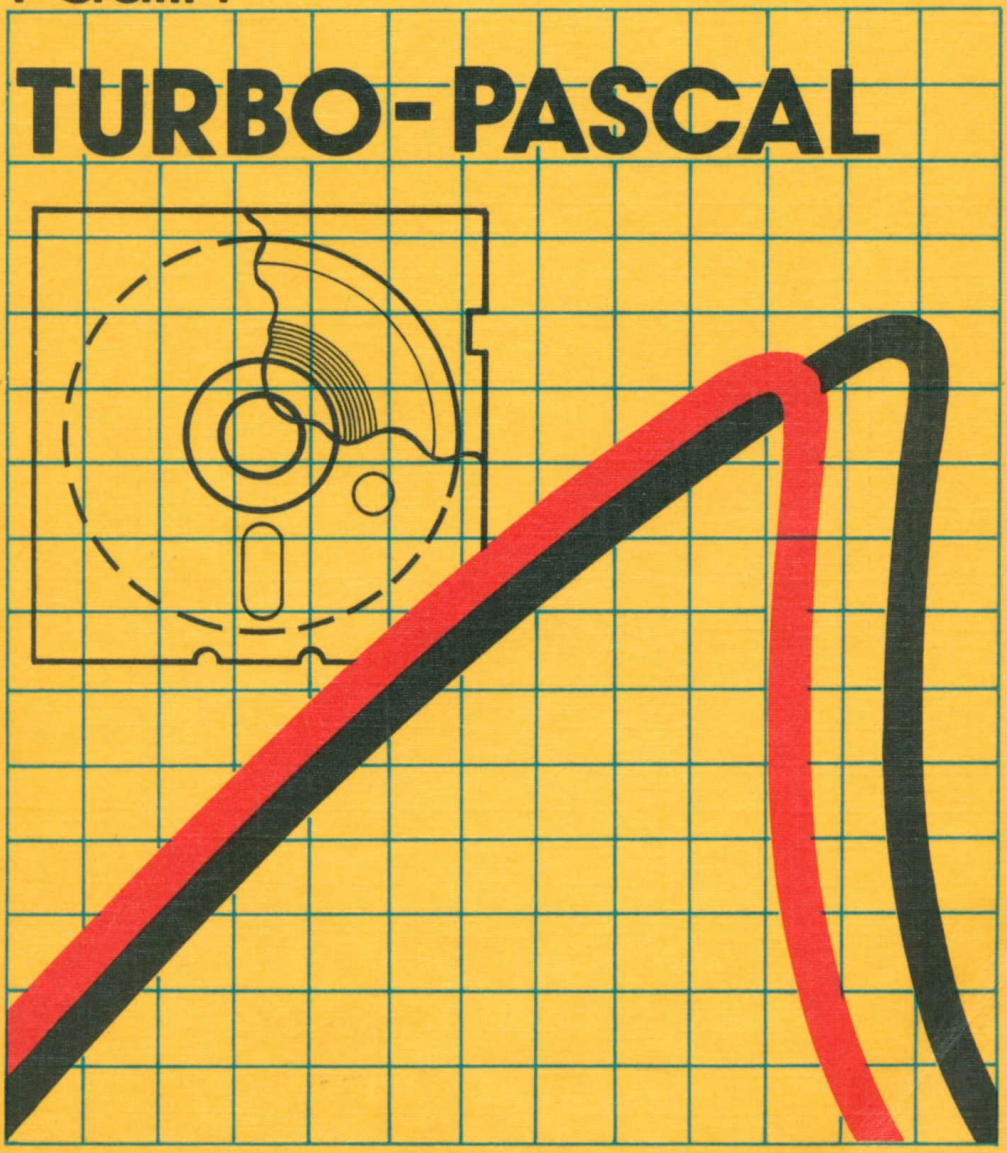
Technische Informatik

Paulin

TURBO-PASCAL



VEB Verlag Technik Berlin

Paulin

TURBO-PASCAL

Technische Informatik

Herausgegeben von
Dr.-Ing. Ludwig Claßen
Dr.-Ing. Dieter Nedo
Dr. rer. nat. Gerhard Paulin

TURBO-PASCAL

Dr. rer. nat. Gerhard Paulin

VEB VERLAG TECHNIK BERLIN

Paulin, Gerhard:
TURBO-PASCAL / Gerhard Paulin. - 1. Aufl. - Berlin : Verl. Technik, 1988. - 206 S. : 20 Bilder, 3 Taf. - (Technische Informatik)
ISBN 3-341-00521-8

ISBN 3-341-00521-8
ISSN 0863-0860 Technische Informatik

1. Auflage

Lizenz 201 · 370/76/88
Printed in the German Democratic Republic
Druck und buchbinderische Weiterverarbeitung:
Druckerei Neues Deutschland, Berlin
Lektor: Karl Belter
Einbandgestaltung: Gabriele Schwesinger
LSV 3043 · VT 3/5953-1
Bestellnummer: 553 921 0
02000

Vorwort

Von den zahlreichen Programmiersprachen der letzten 25 Jahre sind es nur einige wenige, die sich einen festen Platz in der Software unterschiedlicher Rechnersysteme erworben haben. Zu diesen Sprachen muss PASCAL gerechnet werden. Die mikrorechnergestützten , dialogorientierten Rechnersysteme der achtziger Jahre stellen eine neue Qualität von Rechnerarchitekturen dar, und es ist deshalb nicht verwunderlich, wenn an die Software dieser Systeme andere Anforderungen gestellt wurden und werden.
Mit Standard-PASCAL als Kern ist ein bequem handhabbares Programmiersystem entwickelt worden, das Quelltexterarbeitung und Test von Programmen in einem Arbeitsgang im Dialog erlaubt und dabei durch Geschwindigkeit, d. h. durch kurze Antwortzeiten überzeugt. Dieses Programmiersystem ist mehr als eine Programmiersprache; die konkrete Programmiersprache ist in dem System der Teil zum Formulieren der Algorithmen.
Wer sich in Programmierung einarbeiten und moderne Technik nutzen will, wird gern zu dem TURBO-PASCAL-System greifen. Sicherlich ist die Programmiersprache TURBO-PASCAL nicht ganz so einfach zu erlernen wie etwa BASIC, dafür bietet sie aber durch strukturierte Datentypen und strukturierte Anweisungen die Gewähr, übersichtliche, wartbare Programme zu entwickeln. Da sie eingebettet ist in leistungsfähige Betriebssysteme, wie sie auf Büro- und Personalcomputern benutzt werden, ist TURBO-PASCAL sicherlich ein hervorragendes Arbeitsmittel für sehr verschiedene Anwendungsfälle. Diese Anwendungsfälle reichen von Aufgaben der Textverarbeitung über Programme für technisch-wissenschaftliche Berechnungen bis zur Computergrafik - wenn auch letzteres stark von den hardwaremässig gegebenen Voraussetzungen abhängt.
Ziel dieses Buches ist es, alles Wesentliche des TURBO-PASCAL-Systems bereitzustellen. Die Anhänge sollen demjenigen ein bequemes Nachschlagen gestatten, der schon mit dem System vertraut ist. Das Buch setzt Grundkenntnisse der Sprache PASCAL voraus. (Standard-) PASCAL wird in gedrängter Form - gewissermassen als Informationsspeicher - dargestellt. Ausführlich wird auf die Sprachelemente eingegangen, die über den "Standard" hinausgehen und wesentlich die Sprache TURBO-PASCAL prägen. Aus didaktischen Gründen sind Standard-PASCAL und TURBO-PASCAL-Erweiterungen miteinander verflochten.

Ich möchte allen Kollegen danken, die in Gesprächen über ihre Erfahrungen mit dem System berichtet oder die Teile des Manuskripts gelesen haben. Herr Dipl.-Math. Manfred Krzikalla hat seine grossen Erfahrungen auf dem Gebiet maschinennaher Programmierung innerhalb des TURBO-PASCAL-Systems in das Buch eingebracht und den Abschnitt 2.14.8. übernommen.Ich möchte ihm dafür ganz besonders danken, weil darin dem mit Programmierung im Maschinenkode Vertrauten interessante Möglichkeiten gezeigt werden.
Herrn Dr. Ludwig Classen danke ich für fachliche und methodische Hinweise und - last not least - Herrn Jürgen Reichenbach und Herrn Karl Belter vom Verlag Technik für die sehr gute Zusammenarbeit. In das Dankeschön möchte ich auch meine Familie einschliessen, die einerseits etwas von dem Stress abbekommen hat, den das Erarbeiten eines Manuskripts mit sich bringt, die sich aber auch andererseits tatkräftig an der Herstellung der Druckvorlage beteiligte.

Berlin Gerhard Paulin

Inhaltsverzeichnis

0. Einführung

PASCAL wurde in den Jahren 1968 bis 1970 von N. Wirth an der Eidgenössischen Technischen Hochschule in Zürich entwickelt. Ein Compiler steht seit 1971 zur Verfügung. Die ursprünglich für das Lehren über Programmierung entwickelte Sprache setzte sich sehr schnell weltweit durch. Mit dem praktischen Einsatz von PASCAL wurden Wünsche nach einer Erweiterung zu einer universell einsetzbaren Sprache laut, vor allem von Softwareingenieuren nach einer PASCAL-Version, die auf Arbeitsplatzrechnern ein leistungsfähiges Werkzeug darstellt und bequemer zu handhaben ist als eine Assemblersprache. Eine dieser PASCAL-Erweiterungen, die auf Grund solcher Forderungen entstanden, ist TURBO-PASCAL. Die Urheberschaft ist nicht explizit mitgeteilt worden, als Vertreiber fungiert Borland International, California. TURBO-PASCAL steht u. a. in folgenden Betriebssystemen zur Verfügung: CP/M-80, CP/M-86 und MS-DOS.
TURBO-PASCAL schliesst den durch den PASCAL-Bericht und spätere Standards geschaffenen Sprachumfang im wesentlichen ein.
Die Einschränkungen sind von solcher Art, dass es nur geringe und in vielen Fällen gar keine Mühe machen dürfte, vorhandene PASCAL-Quelltexte so abzuändern, dass diese Abweichungen berücksichtigt sind. Wesentlich anders sieht das mit den in TURBO-PASCAL vorhandenen Erweiterungen aus. Ein mit den sprachlichen Mitteln von TURBO-PASCAL entwickeltes Programm wird von konventionellen PASCAL-Compilern nicht akzeptiert. Zu den wesentlichen Erweiterungen der Sprache PASCAL gehören:

- Das Typsymbol **string**.
- Die Einbeziehung der Steuerzeichen in die Symbolmenge.
- Die Erweiterung des Anwendungsbereichs für logische Operatoren.
- Eine bequeme Konvertierung der Werte von Aufzählungstypen.
- Die Aufhebung der strengen Reihenfolge der Abschnitte des Vereinbarungsteils.
- Die strukturierten Konstanten.
- Die Typkonstanten (typed constants).
- Ein Zufallszahlengenerator.
- Vordefinierte Felder für Hauptspeicherdirektzugriff und Zugriff zu den Ports.
- Prozeduren für die Zusammenarbeit mit externen

Objektkodefiles.
- Files mit wahlfreiem Zugriff.
- Ein Konzept für den Aufbau von Überlagerungstrukturen und von COMMON-Bereichen.
- Die Einbeziehung von Maschinenkode in TURBO-PASCAL-Programme
- Möglichkeiten, von TURBO-PASCAL-Programmen Systemrufe abzusetzen.

Bei TURBO-PASCAL geht es aber nicht nur um ein sehr leistungsfähiges Sprachderivat, sondern um ein System, dass dem Anwender ausser dem Compiler bequeme Möglichkeiten für die Quelltextbereitstellung und das Testen zur Verfügung stellt. Das Textverarbeitungssystem WordStar ist in das TURBO-PASCAL-System integriert.
In bezug auf das zu erzeugende Kodefile kann durch Optionen gesteuert werden, ob das File im Hauptspeicher oder auf Diskette bereitgestellt werden soll. Auch in der Laufzeit können Fehler unmittelbar im Dialog korrigiert werden.
Mittels Compilerdirektiven, die, wie auch bei anderen PASCAL-Versionen, in Form von Pseudokommentaren kodiert werden, können Sonderabläufe vorgesehen werden.
Die Spracherweiterungen und die Hilfsmittel zur Quelltextbereitstellung und zum syntaktischen und logischen Test eines Programms machen diese Programmiersprache zu einem effektiven Arbeitsmittel beim Rechnereinsatz.
Kritisch ist allerdings zu sagen, dass die Erweiterungen die Portabilität gefährden.
Dieses Buch will in den Umgang mit dem Programmiersystem einführen, das durch den Namen TURBO-PASCAL gekennzeichnet ist. Da das Manuskript in den Jahren 1985/86 entstanden ist, spiegelt das Buch die wesentlichen Mittel des Programmiersystems dieser Jahre wider. Da bekanntlich alles, was realisiert ist, die Grundlage für eine Weiterentwicklung und Verbesserung ist - was existiert, ist veraltet - gibt es möglicherweise beim Erscheinen des Buches schon Spracherweiterungen, die für das eine oder andere Einsatzgebiet neue Möglichkeiten schaffen.

1. Die Benutzung von TURBO-PASCAL unter dem Betriebssystem CP/M

TURBO-PASCAL kann in unterschiedlichen Betriebssystemen installiert werden.
Hinsichtlich der Benutzung ergeben sich keine oder nur geringfügige Unterschiede.(So haben z. B. in den Betriebssystemen CP/M-80 und MS-DOS ausführbare Programme den Namenszusatz .COM, im Betriebssystem CP/M-86 dagegen .CMD.)Deshalb ist es für die Behandlung des TURBO-PASCAL-Systems keine Einschränkung, sich auf eine Umgebung des Programmiersystems zu beziehen, und das soll im weiteren CP/M (bzw. jedes CP/M-kompatible Betriebssystem) sein.
Im weiteren wird vorausgesetzt, dass das TURBO-PASCAL-Programmiersystem auf einer Diskette einsatzbereit vorliegt.

1.1. Das Menü

Das Starten des Programmiersystems erfolgt durch Eingabe des Kommandos

TURBO

(Kleinbuchstaben sind dabei gleichwertig).
Auf dem Bildschirm meldet sich das System mit der Ausschrift

```
TURBO Pascal System      Version v.vvw
Copyright (C) 1983,1984 by BORLAND Inc.
```

und der Frage

```
Include error messages (Y/N) ?.
```

Als Antwort wird also Y oder N verlangt; im ersten Fall wird ein File (Umfang etwa 1.5 KByte) mit Fehlernachrichten geladen.
Danach erscheint das TURBO-PASCAL-Hauptmenü, in dem die jetzt zu wiederholenden Kommandos für das Programmsystem angegeben sind (Bild 1.1.).

```
Logged drive: x
Work file:
Main file:

Edit        Compile     Run         Save
eXecute     Dir         Quit        compiler Options

Text : 0 bytes
Free : yyyyy bytes
>
```

Bild 1.1. Hauptmenü

x symbolisiert in dem Bild 1.1. (und im weiteren Text) die Bezeichnung des aktuellen Diskettenlaufwerks,
yyyyy den Umfang des freien Hauptspeicherplatzes in dem benutzten Rechner.
Das Pfeilzeichen '>' ist das Promptzeichen von TURBO-PASCAL.
Die Grossbuchstaben in den Schlüsselwörtern des Menüs die Unterkommandos, die jetzt akzeptiert werden.
Da Missverständnisse nicht möglich sind, werden wir die Unterkommandos des TURBO-PASCAL-System kürzer "Kommandos" nennen.
Es existieren folgende Kommandos in TURBO-PASCAL:

Q (quit) beendet die Arbeit von TURBO-PASCAL. Gibt es beim Beenden der Arbeit mit dem Programmiersystem ein modifiziertes, aber nicht auf einer Diskette gesichertes File, so fragt das System, ob ein Sichern erfolgen soll.

L Mit diesem Kommando kann das aktuelle Laufwerk geändert werden.

D Nach Eingabe von D erfolgt die Frage nach einem Laufwerk:
Dir mask:
Es ist ein Laufwerkbezeichner einzugeben. Danach erfolgt die Ausgabe des Diskettеninhaltsverzeichnisses (directory) auf dem Bildschirm.

W gestattet die Auswahl eines Files, mit dem gearbeitet werden soll. Der Filename ist in der Form filename.typ einzugeben. filename symbolisiert eine maximal achtstelligen Namen (gemäss CP/M-Konvention), typ eine maximal dreistellige Namenserweite-

rung. Fehlt .typ, so wird .PAS ergänzt. Mittels W kann zu einem anderen Arbeitsfile übergegangen werden. Sollte das zuletzt benutzte File verändert, aber nicht gespeichert worden sein, so wird vor dem Laden des nächsten Arbeitsfiles gefragt, ob das zuletzt benutzte File gespeichert werden soll.

E Ein mittels W bereitgestelltes File soll ediert werden. Bei dem Editor handelt es sich um eine abgerüstete Version des Textverarbeitungssystems WordStar, d.h., dass das Edieren wie beim Arbeiten mit dem Textverarbeitungssystem erfolgt. Der Editor kann (entsprechend) mit ^kd verlassen werden. Die am häufigsten benötigten Edierkommandos sind im Anhang G zusammengestellt.
Ist kein Arbeitsfile bekannt, so wird nach der Eingabe von E die Eingabe eines Arbeitsfilenamens angefordert. Hat der Name des Arbeitsfiles keinen Namenszusatz, so fügt der Editor .PAS an.

S Durch Eingabe von S wird das Arbeitsfile auf der Diskette gesichert. Das alte File wird aufgehoben, es erhält die Namenserweiterung .BAK.

C Mit dem Unterkommando C wird der PASCAL-Compiler aufgerufen. Verarbeitet werden Quelltextfiles mit dem Namenszusatz .PAS. Im Ergebnis des Compilerlaufs wird ein abarbeitungsfähiges Programmfile erzeugt (syntaktische Korrektheit des Quelltextes vorausgesetzt).
Dieses Programmfile befindet sich im Hauptspeicher und kann sofort mittels R abgearbeitet werden. Soll das Programmfile auf der Diskette gesichert werden, so ist die (bisher beschriebene) Standardsituation mit Hilfe des Kommandos O zu ändern (vgl. die Ergänzungen bei den Erläuterungen zu dem Kommando O).
Werden bei der Übersetzung Syntaxfehler erkannt, so wird die Übersetzung unterbrochen mit einem Fehlerhinweis und der Aufforderung, die ESCAPE-Taste zu drücken (ESC oder ^[). Es erscheint dann auf dem Bildschirm der Quelltextausschnitt, und der Kursor zeigt die Stelle an, an der ein Fehler bemerkt worden ist. Die zu korregierende Stelle kann davor liegen!

R startet die Abarbeitung eines durch Compilation im Hauptspeicher entstandenen Files oder eines Files, das auf einer Diskette abgelegt wurde. R ruft implizit den Compiler auf, wenn sich das Programmfile nicht im Hauptspeicher befindet.

O Standardwerte des Compilers können nach Eingabe von O verändert werden. Es erscheint dann folgendes Menü (Bild 1.2.):

```
compile   ->  Memory
              Com-file
              cHn-file

Find run-time error Quit
```

Bild 1.2. Optionsmenü

Dieses Menü ist folgendermassen zu verstehen:
Standardmässig erfolgt die Erzeugung des Programms im Hauptspeicher (--> M , M von memory).
Durch Eingabe von C kann diese Standardfestlegung verändert werden, es wird dann das Programmfile auf der Diskette abgelegt, das File erhält den Namenszusatz .COM. Das so erzeugte Programm kann mit EXECUTE aufgerufen werden.
Durch Eingabe von H wird ein Programmfile erzeugt, das mit dem Aufruf CHAIN in ein Programm eingefügt werden kann, die Programmfiles erhalten den Namenszusatz .CHN. Näheres über die Arbeit mit .COM-Files und .CHN-Files in den Abschnitten 2.13. und 2.14.7. Das Kommando O wird mit Q verlassen, auf dem Bildschirm erscheint wieder das Promptzeichen des Programmiersystems.

M In TURBO-PASCAL ist es möglich, den Quelltext in Abschnitte zu zerlegen und diese Abschnitte zu benennen. Solche Quelltextabschnitte können in ein Hauptprogramm mit Hilfe der Compileranweisung "¤i" (vgl. Abschn. 2.15.) eingefügt werden. Das Kommando M dient dazu, ein solches Hauptprogramm (main) zu definieren. Die Programmabschnitte können dann mit dem Kommando W aufgerufen und modifiziert werden. Beim Übergang zum Übersetzen wird das aktuelle Arbeitsfile auf der Diskette abgelegt und das Hauptprogramm geladen.
Wird innerhalb eines solchen Programmabschnitts (Arbeitsfiles) ein syntaktischer Fehler erkannt, so wird wiederum das Arbeitsfile zur Korrektur auf dem Bildschirm angezeigt, nach der Korrektur kann sofort die Compilation des Hauptprogramms neu gestartet werden.

1.2. Beispiel einer Programmentwicklung mit dem TURBO-PASCAL-System

Das folgende Beispiel ist für denjenigen gedacht, der sich in die Technologie einarbeiten will, die ihm mit dem TURBO-PASCAL-Programmiersystem in die Hand gegeben ist. Es enthält notwendige und typische Aktionen. Wer in dialogorientierter Programmentwicklung wenig Erfahrung hat, sollte das Beispiel an einem Bürorechner durchspielen. Die Ausgaben des Rechners sind unterstrichen, die Eingaben des Nutzers nicht, das Zeilenendezeichen CR (carriage return) ist nicht angegeben, x symbolisiert das Laufwerk, auf dem sich das TURBO-PASCAL-System befindet. Angaben in {{..}} sind Bemerkungen zu den dort vorzunehmenden Aktionen.

```
x> turbo
> E
Work file name :beispiel
    .
    .
    .
program 1.Versuch;
begin
WRITELN("1. Versuch mit TURBO-PASCAL");
end.
^Kd
>C
Compiling
 1 lines
 Error 58: Illegal character in identifier. Press<ESC>
    {{Es meldet sich der Editor mit dem Quelltext.
     Korrektur: "1." streichen.}}
^Kd
>C
Compiling
 3 lines
 Error 41: Unknown identifier or syntax error. Press <ESC>
    {{Es meldet sich der Editor mit dem Quelltext.
     Zu korrigieren sind Anführungszeichen in Apostrophe.}}
^Kd
>C
Compiling
  4 lines
  Code:     52 bytes (8061-8095)
  Free:  20332 bytes (8096-D002)
  Data:      3 bytes (D003-D006)
```

```
>R
 Running
  1. Versuch mit TURBO-PASCAL
>S
Saving:x:beispiel.pas
>Q
x>
```

2. Die Programmiersprache TURBO-PASCAL

2.1. Basissymbole

Für die Programmiersprache TURBO-PASCAL ist eine Zeichenmenge zugrundegelegt, die aus einfachen Zeichen(ISO-Kode, ASCII) - wie sie auf einer Tastatur vorliegen -, aus Zeichenpaaren und aus Wortsymbolen besteht. Zeichenpaare bzw. Wortsymbole sind nur Darstellungsformen eines Zeichens. Wortsymbole werden in diesem Buch in Beispielprogrammen durch halbfetten Druck dargestellt.
Beim Programmieren wird so vorgegangen, dass man die Wortsymbole durch die entsprechenden Buchstabenfolgen ersetzt und diese Buchstabenfolgen zu "Schlüsselwörtern" erklärt. Diese Festlegung enthält die Forderung, solche Buchstabenfolgen nur in der Bedeutung des Wortsymbols (also eines Basissymbols) zu benutzen.
Die Basissymbolmenge von TURBO-PASCAL enthält:

Ziffernzeichen	0 bis 9
Buchstaben	a bis z und A bis Z und das Unterstreichungszeichen
Spezialzeichen	( ) [ ] { } . , : ; ' # ¤ + - * / ^ = < >
Zeichenpaare	:= <> <= >= .. (. .) (* *) (Die Zeichenpaare (. und .) dürfen für [und], die Zeichenpaare (* und *) für { und } verwendet werden.)

Wortsymbole			
	absolute	**function**	**record**
	and	**goto**	**repeat**
	array	**if**	**set**
	begin	**in**	**shl**
	case	**inline**	**shr**
	const	**label**	**string**
	div	**mod**	**then**
	do	**nil**	**to**
	down	**not**	**type**
	else	**overlay**	**until**
	end	**of**	**var**
	external	**or**	**while**
	file	**packed**	**with**
	for	**procedure**	**xor**
	forward	**program**	

Dieser Aufstellung sind vor allem zwei Informationen zu entnehmen:
1. Es können Klein- und Grossbuchstaben benutzt werden, allerdings werden sie nur in Zeichenketten unterschieden.
2. Unter den Wortsymbolen sind einige, die aus anderen Programmiersprachen, insbesondere aus "klassischem" PASCAL und seinen Varianten, nicht bekannt sind. Das sind die Symbole
absolute, inline, overlay, shl, shr, string, xor.

2.2. Darstellungskonventionen

Im allgemeinen erschwert die Vorgehensweise, Sprachelemente vollständig hinsichtlich Syntax und Semantik darzustellen, das Erlernen der jeweiligen Sprachkonstruktion. Da hier ohnehin Grundkenntnisse der Sprache PASCAL vorausgesetzt werden, soll eine Beschränkung in dem Sinne erfolgen, dass die Sprachelemente symbolisch dargestellt werden. "Symbolisch" heisst, dass eine intuitiv verständliche Notation erfolgt, auf Syntaxdiagramme oder Definitionsgleichungen aber verzichtet wird. Eine formale Syntaxbeschreibung ist im Anhang B angegeben.
Hier werden wir uns meist darauf beschränken, Erzeugungsregeln verbal zu formulieren. Wir meinen, dass das ein praxisorientierter Standpunkt ist: Auch eine natürliche Sprache erlernt man nicht dadurch, dass man grammatische Regeln auswendig lernt. Solide Grundkenntnisse und Übung führen zur Beherrschung. In unserem Falle bedeutet "Übung", Programme zu entwerfen, zu korrigieren und letztlich logisch zu testen.
Wo eine strenge Syntaxnotation erforderlich scheint, werden wir Backus-Naur-Form benutzen, wie sie im Anhang B erklärt ist.
Den Objekten eines Programms können Namen gegeben werden. Wir werden anstelle von Namen Bezeichner sagen. Unter Bezeichner ist eine Folge von Buchstabenzeichen und Ziffernzeichen (gemäss Abschn. 2.1.) zu verstehen, die mit einem Buchstabenzeichen beginnt. (Diese Erzeugungsregel für Bezeichner ist im Anhang B in der Form

```
<bezeichner> ::= <buchstabe>| <bezeichner><buchstabe>|
                               <bezeichner><ziffer>
```

formuliert. Für denjenigen, der sich daran gewöhnt hat, ist das natürlich kurz und klar.)
Diese Regel für das Bilden von Bezeichnern wird in gewissem Sinne dadurch eingeschränkt, dass bestimmte Bezeichner dem System bekannt

sind und eine definierte Bedeutung haben: Konstantenbezeichner, Typbezeichner, Routinenbezeichner. Prinzipiell können diese "vordefinierten" Bezeichner neu definiert werden, doch ist davon abzuraten, und es gibt eigentlich auch keine überzeugenden Gründe, es zu tun.
In diesem Buch werden vordefinierte Bezeichner (Standardbezeichner) durch Grossbuchstaben wiedergegeben. Diese Festlegung erfolgt aber nur aus methodischen Gründen und ist beim Programmieren belanglos. Für Programmtexte gilt also:

halbfett	Wortsymbole (aus der Menge der Basiselemente)
Grossbuchstaben	Standardbezeichner
Kleinbuchstaben	vom Programmierer definierte Bezeichner

In Zeichenketten können beliebige Zeichenfolgen verwendet werden. Das korrigierte Beispiel aus Abschn. 1.2. wäre also nach diesen Festlegungen so wiederzugeben:

```
program versuch;
begin
WRITELN('1. Versuch mit TURBO-PASCAL');
end.
```

2.3. Aufbau eines TURBO-PASCAL-Programms und seine Verbindung mit anderen Systemkomponenten

Der Kern des TURBO-PASCAL-Systems besteht aus einem Programmfile mit dem Namen TURBO.COM, das durch CP/M geladen und gestartet wird. Von den weiteren zum TURBO-PASCAL-System gehörenden Files sind TURBO.MSG und TURBO.OVR zu nennen. TURBO.MSG enthält die Fehlernachrichten des Systems (ungefähr 1.5 KByte Speicherbedarf), TURBO.OVR wird benötigt, wenn .COM-Files von Diskette verarbeitet werden sollen. Die Grobstruktur dieser Verbindungen ist im Bild 2.1. skizziert.

