm89] Emmelmann, H., Automatische Erzeugung effizienter Codegeneratoren, GMD Studien Nr. 158,1989.

[Ers58] Ershov, A.P., On programming of arithmetic operations, Comm. ACM Vol. I, No.8, 1958.

[EuI88] Eulenstein, M., Generierung portabler Compiler - Das portable System POCO, Informatik-Fachberichte 164, Springer Verlag 1988.

[Fra95] Fraser, C., Hanson, D., A retargetable C Compiler: Design and Implementation, The Be˛aminiCummings Publishing Company 1995.

[IBM94] IBM OS/2 32 Bit Object Module Format (OMF) and Linear eXecutable Module Format (LX), Draft 3, Boca Programming Center, Boca Raton, Florida, IBM DAP library volume I, August 1994.

[Job92] Jobst, F., Compilerbau: Von der Quel1e zum professionel1en Assemblertext, Hanser Verlag 1992.

[Kai94] Kaiser, U., Anders, J., Loos, A., Script zur Vorlesung "Grundlagen der Compilertechnik", TV Chennitz 1994.

[Kas90] Kastens, u., Übersetzerbau, Oldenbourg Verlag 1990.

[KIo91] KIoppel, B., Compilerbau am Beispiel der Programmiersprache SIMPL, Vogel Verlag 1991.

[Mic94] Microsoft Portable Executable and Common Object File Format Specification 4.1, Microsoft Developer Network, Microsoft August 1994

[MS94] The Microsoft Developer Network, Library October 1994, Microsoft 1994.

[Nau60] Naur, P., Report on the algorithmic language Algol 60, Comm. ACM 3, 1960

[OS2/95] OS/2 Warp Engine, Pascal Entwicklung unter OS/2, Sonderheft der OS/2 Inside, Februar 1995

[OS2/96] OS/2 Spezial, Speed statt Turbo - Speed-Pascal 1.5, Sonderheft der CHIP, Februar 1996

[OS7/96] OS/2 Inside 7/1996, Turbos Rückkehr - Pascal für OS/2, 1996

[Pau76] Paulin, G., Hohberg, B., Compilertechnik, VEB Verlag Technik Berlin 1976

[Str58] J. Strong, The Problem of Programming Communication with Changing Machines. A Proposed Solution., Part 1 und 2, Comm. ACM Vol. 1, No. 8 und 9,1958

[Thi92] Thies, K.D., 80486 Systemsoftware-Entwicklung, Carl Hanser Verlag 1992

[To4/93] Toolbox, Grosser Compilervergleich: Das leisten C++-Boliden, Ausgabe 4, 1993

[To5/96] Toolbox, Compilervergleich Speed-Pascal/2 und Virtual-Pascal für OS/2, Ausgabe 5, 1996

[Wir95] Wirth, N., Grundlagen und Techniken des Compilerbaus, Addison Wesley 1995

[Wir77] Wirth, N., What can we do about the unnecessary diversity of notation for syntactic definitions?, Comm. ACM Vol. 1, No. 20, 1977