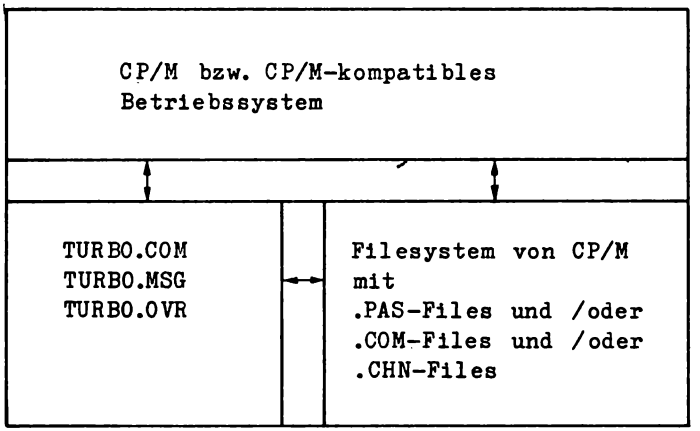


Bild 2.1. CP/M mit TURBO-PASCAL-System

(Grundsätzlich kann der Editor des TURBO-PASCAL-Systems für beliebige Textfiles benutzt werden.)
Hinweise für die Aufteilung des Hauptspeichers des TURBO-PASCAL-Systems werden im Abschnitt 2.13.6. gegeben.

Ein TURBO-PASCAL-Programm besteht aus einer Kopfzeile,einem Vereinbarungsteil und einem Anweisungsteil.

Die Kopfzeile ist durch das Basissymbol **program** gekennzeichnet, dem der Programmbezeichner folgt. Ausserdem können sog. Programmparameter angegeben werden. Das sind Bezeichner für Files, die im Programm benutzt werden. Wenn das Programm abgearbeitet wird, muss zwischen den Programmfiles und den im CP/M existierenden oder anzulegenden Files eine Verbindung hergestellt werden. (Wie das im Programm kodiert wird, ist im Abschn. 2.11. erläutert.) Es erfolgt also eine Zusammenarbeit zwischen dem Laufzeitsystem des Programmiersystems und dem Filesystem von CP/M.

In dem Vereinbarungsteil werden Programmobjekte beschrieben. Programmobjekte steht hier für

- Marken (sie dienen der Programmablaufsteuerung),
- Konstanten,
- Typen (darunter werden Beschreibungen für Definitionsbereiche verstanden),
- Variablen und

Routinen (Prozeduren und Funktionen).

Im Anweisungsteil ist kodiert, welche Aktionen mit den (oder mit Hilfe der)im Vereinbarungsteil fixierten Programmobjekte ausgeführt werden sollen. Es dürfen im Anweisungsteil nur solche Objekte benutzt werden, die entweder im Vereinbarungsteil beschrieben wurden oder die in dem TURBO-PASCAL-System vordefiniert sind, dem Compiler also bekannt sind (z.B. Bezeichner für Standardtypen und Standardprozeduren).
In TURBO-PASCAL ist die Reihenfolge der Vereinbarungen nicht festgelegt (im Gegensatz zu PASCAL).

2.4. Einfache TURBO-PASCAL-Programme

Es soll zunächst das dargestellt werden, was benötigt wird, um übersetzbare und abarbeitbare Programme zu schreiben.

2.4.1. Einfache Typen- und Variablendeklarationen

Im TURBO-PASCAL-System sind einige Bezeichner, die Definitionsbereiche beschreiben, vordefiniert. Solche Bezeichner sind:

INTEGER Alle in dem konkreten TURBO-PASCAL-System darstellbaren ganzen Zahlen.

REAL Alle in dem konkreten TURBO-PASCAL-System darstellbaren reellen Zahlen (Gleitkommazahlen).

CHAR Alle Zeichen des Maschinenkodes (i. allg. ASCII, vgl. Anhang A).

BYTE Der Definitionsbereich umfasst 0 bis 255, ist also ein Teilbereich von INTEGER. Deshalb können Variablen dieses Types wie solche des Typs INTEGER benutzt werden.

BOOLEAN Der Definitionsbereich besteht aus den beiden vordefinierten Bezeichnern FALSE und TRUE.

Sollen Definitionsbereiche eingeführt werden, die sich von diesen unterscheiden, so sind dafür spezielle Sprachelemente zu verwenden (Typen bzw. Typdefinitionen).
Werden nun Variablen benötigt, so muss der Programmierer entscheiden, welchen Definitionsbereich er für eine jede Variable festlegen will. Das geschieht dadurch, dass er für Variablen Bezeichner einführt und ihnen - durch Doppelpunkt getrennt - einen Typ zuordnet, also

```
var
     i1, i2   INTEGER;
     x, y, z : REAL;
     c0,c1,c2: CHAR;
     p : BOOLEAN;
```

Die Variablendeklarationen werden durch Semikolon getrennt, das Basissymbol **var** ist voranzustellen. Insgesamt heisst die obige Konstruktion "Variablendeklarationsteil". Die so eingeführten Variablenbezeichner können in dem folgenden Anweisungsteil benutzt werden.
Bezeichner dürfen nicht mehrfach definiert sein; über Ausnahmen s. Abschn. 2.9.3. (Record-Typ) und Abschn. 2.14.3. (Gültigkeitsbereiche).

2.4.2. Dateneingabe und -ausgabe im Dialog

Was ist in einem Programm zu notieren, damit am Terminal Daten angefordert bzw. auf den Bildschirm Daten ausgegeben werden? In TURBO-PASCAL gibt es dafür Standardprozeduren, von denen an dieser Stelle

```
     WRITELN(a1, a2, ..., am)
und  READLN (v1, v2, ..., vn)
```

behandelt werden sollen.

In der Notation

```
    WRITELN(a1, a2, ..., am)
```

stehen a1, a2, ..., am für Werte, die auf dem Bildschirm sichtbar gemacht werden sollen. Diese Werte können Zeichenfolgen oder numerische oder logische Werte sein.
Beispiele erklären das sofort.
Durch die Abarbeitung von

```
    WRITELN('1. Versuch mit TURBO-PASCAL')
```

erscheint auf dem Bildschirm der in Apostrophe eingeschlossene Text, also

```
     1. Versuch mit TURBO-PASCAL
```

WRITELN(11) überträgt 11 auf den Bildschirm.
WRITELN(' j=', i) bewirkt die Ausgabe der Zeichenfolge j= und anschliessend den Wert der Variablen i.

Die Werte der Variablen x und y können, durch Komma getrennt, mittels WRITELN(x, ',' , y) ausgegeben werden.
Die Anzahl der Positionen, die für die zu druckenden Werte verwendet werden, ist implementationsabhängig. Für ganze Zahlen werden i. allg. soviele Positionen verwendet, wie die Angabe des Wertes erfordert. Gleitkommazahlen werden in sog. halblogarithmischer Form ausgegeben:

```
vz.zzzzzzzzzzEvzz
```

Darin bedeutet v eine Vorzeichenstelle,
z eine Ziffer,
E den Beginn des Skalierungsfaktors.
Hat i den Wert 12 und haben x bzw. y die Werte 12.5 bzw. -900, so wären die Ausgabebilder für obige Beispiele

```
j=11
```

bzw. 1.2500000000E+01, -9.0000000000E+02

Die Ausgabebilder kann man durch Formatangaben beeinflussen. Darunter versteht man einen Zusatz zu den Wertangaben, in dem die Anzahl der Positionen festgelegt wird, die im Ausgabebild zu benutzen sind. Die Formatangabe hat die Form

```
:k
```

wobei k eine positive ganze Zahl bedeutet (eine vollständige Beschreibung erfolgt im Abschn. 2.11.2.).

```
WRITELN('!', 11:4,'!', 12.5, '!')
```

liefert die Ausgabe

```
!  11!  1.2500000000E+01!
```

Die Anweisung

```
READLN(v1, v2, ..., vn)
```

fordert vom Bildschirm n Werte an. v1, v2, ..., vn stehen symbolisch für Variablen. Die Anforderung wird auf dem Schirm durch das Erscheinen des Kursors sichtbar. Der Programmierer muss dann wissen, wieviel Werte er, durch ein oder mehrere Leerzeichen getrennt, eintasten muss. Die Eingabe ist mit der Wagenrücklauftaste (CR oder ET1 auf einigen Tastaturen) zu beenden. Es ist eine normale Vorgehensweise, zuerst einen Text auszugeben, der die folgende Anforderung erklärt, etwa

```
WRITE('Werte fuer x und i eingeben:');
READLN (x,i);
```

Werden Werte für eine Variable des Typs INTEGER angefordert, so ist eine Ziffernfolge einzugeben, der ein Vorzeichen vorangestellt sein kann. Als REAL-Werte werden Werte der INTEGER-Form akzeptiert, es darf jedoch auch ein gebrochener Teil und eventuell ein Exponententeil (wie in den obigen Ausgabebeispielen) angefügt werden. Das Beispiel

```
     READLN(x,i);
```

könnte also durch folgende Eingabewerte befriedigt werden:

```
     12.5    11
oder -9   0
oder 1.1e-3   -21
```

Mit dieser Information ist es möglich, einfachste Quellprogramme zu erzeugen, zu übersetzen und abzuarbeiten. Die Anweisungen des Programms werden in die Basissymbole **begin** und **end** eingeschlossen. Kommentare können eingefügt werden.
Das folgende Protokoll illustriert die Terminalarbeit, nachdem das Promptzeichen '>' des TURBO-PASCAL-System erschienen ist.
Beispiel:

```
>E
Work file name: inout
New file: INOUT.PAS
program inout;
  {Berechnungen am Dreieck}
var a,b,c,s:REAL;
begin
  WRITELN('Dreiecksseiten eingeben');
  WRITE('a='); READLN(a);
  WRITE('b='); READLN(b);
  WRITE('c='); READLN(c);
  s:=(a+b+c)*0.5;
  WRITELN('Dreiecksflaeche =',SQRT(s*(s-a)*(s-b)*(s-c)));
end.
^Kd
>C
Compiling
    12 lines
Code:   358 bytes (8148-82AE)
Free: 19771 bytes (82AF-CFEA)
Data:    27 bytes (CFEB-D006)
>R
Running
```

```
Dreiecksseiten eingeben
a= 3.5
b= 4.8
c= 6.2
Dreiecksflaeche= 8.3630044690E+00
```

Bemerkung: Werden für a,b,c Werte eingegeben, die nicht Masszahlen der Seiten eines Dreiecks sein können, so bricht die Abarbeitung des Programms mit einem Fehler ab.

2.4.3. Kommentare

Um die Lesbarkeit von Programmen zu erleichtern, können Kommentare eingefügt werden. Mit Kommentar bezeichnet man einen beliebigen Text, der in die Klammersymbole { und } eingeschlossen ist. Innerhalb von Kommentaren darf die schliessende Klammer } nicht auftreten.
Beispiel:

```
program test; {Pruefen von Daten} .... begin ... end.
```

2.5. Konstantendefinitionsteil I

Der Sinn einer Konstantendefinition besteht darin, für unveränderliche Programmgrössen Bezeichner einzuführen und im weiteren diese Bezeichner für die Programmgrössen zu benutzen. Sicherlich ist es bequemer pi zu schreiben als 3.1415926. Programme sind dadurch auch einfacher zu ändern, weil die Änderung des Wertes einer Konstanten an jeder Programmstelle wirksam wird, wo der Bezeichner verwendet wird.
Ein Konstantendefinitionsteil beginnt mit dem Basissymbol **const**, anschliessend werden in Form von Gleichungen die Bezeichner und die Werte, die sie repräsentieren, notiert.
Ein Semikolon schliesst jede Konstantendefinition ab.
Ganzzahlige Konstanten können in dezimaler oder hexadezimaler Notation angegeben werden; in Hexaform muss das Zeichen ¤ vorangestellt werden.
Beispiele:

```
const
     n= 20;
     r= 1.5;
     t= ' Anhang ';
     h= ¤80;
```

n ist vom Typ INTEGER, r vom Typ REAL, t symbolisiert eine

Zeichenkette (mehr über den Typ von t im Abschn. 2.7.).
In TURBO-PASCAL wird der Terminus Konstante noch in einer zweiten Bedeutung benutzt. In einer Konstantendefinition kann einem Bezeichner explizit ein Typ zugeordnet werden: Dem Bezeichner wird - durch Doppelpunkt getrennt - ein Typbezeichner nachgestellt, also

```
const
    rr: REAL = 12;
```

So eingeführte Bezeichner (einfache Typkonstanten) sind nicht mehr Konstanten wie oben beschrieben. Sie haben die Funktion von Variablen, für die ein Anfangswert festgelegt wurde. Der Wert dieser Typkonstanten kann im Programm verändert werden. (Typkonstanten werden vollständig im Abschn. 2.10. behandelt.)

2.6. Formelnotation

Mathematische Formeln, die mittels der Operationen Addition, Subtraktion, Multiplikation und Division gebildet werden, sind "wie üblich" zu notieren. Zu beachten ist lediglich, dass der Multiplikationsoperator mitgeschrieben werden muss und dass zwei Divisionsoperatoren vorhanden sind, nämlich

- `/` für INTEGER- und REAL-Operanden, der Quotient ist vom Typ REAL,
- `div` für INTEGER-Operanden, das Ergebnis ist der ganzzahlige Anteil des Quotienten.

Als Operanden dürfen neben Konstanten und Variablen auch Funktionen (Betragsbildung, Wurzel u. ä. sind u. a. vordefiniert) benutzt werden.
Letztlich ist also die Formelnotation dem gewohnten Umgang weitgehend angepasst. In den folgenden Abschnitten werden wir diese globalen Aussagen präzisieren und an Beispielen erläutern.

2.6.1. Arithmetische Ausdrücke und numerische Standardfunktionen

"Ausdruck" ist in TURBO-PASCAL die Bezeichnung für eine Konstruktion, die berechnet (ausgewertet) werden kann und einen Wert liefert. Die Vorrangregeln entstehen dabei durch eine stufenweise Definition des Ausdrucks (vgl. Anhang B).
Ausgegangen wird von elementaren Bauelementen, das sind Konstanten, Variablen, in runde Klammern eingeschlossene Formelnotationen (d. h. eingeklammerte Ausdrücke) und sog. Funktionsaufrufe.
Beispiele für solche elementaren Bausteine sind:

12.9 100 x (y+1.4) SIN(t)

Diese Bausteine heissen Faktoren.
Faktoren können durch Multiplikationsoperatoren verknüpft werden; (arithmetische) Multiplikationsoperatoren sind * / div mod.

* liefert das Produkt der Operanden. Da die Operanden von einem bestimmten Typ sind, muss etwas ausgesagt werden über den Typ des Produkts. Es gilt: Das Produkt ist vom Typ REAL, wenn wenigstens einer der Operanden von diesem Typ ist, sonst ist es vom Typ INTEGER.

/ liefert den Quotienten. Die Operanden können von beliebigem arithmetischem Typ sein, der Quotient ist vom Typ REAL.

div kann nur benutzt werden, wenn beide Operanden vom Typ INTEGER sind; als Ergebnis erhält man den ganzzahligen Teil des Quotienten gemäss folgender Festlegung:

i div j liefert die grösste ganze Zahl, die nicht grösser als i/j ist, wenn i und j beide positiv oder beide negativ sind, liefert das eben beschriebene Resultat mit negativem Vorzeichen, wenn genau ein Operand negativ ist.

mod ist nur anwendbar auf INTEGER-Operanden. Das Ergebnis ist der ganzzahliger Rest bei ganzzahliger Division, es berechnet sich folgendermassen:

```
i mod j = i - (i div j) * j
```

Nachdem in einem Ausdruck alle Multiplikationen und/oder Divisionen ausgewertet wurden, werden Additionen und/oder Subtraktionen ausgeführt.
Die mathematische Formelnotation

$$\frac{(b-a)(ab-4a+2b)}{6n}$$

wird also notiert als

```
(b - a)*(a*b - 4*a + 2*b)/(6*n)     oder
(b - a)/(6*n) * (a*b - 4*a + 2*b)   oder
(b - a)/(6.0*n)* (a*b - 4.0*a + 2.0*b)
```

Das Ergebnis der Berechnung dieses Ausdrucks ist vom Typ REAL (wegen /).

Oben wurde schon gesagt, dass es in TURBO-PASCAL vordefinierte arithmetische Funktionen gibt. Die Bezeichner dieser Funktionen sind also dem TURBO-PASCAL-System bekannt.
Die Benutzung dieser Funktionen in Ausdrücken erfolgt dadurch, dass der Bezeichner angegeben wird, dem, in runde Klammern eingeschlossen, ein Funktionsargument folgt. Dieses Funktionsargument kann selbst ein Ausdruck sein. Folgende Bezeichner für arithmetische Funktionen sind vordefiniert(Standardfunktionsbezeichner):

ABS	Betrag	ARCTAN	Arcustangens
INT	ganzzahliger Teil	COS	Kosinus
		SIN	Sinus
FRAC	gebrochener Teil	EXP	Exponentialfunktion
		LN	natürlicher Logarithmus
		SQR	Quadrat
		SQRT	Quadratwurzel

Ausserdem gibt es zwei Funktionen, die auf Werte vom Typ REAL angewendet werden können und einen Wert vom Typ INTEGER liefern. TRUNC schneidet den gebrochenen Teil einer Zahl ab und liefert den ganzzahligen Teil (wie auch INT). ROUND ist wie folgt definiert:

```
ROUND(x) = TRUNC(x+0.5)   , falls x >= 0
           TRUNC(x-0.5)   , falls x < 0 ist.
```

Beispiele zur Formelnotation:

```
1.  ABS(x - 1)
2.  (((a*x + b)*x +c)*x +d)*x +e
3.  (-b +SQRT(SQR(b) - 4.0*a*c))/(2.0*a)
4.  EXP(-0.2*t)*(SIN(2.0*t + pi/4.0) +SIN(3.0*t) - 0.1)
5.  (i + j - 2)* ROUND(d)
```

2.6.2. Vergleiche, logische Verknüpfungen, logische Standardfunktionen und Shift-Operationen

Entsprechend der im letzten Abschnitt behandelten Notation mathematischer Formeln gibt es Ausdrücke, die nach ihrer Auswertung einen logischen Wert liefern, d. h. falsch oder wahr sind. Im Sprachgebrauch der Programmiersprachen heisst das, sie liefern entweder den Wert FALSE oder den Wert TRUE, das sind zwei vordefinierte Konstanten, sie bilden den Definitionsbereich des Standardtyps BOOLEAN.
Ausdrücke, die einen logischen Wert liefern, werden wiederum stufenweise definiert, um Vorrangregeln für logische Operatoren festzulegen.

Elementare Bausteine solcher "logischen Formelnotationen" sind die Konstanten(bezeichner) FALSE und TRUE, ferner Variablen vom Typ BOOLEAN, booleanwertige (Standard-) Funktionen, Vergleiche und in runde Klammern eingeschlossene logische Ausdrücke.

Zur Notation von Vergleichen dienen sechs Vergleichsoperatoren, nämlich

```
<, <=, =, >, >=, <> .
```

Beispiele für Vergleiche:

```
x < y
ABS(t) <= 1.0
SQRT( SQR (x) + SQR (y) ) > r
```

Als Vertreter booleanwertiger Standardfunktionen sei ODD genannt. Als Argument kann ein Ausdruck angegeben werden, der einen ganzzahligen Wert liefert:

ODD(i) liefert den Wert TRUE, wenn i ungerade ist, sonst FALSE.

Auf diese Bausteine kann der Operator **not** angewandt werden. Symbolisiert q einen logischen Ausdruck, so gilt:

q	**not** q
FALSE	TRUE
TRUE	FALSE

Die Werte elementarer Bausteine oder ihrer Negation können nun mittels Operatoren verknüpft werden. Für die "logische Multiplikation" (Konjunktion) dient der Operator **and**. Symbolisieren p und q logische Faktoren, so ist die Konjunktion durch folgende Tabelle definiert:

p	FALSE	FALSE	TRUE	TRUE
q	FALSE	TRUE	FALSE	TRUE
p **and** q	FALSE	FALSE	FALSE	TRUE

Logische Faktoren und durch **and** verknüpfte Faktoren heissen logische Terme.
Desweiteren steht der Operator **or** für die sog. "logische Addition" (Disjunktion) zur Verfügung. Symbolisieren wiederum p und q logische Terme, so ist die Wirkung von **or** durch folgende Tabelle erklärt:

p	FALSE	FALSE	TRUE	TRUE
q	FALSE	TRUE	FALSE	TRUE
p or q	FALSE	TRUE	TRUE	TRUE

Logische Terme oder durch **or** verknüpfte logische Terme heissen logische Ausdrücke.
Ausser dem Operator für die logische Addition gibt es mit gleicher Priorität den Operator **xor** für das exklusive Oder. Symbolisieren wiederum p und q logische Terme, so ist die Wirkung von **xor** durch folgende Tabelle erklärt:

p	FALSE	FALSE	TRUE	TRUE
q	FALSE	TRUE	FALSE	TRUE
p xor q	FALSE	TRUE	TRUE	FALSE

Die Berechnung eines booleanwertigen Ausdrucks wird demnach gemäss folgender Prioritäten ausgeführt:

not vor **and** vor **or** bzw. **xor** vor Vergleichsoperationen

Man beachte, dass **not**, **and** und **or** bzw. **xor** stärker binden als Vergleichsoperatoren!

```
a < 1 or b > 2
```

ist also falsch wegen des Vorrangs von **or**. Der Ausdruck würde als a<(1 **or** 2)> b aufgefasst werden, was syntaktisch falsch ist. Es muss notiert werden:

```
(a < 1) or (b > 2)
```

Die logischen Operatoren können aber nicht nur auf Operatoren vom Typ BOOLEAN angewendet werden, sondern auch auf Operatoren der Typen INTEGER oder BYTE. Die Verknüpfung ist dann bitweise definiert, wobei ein mit 0 belegtes Bit äquivalent zu FALSE, ein mit 1 belegtes äquivalent zu TRUE ist.
Das folgende Programmbeispiel zeigt die Anwendung der logischen Operatoren auf INTEGER-Werte und auf boolesche Variablen. Die Ergebniswerte sind wiederum als INTEGER-Werte ausgegeben worden. Man beachte, dass der Negationsoperator auf die ganze Zahl 13 angewendet, den Wert -14 liefert. Das liegt an der Komplementdarstellung der negativen ganzen Zahlen, für die gilt, dass die negative Zahl k dargestellt wird als die bitweise Negation von |k|-1, also -14 durch die Negation von |-14|-1 = 13. Die interne Darstellung der Zahlen 13 und 68 sind als Kommentar angegeben.

```
(*LOGOP.PAS*)
program logischeoperatoren(lst);
```

```
var i,j:INTEGER;
    p,q:BOOLEAN;
begin
    i := 13; {00000000 00001101}
    j := 68; {00000000 01000100}
    p := true;
    q := false;
    WRITELN(LST,i:12, j:12, p:12, q:12);
    WRITELN(LST,not i : 12,not j :12);
    WRITELN(LST, i and j : 12, p and q :24);
    WRITELN(LST, -i or j  : 12, p or  q :24);
    WRITELN(LST, i xor j   12, p xor q :24);
    WRITELN(LST);
    WRITELN(LST,i and not j : 24, p and not q : 24);
    WRITELN(LST,i or  not j : 24, p or  not q : 24);
    WRITELN(LST,i xor not j : 24, p xor not q : 24);
    end.
```

Resultatausgabe:

```
          13          68        TRUE        FALSE
         -14         -69
           4                   FALSE
          -9                    TRUE
          73                    TRUE

                       9                     TRUE
                     -65                     TRUE
                     -74                    FALSE
```

TURBO-PASCAL enthält darüber hinaus zwei Verschiebeoperatoren, die auf INTEGER-Werte angewandt werden können. Der Operator shr verschiebt einen ganzzahligen Wert um so viele Bits nach rechts (shift right), wie durch einen ganzzahligen Wert angegeben wird. Eine Verschiebung um eine Bitstelle entspricht also einer ganzzahligen Division durch 2. Der Operand wird dabei als vorzeichenlose Zahl aufgefasst, d. h., die linken Bits werden mit Nullen aufgefüllt.
Der Operator shl verschiebt einen ganzzahligen Wert entsprechend nach links. Jede Verschiebung um eine Bitstelle entspricht also einer Multiplikation mit 2, die rechten Bits werden mit Nullen aufgefüllt.

Beispiel:

```
program shrshl;
var i, j: INTEGER;
begin
   i := 18;  j := -6;
   WRITELN(i, j:15);
   WRITELN( i shr 2, j shr 2   15);
   WRITELN( i shl 2, j shl 2   15);
end.
```

Bei der Abarbeitung erhält man folgende Ergebnisse:

```
18              -6
4           16382
72             -24
```

2.7. Das Zeichenkettenkonzept

Zeichenketten (Strings) können in TURBO-PASCAL als Konstanten, Variablen und als Werte von Ausdrücken auftreten. Stringkonstanten sind Folgen von Zeichen des ASCII-Zeichensatzes, d. h. von druckbaren Zeichen und Steuerzeichen. Wir betrachten zuerst druckbare Zeichen. Folgen von druckbaren Zeichen werden in Apostrophe eingeschlossen.

Beispiele:

```
'TURBO-PASCAL'   '***12***'
```

Ein Apostrophzeichen in einer Stringkonstanten muss doppelt geschrieben werden, z. B. 'cs'''.
Stringkonstanten kann innerhalb eines Konstantendefinitionsteils ein Bezeichner zugeordnet werden, z. B.

```
const bsp = 'Beispiel';
```

In TURBO-PASCAL ist für das Arbeiten mit Zeichenkette ein spezieller Datentyp aufgenommen worden, der durch das Basissymbol **string** notiert wird. Diesem Basissymbol folgt, in eckige Klammern eingeschlossen, eine Angabe über die maximale Anzahl von Zeichen, also eine Präzisierung des konkreten Zeichenkettentyps. Durch die Variablendeklaration

```
var s1, s2: string[64];
```

werden also die Variablen s1 und s2 eingeführt, von denen jede 0 bis 64 Zeichen aufnehmen kann. Die Anzahl der aktuell vorhandenen Zeichen heisst Länge der Stringvariablen. Die Positionen der Stringvariablen sind einzeln adressierbar, in unserem Beispiel mit

s1[1], ..., s2[64]. Ausserdem belegt jede Stringvariable noch einen zusätzlichen Speicherplatz mit dem Index 0, auf dem die augenblickliche Länge der Stringvariablen gespeichert ist. Es sind also Längenangaben von 0 bis 255 möglich.
Die Komponenten einer Stringvariablen sind vom Standardtyp CHAR. In diesem Sinne sind die Typen CHAR und **string** miteinander verträglich.
Stringkonstanten und/oder Stringvariablen können miteinander verkettet werden, als Verkettungsoperator dient das +-Zeichen, das Ergebnis ist eine Zeichenkette.

Beispiel:
'31.' + 'Dez.' + '1986' ergibt '31.Dez.1986'

Zeichenketten können miteinander verglichen werden, dabei gelten für die Berechnung der Zeichen die Werte nach ASCII. Zeichenketten sind nur dann gleich, wenn sie gleiche Länge haben und gleiche Zeichenfolgen darstellen.

Beispiele:

'a' < 'A'	liefert den Wert FALSE
'3' < '15'	liefert den Wert FALSE, weil '3' > '1' ist
'xy' < 'xyz'	liefert den Wert TRUE (bei Gleichheit des führenden Teilstrings wird der kürzere als < behandelt).

Analog zu den arithmetischen und logischen Standardfunktionen der Abschnitte 2.6.1. und 2.6.2. enthält das Zeichenkettenkonzept Standardfunktionen für das Arbeiten mit Zeichenketten. In der folgenden Aufzählung bedeuten

st, st1, st2	Zeichenkettenausdrücke
position	einen Wert wischen 1 und 255 (sonst Laufzeitfehler)
anzahl	einen positiven INTEGER-Wert.

TURBO-PASCAL hat folgende Standardfunktionen für Strings:

LENGTH(st)	Ergebnis ist die Länge von st
POS(st1,st2)	Die Funktion bestimmt das erste Auftreten von st1 in st2 und liefert als Ergebnis den Index des ersten Zeichens dieser übereinstimmenden Sequenz. Tritt st1 in st2 nicht auf, so ist der Funktionswert 0. Beispiel:

```
ianfang := POS('{', zeile);
iende := POS('}',zeile);
DELETE(zeile, ianfang, iende-ianfang-1);
```

CONCAT(st1,st2,...)
Die Funktion verknüpft die Werte der Stringausdrücke gemäss der zuvor beschriebenen Operation + und liefert die dadurch erzeugte Zeichenkette. Ist LENGTH(st1+st2+...) > 255, so tritt ein Laufzeitfehler auf.

COPY(st, position, anzahl)
Von der durch position beschriebenen Komponente des Stringausdrucks st werden anzahl Zeichen als Funktionswert geliefert;
COPY('123.45', 3, 3) liefert also '3.4'. Ist der Wert von position > LENGTH(st), so ist der Funktionswert die leere Zeichenkette, ist
position + anzahl > LENGTH(st),
so wird der Rest von st ab Index position als Funktionswert geliefert. Liegt der Wert von position nicht in 1..255, so wird ein Laufzeitfehler angezeigt.

Diese Standardfunktionen können in Zeichenkettenausdrücken wie Zeichenkettenkonstanten oder Zeichenkettenvariablen benutzt werden. Neben den eben beschriebenen Zeichenkettenfunktionen sind in TURBO-PASCAL vier Standardprozeduren für Zeichenketten aufgenommen worden. In der folgenden Aufzählung haben die Argumente die gleiche Bedeutung wie die Standardfunktionen, ausserdem werden benutzt:

wert	Ausdruck, der einen numerischen Wert liefert
variable	INTEGER- oder REAL-Variable
stvar	Variable von einem Stringtyp
inf	INTEGER-Variable.

Die Standardprozeduren für Zeichenketten sind:

DELETE(stvar, position, anzahl)
In der Stringvariablen werden anzahl Stellen gelöscht, beginnend an der Stelle mit dem Index position. Ist position > LENGTH(stvar), so werden keine Zeichen gelöscht;
ist der Wert des ganzzahligen Ausdrucks position > 255, so tritt ein Laufzeitfehler auf.

INSERT(st, stvar, position)

Der Wert des Stringausdrucks st wird von der durch position beschriebenen Stelle in stvar eingefügt. Wenn position > LENGTH(stvar), so wird st an stvar anfgefügt. Wirdnach dem Einfügen oder Anfügen die in der Deklaration von stvar definierte Länge überschritten, so wird der "Überhang" abgeschnitten. Ein Laufzeitfehler tritt wie bei DELETE auf.

Beispiele:

```
Einfügen   is := POS('teln(', zeile);
              INSERT('lst,',zeile, is + 5);
Ersetzen    DELETE(zeile, is, LENGTH('array[1..8] of CHAR'));
           INSERT('string[8]',zeile, is);
```

STR(wert, stvar)

Der Wert von wert wird in eine Zeichenkette umgewandelt, wobei die Standarddarstellung numerischer Konstanten benutzt wird; diese Zeichenkette wird stvar als Wert zugewiesen. (Das Format der erzeugten Zeichenkette kann durch zusätzliche Angaben, wie sie in der WRITE-Anweisung (s. Abschn. 2.11.2.) möglich sind, beeinflusst werden .)

Beispiel:

```
summe: string[10];     i := j*k + 1; STR(i, summe:10);
```

VAL(st, variable, inf)

Diese Prozedur konvertiert den Wert des Zeichenkettenausdrucks in eine interne Zahlendarstellung und weist diesen numerischen Wert der Variablen variable zu.
inf erhält den Wert 0, wenn diese Zuweisung korrekt ausgeführt wird, sonst erhält inf den Index der Stelle in st, die als falsch (Verstoss gegen die Syntax der Zahlendarstellung) erkannt wurde.

Beispiel: VAL(st +'.00', rval, ok);

2.8. Anweisungen

Die Aktionen, die mit den zuvor im Programm deklarierten Bezeichnern von Objekten bzw. den vordefinierten Namen ausgeführt werden sollen, werden als Anweisungen notiert. In höheren Programmiersprachen - und so auch in TURBO-PASCAL - wird zwischen einfachen und strukturierten Anweisungen unterschieden. Strukturierte Anweisungen enthalten als Teile der Konstruktion wiederum Anweisungen, und zwar einfache und/oder strukturierte; einfache Anweisungen enthalten keine weiteren Anweisungen. Der Begriff Anweisung unfasst sowohl

einfache als auch strukturierte Anweisungen, i. allg. genügt es, diesen Begriff zu verwenden.

2.8.1. Die Wertzuweisung

Mittels der Wertzuweisung wird einer Variablen der Wert eines Ausdrucks zugewiesen. Der Ausdruck ist eine solche Formelnotation, wie sie im Abschn. 2.6. beschrieben wurde. Für die Zuweisung wird das Operationssymbol := eingeführt, so dass die Wertzuweisung insgesamt (in symbolischer Schreibweise) die Gestalt

```
Variable := Formelnotation
```

hat.

Beispiele:
Mit den Variablendeklarationen

```
var x,y:REAL;
    i:INTEGER;
    c:CHAR;
    st: string[12];
    p:BOOLEAN;
```

sind folgende Wertzuweisungen korrekt:

```
x := 1.6;   y := SQRT(x) + 0.5;
i := 20;  c := '*';
st := 'FELD.U';
p := (i*x + y) < 100;
```

Betrachten wir aber die Wertzuweisung

```
Variable := Formelnotation
```

allgemein, so muss man genauere Aussagen über den Typ der Variablen und über den Typ des Wertes machen, der sich nach der Auswertung der Formelnotation ergibt. Sind diese beiden Typen gleich, so kann die Wertzuweisung ausgeführt werden. Im Falle von Ungleichheit ist jedoch zwischen den Fällen zu unterscheiden, in denen automatisch eine Typanpassung erfolgt - das TURBO-PASCAL-System sieht entsprechende Konvertierungen vor - und den Fällen, wo vom System eine Typanpasssung nicht durchgeführt wird.Im letzten Fall werden bei der Compilation Fehler angezeigt.
In der Tafel 2.1. ist Information über Typverträglichkeit bei Wertzuweisungen zusammengestellt.

Tafel 2.1. Typverträglichkeit bei Wertzuweisungen

Formeltyp

Var.-Typ	INTEGER	BYTE	REAL	CHAR	BOOLEAN	string
INTEGER	+	+	-	-	-	-
BYTE	+	+	-	-	-	-
REAL	+	+	+	-	-	-
CHAR	-	-	-	+	-	-
BOOLEAN	-	-	-	-	+	-
string	-	-	-	+	-	+

+ bedeutet dabei, dass die Zuweisung erlaubt ist.

2.8.2. Die Prozeduranweisung

Unter Prozeduren werden Routinen verstanden, die mehr oder minder umfangreiche Aktionen innerhalb des Programms ausführen. Prozeduren können im Vereinbarungsteil deklariert oder als vordefinierte Prozeduren des TURBO-PASCAL-Systems bekannt sein. Das Deklarieren von Prozeduren und ihre Verwendung wird im Abschn. 2.13. besprochen. Hier soll an einige Standardprozeduren erinnert werden, wie sie in den Abschnitten 2.4. und 2.7. behandelt wurden: WRITE, READ, DELETE, INSERT u. ä.
Die Verwendung dieser Prozedurbezeichner mit den geforderten Argumentangaben nennt man eine Prozeduranweisung.

Beispiele (es werden die Bezeichner aus Abschn. 2.8.1. verwendet):

```
WRITE(st + '[', i, ']=', x);
READ(c, y);
DELETE(st, 5, 2);
INSERT('01', st, 5);
```

2.8.3. Die Sprunganweisung

Der Sinn von Sprunganweisungen ist es, die sequentielle Abarbeitung der Anweisungen eines Programms zu unterbrechen und an einer markierten Stelle fortzusetzen. Um das zu realisieren, werden einerseits Markierungen benötigt, um Fortsetzungsstellen (das sind Anweisungen) zu kennzeichen, andererseits Anweisungen, die die Programmfortsetzung steuern.
Zur Markierung von Anweisungen werden Marken verwendet, sie müssen im Vereinbarungsteil definiert werden. Das erfolgt in der Markende-

finition

```
label m1, m2, ..., mn;
```

label ist das die Markendefinition kennzeichnende Basissymbol, m1, m2, ... symbolisieren Marken. Als Marken dürfen Folgen aus Ziffern und Buchstaben verwendet werden. Kommas trennen die Marken.

Beispiel:

```
label 1, 11, 111, wiederhole, ende;
```

So definierte Marken dürfen dann im Anweisungsteil Anweisungen vorangestellt werden, wobei der Doppelpunkt als Trennzeichen dient.

Beispiel:

```
1: x := 0;
wiederhole: READ(y);
 ende: WRITELN('y=',y);
```

Eine so gekennzeichnete Anweisung wird in einem Programm durch

```
goto marke
```

erreicht.
Prinzipiell ist die Verwendung von Sprunganweisungen nicht zu empfehlen, weil dadurch leicht die logischen Zusammenhänge im Programm zerstört werden. (In bezug auf den Gültigkeitsbereich von Marken s. Abschn. 2.14.3.)

2.8.4. Die Verbundanweisung

Mehrere Anweisungen, einfache oder strukturierte, können zusammengefasst werden, so dass sie syntaktisch als eine Anweisung fungieren. Das geschieht einfach dadurch, dass die zusammenzufassenden Anweisungen durch Semikola getrennt aufgezählt und insgesamt in die Basissymbole **begin** und **end** eingeschlossen werden.

Beispiel:

```
begin READ(x);
      y := x + 2.5
end
```

Die Verbundanweisung ist also eine strukturierte Anweisung.
Man beachte, dass das Semikolon als Trennzeichen für Anweisungen steht. Wird das letzte Beispiel abgeändert in

```
begin READ(x);
      y := x + 2.5;
end
```

so sind das drei einfache Anweisungen. Das Fehlen einer Anweisung zwischen dem letzten Semikolon und dem abschliessenden **end** wird als Leeranweisung aufgefasst. Die Leeranweisung ist also eine spezielle einfache Anweisung, sie kann auch markiert werden.

Beispiel:

```
begin i := 0;
      ...
      st := ' '; 111:
end
```

Der Anweisungsteil eines Programms als Ganzes ist eine Verbundanweisung. Ansonsten benötigt man Verbundanweisungen in bedingten Anweisungen (s. Abschn. 2.8.5.), in Zyklenanweisungen (s. Abschn. 2.8.6.) und in Routinendeklarationen (s. Abschn. 2.13.).

2.8.5. Bedingte Anweisungen

In TURBO-PASCAL gibt es zwei Formen der bedingten Anweisung, nämlich die IF-Anweisung und die CASE-Anweisung.
In der IF-Anweisung wird die Ausführung einer Anweisung von dem Erfülltsein oder Nicht-Erfülltsein einer Bedingung, d. h. vom Wert eines logischen Ausdrucks, abhängig gemacht.
In symbolischer Notation hat die IF-Anweisung eine der folgenden Formen:

```
if bedingung then anweisung
```

oder

```
if bedingung then anweisung1 else anweisung2
```

Die Anweisungen können dabei einfach oder strukturiert sein, insbesondere können es wieder bedingte Anweisungen sein.
Ein auftretendes **else** gehört stets zu dem letzten **then**.

Beispiele:

```
if a > x then WRITELN('a > x erreicht');

if ABS(t) <= 1.0 then begin i := i + 1;
                            t := 2*t;
                      end;

if (x<0) or (x>4) then goto 11
```

```
                    else WRITELN('x=', x);

    if  (gehalt<=600.00) or (gehalt <= 1200) and zvr
                          then sv  :=  0.1*gehalt
                          else
              if not zvr then sv := 60
                         else {(gehalt > 1200) and zvr} sv := 120
```

Zur Kodierung von Mehrfachverzweigungen dient die CASE-Anweisung. Sie besteht aus einer Folge von einfachen oder strukturierten Anweisungen, die durch Auswahlmarken gekennzeichnet sind. Die Auswahl wird gesteuert durch den Wert eines Ausdrucks. Es wird dann die Anweisung abgearbeitet, deren Auswahlmarke dem Wert des auswählenden Ausdrucks gleich ist oder ihn enthält. Tritt der Wert des auswählenden Ausdrucks nicht unter den Anweisungskennzeichen auf, so wird keine Anweisung aus der CASE-Anweisung ausgeführt, es sei denn, die CASE-Anweisung endet mit einem ELSE-Zweig.

Symbolische Darstellung:

```
case auswahl of
 m1: anweisung1;
 m2: anweisung2;
 ...
 mn: anweisungn;
 else anweisung
end
```

(Der ELSE-Zweig kann fehlen.)
Die mi sind Werte vom gleichen Typ wie auswahl, der Typ muss einfach und nicht REAL sein. Als Auswahlmarken können auch Teilbereiche angegeben werden.
Beispiel:

```
case zeichen of
 'a'..'z': kb := kb + 1;
 'A'..'Z': gb := gb + 1;
 '0'..'9': z := z + 1;
 ' '     : s := s + 1;
 else sonst := sonst + 1
end
```

2.8.6. Kodierung von Zyklen

Für die Notation von zyklischen Abläufen gibt es ein Sprachelement - die FOR-Anweisung -, das durch eine Zählvariable gesteuert wird.

Diese Zählvariable wird in ihrem Definitionsbereich jeweils um +1 oder -1 verändert, bis ein Grenzwert überschritten wird.
Zwei weitere Sprachelemente - WHILE-Anweisung und REPEAT-Anweisung - sind so aufgebaut, dass die Wiederholung des zyklischen Teils vom Wert eines logischen Ausdrucks abhängig gemacht wird.

Wir behandeln hier den Aufbau dieser drei Sprachelemente und geben Prinzipbeispiele an; weitere Beispiele findet man im Abschn. 2.9.4.
Die Zyklenanweisungen sind im einzelnen wie folgt aufgebaut:

1. FOR-Anweisung
Symbolische Darstellung:

```
for steuervariable := anfangswert to endwert do anweisung
```

bzw.

```
for steuervariable := anfangswert downto endwert do anweisung
```

steuervariable bedeutet darin eine einfache Variable, deren Typ nicht REAL ist (meist INTEGER).
anfangswert und endwert symbolisieren Ausdrücke, deren Werte mit dem Typ der steuervariablen verträglich sind.

Beispiele:

```
(1)  for j := 1 to 9 do WRITELN(j, SQR(j):10, j*j*j:10)
(2)  i := 65;
     for c := 'A' to 'Z' do begin
                            WRITELN(c, i:10);
                            i := i+1;
                            end
(3)  t := 1.0;
     for n := m1 downto 0 do t := t*x
```

Anmerkung:
Ist der endwert von vornherein überschritten bzw. unterschritten, so wird die anweisung nicht ausgeführt.

2. WHILE-Anweisung
Symbolische Darstellung:

```
while bedingung do anweisung
```

anweisung wird solange ausgeführt, wie die Berechnung von bedingung den Wert TRUE liefert (möglicherweise also gar nicht). Die Werte von einer oder mehreren Variablen werden in anweisung - meist ist

es eine Verbundanweisung - geändert.

Beispiele:

```
(1)  while ABS(a-b) > 0.001 do
          begin c := (a+b)/2;  n := n+1;
                if ODD(n) then a := c
                          else b := c
          end


(2)  x1 := a;  x0 := 0.0;
     while ABS(x1-x0) > 1.0e-9 do
          begin x0 := x1;
                x1 := (SQR(x0) +a/SQR(x0)) /3.0;
          end
```

3. REPEAT-Anwisung
Symbolische Darstellung:

```
     repeat
     anweisung1;
     .
     .
     .
     anweisungn
until bedingung
```

Die Anweisungsfolge

```
     anweisung1;...;anweisungn
```

wird mindestens einmal durchlaufen. Hat dann der logische Ausdruck bedingung den Wert FALSE, so wird die Anweisungsfolge ein weiteres Mal ausgeführt.
Die REPEAT-Anweisung ist abgearbeitet, wenn bedingung den Wert TRUE liefert.

Beispiele:

```
(1)  repeat
     READ(c); st := st+c;
     until c=' '
(2)  x1 := a;
     repeat
     x0 := x1;
     x1 := (SQR(x0) +a/SQR(x0)) /3.0;
     until ABS(x1-x0) < 1.0e-9
```

Dieses Beispiel ist hinsichtlich der Wirkung dem Beispiel (2) zur WHILE-Anweisung äquivalent.

Übungen:
Im folgenden werden Aufgaben formuliert, die geeignet sind, die Verwendung der bisher besprochenen Sprachelemente zu üben. Da man eine Programmiersprache am besten dann erlernt, wenn man in ihr Algorithmen formuliert - also ein paar Programmzeilen notiert - sei dem Anfänger empfohlen, diese Aufgaben in TURBO-PASCAL-Notation umzusetzen.

A2.8.-1: Erzeugen einer Zahl
Eine Folge von Ziffern werde zeichenweise eingelesen. Die Ziffernfolge ist in eine INTEGER-Zahl zu verwandeln. Ein Zeichen, das nicht Ziffernzeichen ist, schliesst die Eingabe ab.
Anleitung: Eine Variable c vom Typ CHAR kann durch

```
INTEGER(c)
```

in eine ganze Zahl umgewandelt werden, der dem Wert des in der Variablen c gespeicherten Zeichens entspricht, den dieses Zeichen im ASCII (vgl. Anhang A) hat.

A2.8.-2: Tabelle des Werteverlaufes einer Funktion
Die Funktion

$$y = (0.4e^{-0.2t}+0.1)(2\cos t + 0.5\cos 2t)$$

ist im Intervall [a,b] mit der Schrittweite dt zu tabellieren. Die Wertausgabe soll auf dem Bildschirm erfolgen. Nach der Ausgabe von 10 Werten ist die Ausgabe bis zum Drücken der Return-Taste (carriage return) zu unterbrechen. Die Werte für a,b und dt sind vom Bildschirm einzugeben.

A2.8.-3: Grösster gemeinsamer Teiler
Einzugeben sind Zahlenpaare (a,b), a,b ganzzahlig. Es ist für jedes Zahlenpaar der grösste gemeinsame Teiler ggt(a,b) zu berechnen und auszugeben. Der Programmzyklus soll beendet werden, wenn 0 für a eingegeben wird.

A2.8.-4: Berechnen eines Funktionswertes
Die Funktion

$$f(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^n}{(n+1)(n+2)} \quad , \quad |x| < 1$$

ist an der Stelle x = x0 zu berechnen.

2.9. Typendefinitionen und Variablendeklarationen

Im Abschn. 2.4.1. wurden solche Typen und Variablen behandelt, deren Bedeutung unmittelbar klar ist.
In diesem Abschnitt wird gezeigt, welche weiteren Möglichkeiten es in TURBO-PASCAL gibt, Datentypen zu beschreiben und Variablen einzuführen, die diese Datentypen repräsentieren.

2.9.1. Teilbereichstypen, Skalartypen, Typendefinition und Typwandlung

Typen beschreiben Definitionsbereiche.
Die im Abschn. 2.4.1. definierten Typen INTEGER, CHAR, BOOLEAN werden i. allg. Aufzählungstypen genannt. Von diesen Typen können Teilbereichstypen gebildet werden, z. B.

```
     0..99
oder 'A'..'Z'
```

Teilbereichstypen kann ein (Typ-)Bezeichner zugeordnet werden. Das erfolgt in einer Typdefinition, die für obige Beispiele so aussehen könnte:

```
     type index = 0..99;
          buchst = 'A'..'Z';
     type BYTE = 0..255;
```

(Der Typ BYTE ist in TURBO-PASCAL vordefiniert.)
Die Typen bzw. Typbezeichner können nun zum Festlegen der Definitionsbereiche von Variablen benutzt werden (den Variablen werden Typen zugeordnet).

Beispiel:

```
     var i1, i2: index;
         b1, b2, b3: buchst;
         b: 'A'..'Z';
```

Man beachte: Variablen sind in Operationen nur verträglich, wenn sie sich auf den gleichen Typbezeichner beziehen oder gemeinsam in einer Variablendeklaration vereinbart wurden oder Teilbereiche des gleichen Aufzählungstyps sind.

Als Typ kann weiterhin eine in Klammern eingeschlossene Aufzählung von Bezeichnern benutzt werden, sog. Skalartypen.

Beispiele:
(rot, blau, gelb, gruen)
(Koenig, Dame, Laeufer, Springer, Turm)

Für Skalartypen werden wie oben Typbezeichner eingeführt, also

```
type figur = (Koenig, Dame, Laeufer, Springer, Turm)
```

und entsprechend Variablen

```
var g1, g2: figur;
```

denen dann Werte aus dem Typ figur zugewiesen werden können, z. B.

```
g1 := Dame;
```

Die Bezeichner in der Typdefinition sind Konstantenbezeichner, und es gilt für obiges Beispiel

```
Koenig < Dame < Laeufer < Springer < Turm
```

(was der Bewertung beim Schachspiel nicht entspricht).
Der im Abschn. 2.4.1. eingeführte Skalartyp BOOLEAN ist vordefiniert durch

```
type BOOLEAN = (FALSE,TRUE)
```

Skalartypen sind ebenfalls Aufzählungstypen.
Für die Verwendung von Aufzählungstypen sind weitere Standardfunktionen erklärt (a symbolisiert einen Wert aus einem Aufzählungstyp).

PRED(a)	Vorgänger von a
SUCC(a)	Nachfolger von a
ORD(a)	Position von a in der Typdefinition, wobei die Zählung mit 0 beginnt.

Besonders bequem ist in TURBO-PASCAL die Konvertierung des Wertes eines Typs in einen anderen. Dafür können die Typbezeichner wie Standardfunktionen benutzt werden. Die Funktion wandelt dann das Argument in den entsprechenden Wert des Typs, den der Typbezeichner angibt.

Beispiele:
figur(2) ergibt den Wert Laeufer

```
INTEGER ('a') ergibt den Wert 97 (vgl. Anhang A)
CHAR(¤5B) ergibt das Zeichen '['
```

2.9.2. Array-Typen und Array-Variablen

Objekte gleichen Typs können zu Feldern (arrays) zusammengefasst werden. Zur Beschreibung dieser Zusammenfassung wird ein weiterer Typ benutzt, der Array-Typ.
Der Array-Typ wird wie folgt notiert:

```
array[indexliste] of komponententyp
```

komponententyp ist der Typ der Objekte, die in dem Feld zusammengefasst sind. Die Komponenten können von einfachem oder strukturiertem Typ sein. indexliste besteht aus einem Index oder aus einer durch Kommas getrennten Aufzählung mehrerer Indizes. Jeder Index ist von einem Aufzählungstyp. Der Typbezeichner INTEGER ist als Indextyp verboten.

Beispiele:

```
array[0..127] of REAL
array [BYTE] of CHAR
array[1..12] of array [1..n] of INTEGER
```

Im dritten Beispiel ist der Komponententyp auch ein Array-Typ, dafür ist die kürzere Notation

```
array[1..12, 1..n] of INTEGER
```

gestattet. n bezeichnet eine (bereits definierte) Konstante. Array-Typen können in einer Typendefinition an einen Typbezeichner gebunden werden.

Beispiele:

```
type vektor = array[0..127] of REAL;
     brett = array['a'..'h', 1..8] of figur;
     matrix = array[1..12, 1..n] of INTEGER;
```

Variablenbezeichner, die an einen Array-Typ gebunden sind, heissen Array-Variablen.

Beispiel:

```
var x,y: vektor;
    tt: array[1..20, 1..20] of REAL;
```

```
a: matrix;
s: array[0..5] of string[14];
```

Zu den Komponenten eines Feldes erfolgt der Zugriff durch Angabe der Array-Variablen und durch Positionsangaben. Die Positionsangaben sind Indexausdrücke, d. h. Ausdrücke, die nach ihrer Berechnung einen Wert ergeben, der in dem im Typ definierten Indexbereich liegt. (Wird der Indexbereich verlassen, so wird die Programmabarbeitung fehlerhaft sein.) Durch Direktiven (vgl. Abschn. 2.15.) kann erreicht werden, dass das Verlassen des Indexbereichs gemeldet wird.
Die Feldkomponenten heissen "indizierte Variablen".

Beispiele:

```
x[1]
y[(i+j)*2]
tt[k, k+1]
a[j, j]
s[0]
```

Der Typ der indizierten Variablen ist der entsprechende Komponententyp, der Typ von s[0] ist also **string**[14].
Indizierte Variablen können wie einfache Variablen benutzt werden (Beispiele s. Abschn. 2.9.5.).
Felder mit dem Komponententyp CHAR machen in dieser Behandlung eine Ausnahme. Solche Felder dürfen nur einen Index haben. TURBO-PASCAL behandelt sie als verträglich mit Variablen des String-Typs, wobei das Zeichenfeld (**array**[...] **of** CHAR) allerdings eine konstante Länge hat. Felder des Komponententyps CHAR können in String-Ausdrücken verwendet werden; String-Konstanten können Zeichenfeldern bei Längengleichheit zugewiesen werden; String-Ausdrücke können Zeichenfeldern nicht zugewiesen werden.

Für Variablen von einem Array-Typ sind folgende Operationen erklärt:

:=		Der Wert einer Array-Variablen kann einer andereren gleichen Typs zugewiesen werden.
=	<>	Die Werte von eindimensionalen Array-Variablen des Komponententyps CHAR können hinsichtlich Gleichheit oder Ungleichheit verglichen werden(lexikografisch).

In TURBO-PASCAL gibt es darüber hinaus zwei Standardfelder (MEM und PORT), mit deren Hilfe Hauptspeicherplätze bzw. Ports direkt adressiert werden können. Auf diese vordefinierten Felder wird im Abschn. 2.13. eingegangen.

Beispiele und Übungen

Da das Arbeiten mit Matrizen bei mathematischen, technischen aber auch ökonomischen Aufgaben heute zu jeder Grundlagenausbildung gehört, sollen auch an dieser Stelle Aufgaben genannt werden, um Algorithmen zu kodieren, die sich dieser Objekte bedienen. (Lösungen im Abschn. 4.)

A2.9.2.-1: Zeilensumme und Spaltensumme einer Matrix
Gegeben sei eine Matrix mit m Zeilen und n Spalten. z und s seien zwei Vektoren. Es sollen die Komponenten von z und s berechnet werden, und zwar so, dass z[i] die Summe der Elemente der i-ten Zeile der Matrix ist, s[j] entsprechend die Summe der j-ten Spalte.

A2.9.2.-2: Maximale Komponente eines Vektors
Gegeben sei ein Vektor a mit n Komponenten. Man gebe den Wert der maximalen Komponente und ihren Index an!

A2.9.2.-3: Summe der Diagonalelemente einer quadratischen Matrix
a sei n*n-Matrix. Man berechne die Summe der Elemente in den Diagonalen und gebe sie aus!

A2.9.2.-4: Summe der Elemente oberhalb der Hauptdiagonalen
a sei n*n-Matrix. Es ist die Summe der Elemente zu bilden, deren Zeilenindex kleiner als der Spaltenindex ist.

2.9.3. Record-Typen, Record-Variablen und With-Anweisung

Während im Array-Typ nur Komponenten mit gleichen Attributen (Komponenten gleichen Typs) zusammengefasst wurden, dient der Record-Typ dazu, eine feste Anzahl von Objekten beliebigen Typs zu einer Datenstruktur zusammenzufassen. Jede Komponente wird durch einen Bezeichner mit nachgestellter Typangabe gekennzeichnet. die Komponenten heissen Datenfelder, die Komponentenbezeichner entsprechend Datenfeldbezeichner. Datenfelddefinitionen eines Record-Typs werden mit Semikolon abgeschlossen. Die Zusammenfassung der Datenfelder zu einem Record erfolgt durch die Basissymbole **record** und **end**.

Beispiel:

```
record schl_nr: array [1..12] of CHAR;
       soll, ist: INTEGER;
       st_preis:INTEGER;
       lagerplatz: CHAR;
end;
```

Einem Record-Typ kann in einer Typdefinition ein Typbezeichner zugeordnet werden.

Beispiel:

```
type material = record schl_nr: array [1..12] of CHAR;
                              . . .
                end;
```

In dem Beispiel ist die Komponente schl_nr selbst strukturiert. Das ist uneingeschränkt möglich, d. h., ein Record-Typ kann Datenfelder enthalten, die selbst wieder von einem Record-Typ sind.

Beispiel:

```
type probe = record x: array[1..15] of REAL;
                    mw: REAL;
             end;
     charge = record umfang: INTEGR;
                    herstellungsdatum:
                           array[1..6] of charge;
                    kontrolle: probe;
              end;
```

Jeder Record-Typ eröffnet ein neues Bezeichnungsniveau, d. h., es dürfen auch solche Bezeichner für Datenfelder verwendet werden, die in vorhergehenden Programmteilen schon benutzt wurden.

Beispiel:

```
var z: REAL;
type yz = record y: REAL;
             z: array [1..8] of INTEGER;
          end;
```

Record-Variablen können in Variablendeklarationen dadurch eingeführt werden, indem Variablenbezeichnern ein Record-Typ zugeordnet wird.

Beispiel:

```
var x, y: record a, b: INTEGER;
                 abst: REAL;
          end;
    probe0, probe1: probe;
    prod: charge;
    monatsprod : array[1..31] of charge;
    y_z: yz;
```

Der Zugriff zu den Komponenten von Record-Variablen erfolgt so, dass der Bezeichner der Record-Variablen notiert wird, es folgt, durch einen Punkt getrennt, der Datenfeldbezeichner. Der Punkt dient also zum Selektieren eines Datenfeldes. Ist das Datenfeld ebenfalls strukturiert, so ist durch Indexangaben oder weiteres Selektieren die Komponente der inneren Struktur auszuwählen. In bezug auf obige Variablendeklarationen bedeuten also die folgenden Notationen sinnvolle Zugriffe zu Komponenten der Variablen:

Beispiele:

Variable	Typ
x.a	INTEGER
y.abst	REAL
probe0.x	**array**[1..15] **of** REAL
probe0.x[3]	REAL
prod.umfang	INTEGER
prod.kontrolle	probe
prod.kontrolle.mw	REAL
monatsprod[9].herstellungsdatum[1]	CHAR
y_z.z[j]	INTEGER

Für Variablen, die als Komponenten von Record-Variablen beschrieben werden, also mittels einer Punktschreibweise kodiert sind, wird die Bezeichnung "qualifizierte Variable" (im Gegensatz zu "indizierte Variable" für Array-Komponenten) benutzt.

Bisher haben wir einen Record-Typ eingeführt, von dem während der Übersetzung alle Datenfelder mit ihren Attributen bekannt sind. Da der Record-Typ eine Datenstruktur beschreibt, ist dem Compiler also die Gesamtlänge (Anzahl der Datenbytes) bekannt, die für Objekte dieses Record-Typs benötigt wird. Ebenso liegen die relativen Adressen der Datenfelder fest. Das Bild 2.2. illustriert das an Hand des oben beschriebenen Record-Typs material.

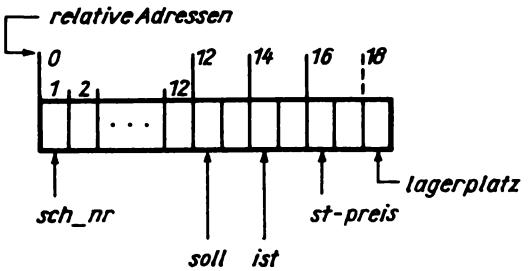


Bild 2.2. Beispiel einer durch einen Record-Typ beschriebenen Datenstruktur

Nun ist es in PASCAL auch möglich, diesem bisher beschriebenen sog. festen Teil eines Records einen Variantenteil anzufügen. Zum Verständnis dieser Konstruktion betrachten wir folgendes: Die Daten eines Produkts enthalten die Angaben, wie sie zuvor im Typ charge beschrieben wurden. Darüber hinaus sollen weitere Daten aufgenommen werden, die jedoch davon abhängen, ob dieses Produkt für den Inlandbedarf vorgesehen ist oder für den Export in gewisse Ausfuhrländer. Die in jedem Falle zu speichernde Information hängt von spezifischen Bedingungen ab (Verträge, Prioritäten, Verrechnungskurs u. dgl.). Nehmen wir an, dass neben dem Inlandbedarf Export nach Bulgarien, Ungarn, CSSR und BRD vorgesehen ist, so könnte für jedes dieser Länder der aufzubauende Datensatz anders aussehen, etwa so:

```
Inland:(abnehmer: array [1..6] of string [30];
        bedarf: array [1..6] of INTEGER)
Bulgarien:(importeur: string[30]; vertragskennz: string[20];
           kritische_daten: array [1..2] of string [6])
```

Für weitere Länder sind entsprechende individuelle Datensätze erforderlich. Zusammen mit dem festen Teil über die Daten des Produkts lässt sich dieser Typ so darstellen:

```
type konsumenten = (Inland, Bulg, Ung, CSSR, BRD);
     produkt = record
               stammdaten: material;{weiter oben definiert}
               case k: konsumenten of
               Inland:(abnehmer:array [1..6] of string [30];
                       bedarf:array [1..6] of INTEGER);
```

```
Bulg:  (importeur: string[30];
        vertragskennz: string [20];
        kritische_daten:array [1..2] of string[6]);
{Hier folgen weitere Datenstrukturen für Ung,
 CSSR und BRD}
end
```

Der in diesem Beispiel auftretende Teil, eingeleitet durch das Basissymbol **case** und endend vor dem Basissymbol **end**, heisst Variantenteil.
In jeder Variante wird eine spezielle Datenstruktur beschrieben, deren Eigenschaften jedoch wegen der darin vorhandenen Bezeichner und den zugeordneten Typattributen vollständig beschrieben ist. Insbesondere ist dem Compiler die Bytezahl bekannt, die für jede Variante benötigt wird. Dem Compiler ist damit die maximale Länge der Varianten bekannt.
Ein solcher Record-Typ mit Variantenteil beschreibt nun eine Datenstruktur, in der der Variantenteil die maximale Länge hat, so dass in diesem Speicherbereich jede Variante Platz fände. Bild 2.3. zeigt für unser Beispiel, wie ein Record des Typs produkt aufgebaut ist.

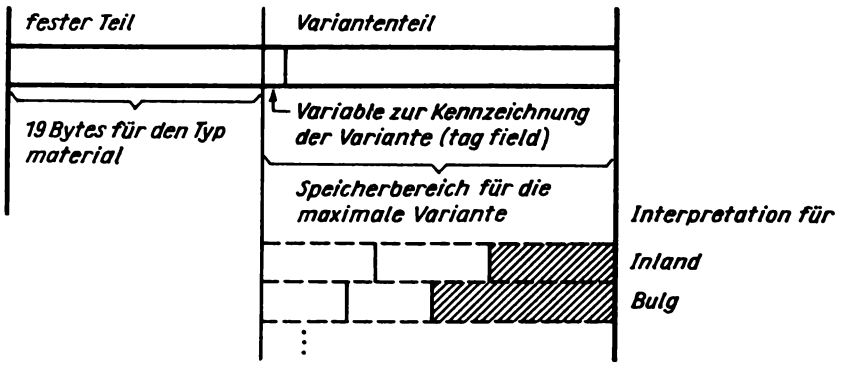


Bild 2.3. Record mit Variantenteil

Nach diesem Beispiel, dass die Motivierung solcher Datenstrukturen erläutern sollte, soll zuerst die Syntax des Record-Typs korrekt beschrieben werden. Dazu seien Syntaxdiagramme verwendet, die so zu verstehen sind, dass ein Durchlaufen des Diagramms eine gültige Konstruktion eines Record-Typs liefert(Bild 2.4.).

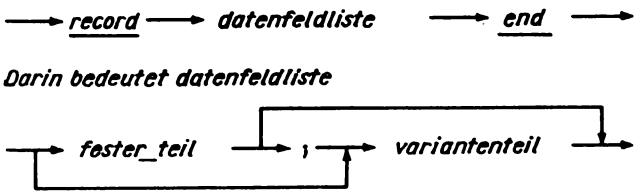


Bild 2.4. Syntax des Record-Typs

Man entnimmt dem Diagramm, dass sowohl der feste Teil als auch der Variantenteil fehlen können. (Es wäre also auch **record end** ein gültiger Record-Typ, jedoch hat er in der Programmierung keinen Sinn.) Was im Bild 2.2. als fester_teil symbolisiert ist, entspricht den eingangs verwendeten Beispielen, also einer Folge von Daten- feldbezeichnern mit zugeordneten Typattributen.
Jede Variante des Variantenteils ist durch eine Konstante gekennzeichnet, ihr folgt, getrennt durch einen Doppelpunkt und in runde Klammern eingeschlossen, eine Datenfeldliste im Sinne von Bild 2.2. (In dieser Datenfeldliste dürfte also wiederum am Ende ein Variantenteil vorhanden sein. Nur nach einer gewissen Programmierpraxis wird der Wunsch auftreten, solche Strukturen zu benutzen.)
Der Variantenteil hat in symbolischer Notation folgende Form:

```
case einfache Variable : Typbezeichner of
konstante_1: (Datenfeldliste_1);
konstante_2: (Datenfeldliste_2);
...
konstante_n: (Datenfeldliste_n)
```

konstant_1, konstant_2, ... sind dabei Werte aus dem mit Typbezeichner beschriebenen Definitionsbereich, also Werte, die auch der einfachen Variablen zugewiesen werden können.
Werden nun Variablen von einem Record-Typ definiert, der einen Variantenteil hat, so sehe man hinter diesen Variablen Speicherbereiche, in denen die maximale Variante Platz findet. Auch die zur Spezifizierung der Varianten benutzte Variable (zwischen case und of) benötigt Speicherplatz wie alle anderen Datenfelder des Records.
Die Varianten belegen den gleichen Speicherplatz. Zu den Komponenten der Varianten wird wie zu jedem anderen Datenfeld des Records zugegriffen. Das heisst aber, dass zu den gleichen Speicherstellen zugegriffen werden kann, wobei die Inhalte der Speicherplätze aber

unterschiedlich interpretiert werden, wenn die Datenfeldbezeichner unterschiedliche Attribute haben. Diesen Sachverhalt soll das folgende Beispiel verdeutlichen.

Beispiel:

```
type beispiel = record
                  nr: 1..1000;
                  case q : BOOLEAN of
                   FALSE:(j : INTEGER);
                   TRUE: (a, b:CHAR)
                  end;
var bsp: beispiel;
    ...
    bsp.q := TRUE;
    bsp.a := 'N'; bsp.b :=  '|';
    WRITELN(bsp.j);
```

Ausgegeben wird die Zahl 31822.
Das erklärt sich so: Für j werden 2 Bytes Speicherplatz verwendet. Dieser Speicherplatz ist identisch mit dem für die beiden CHAR-Variablen a und b. Da 'N' in ASCII (vgl. Anhang A) den Wert ¤4E, '|' den Wert ¤7C hat, ist dieser Speicherplatz mit ¤4E7C belegt. In der Ausgabeanweisung wird diese Belegung als Wert einer ganzen Zahl aufgefasst, in eine Dezimaldarstellung konvertiert und dann ausgegeben. Nun ist aber auf Grund der Darstellung ganzer Zahlen /6/

$$\begin{aligned} ¤4E7C &= 7*16^3 + C*16^2 + 4*16^1 + E*16^0 \\ &= 28672 + 3072 + 64 + 14 \\ &= 31822 \end{aligned}$$

Bemerkung: Der Zugriff zu den Datenfeldern der Varianten ist unabhängig von dem Wert des Kennzeichenfeldes (tag field, im obigen Beispiel bsp.q). Daraus folgt, dass die Datenfeldbezeichner in den Varianten voneinander verschieden sein müssen.
Für Record-Variablen ist folgende Operation erklärt:

:= Der Wert der Record-Variablen wird kopiert. Die Record-Variablen auf der linken und auf der rechten Seite der Wertzuweisungen müssen vom gleichen Typ sein.

Beispiele:

```
(1) var probe0, probe1: record ... end; ... ; probe1 := probe0;...
(2) program rec (OUTPUT);
    var r,t: record x : REAL;
```

```
                y : array [1..8] of INTEGER;

        end;
begin r.x := 100.0;
      r.y[6] := 2000;
      t := r;
      WRITELN('Ausgabe der kopierten Werte');
      WRITELN('t.x=', t.x, ' t.y[6]=', t.y[6]);
end.
```

Bei kompliziertern Datenstrukturen kann es lästig sein , in einer Folge von Anweisungen, die sich auf die Datenfelder der gleichen Record-Variablen beziehen, stets die Record-Variable vollständig zu notieren. Als Beispiel seien die Komponenten des Feldelementes j von monatsprod -das ist ein Record- zeilenweise ausgegeben.

```
WRITELN(monatsprod[j].umfang);
WRITELN(monatsprod[j].herstellungsdatum);
WRITELN(monatsprod[j].kontrolle.x[1]);
WRITELN(monatsprod[j].kontrolle.x[2]);
. . .
WRITELN(monatsprod[j].kontrolle.mw);
```

Dafür ist eine verkürzende Notation eingeführt worden, und zwar darf die Record-Variable in einer WITH-Klausel den Anweisungen vorangestellt werden.

Beispiel:

```
with monatsprod[j] do
  begin WRITELN(umfang);
        WRITELN(herstellungsdatum);
        WRITELN(kontrolle.x[1];
        WRITELN(kontrolle.x[2]);
        . . .
        WRITELN(kontrolle.mw);
  end;
```

Die Record-Variable monatsprod[j].kontrolle kann ebenfalls in der WITH-Klausel notiert werden, also

```
with monatsprod[j], monatsprod[j].kontrolle do
  begin WRITELN(umfang);
        WRITELN(herstellungsdatum);
        WRITELN(x[1]);
```

```
            WRITELN(x[2]);
            . . .
            WRITELN(mw);
      end;
```

Das Vorhandensein von zwei Record-Variablen in der WITH-Klausel wird wie folgt behandelt:

```
     with v1, v2 do  begin ... end
```

bedeutet

```
     with v1 do
       with v2 do  begin ... end
```

Standardmässig ist eine tiefere Verschachtelung nicht möglich (s. Abschn. 2.15.).
Es sei nochmals darauf hingewiesen, dass mit jedem Record-Typ ein neues Namensniveau eingeführt wird. Das folgende (formale) Beispiel ist also syntaktisch korrekt, lässt sich also übersetzen und abarbeiten.

Beispiel:

```
     program r;
     var a: record a: REAL;
                 .i: INTEGER;
            end;
         b: array [1..3] of record b: array [1..4] of REAL;
                                    c: BOOLEAN;
                            end;
         c: record j: INTEGER;
                   c: record j: INTEGER;
                              c: REAL;
                      end
            end;
     begin
       b[1].b[1] := 1.0;
       a.a := 0.0;
       with b[2], a do begin b[2] := 2.0;
                             a := 12.0;
                       end;
       c.c.c := 1000;
       WRITELN('a.a=', a.a);
       WRITELN('b[2].b[2]=', b[2].b2[]);
       WRITELN('c.c.c=', c.c.c);
     end.
```

Zur Selbstkontrolle des Behandelten möge folgende Aufgabe dienen.

A2.9.3.-1: Zugriff zu den Komponenten strukturierter Variabler
Die folgenden Wertzuweisungen seien in einem Programm syntaktisch korrekt. Man gebe Definitionen bzw. Deklarationen für die strukturierten Variablen an, die im Vereinbarungsteil erklärt sein müssen. (Lösungen im Abschn 4.)

```
(1)  x.y[i][j] := 0.0;
(2)  x1.x2.q[7] := a<b;
(3)  v1[1].v2[2] := i/(i+1);
(4)  tk[m*n+1].r := 'abcd';
(5)  u[s[1]] := d.u[s[j]];
```

Weitere Beispiele für das Arbeiten mit Record-Variablen findet man im Abschn. 2.9.5.

2.9.4. Set-Typen und Set-Variablen

In einem gewissen Umfang kann in TURBO-PASCAL mit Mengen gearbeitet werden. Für die Mengenbildung wird ein Basistyp zugrundegelegt. Dieser Basistyp darf in TURBO-PASCAL nicht mehr als 256 Elemente enthalten, sie müssen in dem Zahlenbereich von 0 bis 255 liegen. Die numerische Interpretation der Elemente des Typs CHAR liefert Werte <=127, also kann CHAR (oder Teilbereichstypen davon) Basistyp für Mengenbildung sein. Die in einem Skalartyp vereinbarten Bezeichner sind ihrer Funktion nach Konstantenbezeichner, denen in aufsteigender Folge die ganzen Zahlen beginnend mit 0 zugeordnet werden. Auch solche Skalartypen können also als Basistypen für Mengen verwendet werden.
Ein Mengentyp wird definiert durch

```
set of Basistyp
```

Der damit festgelegte Wertevorrat ist die Menge aller Teilmengen des Basistyps (die leere Menge eingeschlossen), die sog. Potenzmenge. Durch eine Typdefinition kann einem Mengentyp wiederum ein Typbezeichner zugeordnet werden.

Beispiel:

```
type bsp = set of 0..7;
```

Elemente des Typs bsp sind z. B. [0,1], [1,2,7], [0,1,2,3,4,5,6,7] und die durch [] notierte leere Menge. Set-Typen können Komponenten von Array- und Record-Typen sein. Set-Variablen (Mengenvariablen)

sind Variablen von einem Set-Typ.

Beispiel:

```
type knoten = 0..31;
var bsp1, bsp2: bsp;
    chset: set of CHAR;
    buchst: set of 'A'..'Z';
    woche: set of (mon, die, mittw, don, frei);
    s: array [0..9] of set of knoten;
```

Die Erzeugung eines Wertes von einem Mengentyp folgt den im Abschn. 2.6. behandelten Berechnungs- und Notationsregeln für Ausdrücke. Zunächst einmal kann der Wert einer Menge durch eine sog. Mengenkonstruktion angegeb en werden. Eine Mengenkonstruktion besteht aus der durch Kommas getrennten Aufzählung der Elemente, wobei auch die Notation von Teilbereichen verwendet werden darf.

Beispiele:

```
[0,1]  [0..7]  [1,5..7]      Basistyp 0..7
['0','1']  ['0'..'9']        Basistyp CHAR
['M'..'0', 'Z']              Basistyp 'A'..'Z'
[mon,don] [die..frei]        Basistyp (mon,die,mittw,don,frei)
```

In diesen Beispielen wurden Konstanten benutzt.
Es sind aber auch Ausdrücke erlaubt, deren Wert zu dem Basistyp gehört.

Beispiele:

```
[i, j]   [i, i+2 .. j]
['c' .. SUCC(ch)]
```

Eine Mengenkonstruktion (das ist ein Wert von einem Mengentyp) kann einer Set-Variablen zugewiesen werden.

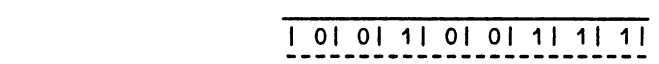
Beispiele:

```
bsp1 := [2, 5..7];
chset := ['.', ',', ';'];
buchst := [];
woche := [die];
g[0] := [30, 31];
```

Zum Verständnis stelle man sich vor, dass für eine Set-Variable ein Speicherbereich bereitzustellen ist, der (mindestens) soviele Bits umfasst, wie der Basistyp Elemente hat. Jedes Bit ist dann einem Element des Basistyps zugeordnet. Ist dieses Bit mit 0 belegt, so

bedeutet das, dass das Element nicht in der Menge enthalten ist, bei Belegung mit 1 ist es enthalten.

Beispiel:
Die Variable bsp1 hat nach bsp1 := [2, 5..7] folgende Belegung:



Werte eines Mengentyps können miteinander verknüpft werden, und zwar durch die Operationen Konjunktion oder Mengendurchschnitt (Operator ist *), durch die Operation Disjunktion oder Mengenvereinigung (Operator ist +) und die Mengendifferenzbildung (Operator ist -).
Symbolisieren a und b Mengen, so liefert

a * b die Menge aller der Elemente, die sowohl in a als auch in b enthalten sind,
a + b die Menge aller der Elemente, die in a oder in b oder in a und in b enthalten sind,
a - b die Menge aller der Elemente, die in a aber nicht in b enthalten sind.

Beispiel:
Es seien folgende Anweisungen vorausgesetzt:

```
a := [1, 3..5]; b := [4..6];
```

Dann ist das Resultat von

```
a * b    [4, 5]
a + b    [1, 3..6]
a - b    [1, 3]
```

Ausserdem sind Mengenvergleiche definiert, und zwar

= Test auf Gleichheit,
<> Test auf Ungleichheit,
>= Test, ob 1. Operand alle Elemente des 2. Operanden enthält,
<= Test, ob alle Operanden des 1. Operanden im 2. enthalten sind.

Die Tests liefern als Resultat einen Wert des Typs BOOLEAN. Die Enthaltenseinbeziehungen a < b bzw. a > b (a, b bedeuten Mengen) sind nicht erklärt. Sie müssen mittels des Operators and notiert werden, also (a <= b) and (a <> b) bzw. (a >= b) and (a <> b).
Weiterhin gibt es den Operator in um zu prüfen, ob ein Wert aus

einem Basistyp in einer Menge dieses Basistyps enthalten ist. Der Operator liefert einen logischen Wert.

Beispiele:

```
var c: CHAR;
    i: INTEGER;
    q: BOOLEAN;
    g: array[1..26] of CHAR;
. . .
if not (c in ['a'..'z']) then ...;
while i in [1..5, 25..30] do ...;
q := i in [g[i]..g[i+3]];
```

Weiter Beispiele zum Arbeiten mit Mengen im Abschn. 2.9.5.

2.9.5. Beispiele

Diese Beispiele sollen die Verwendung von strukturierten Anweisungen und strukturierten Variablen, wie sie in den Abschnitten 2.8. bzw. 2.9. behandelt wurden, illustieren.
Es werden numerische und alphanumerische Beispiele herangezogen.

1. Werteverlauf einer Funktion y = P(x)
Die Koeffizienten eines Polynoms n-ten Grandes, n <= 9, sind in der Reihenfolge n, a[0], a[1], ... , a[n] einzugeben.
Auszugeben sind Argumente und Funktionswerte im Intervall 0 bis 1 mit einer Schrittweite von 0.05.

```
{tab.pas}
program tab;
const dx = 0.05;
      u = 0.0;
      b = 1.0;
var x, y : REAL;
    n, i, iz : INTEGER;
    a : array[0..9] of REAL;
    ok : BOOLEAN;

begin
WRITE('Grad des Polynoms eingeben: ');
READLN(n);
iz := 1;
repeat
ok := TRUE;
```

```
if not (n in [1..9]) then
   begin WRITE('Wert unzulaessig, Eingabe wiederholen');
         READLN(n);
         ok := FALSE;  iz := iz +1;
   end
                      else
   begin WRITELN('Koeffizienten beginnend mit a[0] eingeben');
         for i := 0 to n do READLN(a[i]);
         WRITELN(lst,'       x', 'P(X)':28);
         x := u;
         while x < b +0.01 do
            begin
               y := a[n];
               for i := n-1 downto 0 do y := y*x +a[i];
               WRITELN( lst,x: 16: 5, y: 24);
               x := x +dx;
            end;
    end;
until ok or (iz = 5);
if iz = 5 then WRITELN('Programm abgebrochen',
                       ' da 5 falsche Eingaben');
end.
```

Bemerkungen:
(1) Bei unzulässigem Grad darf die Eingabe bis zu fünfmal wiederholt werden.
(2) Die Berechnung des Funktionswertes erfolgt nach dem Hornerschen Schema:

$$y = (\ldots (a[n]x + a[n-1])x + \ldots + a[1])x + a[0].$$

2. Sortieren von Datensätzen

Gegeben ist ein Vektor von n<= 50 Datensätzen. Jeder der Datensätze enthält ein sechsstelliges alphanumerisches Merkmal. Die erste nicht mehr benutzte Vektorkomponente ist durch 6 Leerzeichen im Datenfeld m gekennzeichnet. Die Datensätze sind in dem Vektor hinsichtlich des Merkmals in aufsteigender Folge zu sortieren. Zur Kontrolle ist die Folge der sortierten Merkmale auszugeben.
Es werden im folgenden nur die Vereinbarungen und Anweisungen angegeben, die dafür erforderlich sind.

```
program sort_records;
type key = array[1..6] of CHAR;
     satz = record
        {Datenfelder des Records}
```

```
        m : key
        end;
var m0 : key;
    hi : satz;
    adrj, i, j : 1..50;
    tafel : array[1..50] of satz;
const leer = '        ';

begin {Erzeugen des Vektors nicht dargestellt}
i := 1;
if tafel[1].m <> leer then
repeat
with tafel[i] do
     begin m0 := m; adrj := i;
           j := i+1;
           repeat
           with tafel[j] do
              if (m <> leer) and (m < m0) then
                 begin m0 := m;
                       adrj := j
                 end;
           j := j+1;
           until (j > 50) or (tafel[j].m = leer);
           {Es folgt die Umspeicherung}
           hi := tafel[adrj];
           tafel[adrj] := tafel[i];
           tafel[i] := hi;
           i := i+1;
     end {with tafel};
until tafel[i].m = leer;
i := 1;
while tafel[i].m <> leer do begin WRITELN(tafel[i].m);
                                  i := i+1;
                            end;
end.
```

3. Ganzzahlige Division

Das folgende Programm zeigt, wie allein mit Addition, Subtraktion, Links- und Rechtsverschiebung die Division ganzer positiver Zahlen realisiert werden kann. Der Algorithmus geht davon aus, dass für zwei natürliche Zahlen a und b, a > b, die Darstellung

a = 0 . b + a

gilt. Ist nun b < a, so lässt sich das schreiben als

```
     a = 0 . b + b + (a-b)
         1 . b + a'  .
```

Für b < a' lässt sich das wiederholen, so dass der Faktor von b jeweils um 1 erhöht wird, während von a, a' usw. jeweils b subtrahiert wird. Das führt, wenn man den Rest jetzt r nennt, zu

```
     a = m . b + r, b > r.
```

Die Umformung

```
     a = 2m(b/2) + r
       =  2m . b' + r
       =  m' . b' + r
```

gestattet nun, den obigen Algorithmus zu wiederholen, wenn b' < r ist.

```
{div.pas}
program ganzz_division;

var  a, b, quot, rest, i  : INTEGER;

begin
WRITE('a eingeben, a=0 bricht das Programm ab  ');
READLN(a);
while a <> 0 do
      begin WRITE('b eingeben  '); READLN(b);
            rest := a;  quot := 0;  i := b;
            while i <= rest do i := 2*i;
            while i <> b do
                  begin
                  i := i shr 1;  quot := quot shl 1;
                  if i <= rest then
                              begin
                              rest := rest -i;  quot := quot +1;
                              end;
                  end;
WRITELN(a, b:8, quot:8, rest:8);
WRITE('Naechsten Wert fuer a oder 0 eingeben  ');
READLN(a);
end;
end.
```

4. Darstellung eines Graphen

Gegeben ist folgender gerichteter Graph(Bild 2.5.):

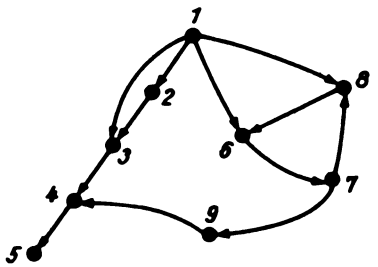


Bild 2.5. Gerichteter Graph

Die den Graph bestimmenden Beziehungen sind als Paare von Knoten einzugeben: (Vorgänger, Nachfolger). Es ist ein Vektor zu erzeugen, dessen Elemente folgende Information für jeden Knoten enthalten:

- Attribute des Knotens (maximal 20 Zeichen)
- Menge der Vorgängerknoten
- Menge der Nachfolgerknoten.

Die folgenden Programmzeilen enthalten die notwendigen Vereinbarungen und Anweisungen zur Belegung des Vektors. (z wird nur initialisiert.)

```
{graph.pas}
program graph;
type anzahl = 1..9;
     knoten = record
              z: string[20];
              vg, nf: set of anzahl;
              end;
var i, j, von, nach: anzahl;
    g: array [anzahl] of knoten;
    kn: anzahl;
    { ... }
begin
WRITE('Anzahl der Knoten eingeben:');
READLN(kn);
for i := 1 to kn do {Initialisierung}
                with g[i] do
                   begin z := '';
                         vg := [];  nf := [];
```

```
                    end;
WRITELN('Knotenpaar eingeben oder 0 0 fuer Ende');
READLN(von, nach);
while (von <> 0) and (nach <> 0) do
   begin
      g[von].nf := g[von].nf + [nach];
      g[nach].vg := g[nach].vg + [von];
      READLN(von, nach);
  end;

{ ... }
for i := 1 to kn do
   with g[i] do
      begin WRITE(i:4, '  :vg:');
            for j := 1 to kn do
               if j in vg then WRITE(j:3);
            WRITELN;
            WRITE('      :nf:');
            for j := 1 to kn do
               if j in nf then WRITE(j:3);
            WRITELN;
     end;
{ ... }
end.
```

5. Primzahlen

Unter dem Namen "Sieb des Eratosthenes" (282 - 202 in Kyrene) ist ein Verfahren zur Primzahlbestimmung bekannt, das wie folgt vorgeht: Man geht von der Folge der natürlichen Zahlen 2, 3, ... aus. Beginnend mit 2 wird jede weitere Zahl gestrichen. Die nächste nicht gestrichene Zahl ist 3 (2. Primzahl). Jetzt wird jede 3. Zahl gestrichen. Die nächste nicht gestrichene Zahl ist 5, wiederum erfolgt das Streichen von Zahlen. Die Zahlen, die bei diesem Vorgehen nicht gestrichen worden sind, sind Primzahlen.
Das folgende Programm führt diesen Algorithmus aus. Es steht hier als ein Beispiel für das Arbeiten mit Mengen. Die Beschränkung n = 255 ist TURBO-PASCAL-spezifisch.

```
{prim.pas}
program pz;
{Programm nach Hoare}

const n = 255;
var sieb, primz: set of 2 .. n;
    next, j: INTEGER;
```

```
begin sieb := [2..n]; primz := [];
      next := 2;
      repeat
      while not (next in sieb) do
              next := SUCC(next);
      primz := primz + [next];
      j := next;
      while j <= n do  {eliminieren}
        begin
         sieb := sieb - [j];
         j := j + next;
        end;
     until sieb = [];
for j := 2 to n do
    if j in primz then WRITELN(j);
end.
```

2.9.6. Typverträglichkeit und Konvertierungsfunktionen

Oben wurde gesagt, dass durch die Typattribute der Definitionsbereich von Variablen festgelegt wird. Ebenso hat jede Konstante einen Typ, der sich unmittelbar aus ihrer Notation ergibt. Auch für die bisher behandelten Standardfunktionen wurde der Typ des Funktionswertes angegeben (event. in Abhängigkeit vom Typ des Arguments). Werden nun zwei Operanden - Konstanten, Variablen, Funktionsaufrufe oder in Klammern eingeschlossene Ausdrücke - miteinander verknüpft, so ist die Frage berechtigt, was geschieht, wenn sich die Operandentypen unterscheiden. Das ist in TURBO-PASCAL (wie in anderen Programmiersprachen) durch Festlegungen über Typverträglichkeit und Typanpassungen geregelt. Im einzelnen gelten folgende Festlegungen:

1. Variablen sind verträglich, wenn sie in der gleichen Variablendeklaration eingeführt wurden.

Beispiel:

```
     var a, b: array [1..n] of REAL;
         u, v, w: (ja, nein, vielleicht);
```

Verträglich sind a und b bzw. u, v und w.

2. Variablen sind verträglich, wenn sie sich auf die gleiche Typdefinition beziehen.

Beispiel:

```
type m = array [1..m, 1..k] of INTEGER;
var a:m; ... ; b:m;
```

3. Operanden sind verträglich, wenn die zugeordneten Typen Teilbereiche desselben Aufzählungstyps sind.

Beispiel:

```
type jahr = (Jan, Feb, Maerz, Apr, Mai, Juni, Juli, Aug,
             Sept, Okt, Nov, Dez);
var c1, c2: CHAR;
    c3: 'a'..'z';
    b: BYTE;
    i, j: INTEGER;
    it, jt: 0..maxint;
    j1987: jahr;
    quartal: Juli..Sept;
```

Verträglich sind c1, c2 und c3, ferner b, i, j, it und jt und letztlich j1987 mit quartal.

4. Set-Variablen bzw. Set-Operanden sind verträglich, wenn ihre Basistypen verträglich sind. Die leere Menge ist mit jedem Set-Typ verträglich.

5. String-Operanden unterschiedlicher Typen sind verträglich.

6. Variablen vom Typ **array** [...] **of** CHAR sind mit Zeichenkettenkonstanten verträglich, wenn die Länge der Zeichenkettenkonstanten der Anzahl der Array-Komponenten entspricht.

7. String-Variablen sind zuweisungsverträglich mit eindimensionalen CHAR-Feldern, d. h., der Wert von Char-Feldern kann String-Variablen zugewiesen werden, nicht umgekehrt.

Neben diesen Festlegungen gibt es einige Situationen, in denen automatisch eine Typwandlung erfolgt (d. h., dass der Compiler entsprechende Konvertierungen vorsieht).:

1. In arithmetischen Ausdrücken (vgl. Abschn 2.6.1.) werden ganzzahlige Werte in den Typ REAL konvertiert, wenn wenigstens einer der Operanden vom Typ REAL ist oder der Divisionsoperator / benutzt wird.

2. Variablen vom Typ REAL können Werte vom Typ INTEGER zugewiesen

werden; die INTEGER-Werte werden vor der Zuweisung konvertiert. Es ist aber nicht gestattet, INTEGER-Variablen Werte des Typs REAL zuzuweisen.

Sollen Operanden unterschiedlichen Typs verknüpft werden, für die durch obige Festlegungen keine Regelung getroffen wurde, so muss eine Typanpassung explizit programmiert werden.
Das Prinzip dieses Programmierens für Aufzählungstypen wurde schon im Abschn. 2.9.1. angedeutet. Es besteht darin, dass die Bezeichner von Aufzählungstypen wie Funktionsbezeichner benutzt werden können, um einen Wert aus dem durch den Funktionsbezeichner genannten Typ zu erzeugen. Ist also der Typ jahr wie im obigen Beispiel definiert, so wäre die Ergibtanweisung

```
j1987 := jahr(i);
```

korrekt, ebenso

```
i := INTEGER(j1987);
```

Diese Typwandlungsmöglichkeit besteht zusätzlich zu den Möglichkeiten aus Standard-PASCAL. Es existieren also weitere Konvertierungsfunktionen, nämlich:

ORD(a)
Die Funktion wandelt den Wert eines Ausdrucks a (a von einem Aufzählungstyp) in den Typ INTEGER. Ist a insbesondere vom Typ CHAR, so ist der Funktionswert durch den Wert von a in ASCII gegeben.
Ist a von einem Skalartyp, so ist ORD(a) gleich der Position von a in der Definition des Skalartyps, wobei die Zählung mit 0 beginnt.

CHR(i)
Die Funktion wandelt den ganzzahligen Ausdruck i in das ASCII-Zeichen um, das dem Wert entspricht (0<= i <= 255).

Zur Wandlung von REAL in INTEGER dienen die Funktionen TRUNC und ROUND.

TRUNC(n)
n symbolisiert einen Ausdruck von numerischem Typ. Der Funktionswert ist der ganzzahlige Teil dieses Ausdrucks, d. h., er entsteht durch Abschneiden des gebrochenen Teils. Das Ergebnis ist vom Typ INTEGER.

ROUND(n)
Die Funktion realisiert das übliche Runden auf die nächste ganze Zahl, wobei der gebrochene Teil 0.5 "aufgerundet" wird. Es gilt also

```
ROUND(n) = { TRUNC(n+0.5), falls n >= 0,
           { TRUNC(n-0.5), falls n < 0 ist.
```

2.10. Konstantendefinitionsteil II

In TURBO-PASCAL können Variablen bei ihrer Deklaration initialisiert werden. Diese Vereinbarungen heissen - etwas irreführend - typgebundene Konstanten (typed constants), benutzt werden sie wie Variablen, insbesondere können sie nicht dort verwendet werden, wo Konstanten verlangt sind, z.B. in Typdefinitionen. Die Vereinbarung erfolgt in einem Konstantendefinitionsteil. Sie unterscheidet sich von denen im Abschn. 2.5. behandelten nur durch die zusätzliche Angabe eines Typs. Allerdings können in TURBO-PASCAL diese typgebundenen Konstanten auch strukturiert sein; für die Initialisierung sind dann spezielle Sprachelemente erforderlich.

2.10.1. Unstrukturierte Konstanten

Unstrukturierte typgebundene Konstanten sollen im weiteren kürzer "Typkonstanten" genannt werden.
Als Typbezeichner dürfen Skalartypen und der Typ **string** benutzt werden.

Beispiele:

```
     const  dr0=¤5F;
            dr :BYTE= ¤5F;
            cr :CHAR= ^M;
            limit :INTEGER= 127;
            e :REAL= 2.7182;
            zeile :string[30]= '|     x       |            y     |'
```

Die Variablenvereinbarung

```
    var a: array[1..dr0] of INTEGER;
```

ist korrekt, hingegen ist

```
    var b:array[1..dr] of INTEGER;
```

unzulässig.

2.10.2. Strukturierte Konstanten

Strukturierte typgebundene Konstanten bezeichnen Programmgrössen, die in Konstantendefinitionen mit Initialisierung eingeführt wurden und wie strukturierte Variablen benutzt werden. Es soll im weiteren die kürzere Bezeichnung "strukturierte Typkonstanten" benutzt wer-

den.

Für die Notation der Initialisierung sind für die drei Formen strukturierter Konstanten, nämlich Array-Konstanten, Record-Konstanten und Set-Konstanten Sprachelemente aufgenommen worden, sie werden Aggregate genannt. In symbolischer Schreibweise hat dann die Definition einer strukturierten Typkonstanten die Form:

```
const Bezeichner: strukturierter Typ = Aggregat;
```

Die Aggregate spiegeln den Aufbau der Konstanten wider. Ihr Aufbau wird an Hand der folgenden Beispiele erklärt:

Beispiele:

(1)
```
type vektor = array[1..4] of INTEGER;
const v1: vektor = (1,2,3,4);
```
Die Komponenten der strukturierten Typkonstanten v1 haben die Anfangswerte 1,2,3 und 4.
Die Trennung der Komponentenwerte erfolgt in dem Array-Aggregat durch Kommas.

(2)
```
const
      hexaziffer: array [10..15] of CHAR =
                                  ('A','B','C','D','E','F');
```
Die Zuweisung zu einem CHAR-Feld kann auch durch Angabe einer Zeichenkette erfolgen, d.h., gleichwertig mit obiger Darstellung kann kodiert werden.
```
const hexaziffer: array[10..15] of CHAR = ('ABCDEF');
```

(3)
```
type matrix = array[1..2,1..3] of REAL;
const xm: matrix = ((0,1,1),(-1,-1,1));
```
In dem Aggregat sind die Anfangswerte zeilenweise zusammengefasst anzugeben. Das entspricht der Reihenfolge der Abspeicherung der Elemente im Variablenspeicher.
Bei mehrdimensionalen Feldern ist das entsprechend anzugeben, d.h. so,dass für den am weitesten rechts stehende Index die Elemente zuerst angegeben werden. Das nächste Beispiel zeigt das.

(4)
```
type silo = array[0..1,0..1,0..2] of BYTE;
const vsilo: silo =(((¤50,¤51,¤52),(¤B,¤0C,¤5F)),
                    ((¤00,¤01,¤02),(¤C1,¤19,¤F0))
                   );
```
Diese Initialisierung speichert:
```
vsilo[0,0,0]:=¤50; vsilo[0,0,1]:=¤51; vsilo[0,0,2]:=¤52;
vsilo[0,1,0]:=¤B ; vsilo[0,1,1]:=¤0C; vsilo[0,1,2]:=¤5F;
```

```
vsilo[1,0,0]:=¤00; usw.
```

(5)
```
type komplex = record  re, im : REAL end;
const z : komplex = (re:2.0;im:1.0E-9);
```
Im Record-Aggregat wird jedem Datenfeld des Typs ein Wert zugewiesen; die Initialisierung der Datenfelder wird durch Semikolons getrennt.
Datenfeldbezeichner und Aggregat werden durch Doppelpunkt getrennt.

(6)
```
type st = record x: array[1..7] of INTEGER;
                 xx: REAL;
          end;
const charge = record nr: INTEGER;
                test: st;
                 end = (nr: 1;
                    test: (x:(0,0,0,0,0,0,0);
                       xx:-1)
                      );
```

Enthält eine Record-Konstante Datenfelder von einem Array-Typ oder Record-Typ, so sind diese Datenfelder wie Array-Konstanten oder Record-Konstanten zu initialisieren.
Enthält eine Record-Konstante einen Variantenteil, so kann eine beliebige Variante initialisiert werden, jedoch muss das Aggregat einen Anfangswert für die Auswahlvariable (tag field) enthalten.

(7)
```
type digitsymbol  =  set of'0'..'9';
     digit = set of 0..9;
const digit_symbol  :digitsymbol  =  ['0'..'2','7'..'9']
      voktalsymbol   :digitsymbol   =['0'..'7'];
      dez: digit = [0..9];
      okt: digit = [0..7];
      vdigit: digit = [];
```

Dieses Beispiel zeigt die Vereinbarung von Set-Konstanten. In den Aggregaten dürfen nur Konstanten benutzt werden, im übrigen entspricht die Notation der im Abschn. 2.9.4.

2.11. Das Filekonzept

Ein Programm übernimmt i. allg. bereitgestellte Problemdaten und gibt Resultatdaten aus. Wie das im einfachsten Fall in einem Programm notiert werden kann, wurde im Abschn. 2.3. erläutert.

Betrachtet man den Datenaustausch zwischen dem Programm und der Umgebung, so können Eingabedaten entweder über Kanäle bezogen werden, die im System durch vordefinierte Namen gekennzeichnet sind, oder sie können von einer Diskette gelesen werden. Für die vordefinierten Kanäle ist im TURBO-PASCAL-System der Terminus "logisches Gerät" (logical device) üblich. INPUT ist ein solches logisches File, das das Standardeingabegerät des TURBO-PASCAL-System beschreibt. Die Files auf der Diskette haben einen Namen, der keinem Filebezeichner aus dem Programm entsprechen muss. Im Programm kann ein File wie eine Variable deklariert und in speziellen Anweisungen benutzt werden. Die Komponenten eines Files sind als fortlaufend numeriert zu denken, die Numerierung beginnt mit 0. Ein implizit zugeordneter Zeiger (pointer) adressiert die aktuelle Komponente (Lese- oder Schreibstelle). Der Zusammenhang zwischen der File-Variablen im Programm - häufig wird von einem logischen File gesprochen - und einem realen File auf einer Diskette wird in der Laufzeit hergestellt.
Die Datenausgabe ist entsprechend organisiert. Es ist entweder die Ausgabe an ein logisches Gerät (z.B. OUTPUT, das ist das Standardausgabegerät des TURBO-PASCAL-System) oder in ein Diskettenfile möglich.
Files bestehen aus Komponenten gleichen Typs; eine Ausnahme machen Textfiles.
Aktionen für File-Variablen bzw. File-Komponenten sind mit Standardprozeduren und Standardfunktionen möglich.

2.11.1. File-Typen und File-Variablen

Ein File-Typ hat die Form

```
file of filekomponententyp
```

filekomponententyp bezeichnet dabei einen beliebigen Typ, der jedoch nicht selbst ein File-Typ ist oder einen File-Typ als Komponente enthält. Ein File-Typ kann Komponententyp eines Array-Typs oder eines Record-Typs sein.

Beispiele:

```
type a = array[1..n, 1..n] of INTEGER;
         r = record s:string[80];
                    nr:INTEGER;
             end;
        fa = file of a;
        fr = file of r;
```

```
        freal = file of REAL;
var vfa : fa;
    vfr : fr;
    vfreal : freal;
    vfchar : file of CHAR;
    vffa : array[1..3] of fa;
```

Die beiden folgenden Vereinbarungen sind fehlerhaft, da der Komponententyp der File-Variablen wiederum Komponenten von einem File-Typ enthält.

```
type f0 = file of r;
     r0 = record ...
          df0:f0;  {korrekt}
          ...
          end;
var  f0falsch: file of f0;
     r0falsch: file of r0;
```

2.11.2. Funktionen und Prozeduren für Files

Wenn auch die Deklaration von File-Variablen der anderer Variabler entspricht, so dürfen File-Variablen doch nicht in Ausdrücken oder Ergibtanweisungen benutzt werden. Für File-Variablen sind weder die Operatoren (arithmetische, logische, Vergleichsoperatoren) noch das Zuweisungssymbol definiert. File-Variablen werden ausschliesslich als Parameter von Funktionen und Prozeduren benutzt. Das TURBO-PASCAL-System enthält eine Anzahl von Standardroutinen, die File-Variablen als Argumente haben; darüber hinaus können in Prozedur und Funktionsvereinbarungen formale Parameter von einem File-Typ verwendet werden (s. Abschn. 2.14.). In diesem Abschn. werden die Standardroutinen dargestellt.

Die Argumente der Standardroutinen haben folgende Bedeutung:

filevar	Name einer File-Variablen.
str_ausdr	Ausdruck von einem String-Typ; i. allg. wird es sich dabei um eine Zeichenkettenkonstante oder um den Namen einer String-Variablen handeln.
m	Ausdruck, der einen Wert vom Typ INTEGER liefert.
var_liste	Liste von Variablenbezeichnern des Komponententyps des Files; die Variablenbezeichner werden durch Kommas getrennt; es muss mindestens ein Variablenbezeichner angegeben sein.

Zuordnung von File-Variablen zu Files der Programmumgebung

ASSIGN(filevar, str_ausdr)

Der Wert von str_ausdr muss eine Zeichenkette sein, wie sie im Betriebssystem zur Identifikation eines Files benötigt wird, also - falls erforderlich - mit Angabe des Diskettenlaufwerks und eines Namenszusatzes. Im allgemeinen wird die Angabe einer entsprechenden Zeichenkette im Dialog angefordert, so dass folgende Vorgehensweise typisch ist:

```
var filename : string[14];
    f : file of komponententyp;
    ...
begin ...
    WRITE('Filename eingeben:');
    READLN(filename);
    ASSIGN(f, filename);

end.
```

Danach bedeutet jede Verwendung des logischen Files f eine Aktion mit dem File, dessen Identifikation in der READLN-Anweisung übernommen wurde (weitere Beispiele s. Abschn. 2.11.5.).

Eröffnen eines Files zum Lesen

RESET(filevar)

Die durch die ASSIGN-Prozedur zugewiesene Zeichenkette muss ein vorhandenes Diskettenfile bezeichnen. Existiert das File nicht, so wird das Programm wegen eines Eingabe/Ausgabe-Fehlers abgebrochen. (Der Abbruch kann durch die I-Direktive verhindert werden, vgl. dazu Abschn. 2.15.). Existiert das File, so trifft das TURBO-PASCAL-System Vorkehrungen, den ersten Satz des Files zu lesen, d. h., der Filepointer zeigt auf die erste Komponente (Komponente mit der Nummer 0).

Eröffnen eines Files zum Schreiben

REWRITE(filevar)

Es wird ein Diskettenfile eingerichtet, das leer ist und dessen Filepointer die Komponente 0 adressiert. Existiert ein File gleichen Namens, so wird es gelöscht.

Abschliessen eines Files

CLOSE(filevar)

Der Zugriff zum File wird unterbunden, im File-Verzeichnis der

Diskette werden die notwendigen Eintragungen vorgenommen. Ein abgeschlossenes File kann neu eröffnet werden. Jedes durch die RESET- oder REWRITE-Anweisung eröffnete File soll durch die CLOSE-Anweisung explizit abgeschlossen werden, da keine automatische File-Abschliessungsbehandlung am Programmende erfolgt und bei späterem Zugriff zu dem File nicht File-Ende erkannt wird oder der Name gar nicht in das File-Verzeichnis eingetragen wurde.

Leeren und Füllen des internen File-Puffers

FLUSH(filevar)

Ist filevar zum Schreiben eröffnet, so wird durch die FLUSH-Anweisung der Inhalt des internen File-Puffers auf die Diskette übertragen, was i.allg. dann geschieht, wenn der interne File-Puffer gefüllt ist.
Ist filevar zum Lesen eröffnet, so bewirkt die Anweisung, dass beim nächsten Lesen ein Zugriff zur Diskette erfolgt, der interne Puffer also neu gefüllt wird.

Löschen eines Files

ERASE(filevar)

Das File wird gelöscht.

Schreiben in ein File

WRITE(filevar, var_liste)

Die Werte der Variablen von var_liste werden auf das Diskettenfile übertragen, die Position des Filepointers wird korrigiert.

Lesen von einem File

READ(filevar, var_liste)

Es werden dem durch filevar bezeichneten File Komponenten entnommen und den Variablen zugewiesen. Die Übernahme beginnt an der durch den Filepointer bezeichneten Stelle, nach jedem Zugriff wird die Stellung des Filepointers korrigiert. Die Anzahl der zu übernehmenden Datenbytes wird durch die Länge des Komponententyps festgelegt.

Positionieren des Filepointers

SEEK(filevar, m)

Durch die SEEK-Anweisung wird der Filepointer auf die m-te Komponente positioniert; die Zählung der Komponenten beginnt mit 0. In bezug auf die Positionierung am File-Ende vgl. Erläuterungen zur Standardfunktion FILESIZE (s. u.).

Umbenennen eines Files

RENAME(filevar, str_ausdr)
Der reale Name des logischen Files filevar wird geändert, das File hat nach Ausführung der RENAME-Anweisung den Namen, der durch den Wert von str_ausdr fixiert ist.

Beispiel: RENAME(f0, 'e:exper.dat')

Diese Prozedur kann benutzt werden, um zusammen mit der I-Direktive (s. Abschn. 2.15.) das unbeabsichtigte Überschreiben von Files zu verhindern.

Endetest

EOF(filevar)
Die boolean-wertige Funktion EOF hat den Wert TRUE, wenn der Filepointer des Files filevar hinter die letzte Komponente positioniert wird, sonst liefert die EOF-Funktion den Wert FALSE. Mit Hilfe der EOF-Funktioon kann die zyklische Verarbeitung aller Komponenten eines Files bequem organisiert werden.

Komponentenanzahl

FILESIZE(filevar)
Die Standardfunktion liefert als Wert die Anzahl der Komponenten des Files. Wenn FILESIZE(filevar) = 0, so heisst das, dass das File leer ist.
FILESIZE(filevar) = m bedeutet, dass m Komponenten, numeriert von 0 bis m-1 vorliegen. Die Funktion FILESIZE kann dort verwendet werden, wo ein ganzzahliger Ausdruck verlangt wird. Häufig erfolgt die Verwendung in der SEEK-Prozedur (s. o.), um den Filepointer hinter die letzte Komponente zu positionieren und damit die Voraussetzungen zu schaffen, um das File zu verlängern (s. Abschn. 2.11.5.).
Beispiel: SEEK(f0, FILESIZE(f0));

Position des Filepointers

FILEPOS(filevar)
Diese Funktion liefert einen ganzzahligen Wert, nämlich die Nummer der vom Filepointer adressierten Komponente. Liefert die FILEPOS-Funktion den Wert 0, so bedeutet das, das die erste Komponente adressiert ist.

Die Standardprozeduren bzw. -funktionen gestatten das Ändern einzelner File-Komponenten (update); ein Beispiel dazu s. Abschnitt 2. 11.5.
Wie man mit diesen Standardroutinen ein File erzeugt, wieder einliest und verlängert zeigt das folgende Programmbeispiel. Die Ausgabeanweisungen dienen nur der Kontrolle des korrekten Ablaufs.

```
{Arbeiten mit Files}
program FILES;
var i:INTEGER;
a:array [1..25] of INTEGER;
fint: file of INTEGER;
s: string[14];
begin
WRITE('Filename eingeben:');
READLN(s);
ASSIGN(fint,s);
REWRITE(fint);
for i := 1 to 20 do WRITE(fint,i);
CLOSE(fint);
RESET(fint);
for i := 1 to 20 do READ(fint,a[i]);
WRITELN(lst,1, a[1]:10);
WRITELN(lst,19,a[19]:10);
WRITELN(lst,'filesize=',filesize(fint));
SEEK(fint,filesize(fint));
for i := filesize(fint)+1 to 25 do WRITE(fint,i);
SEEK(fint,20);
for i := 21 to 25 do begin READ(fint,a[i]);
                           WRITELN(lst,a[i]);
                     end;
CLOSE(fint);
READ(s);
RENAME(fint,s);
end.
```

2.11.3. Standard-Eingabe/Ausgabe

2.11.3.1. Textfiles

Als Textfiles werden spezielle, nur sequentiell verarbeitbare Files bezeichnet, deren Komponenten vom Typ CHAR sind. Von Files des Typs "file of CHAR" unterscheiden sich Textfiles dadurch, dass in die Zeichenfolgen Strukturierungszeichen eingefügt wurden (bzw. wer-

den), so dass das File eine sog. Zeilenstruktur hat. Zur Strukturierung dient das Steuerzeichenpaar "carriage return, line feed", d. h. ^M^J bzw. ¤0D0A (vgl. Anhang A). Dem letzten Zeilenendezeichen folgt eine Kennzeichnung des File-Endes (^Z, ¤1A). Bild 2.6. zeigt den Aufbau eines Textfiles, wobei eoln und eof End-of-line-Zeichen bzw. End-of-file-Zeichen symbolisieren.

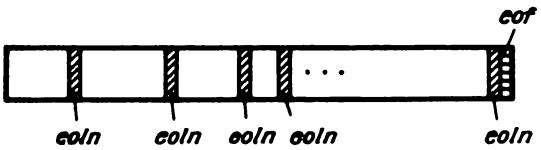


Bild 2.6. Aufbau eines Textfiles

Der Typ TEXT ist ein Standardtyp in den Programmiersprachen PASCAL und TURBO-PASCAL. File-Variablen dieses Typs werden also durch

```
var t0, t1: TEXT;
```

vereinbart.
Im Gegensatz zu den in den Abschnitten 2.11.1. und 2.11.2. behandelten Files gibt es hier nicht Komponenten, deren Attribute durch den Komponententyp festgelegt sind. Die Zeilenlängen sind unterschiedlich, die Zeilen sind nicht als numeriert zu verstehen. Es ist also nicht möglich, den File-Pointer auf eine bestimmte Zeile zu positionieren: Textfiles können nur sequentiell verarbeitet werden.
Wie für Nicht-Textfiles sind Prozeduren und Funktionen vordefiniert. Die Zuordnung eines logischen Textfiles zu einem Diskettenfile erfolgt - der allgemeinen Handhabung von Files entsprechend - durch die Prozeduranweisung

```
ASSIGN(filevar, str_ausdr)
```

(s. Abschn. 2.11.2.). Entsprechendes gilt für das Eröffnen mittels RESET(filevar) bzw. REWRITE(filevar).
Die Standardfunktion

```
EOF(filevar)
```

hat den Wert TRUE, wenn das eof-Zeichen erkannt ist, sonst den Wert FALSE.
Für Textfiles gibt es eine zweite booleanwertige Standardfunktion, nämlich

```
EOLN(filevar)
```

Diese Funktion hat den Wert TRUE, wenn als nächstes zu verarbeitendes Zeichen das eoln-Zeichen erkannt wurde, sonst hat die Funktion den Wert FALSE.

Die EOLN- und die EOF-Funktion machen es bequem, Textfiles zu verarbeiten. Typische Programmkonstruktionen sind

```
(1) while not EOF(filevar) do
     begin
      {Es folgen Anweisungen.}
      while not EOLN(filevar) do
        begin
          {Eingabe und Verarbeitung eines Zeichens.}
        end;
      {Möglicherweise weitere Anweisungen.}
      {Filepointer auf Zeichen nach eoln-Zeichen setzen.}
   end;

(2) repeat {Lesen eines Zeichens.}
           {Anweisungen}
   until EOLN(filevar)
```

Für das Suchen des eoln- bzw. eof-Zeichens gibt es die Standardfunktionen

SEEKEOLN(filevar) und SEEKEOF(filevar)

Diese Funktionen übergehen von der aktuellen Position des Filepointers Leerzeichen und Tabulatorzeichen, SEEKEOF auch ein eoln-Zeichen und liefern dann den Wert TRUE oder FALSE, je nachdem, ob nach diesem Übergehen der Filepointer auf ein entsprechendes Zeichen weist.

Die Datenübernahme erfolgt mittels

```
READ(filevar, var_liste) bzw.
READLN(filevar, var_liste)
```

Die Variablen in var_liste müssen von einem der folgenden Typen sein: CHAR, INTEGER oder REAL. Ganze Zahlen und reelle Zahlen müssen in dem Textfile so dargestellt sein, wie es für entsprechende Konstanten in einem Programm gefordert ist. Leerzeichen und eoln-Zeichen trennen Zahlen; führende Leerzeichen werden überlesen.

Beispiel:
Es mögen folgende Vereinbarungen gelten:

```
var i : array [1..3] of INTEGER;
    j : INTEGER;
    c : array [1..5] of CHAR;
    x1, x2 : REAL;
    q : BOOLEAN;
    t : TEXT;
```

Das Textfile t sei wie folgt belegt:

```
BSP1:  3   4   -1   5.2   eoln   9.0E-1   eoln eof
```

Dann können die Werte durch

```
for j := 1 to 5 do READ(t, c[j]);
READ(t,i[1], i[2], i[3], x1, x2);
```

übernommen werden. Der Filepointer würde danach das Leerzeichen hinter -1 adressieren.
Da das Testen dieses Beispiels noch das zusätzliche Bereitstellen des Datenfiles erfordert, geben wir das Beispiel als vollständiges Programm an. Das Datenfile sei mit dem TURBO-Editor unter dem Namen tfile1.dat erzeugt. Dieser Name würde - mit Angabe des Diskettenlaufwerks - einzugeben sein.

```
{f1test.pas}
program tfile1_test;
var i : array[1..3] of INTEGER;
    j :INTEGER;
    c : array [1..5] of CHAR;
    x1, x2: REAL;
    q :BOOLEAN;
    s : string[14];
    t: TEXT;
begin
     WRITE('Filename eingeben: '); READLN(s);
     ASSIGN(t,s); RESET(t);
     for j := 1 to 5 do READ(t, c[j]);
     READ(t, i[1], i[2], i[3], x1, x2);
     WRITELN('Kontrollausgaben: ', c);
     WRITELN(i[1]:20, i[2]:3, i[3]:3);
     WRITELN(x1 :34, x2 :16);
     q := EOLN(t); WRITELN(q : 25);
     q := EOF(t); WRITELN(q : 25);
     CLOSE(t);
end.
```

Das Einlesen mit READLN hat die gleiche Wirkung wie das Einlesen mit READ, nur wird zusätzlich der Filepointer hinter dem nächsten eoln-Zeichen positioniert.
In der READLN-Anweisung kann var-liste fehlen, also

```
READLN(filevar)
```

Die Wirkung dieser Anweisung ist das Positionieren des Filepointers auf das Zeichen nach dem nächsten eoln-Zeichen; eine Datenübernahme erfolgt dabei nicht.

Die Übertragung von Daten in ein Textfile erfolgt mittels

```
WRITE(filevar, ausgabeliste) bzw.
WRITELN(filevar, ausgabeliste)
```

Beispiel: Umwandlung eines Text-Files in ein Stringfile

```
{Textfile in File of string verwandeln.}
program text_to_stringfile;
type string128=string[128];
var ftext:text;
    fstring: file of string128;
    i:integer;
    s128:string128;
    fn:string[14];
begin
   WRITELN('Name des Textfiles eingeben:  ');
   READLN(fn);
   ASSIGN(ftext,fn); RESET(ftext);
   WRITELN('Name des Stringfiles eingeben.');
   READLN(fn);
   ASSIGN(fstring,fn); rewrite (fstring);

   while not eof(ftext) do
     begin i := 0;
           while not eoln(ftext) do
             begin i := i + 1;
                   read(ftext,s128[i]);
             end;
           READLN(ftext);
           s128[0] := char(i);
           WRITE (fstring,s128);

      end;
   CLOSE(fstring);
   RESET(fstring);
   WRITELN('Strings in fstring:',filesize(fstring));
   CLOSE (fstring);
end.
```

Wird WRITELN ohne ausgabeliste benutzt, so wird nur ein eoln-Zeichen geschrieben.

2.11.3.2. Standardfiles

Im Abschn. 2.4.2. wurden die beiden Standardfilebezeichner INPUT und OUTPUT eingeführt. Es handelt sich um Textfiles, die aber von vornherein Geräten zugeordnet sind und für die Eröffnungs- und Abschliessungsroutinen automatisch ausgeführt werden. Diese Geräte heissen in der Terminologie von TURBO-PASCAL "logische Geräte"; in bezug auf diese beiden Files handelt es sich um die Konsole (CON:) oder das Terminal (TRM:). Letztlich heisst das, dass Eingabedaten von der Tastatur erwartet werden und Ergebnisdaten auf dem Bildschirm erscheinen. Dabei werden die über die Tastatur eingegebenen Daten auf dem Bildschirm angezeigt (Bildschirmecho).
Die Dateneingabe erfolgt durch die READ- bzw. READLN-Prozedur. In der Variablenliste von Eingabeanweisungen sind nur Bezeichner für Variablen der Typen INTEGER, REAL, CHAR und **string** gestattet. Die Datenausgabe erfolgt durch die WRITE- bzw. WRITELN-Prozedur. In der Ausgabeliste von Ausgabeanweisungen sind nur Ausdrücke gestattet, die einen Wert eines der folgenden Typen liefern: INTEGER, BYTE, REAL, CHAR, **string**, **array**[..] **of** CHAR, BOOLEAN. In den Ausgaben sind ausserdem Angaben möglich, die die Anzahl der Positionen festlegen, die für die Ausgabe des Wertes zu verwenden sind. Diese Formatangabe besteht aus einem Doppelpunkt, gefolgt von einem ganzzahligen Wert. (Es wurde davon schon Gebrauch gemacht.) Dieser ganzzahlige Wert kann als Ausdruck angegeben werden, bisher wurden nur Konstanten benutzt.
Ist also die Länge des Ausgabebereichs durch ein solches Format angegeben, so wird der auszugebende Wert rechtsbündig eingeordnet. Links wird erforderlichenfalls mit Leerzeichen aufgefüllt.
Das folgende Beispielprogramm zeigt das.
Bei der Ausgabe von Werten des Typs REAL gibt es zwei Ausgabeformate, nämlich eine normierte Zahlendarstellung mit einem Skalierungsfaktor und eine Festkommadarstellung, für die die Angabe der auszugebenden Dezimalstellen festgelegt werden kann (vgl. letzte FOR-Anweisung im folgenden Beispielprogramm).

```
{writeform}
program writeformat;
const
   t= '!Beispiel!';

var i,j: INTEGER;
```

```
    x: REAL;
    drbild: TEXT;
    s: string[14];
begin
  WRITELN('Filename eingeben');
  READLN(s);
  ASSIGN(drbild,s); REWRITE(drbild);
  WRITELN(drbild,'Ausgabe mit Formaten auf ein Textfile');
  j := 5;
  for i := 1 to 4 do WRITE (drbild,t: i*j);
  WRITELN(drbild);
  for i := 1 to 5 do WRITELN(drbild,'!',123:i*j,'!',-123:i*j,'!');
  WRITELN(drbild);
  x:= 2.6;
  for i := 1 to 5 do WRITELN(drbild,'!', x:i*j,'!',-x:i*j,'!');
  WRITELN(drbild);
  for i := 1 to 5 do WRITELN(drbild,'!',x:i*j:i,'!',-x:i*j:i,'!');
  WRITELN(drbild);
  CLOSE(drbild);
end.
```

```
Ausgabe mit Formaten auf ein Textfile
!Beispiel!!Beispiel!     !Beispiel!          !Beispiel!
!  123! -123!
!       123!      -123!
!            123!           -123!
!                 123!                -123!
!                      123!                     -123!

!2.6E+00!-2.6E+00!
!2.6000E+00!-2.600E+00!
!2.600000000E+00!-2.60000000E+00!
!    2.6000000000E+00!   -2.6000000000E+00!
!         2.6000000000E+00!        -2.6000000000E+00!

!  2.6! -2.6!
!      2.60!     -2.60!
!          2.600!         -2.600!
!              2.6000!             -2.6000!
!                  2.60000!                -2.60000!
```

Das Beispiel zeigt die Benutzung der Formatangaben. Als Ausgabefile wird ein Textfile verwendet, sein Inhalt ist wiedergegeben. Die Ausgabe auf dem Bildschirm erfolgt ebenso, wenn überall die Bezüge

auf das File drbild entfernt werden. Dann wird vom TURBO-PASCAL-System das Standardfile OUTPUT eingesetzt, das mit dem Terminal identifiziert wird.
Ausser diesen Standardfilebezeichnern gibt es folgende vordefinierte Namen für Textfiles: CON, TRM, LST, KBD.
Zur Bedeutung dieser Standardfiles (Textfiles):

CON wird der Gerätekonsole zugeordnet. Die Dateneingabe erfolgt dabei zeilenweise von einem (Zeilen-)Puffer, in dem die eingegebenen Daten noch modifiziert werden können. Die Datenausgabe in das File CON wird durch die Ausgabe auf den Bildschirm realisiert.

TRM ist Abkürzung für das Terminalgerät. Die eingegebenen Daten werden übernommen. Von eingegebenen Steuerzeichen erscheint nur das Echo des eoln-Zeichens auf dem Schirm.

LST bezeichnet das Druckerfile.

Die folgenden Programme zeigen die Anlage eines Nicht-Textfiles (File of Records) und die Ausgabe dieses Files an den Drucker.

```
{File fuer printrec anlegen}
program f_auto;
type nr = string [7];
     farbe = (blau, gruen, weiss, rot, lila );
     auto = record
            kennz:nr;
            typ:string[20];
            baujahr:40..90;
            aktuell:farbe
            end;

var d:auto;
    autos: file of auto;
    s:string[14];
    vfarbe:string[5];
    c:CHAR;
begin
  WRITE('Name des Files eingeben: ');
  READLN(s);
  ASSIGN(autos,s); REWRITE(autos);
    repeat
      WRITE('Kennz. eingeben: ');
      READLN(d.kennz);
      WRITE('Typ eingeben: ');
```

```
      READLN(d.typ);
      WRITE('Farbe eingeben: ');
      READLN(vfarbe);
     case vfarbe[1] of
      'b': d.aktuell := blau;
      'g':d.aktuell := gruen;
      'w':d.aktuell := weiss;
      'r':  d.aktuell := rot;
      'l':d.aktuell := lila ;end;
      WRITE(autos,d);
      WRITELN('Weitere Datensaetze? (n/<cr>): ');
      READLN(c);
    until c ='n';
    WRITELN('Anzahl der Saetze=', FILESIZE(autos));
 CLOSE(autos);RESET(autos);
 READ(autos, d);
 WRITELN(d.kennz); WRITELN(d.typ);
 end.


{printrec}
program printrec;
type nr = array [1..7] of CHAR;
     farbe =(blau, gruen ,weiss, rot, lila);
     auto = record
            kennz:nr;
            typ:string[20];
            baujahr:40..90;
            aktuell:farbe
            end;

var daten :auto;
    autofile: file of auto;
    s : string[14];

const vfarbe: array [0..4] of string [5] =
                      ('blau',  'gruen',  'weiss','rot',  'lila');

begin
  WRITELN('Name des zu verarbeitenden Files eingeben');
  READLN(s);
  ASSIGN(autofile, s);                                    {¤i-}
  RESET(autofile);                                        {¤i+}
if IORESULT<>0 then begin
                           WRITELN('kann nicht eroeffnen'); halt
```

```
                              end;
  WRITELN(LST, 'Kennzeichen','Typ':8,'Jahr':17,'Farbe':7);
  WRITELN;
  while not EOF(autofile) do
    begin
    READ(autofile,daten);
    with daten do
      begin
        WRITE(LST,kennz:9,typ:22, 1900 + baujahr:5);
        WRITELN(vfarbe[ORD(aktuell)]:7);
      end;
    CLOSE(autofile);
    end;
end.
```

In der dialogorientierten Programmentwicklung ist es meist sinnvoll, während des Tests Ergebnisdaten nur auf dem Bildschirm auszugeben (z.B. durch WRITELN('x=',x) u.ä.), und nach dem Testen des Programms die signifikanten Ausgaben zum Drucker zu lenken. Das ist leicht dadurch möglich, dass in die WRITE-Anweisungen "LST, " eingefügt wird (mit dem TURBO-PASCAL-Editor durch ^QA).

KBD (Abk. für keyboard device) bezeichnet die Eingabetastatur. KBD unterscheidet sich von den Files CON und TRM dadurch, dass kein Echo der Eingabedaten auf dem Schirm erscheint. Es ist also möglich, Zeichenfolgen einzutasten, die nicht auf dem Bildschirm erscheinen sollen (z.B. Kennwörter).

2.11.4. Blockweise Verarbeitung von Files

Ausser den Files eines spezifizierten Komponententyps und Textfiles enthält das Filekonzept von TURBO-PASCAL die Möglichkeit, Daten zwischen Diskettenfiles und Programmvariablen direkt zu transportieren. Dieser Datentransfer wird nur durch die Anzahl der zu übertragenden Bytes gesteuert, nicht aber durch den Typ der Variablen oder den Komponententyp des Files.
Für in diesem Sinne direkten Zugriff zu Diskettenfiles werden Filevariablen ohne Komponententyp verwendet. Solche Variablen erhalten nur das Typattribut **file**, also

```
var f1, f2 : file;.
```

So definierte Files sollen im weiteren typfreie Files genannt werden. Das Schreiben auf bzw. Lesen von typfreien Files erfolgt in

Blöcken von je 128 Bytes.
Für typfreie Files sind die Standardroutinen EOF, FILESIZE, FILEPOS und SEEK erklärt, und zwar so, wie es im Abschn. 2.11.2. dargestellt wurde. Da kein Komponententyp definiert ist, wird die Blocklänge von 128 Bytes als Komponentenlänge verwendet.
Auch typfreie Files müssen mit der RESET- bzw. REWRITE-Anweisung eröffnet und mit der CLOSE-Anweisung abgeschlossen werden.
Für den Schreib- oder Lesezugriff sind an Stelle der WRITE- bzw. READ-Prozedur zwei andere Standardprozeduren zu verwenden, nämlich

BLOCKWRITE und BLOCKREAD.

Beide Prozeduren können mit drei oder vier Parametern benutzt werden. Diese Parameter haben folgende Bedeutung:

```
BLOCKWRITE(varbezeichner {typfreie Filevariable},
           sender        {Variablenbezeichner},
           blockanzahl   {Ausdruck vom Typ INTEGER, der die Anzahl
                          der zu transferierenden Blöcke angibt}
         {,aktuell }     {INTEGER-Variable, die als Wert die Anzahl
                          der wirklich übertragenen Blöcke enthält}
           )
```

Der 4. Parameter kann entfallen. Es werden, beginnend an der Anfangsadresse der Variablen 128 * blockanzahl Bytes übertragen, es sei denn, die Übertragung muss vorher beendet werden.

```
BLOCKREAD(varbezeichner {typfreie Filevariable},
          empfaenger     {Variablenbezeichner},
          blockanzahl    {Ausdruck vom Typ INTEGER, der die Anzahl
                          der zu transferierenden Blöcke angibt}
        {,aktuell }      {INTEGER-Variable, die als Wert die Anzahl
                          der wirklich übertragenen Blöcke enthält}
          )
```

Mittels BLOCKREAD erfolgt die Übertragung vom Diskettenfile zur der Programmvariablen empfaenger, wiederum in Blöcken von 128 Bytes. Der Programmierer muss sicherstellen, dass die Variable empfaenger 128 * blockanzahl Bytes aufnehmen kann.
Diese blockweise Fileverarbeitung erlaubt es, Files beliebiger Typen zu verarbeiten.

Möglichkeiten einer programmierten Fehlerbehandlung bei Eingabe/Ausgabe-Operationen können mit der Compilerdirektive I behandelt werden (vgl. Abschn. 2.15.).

2.11.5. Beispiele

1. Kopieren eines Textfiles

```
{textkopie}
program  kopieren_eines_files;
const satzlaenge=128;
var
  quelle, ziel : file;
  quellname, zielname : string [14];
  puffer : array [1..satzlaenge] of BYTE;
  ok:BOOLEAN;

begin
repeat
WRITE('Quellfilename: ');
READLN(quellname);
ASSIGN(quelle, quellname);
{¤i-}
RESET(quelle);
{¤i+}
ok:=(IORESULT=0);
if not ok then WRITELN('File existiert nicht');
until ok;
WRITE('Zielname: ');
READLN(zielname);
ASSIGN(ziel, zielname);
REWRITE(ziel);
repeat
BLOCKREAD(quelle, puffer, 1);
BLOCKWRITE(ziel, puffer, 1);
until EOF;
WRITELN('Kopieren beendet');
CLOSE(quelle); CLOSE(ziel);
end.
```

2. Beispiel für Array-Komponenten von einem Filetyp

```
{filefeld}
program  filefeld;
type ifile = file of integer;
var af:array [1..3] of ifile;
    st:string[10];
    i,j:integer;
begin
```

```
    READ(st);
    ASSIGN(af[1],st);
    REWRITE (af[1]);
    for j := 1 to 12 do WRITE (af[1],j);
    WRITELN;
    CLOSE(af[1]);
    RESET(af[1]);
    for j := 1 to 12 do begin READ(af[1],i);
                              WRITELN(i);
                        end;
    CLOSE(af[1]);
end.
```

3. Beispiel für das Arbeiten mit einem CHAR-File

```
{writefi}
program charfile;
var h: CHAR;
    i: INTEGER;
    fn: string[14];
    aus: file of CHAR;
begin
   WRITE('Ausgabefile:'); READLN(fn);
   ASSIGN(aus, fn); REWRITE(aus);
   for i := 65 to 74 do
                     begin
                     ch := CHAR(i);
                     WRITE(aus, ch);
                     end;
CLOSE(aus);
RESET(aus);
WRITELN('Kontrolldruck');
for i := 1 to 10 do
                 begin
                 READ(aus,ch);
                 WRITE(ch:2);
                 end;
WRITELN;
CLOSE(aus);
end.
```

4. Auszug aus einem Stichprobenfile

In einem Diskettenfile sind Stichproben gespeichert, von denen jede <= 20 Werte enthält. Der Komponententyp dieses Files ist wie folgt definiert:

```
type probe = array[1..20] of REAL;
       stprobe = record umfang: 1..20;
                                stpr: probe
                 end;
```

Es sind alle Stichproben, die mindestens 15 Werte enthalten, in einem File zusammenzufassen.

{auszug}

```
program a;
const
  limit = 15;
type
  probe = array [1..20] of REAL;
  stpr = record umfang : 1 .. 20;
                 stpr: probe
         end;

var
  alt, auszug: file of stpr;
  r: stpr;
  i:INTEGER;
begin
  ASSIGN(alt, 'f:probe');
  RESET(alt);
  ASSIGN(auszug, 'f:f1.dat');
  REWRITE (auszug);
  while not EOF(alt) do
    begin
      READ(alt,r);
      if r.umfang >= limit then
          WRITE(auszug,r);
   end;
 CLOSE(alt);
 CLOSE(auszug);
 RESET(auszug);
 READ(auszug,r);
 for i := 1 to r.umfang do write(r.stpr[i]);
 WRITELN;
 CLOSE(auszug);
end.
```

5. Mischsortieren

Zwei Files mit Datensätzen gleichen Typs, die nach einem in diesem Datensätzen enthaltenen Merkmal m sortiert sind, sollen zu einem sortierten File zusammengefügt werden. Es wird vorausgesetzt, dass keines der Eingabefiles leer ist.

```
{mischsortieren zweier files: merge.pas}
program merge;
{Sortiert die geordneten Files ein1 und ein2  auf das File aus}
 type satz= record {...}
              m: array[1..10] of CHAR;
                  {...}
            end;
var h_ein1, h_ein2: satz;
    fn: string[14];
    q1, q2: BOOLEAN;
    i: INTEGER;
    ein1, ein2, aus: file of satz;
begin
  WRITE('1. Eingabefile:'); READLN(fn);
  ASSIGN(ein1,fn); RESET(ein1);
  WRITE('2. Eingabefile:'); READLN(fn);
  ASSIGN(ein2,fn); RESET(ein2);
  WRITE('name des Ausgabefiles:'); READLN(fn);
  ASSIGN(aus,fn); REWRITE(aus);
  READ(ein1, h_ein1);
  READ(ein2, h_ein2);
  repeat
  if h_ein1.m < h_ein2.m then begin
                              WRITE(aus, h_ein1);
                              q1 := EOF(ein1);
                              if not q1 then READ(ein1,h_ein1);
                              end
                         else begin
                              WRITE(aus, h_ein2);
                              q2 := EOF(ein2);
                              if not q2 then READ(ein2,h_ein2);
                              end;
until q1 or q2;

if q1 then {Umspeichern des Rests von ein2}
           begin
           WRITE(aus, h_ein2);
```

```
            for i := FILEPOS(ein2) to FILESIZE(ein2)-1 do
                     begin
                     READ(ein2, h_ein2);
                     WRITE(aus, h_ein2);
                     end;
            end
       else {Umspeichern des Rests von ein1}
            begin
            WRITE(aus, h_ein1);
            for i := FILEPOS(ein1) to FILESIZE(ein1)-1 do
                     begin
                     READ(ein1, h_ein1);
                     WRITE(aus, h_ein1);
                     end;
            end;
CLOSE(ein1); CLOSE(ein2); CLOSE(aus);
end.
```

6. Modifizieren eines Files (update)

Die Aufgabe besteht darin, gewisse Komponenten eines Files zu korrigieren. Wir betrachten dazu noch einmal das File autos (leicht abgeändert) aus dem Programmbeispiel des Abschn. 2.11.3. Wir setzen voraus, dass die Nummern der Komponenten, die modifiziert werden sollen, bekannt sind, so dass mittels der SEEK-Prozedur der Filepointer auf die Komponente gestellt werden kann. In unserem Beispiel werden wir nur eine Korrektur der Datenfelder kennz und letzte_Wartung vorsehen. Das Programm könnte dann wie folgt angelegt werden:

```
{Update eines Files}
program update;
const flimit=300;

type
  name=string[20];
  objekt=record
         kennz:name;
         anzahl:INTEGER;
         soll:INTEGER;
         letzterzugriff:string[6];
         end;
var ofile : file of objekt;
    vobjekt : objekt;
```

```
    i, nr: INTEGER;
    fname   string[14];
begin
WRITE('Filename eingeben:');
READLN (fname);
ASSIGN (ofile,fname);
RESET  (ofile);
WRITE  ('Nr. des Objekts eingeben, -1 bedeutet Eingabeende:');
READLN (nr);
while (nr>=0) and (nr<=flimit) do
    begin
    SEEK (ofile, nr);
    READ (ofile, vobjekt);
    with vobjekt do
         begin
         WRITE ('Objektkennzeichen eingeben:');
         READLN(kennz);
         READLN(anzahl);
         WRITE ('Soll zur Zeit:', Soll:10,', neues Soll');
         READLN(soll);
         WRITE ('Datum als JJMMTT eingeben');
         READLN(letzterzugriff);
    end {with};
    SEEK (ofile, nr);
    WRITE(ofile, vobjekt);
    WRITE('Nr. des Objektes eingeben, -1 bedeutet Eingabeende');
end {while};
CLOSE(ofile);
end.
```

Das ist ein Prinzipbeispiel. Bei unbekannter Komponentennummer kommt das Auffinden des richtigen Satzes hinzu.
Es sei darauf hingewiesen, dass das File zum Lesen eröffnet wird, dass aber sowohl gelesen als auch geschrieben wird.

2.12. Das Zeigerkonzept

2.12.1. Zur Speicherorganisation in der Laufzeit

Für das Verständnis des Zeigerkonzepts (Pointer-Konzept) ist es zweckmässig zu betrachten, wie ein Programm in der Laufzeit den Hauptspeicher benutzen kann. Neben dem Betriebssystem CP/M und den vom Betriebssystem verwalteten Routinen muss der vom Compiler generierte Objektkode im Hauptspeicher vorliegen, ausser-

dem muss Speicherplatz reserviert sein für Programmvariablen. Es bleibt ein mehr oder minder umfangreicher "ungenutzter" Speicherbereich. Dieser Speicherbereich wird von Routinen des TURBO-PASCAL-System verwaltet und während der Programmabarbeitung benutzt. Diese Benutzung erfolgt auf dreierlei Art, nämlich

1. als Kellerspeicher (stack) für die temporäre Zwischenspeicherung von Werten,
2. als Rekursionsspeicher (recursion stack) für die Arbeitsspeicher rekursiver Prozeduren und Funktionen (s. Abschn. 2.14.6.),
3. als sog. Halde (heap) für Speicherbereiche, die bei Bedarf von dem aktiven Programm angefordert werden.

In dem TURBO-PASCAL-System ist der "ungenutzte" Speicherbereich für diese drei Formen der Speicherplatzanforderungen aufgeteilt. Drei Standardbezeichner für INTEGER-Variable werden von dem TURBO-PASCAL-System benutzt, um diese Bereiche zu verwalten, das sind die Bezeichner HEAPPTR, RECURPTR und STACKPTR.
Der Haldenspeicher schliesst an den Objektkode an und wächst zu höheren Hauptspeicheradressen. Der Kellerspeicher beginnt, wo der Speicherbereich für die Programmvariablen endet und wächst zu den niedrigen Hauptspeicheradressen hin, die Anfangsadresse des Rekursionsspeichers liegt 1KByte vor der Adresse des Kellerspeichers, auch der Rekursionsspeicher wird in Richtung auf niedrigere Hauptspeicheradressen vergeben.
Diese Speicherplatzaufteilung zeigt Bild 2.7.

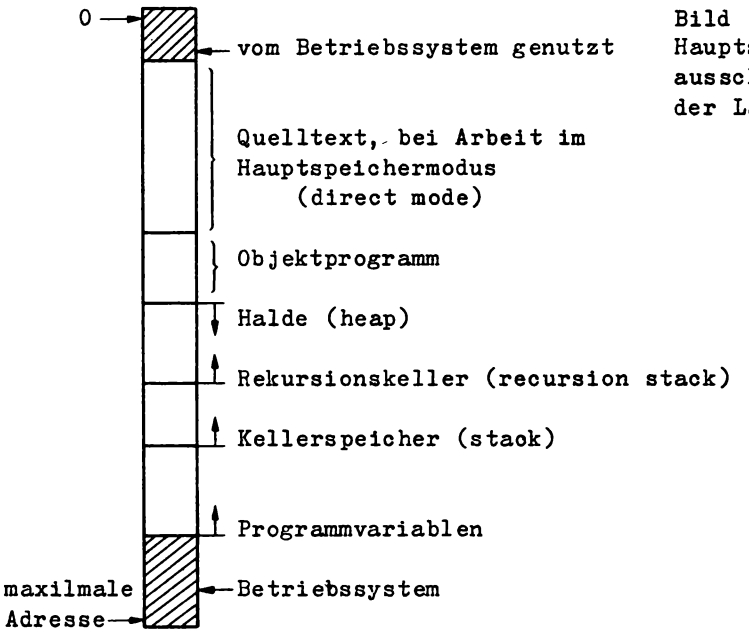


Bild 2.7
Hauptspeicher-ausschnitt während der Laufzeit

Die Standardvariablen erhalten als Anfangswerte die jeweiligen Anfangsadressen der Speicherbereiche, stets gilt aber:

HEAPPTR < RECURPTR < STACKPTR.

2.12.2. Pointer-Variablen und dynamische Variablen

Wenn bisher von Variablen gesprochen wurde, so verband sich damit die Vorstellung, dass in einer Variablendeklaration ein Bezeichner festgelegt wurde, für den vom Compiler für die Laufzeit des Programms Speicherplatz reserviert wurde. Dieser Speicherbereich stand (nach dem bisher Gesagten) während der gesamten Abarbeitungszeit des Programms zur Verfügung (statische Variable), der Zugriff zu dem Speicherbereich erfolgte über den Namen der Variablen, Komponenten konnten durch Indizierung (bei Array-Variablen) oder Qualifizierung (bei Record-Variablen) ausgewählt werden.
Daneben ist es jedoch auch möglich, Speicherbereich für Variablen in der Laufzeit des Programms anzufordern, so dass im konkreten Fall die Erzeugung einer Variablen vom Ablauf des Programms abhängt. Diese Variablen heissen dynamische Variablen, sie werden auf der Halde realisiert und sind nicht in einer Variablendeklaration eingeführt worden. Der Zugriff zu diesen Variablen kann also nicht

über einen Bezeichner erfolgen. Das Arbeiten mit dynamischen Variablen erfolgt über Pointer-Variablen, das sind Variablen von einem Pointer-Typ. Das Prinzip dieses Arbeitens zeigt Bild 2.8.

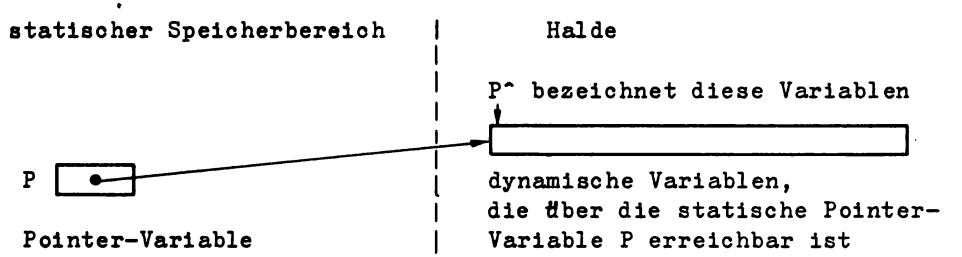


Bild 2.8. Prinzipskizze: statische und dynamische Variablen

In PASCAL und auch in TURBO-PASCAL geht man bei der Realisierung dieser Konzeption wie folgt vor.

1. Es wird ein Typ definiert, dessen Wertevorrat Adressen sind, die auf Objekte eines anderen Typs verweisen.

Beispiel:

```
type maschine = record
                  bezeichnung : string[30];
                  inv_nr : INTEGER;
                  preis   INTEGER;
                end;
pm = ^maschine;
```

Das Symbol ^ ist das Pointer-Symbol von TURBO-PASCAL.

2. Es werden Variablen von einem Pointer-Typ definiert.

Beispiel:

```
var masch1, masch2, masch3 : pm;
    halle1 : array [1..10] of pm;
```

Im Falle der Variablen halle1 sind die 10 Komponenten von dem Pointer-Typ pm, d.h., dass jede Komponente von halle1 auf ein Objekt vom Typ maschine verweisen kann (s. Bild 2.9.).

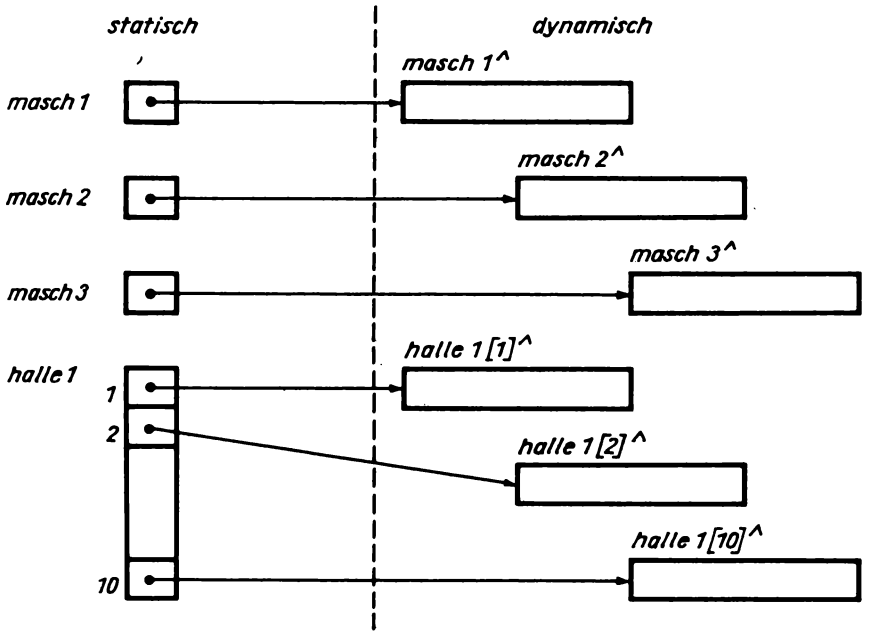


Bild 2.9. Pointer-Variablen und adressierte Objekte

3. Für Pointer-Variablen sind folgende Operatoren erklärt:

:= Der Wert einer Pointer-Variablen kann einer anderen Pointer-Variablen zugewiesen werden, wenn sie vom gleichen Typ sind.

= Die Werte von Pointer-Variablen können auf Gleichheit und auf

<> Ungleichheit verglichen werden.

4. Der Speicherplatz für die dynamische Variable muss angefordert werden. Das erfolgt durch Aufruf der Standardprozedur

```
NEW(pv).
```

pv symbolisiert dabei eine Pointer-Variable.

Wird diese Prozedur abgearbeitet, so werden auf der Halde soviele Bytes zur Verfügung gestellt, wie ein Objekt des Typs, an den die Pointer-Variable gebunden ist, fordert. Die Anfangsadresse dieses Speicherbereich wird der Pointer-Variable als Wert zugewiesen. Der Wert der Variable HEAPPTR wird nun um die Anzahl der bereitgestellten Bytes erhöht. Damit ist eine dynamische Variable erzeugt wor-

den. Der Zugriff zu dieser dynamischen Variable erfolgt durch Benutzung des Namens der Pointer-Variable, gefolgt von dem Pointer-Symbol ^. Die dynamischen Variablen im Bild 11 können z. B. durch masch1^ bzw. halle1[i]^ angesprochen werden.

Beispiele:
Mit den Bezeichnungen aus Bild 11 stellen die folgenden Zeilen gültige TURBO-PASCAL-Anweisungen dar:

```
NEW(masch1);
masch1^.bezeichnung := 'Fraese';
masch1^.inv_nr := 17017;
NEW(halle1[2]);
for j := 1 to 10 do
with halle1[j]^ do
     begin bezeichnung := 'Kompressor';
           READ(inv_nr,preis);
     end;
```

Wird jetzt eine statische Pointer-Variable mittels

```
var vpg : pg;
```

eingeführt, so lässt sich damit eine aus verketteten Datensätzen bestehende Liste im Programm aufbauen. Die folgenden Zeilen zeigen den Aufbau einer solchen "Liste" mit drei Datensätzen, wobei die konkreten Werte vom Terminal angefordert werden.

Beispiel:

```
{...}
type pg = ^geraet;
     geraet = record  {wie oben}
              {...}
              end;
var vpg : pg;
begin {...}
NEW(vpg);
with vpg do
     begin WRITELN('1. Datensatz eingeben ');
           WRITE('Bezeichnung: ');
           READLN(m.bezeichnung);
           WRITE('Inventarnummer und Preis: ');
           READ(m.inv_nr, m.preis);
           WRITE('Feuchtigkeit und Temperatur: ');
           READLN(feuchtigkeit, temp);
     end;  {Eingabe des ersten Satzes beendet}
NEW(vpg^.next);
            {diese Record-Komponente ist ebenfalls vom Typ pg}
```

```
with vpg^.next do
     begin  {wie oben}
          {...}
     end;  {Eingabe des 2. Satzes beendet}
NEW(vpg^.next^.next);
with vpg^.next^.next^ do
     begin  {wie oben}
          {...}
     end;  {Eingabe des 3. Satzes beendet}
```

Offensichtlich wird die Adressierung überaus unbequem. Auf diese Art liesse sich auch nicht die Eintragung von n Sätzen steuern. Um die Aktionen zyklisch ausführen zu können, benötigt man ein Kennzeichen, das aussagt, dass kein weiterer Satz folgt. Dafür gibt es die Pointer-Konstante **nil**, die mit jedem Pointer-Typ verträglich ist. Damit initialisiert man i. allg. Pointer-Variablen, also

```
vpg := nil;
```

oder

```
vpg^.next := nil;.
```

Das Ende einer Sequenz verketteter Sätze kann jetzt dadurch gesucht werden, dass man eine Pointer-Variable entsprechenden Typs die Sequenz durchlaufen lässt und jedesmal prüft, ob in dem gerade betrachteten Satz das Datenfeld next den Wert **nil** hat. In dem Falle ist man beim letzten Glied der Kette angelangt, sonst muss man unter Verwendung des Wertes von next zum nächsten Datensatz übergehen. Das setzt natürlich voraus, dass beim Aufbau der Kette der (jeweils) letzte Datensatz im Datenfeld next mit **nil** belegt wurde.

Beispiel: Suchen des letzten Datensatzes

```
var hi, vpg : pg;
begin  {..}
hi := vpg;
if hi = nil then WRITELN('Keine Saetze eintragen ')
            else while hi^.next <> nil do hi := hi^.next;
                {hi adressiert jetzt den letzten Satz}
{...}
end.
```

Weitere Beispiele s. Abschn. 2.12.4.

Erfahrungsmässig macht das Verständnis des Pointer-Konzepts Schwierigkeiten. Es seien deshalb drei formale Aufgaben zur Selbstkontrolle genannt.

A2.12.1.-1: Bild 2.10a

Man gebe Vereinbarungen und Anweisungen zur Erzeugung der im Bild 2.10a dargestellten Struktur an (Lösungen im Abschn. 4)!

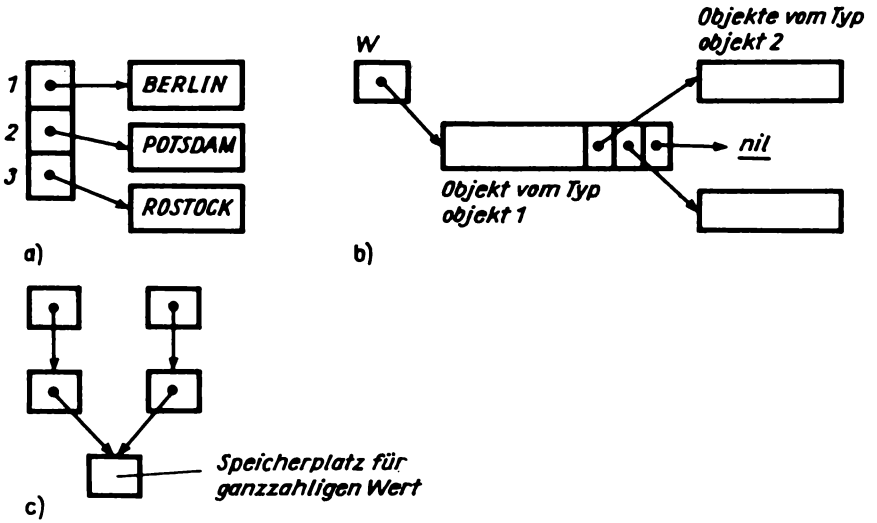


Bild 2.10. Beispiele dynamischer Strukturen

A2.12.2.-2: Bild 2.10b
Man gebe Vereinbarungen und Anweisungen an zur Erzeugung der im Bild 2.10b dargestellten Struktur. Die drei eingetragenen Felder von einem Pointer-Typ sollen Variablen eines Typs objekt2 adressieren.

A2.12.2.3: Bild 2.10c
Man gebe Vereinbarungen und Anweisungen an zur Erzeugung der im Bild 2.10c dargestellten Struktur!

2.12.3. Standardroutinen für die Halde

Im letzten Abschnitt wurde die Speicherplatzbereitstellung durch die NEW-Prozedur erklärt.
Es sollen jetzt weitere Prozeduren und Funktionen beschrieben werden, die eine effektive Nutzung der Halde ermöglichen.
pv symbolisiert in den folgenden Prozeduren eine Variable von einem Pointer-Typ, iausdr einen Ausdruck, der einen ganzzahligen Wert liefert.

MARK(pv)
Der Aufruf dieser Prozedur bewirkt, dass der aktuelle Wert des Haldenzeigers HEAPPTR der Pointer-Variable als Wert zugewiesen wird. Der aktuelle Stand der Halde wird gemerkt. Das ist eine Vorbereitung für die Freigabe des Speicherplatzes, der nach der MARK-Anweisung angefordert wurde.

RELEASE(pv)
HEAPPTR wird mit dem Wert von pv, der in einem MARK-Aufruf gesetzt worden ist, belegt. Das bedeutet, dass der Speicherbereich aller dynamischen Variablen, die nach dem letzten MARK(pv)-Aufruf generiert wurden, freigegeben wird.

DISPOSE(pv)
Diese Prozedur gibt genau den Speicherbereich auf der Halde frei, der durch die Pointer-Variable adressiert und durch die Länge des pv zugeordnenten Typs begrenzt wird. (DISPOSE ist schon in den PASCAL-Bericht /1/ aufgenommen worden, das Paar MARK/RELEASE ist die Alternative zu DISPOSE. In einem Programm ist entweder das Paar MARK/RELEASE oder die Prozedur DISPOSE zu benutzen.) Der durch DISPOSE-Aufrufe freigegebene Speicherplatz wird von Laufzeitroutinen des TURBO-PASCAL-System verwaltet. Bei erneuten Speicherplatzanforderungen wird versucht, die Forderung aus der Freispeichersammlung zu befriedigen.

GETMEM(pv, iausdr)
Der Aufruf dieser Prozedur weist pv die Anfangsadresse eines Speicherbereichs der Länge iausdr zu. Im Gegensatz zur NEW-Prozedur wird hier die Länge nicht durch den der Pointer-Variable pv zugeordneten Typ festgelegt, sondern explizit durch der Wert von iausdr.

FREEMEM(pv, iausdr)
Der Aufruf von FREEMEM gibt iausdr Bytes frei. Der Wert von iausdr muss gleich dem sein, der in GETMEM für die Speicherplatzforderung benutzt wurde.

MEMAVAIL
MEMAVAIL ist Bezeichner einer Standardfunktion. Sie liefert als Wert die Anzahl der auf der Halde zur Verfügung stehenden Bytes (vgl. auch MAXAVAIL).

MAXAVAIL
MAXAVAIL ist Bezeichner einer Standardfunktion vom Typ INTEGER. Die Funktion liefert (unter CP/M-80) die grösste Anzahl der Bytes, die auf der Halde zusammenhängend zur Verfügung gestellt werden können. Wäre dieser Wert grösser als 32 767, so ist der Wert von MAXAVAIL negativ, und der richtige Wert müsste durch 65 536.0 + MAXAVAIL berechnet werden (REAL-wertig, weil 65 536 > MAXINT!).

ORD(pv)
Mit dieser Standardfunktion kann (in CP/M-80) die in einer Pointer-Variable gespeicherte Adresse in einen ganzzahligen Wert konvertiert werden.

PTR(iausdr)
Diese Standardfunktion wandelt einen ganzzahligen Wert in eine Adresse; das Ergebnis kann einer Pointer-Variable zugewiesen werden, unabhängig von dem speziellen Pointer-Typ, d. h., der Wert der PTR-Funktion ist mit jedem Pointer-Typ verträglich. Daraus ergibt sich die Möglichkeit, über eine Pointer-Variable beliebige Speicherbereiche zu adressieren. Das entspricht sicherlich nicht den Gesichtspunkten strukturierter Programmierung und ist deshalb für gewöhnliche Anwenderprogramme auch nicht zu empfehlen. Für Fragen der Systemprogrammierung mit dem TURBO-PASCAL-System kann die Verwendung der Typwandlungen "Adresse in INTEGER" und umgekehrt nützlich sein.

2.12.4. Beispiele

1. Erzeugen einer verketteten Liste von Datensätzen

Im Abschn. 2.11. wurde gezeigt, wie ein Datenfile mit Sätzen des Typs auto angelegt und modifiziert werden kann. Es sei die Existenz eines solchen Files vorausgesetzt. Es sollen nun die Sätze des Files eingelesen und in Form einer Liste gespeichert werden. Unter "Liste" sei dabei eine Struktur verstanden, deren erstes Element durch eine Pointer-Variable adressiert wird, dieses erste Element verweist auf ein zweites usf., das letzte Element ist dadurch gekennzeichnet, dass es statt des Verweises den Wert nil enthält.

```
{Liste.pas}
program liste;
type nr = string[7];
   auto = record
          kennz : nr;
          typ : string[20];
```

```
          baujahr : 40..90
          end;
   peintrag = ^eintrag;
    eintrag =   record
                objekt : auto;
                naechster_eintrag : peintrag
                end;
var d : auto;
    autos : file of auto;
    eintragung : eintrag;
    fuhrpark, letzter_eintrag, hi : peintrag;
    s : string[14];
    kz : nr;
begin
  WRITE('Eingabefile:'); READLN(s);
  ASSIGN(autos,s); RESET(autos);
  {Es wird vorausgesetzt, dass das File nicht leer ist.}
  READ(autos,d);
  eintragung.objekt := d; eintragung.naechster_eintrag := nil;
  NEW(fuhrpark); letzter_eintrag := fuhrpark;
  fuhrpark^ := eintragung;
  while not EOF(autos) do
    begin
      READ(autos,d);
      eintragung.objekt := d; eintragung.naechster_eintrag := nil;
      NEW(letzter_eintrag^.naechster_eintrag);
      letzter_eintrag := letzter_eintrag^.naechster_eintrag;
      letzter_eintrag^ := eintragung;
    end;
  {Zur Kontrolle soll das Datenfeld kennz der eingetragenen Saetze
  ausgegeben werden.}
  letzter_eintrag := fuhrpark;
  repeat
  with letzter_eintrag^ do
    begin WRITELN(objekt.kennz);
          letzter_eintrag := naechster_eintrag;
    end;
  until letzter_eintrag = nil;
  {Es soll nun ein Satz anhand eines eingelesenen Kennzeichens in
  der Liste gesucht und aus der Liste entfernt werden. Die Situa-
  tion zeigt Bild 2.10.}
  WRITE('Kennzeichen eingeben :'); READLN(kz);
  letzter_eintrag := fuhrpark; {Das Suchen muss bei der 1. Eintragung
                                beginnen.}
  if letzter_eintrag^.objekt.kennz = kz then
```

```
                        begin fuhrpark :=
                              letzter_eintrag^.naechster_eintrag;
                              DISPOSE(letzter_eintrag);
                        end
                        else
          if letzter_eintrag^.naechster_eintrag <> nil then
             repeat
             if letzter_eintrag^.naechster_eintrag^.objekt.kennz
                                                        = kz
          then begin hi := letzter_eintrag^.naechster_eintrag;
letzter_eintrag^.naechster_eintrag :=
             letzter_eintrag^.naechster_eintrag^.naechster_eintrag;
                    DISPOSE(hi);
               end
          else
            letzter_eintrag := letzter_eintrag^.naechster_eintrag;
             until letzter_eintrag^.naechster_eintrag = nil;
  {Kontrollausgabe der veraenderten Liste.}
  letzter_eintrag := fuhrpark;
  repeat
  with letzter_eintrag^ do
       begin WRITELN(objekt.kennz);
             letzter_eintrag := naechster_eintrag;
       end;
  until letzter_eintrag = nil;
  end.
```

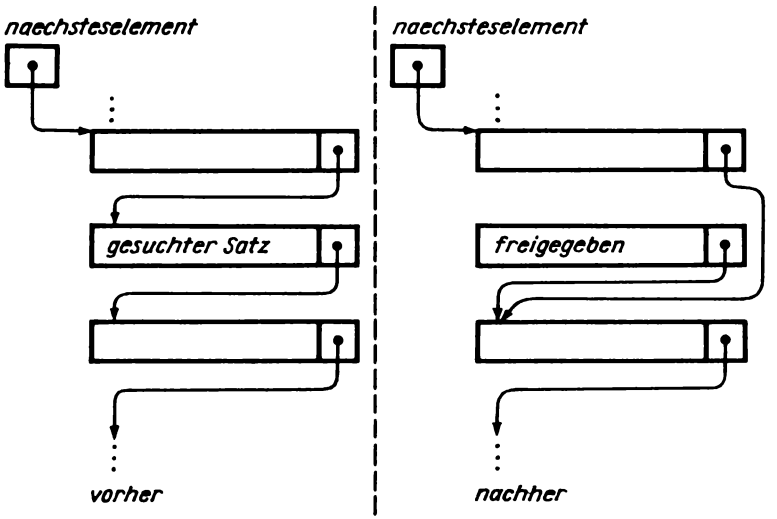


Bild 2.11. Streichen eines Satzes in einer Liste

(Was ändert sich am Programm, wenn auf die Variable eintragung verzichtet wird ?)

2. Dynamische Files

Prizipiell ist es auch möglich, Records zu verwenden, in denen Datenfelder von einem File-Typ enthalten sind. Werden diese Records dynamisch realisiert, so werden die File-Variablen ebenfalls dynamisch erzeugt, wie es folgendes Programm illustriert.

```
{dynfile}
program dynfile;
type df=record x:REAL;
              fi:file of INTEGER
        end;
var pf:^df;
    i,j:INTEGER;
begin
NEW(pf);
ASSIGN(pf^.fi,'f:dynf');
REWRITE(pf^.fi);
```

```
for i := 1 to 14 do WRITE(pf^.fi.i);
CLOSE(pf^.fi);
RESET(pf^.fi);
for i := 1 to 14 do begin READ(pf^.fi,j);
                          WRITELN(j);
                    end;
end.
```

2.13. Direkte Adressierung von Hauptspeicherbereichen

In TURBO-PASCAL ist der direkte Zugriff zu den Speicherstellen des Hauptspeichers möglich. So nützlich das für Fragen der Systemprogrammierung oder der effektiven Nutzung aller vom Betriebssystem gebotenen Möglichkeiten ist, so gefährlich ist es andererseits bei leichtfertiger Verwendung. Direktes Lesen aus bzw. Schreiben in den Hauptspeicher setzt voraus, dass der Programmierer sich gründlich über die aktuelle Speicherbelegung informiert hat.
Die Möglichkeiten, direkt mit Hauptspeicherbereichen zu arbeiten (ohne den Inline-Assembler - vgl. Abschn. 2.14.8. - heranzuziehen), sind:

- Vordefinierte Variablen, die vom TURBO-PASCAL-System verwaltete Speicherbereiche adressieren;
- Vordefinierte Funktionen zum Testen der aktuellen Speicherbelegungssituation und Umwandlung von INTEGER-Werten in Adressen und umgekehrt;
- Direkte Zuordnung von Variablen zu Speicherbereichen.

2.13.1. Vordefinierte Variablen und Felder

Die generelle Situation der Speicheraufteilung während der Abarbeitung des Programms, das im Memory-Modus (Direktmodus) übersetzt worden ist (vgl. Abschn. 1.), zeigte Bild 2.7 im Abschn. 2.12.
Die Arbeitspunkte der drei vom Laufzeitsystem benutzten Speicherbereiche

- Halde (heap),
- Rekursionskeller (recursion stack) und
- Kellerspeicher (stack)

sind in den drei vordefinierten Variablen

- HEAPPTR, RECURPTR und STACKPTR

fixiert. Diese Variablen können wie vom Programmierer definierte INTEGER-Variablen verwendet werden. Es ist selbstverständlich, dass durch ein leichtfertiges Verändern dieser Werte Speicherbereiche

zerstört werden können oder der Zugriff verloren gehen kann.
Das TURBO-PASCAL-System führt Kontrollen aus, um zu verhindern, dass Halde und Rekursionskeller ineinander überlaufen, es prüft nicht, ob der Kellerspeicher in den Rekursionskeller überläuft. Wenn diese Gefahr besteht, dann müsste das durch das Programm selbst ermittelt werden. Abhilfe könnte dadurch geschaffen werden, den Rekursionskeller auf einer niedrigeren Adresse anzulegen.

Weiterhin sind in TURBO-PASCAL zwei Felder vom Komponententyp BYTE vordefiniert; die Bezeichner dieser Felder sind

MEM und PORT.

Der Hauptspeicher wird als ein Feld mit dem Namen MEM aufgefasst, so dass durch

```
MEM[Index]
```

jeder Hauptspeicherplatz ansprechbar ist. Durch

```
MEM[¤F420] := ¤0A
```

wird also auf den Speicherplatz mit der Adresse ¤F420 der Wert ¤0A geschrieben, durch

```
b := MEM[¤F420]
```

kann er wieder gelesen werden (b vom Typ BYTE).

Beispiel:
Das Betriebssystem speichert Information über die Kommandozeile ab Adresse ¤80. Will man diese Kommandozeile in eine Stringvariable s übernehmen, so erfolgt das z. B. durch

```
for i := 1 to 127 do s[i] := MEM[¤7F+i];
```

Anmerkung: Der Speicherplatz MEM[¤80] enthält Information darüber, ob das aktuelle Programm direkt vom Betriebssystem aufgerufen wurde (0 <= MEM[¤80] <= 127) oder ob es durch eine EXECUTE- oder CHAIN-Anweisung aktiviert wurde (127 < MEM[¤80]).

Das zweite vordefinierte Feld kann verwendet werden, um zu den Datenports des Prozessors zuzugreifen. Jede Komponente des Feldes PORT stellt den Datenport mit der Nummer dar, die als Index angegeben wird. Weil die Datenports 8-Bit-Adressen haben, ist der Indextyp des Feldes BYTE. Eine Wertzuweisung zu einer Komponente des Feldes PORT bedeutet die Ausgabe des Wertes an den Datenport; wird andererseits eine Komponente des Feldes PORT in einem Ausdruck benutzt, so wird der an diesem Datenport anliegende Wert verwendet (Lesen eines Wertes von einem Datenport).

Beispiel:

```
PORT[¤12] := ¤04;
```

2.13.2. Vordefinierte Funktionen

Im Zusammenhang mit der Verwaltung der Halde (s. Abschn. 2.12.) wurden die Funktionen besprochen, die Information über den dynamischen Speicherplatz liefern. Es handelte sich um die Funktionen

MAXAVAIL	grösster zusammenhängender Speicherbereich auf der Halde
MEMAVAIL	freier Speicherbereich auf der Halde
PTR(i)	Erzeugen einer Adresse aus einer ganzen Zahl
ORD(pv)	Erzeugen einer ganzen Zahl aus einer Adresse

Darüber hinaus gibt es die Funktion

ADDR(bezeichner).

Sie liefert als Wert die Anfangsadresse eines Speicherbereichs als INTEGER-Wert. bezeichner kann dabei eine Variable sein - auch eine Komponente eines Arrays oder eines Records -, jedoch auch ein Routinenbezeichner oder ein Typbezeichner.

2.13.3. Das Attribut absolute

Die Deklaration einer Variablen hat die Form

```
var ...
        variable: typ;
        ...
```

Diese Form der Deklaration kann ergänzt werden durch die Stelle, wo diese Variable anzulegen ist. Die Angabe kann absolut durch die Adresse des Speicherplatzes oder relativ durch Bezugnahme auf eine bereits definierte Variable erfolgen.
Syntaktisch wird das durch das Basissymbol **absolute** erreicht. Obige Deklaration könnte auch in der Form

```
var
        variable: typ absolute adresse;
```

notiert werden. adresse bedeutet darin entweder eine ganzzahlige Konstante, die den Speicherplatz identifiziert oder einen bereits definierten Variablenbezeichner.

Beispiel:

```
var ...
      st: string[32];
      l_st: BYTE absolute st;
      u: INTEGER absolute ¤2000;
```

Bei der Bezugnahme auf eine andere Variable wird der Speicherbereich überlagert; st und l_st haben also in obigem Beispiel gleiche Speicheradressen. Da auf dem ersten Byte von st (st[0]) die aktuelle Länge der String-Variablen st steht, ist diese Längeninformation über l_st zu erreichen.
Die Festlegung direkter Variablen für Adressen hat vor allem Bedeutung für die Vermittlung von Werten zwischen Programmen, die aus unabhängigen Übersetzungen hervorgegangen sind. Beispiele dafür werden im Abschn. 2.14.7. behandelt.

2.14. Das Routinenkonzept

In jeder höheren Programmiersprache gibt es Sprachelemente, die es gestatten, ein Programm in Abschnitte zu zerlegen, so dass in diesen Abschnitten Teilaufgaben des Gesamtprogramms weitgehend unabhängig von anderen Programmteilen dargestellt werden können. Solche Abschnitte heissen Routinen, sie werden in Prozeduren und Funktionen unterteilt. Besonders häufig vorkommende Routinen sind in die höheren Programmiersprachen integriert. Sie sind dem Benutzer der Programmiersprache durch einen vordefinierten Routinenbezeichner zugänglich, die Algorithmen selbst liegen in abarbeitbarer Form im sog. Laufzeitsystem vor und werden mit dem aus der Compilation hervorgegangenen Programm verbunden. Solche vordefinierten Routinen sind bisher in allen Abschnitten als Standardprozeduren bzw. Standardfunktionen aufgetreten. Beispiele für Standardprozeduren sind WRITE, NEW, INSERT, für Standardfunktionen SQRT, EOLN, COPY, FRAC.
Im Programm treten diese Prozeduren durch die Nennung ihres Namens und durch Angabe von Parametern auf, die dem Austausch von Programmgrössen zwischen der speziellen Routine und der Umgebung, in der die Routine benutzt wird, dienen.
Was in der Routine im Detail geschieht, um die in der Routinenbeschreibung mitgeteilte Wirkung hervorzurufen, ist dem Anwender nicht bekannt, i. allg. ist es für ihn auch ohne Belang.
Sollen problemspezifische Teilaufgaben als selbständige Programmteile formuliert werden (Unterprogramme), so geschieht das in TURBO-PASCAL durch die Vereinbarung von Prozeduren oder von Funktionen. Wie schon der Terminus "Vereinbarung" sagt, erfolgt die Niederschrift im Vereinbarungsteil. Wie diese Vereinbarung zu erfolgen hat, wie die vereinbarten Routinen benutzt werden und welche Probleme beim Austausch von Programmgrössen auftreten, wird in den nächsten Abschnitten ausführlich behandelt.

2.14.1. Prozedurdeklarationen und Prozeduranweisungen

Eine Prozedurdeklaration besteht aus einem Prozedurkopf, dem ein Block folgt. Äusserlich hat eine Prozedurdeklaration Ähnlichkeit mit dem Aufbau des Gesamtprogramms, das ja auch aus dem Programmkopf und einem Block besteht.
Im Prozedurkopf erhält die Prozedur einen Namen, ausserdem werden sog. formale Parameter aufgelistet, die gewissermassen als Attrappen für Programmgrössen stehen, die beim Aktivieren der

Prozedur durch aktuelle Grössen ersetzt werden. Diese formalen Parameter werden durch Bezeichner beschrieben, denen Typattribute zugeordnet sind. Die Bezeichner der formalen Parameter sind nur innerhalb der Prozedurdeklaration gültig, sie haben keine Beziehung zu davor definierten Bezeichnern (neues Definitionsniveau). Innerhalb der Prozedurdeklaration können sie als Variablen benutzt werden, wobei ihre Verwendung ihrem Typ entsprechen muss.
Das Typattribut muss ein Typbezeichner sein, das ergibt sich zwingend aus der Festlegung über Typverträglichkeit (vgl. Abschn. 2.9.6.). Ein Typbezeichner, der zur Spezifizierung formaler Parameter dient, muss bereits erklärt sein.

Beispiele:

```
procedure addiere (a, b: REAL, var c: REAL);
procedure ausgabe (txt: string64);
```

Dem Prozedurkopf schliesst sich ein Block an, das ist im einfachsten Fall eine Verbundanweisung. Obige Beispiele könnten z. B. wie folgt zu vollständigen Prozedurdeklarationen erweitert werden:

```
procedure addiere (a, b: REAL, var c: REAL);
begin
  c := a + b;
end
```

bzw.

```
procedure ausgabe (txt: string64);
begin
  WRITELN(lst, txt);
end
```

2.14.1.1. Typparameter

Nach diesen Vorbetrachtungen soll nun der Aufbau einer Prozedurdeklaration systematisch dargestellt werden. Die Parameterliste kann leer sein (parameterlose Prozedur), sonst besteht sie aus einer Aufzählung von Parametergruppen, die durch Semikolon getrennt sind. Eine Parametergruppe hat eine der folgenden Formen:

```
b1, b2, ..., bn : typbezeichner    ,  n >= 1,
```

oder `var b1, b2, ..., bn : typbezeichner  , n >= 1.`
Der Unterschied, der in diesen Formeln gemacht wird, enthält Information, wie die aktuellen Parameter, die beim Aufruf der Prozedur

die formalen Parameter ersetzen, behandelt werden. Solche formalen Parameter, die nicht durch **var** gekennzeichnet sind, stehen für Werte, die an die Prozedur übergeben werden sollen. Vor dem Aufruf der Prozedur werden die aktuellen Werte berechnet und im Stack (vgl. Bild 2.7.) abgelegt, von wo sie in die Prozedur übernommen werden, wenn die Prozedur aktiv ist. Diese Parameter heissen Wertparameter.
Die durch **var** gekennzeichneten Parameter heissen Variablenparameter oder Referenzparameter. Von den zugeordneten aktuellen Parametern werden im Stack vor dem Aufruf der Prozedur die Adressen hinterlegt. Insbesondere heisst das, dass die Adressen indizierter oder dynamischer Variabler berechnet werden und durch Wertänderungen von Variablen, die für die Berechnung des Indexausdrucks herangezogen wurden, oder durch Veränderung des Wertes der Pointer-Variablen nicht mehr beeinflusst werden. Die aktive Prozedur kann dann einerseits mittels dieser Adressen die Werte der Variablen finden, andererseits aber auch die Werte dieser Variablen verändern (Ausgangsparameter, die Prozedur vermittelt über sie Werte an die aufrufende Umgebung).

Im Falle der Wertvermittlung repräsentiert der formale Parameter eine Variable, die (vom Compiler angelegt) lokal in der Prozedur ist. Das bedeutet, dass diese formalen Parameter so benutzt werden können wie Variablen, die innerhalb der Prozedur deklariert sind, sie werden als lokale Grössen betrachtet.
Wie schon gesagt, folgt dem Prozedurkopf ein Block, der also selbst mit einem Vereinbarungsteil beginnen kann. Betrachten wir das Beispiel der Eingabe der Werte einer Stichprobe vom File INPUT.

Beispiel:

```
     procedure lies_stichprobe (n: INTEGER; var a: vektor);
     var i: INTEGER;
     begin
        WRITELN('Es folgt die Anforderung von ', n:3,
                ' Werten der Stichprobe');
        for i := 1 to n do READLN(a[i]);
     end;
```

i ist hierin eine lokale Variable. Der formale Parameter n wird so behandelt, als wäre er auf dem Niveau dieses Vereinbarungsteils deklariert und beim Aufruf der Prozedur initialisiert.
a muss als Variablenparameter vereinbart sein, da ja in diese Prozedur die Werte nur eingelesen, aber in der Umgebung der Prozedur ausgewertet werden.
Wenn auch der Vereinbarungsteil dieses Beispiels nur eine Deklaration enthält, so könnten dort alle Definitionen und Deklarationen

auftreten, auch wieder Prozedur- und Funktionsdeklarationen. Dadurch entsteht eine Blockschachtelung, die Fragen nach dem Gültigkeitsbereich von Bezeichnern aufwirft. Diese Fragen werden im Abschn. 2.14.3. beantwortet.
Der Aufruf einer Prozedur erfolgt durch eine Prozeduranweisung. Sie besteht nur aus dem Namen der Prozedur und einer Liste von aktuellen Parametern. (Handelt es sich um eine parameterlose Prozedur, so sind selbstverständlich auch beim Aufruf keine Parameter anzugeben.) Die aktuellen Parameter müssen in ihrer Anzahl, im Typ und in der Reihenfolge den korrespondierenden formalen Parametern entsprechen. Es müssen also auf den Positionen von Wertparametern Ausdrücke stehen, die berechnet werden können, so dass der daraus resultierende Wert übergeben wird. Auf der Position von Variablenparametern müssen Variablenbezeichner stehen, das können auch Bezeichner von strukturierten Variablen, dynamischen Variablen, Komponenten dynamischer Variablen oder File-Variablen sein. Man beachte, File-Variablen dürfen nur als Variablenparameter verwendet werden!
Nach Abarbeitung der Prozedur wird das Programm mit der Anweisung fortgesetzt, die auf die Prozeduranweisung folgt.

Beispiele:
1. Maximales und minimales Element einer Stichprobe

```
procedure maxmin (a: vektor; n: INTEGER; max_el, min_el: REAL);
var hmax, hmin: REAL;
            i: INTEGER;
begin
if a[1] < a[2] then begin min_el := a[1]; max_el := a[2] end
               else begin min_el := a[2]; max_el := a[1] end;
for i := 3 to n do
     if a[i] < hmin then hmin := a[i]
                    else
                    if a[i] = hmax then hmax := a[i];
min_el := hmin; max_el := hmax;
end;
```

Bemerkungen:
(1) Die lokalen Variablen hmin und hmax dienen nur der Verbesserung der Abarbeitungsgeschwindigkeit, da der Zugriff zu lokalen Variablen zeitgünstiger ist. Prinzipiell hätte auch mit min_el und max_el gearbeitet werden können.
(2) Da a als Wertparameter behandelt wird, wird eine Kopie unter einer generierten lokalen Variablen abgelegt. Für grössere Objekte ist das speicherplatzaufwendig. In solchen Fällen ist es häufig

besser, einen Variablenparameter zu verwenden, also

```
procedure maximum (var a: vektor; n: INTEGER;
                   max_el, min_el: REAL);
```

Um den Aufruf der Prozedur zu zeigen, sei ein Rahmenprogramm angegeben:

```
{stprobe.pas}
program stichprobenverarbeitung;
const n = 200;
type vektor = array[1..n] of REAL;
var i, laenge: INTEGER;
    maxi, mini: REAL;
    c: CHAR;
    x: vektor;

procedure eingabe(var a: vektor; l: INTEGER);
var i: INTEGER;
begin
     WRITE('Stichprobenwerte: ');
     for i := 1 to l do READ(a[i]);
end;

procedure maxmin (a: vektor; n: INTEGER; max_el, min_el: REAL);
var hmax, hmin: REAL;
            i: INTEGER;
begin
if a[1] < a[2] then begin min_el := a[1]; max_el := a[2] end
               else begin min_el := a[2]; max_el := a[1] end;
for i := 3 to n do
     if a[i] < hmin then hmin := a[i]
                    else
                    if a[i] = hmax then hmax := a[i];
min_el := hmin;
max_el := hmax;
end;

begin
repeat
     c := 'J';
     WRITE('Umfang der Stichprobe eingeben: '); READLN(laenge);
     if laenge in[7..200] then {Mindestlaenge der Stichprobe sei 7}
                       begin
                       eingabe(x, laenge);
```

```
                    WRITELN('   ');
                    maxmin(x, laenge, maxi, mini);
                    WRITELN('Max = ', maxi:8:4, ', Min = ',mini:8:4);
                  {Anweisungen zur weiteren Auswertung der Stichprobe}
                    WRITE('Weitere Stichprobe? (J/N)');     READLN(c);
                    end  else
                    WRITELN('Angabe falsch. Wiederholen!');
until not ( c in ['j', 'J']);
{Weitere Anweisungen des Programms.}
end.
```

(Was müsste an dem Programm geändert werden, wenn die Eingabe nicht von INPUT, sondern von einem Textfile stpr.dat erfolgen soll?)

2. Einlesen einer File-Komponente und Anfügen dieser Komponente an eine dynamisch erzeugte Liste

```
{Verlaengern einer Liste}
procedure verlaengere(var f: filetyp; var p: pointer);
{Die Prozedur setzt voraus, dass p eine Pointer-Variable ist, die
ein Objekt adressieren kann, das mit dem Komponententyp des Files
identisch ist, also
type pointer = ^komptyp;
     komptyp = record ...
                      nf: pointer; ...
               end;
     filetyp = file of komptyp;
var w: pointer;
}

var hp: pointer;
begin
     if not EOF(f) then begin hp := p;
                        if hp = nil then begin NEW(p);
                                             READ(f, p^);
                                             p^.nf := nil;
                                        end
                                   else begin
                                          hp := p;
                                          while hp^.nf <>nil do
                                              hp := hp^.nf;
                                          READ(f, hp^);
                                          hp^.nf := nil;
                                        end;
                     end
```

```
                else WRITELN('File ist leer');

end;
```

Weitere Beispiele zur Unterprogrammtechnik im Abschn.2.14.5.

2.14.1.2. Typfreie Parameter und Typanpassung

Zu den bisher behandelten formalen Parametern gehörte ein Typattribut. Nun gibt es aber Fälle, wo es auf den Typ nicht ankommt. Solche Fälle können z. B. sein, dass die Inhalte von zwei Speicherbereichen vertauscht werden sollen (Anfangsadresse und Länge sind hinreichend) oder das Files mit BLOCKREAD oder BLOCKWRITE transportiert werden sollen u. dgl. Für die Programmierung solcher Fälle können typfreie Parameter verwendet werden. Typfreie Parameter können beim Aufruf nicht durch Typvariablen ersetzt werden (unverträglich mit allen Typen), es sei denn, es handelt sich um solche Standardprozeduren, für die der Typ der Parameter bedeutungslos ist (BLOCKREAD, BLOCKWRITE, MOVE, FILLCHAR oder ADDR).
Auf der Position der zugeordneten aktuellen Parameter werden i. allg. absolute Variablen benutzt.

Beispiel: Umspeichern eines Teils zweier Vektoren

```
{typfreie Parameter}
program typfrei;
var s1, s2: string[16];
    s1abs: BYTE absolute s1;
    s2abs: BYTE absolute s2;
    j: INTEGER;
procedure vertausche( laenge: INTEGER; var absvar1, absvar2);
    type vektor = array [1..512] of BYTE;
    var v1: vektor absolute absvar1;
        v2: vektor absolute absvar2;
        i: INTEGER;
        hi: BYTE;
    begin
      for i := 1 to laenge do
      begin
         hi := v1[i+1];
         v1[i+1] := v2[i+1];
         v2[i+1] := hi;
      end;
   end {vertausche};

begin {Rahmen zum Testen der Prozedur}
```

```
  WRITE('16 Zeichen fuer s1 eingeben: ');
  READLN(s1);
  WRITE('16 Zeichen fuer s2 eingeben: ');
  READLN(s2);

  vertausche(10,s1abs, s2abs);
  WRITELN(s1);
  WRITELN(s2);
end.
```

```
Running
16 Zeichen fuer s1 eingeben: 1111222233334444
16 Zeichen fuer s2 eingeben: aaaabbbbccccdddd
aaaabbbbcc334444
1111222233ccdddd
```

In den zuletzt behandelten Situationen wurden keine Typparameter benötigt. Im Falle der Verwendung von Typparametern gibt es einen Fall, wo bei Nichtübereinstimmung der Typen der formalen Parameter mit den aktuellen Parametern automatisch eine Typanpassung erfolgt. (Die Zuweisung real:= integer ist stets erlaubt.) Es handelt sich dabei um Parameter von einem String-Typ. Standardmässig ist gefordert, dass die Länge des aktuellen Parameters dem Längenattribut des formalen Parameters entspricht. Diese Standardfestlegung kann jedoch durch die Compilerdirektive V aufgehoben werden (s. auch Abschn. 2.15.).

Beispiel:

```
program b;
{¤V-}
type string128 = string[128];
 ...
procedure stringparameter(s:string128;...);
 ...
begin
 ...
stringparameter('Testbeispiel',...);
 ...
end.
```

Der Aufruf der Prozedur wäre ohne die Compilerdirektive {¤V-} falsch.

2.14.2. Funktionsdeklarationen und Funktionsaufrufe

Solche Unterprogramme, die genau einen Wert liefern, der unmittelbar in Ausdrücken weiterverwendet werden soll, werden in der Form von Funktionsvereinbarungen deklariert. Auch Funktionsvereinbarungen bestehen aus einer Kopfzeile, die sich von dem Prozedurkopf nur durch folgendes unterscheidet:

Kennzeichnendes Basissymbol ist **function**,

der Typ des Funktionswertes ist, durch Doppelpunkt getrennt, hinter der Parameterliste anzugeben. Der Typ muss ein Typbezeichner sein. Als Typen sind nur die Standardtypen REAL, INTEGER, BYTE, BOOLEAN, CHAR, ferner Skalartypen, Teilbereichstypen und String-Typen erlaubt.

Der dem Funktionskopf folgende Block entspricht in seinem Aufbau dem der Prozedur. Es muss allerdings mindestens eine Wertzuweisung geben, worin der Funktionsbezeichner auf der linken Seite vorkommt (worin also dem Funktionsbezeichner ein Wert zugewiesen wird).

Beispiele:

(1)
```
function abstand(x,y:REAL):REAL;
begin
  abstand:=SQRT(SQR(x)+SQR(y));
end;
```

(2)
```
function reziprokfak(n:INTEGER):REAL;
var i:INTEGER;
    y:REAL;
begin
  if (n=0) or (n=1) then reziprokfak:=1.0
                    else
                    begin y:= 1;
                          for i:=2 to n do y:=y/i;
                          reziprokfak:= y;
                    end;
end;
```

(3)
```
function vorhanden(var sv:stringvektor;
                                  n:INTEGER; s:string8):BOOLEAN;
var i:INTEGER;
   ok:BOOLEAN;
begin
i:=1; ok:=FALSE;
repeat
```

```
if sv[i]=s then begin ok:=TRUE;
                      vorhanden:=TRUE;
                end
           else i:=i+1;
until ok or (i>n)
if not ok then vorhanden:=FALSE;
end;
```

Weitere Beispiele zu Funktionsunterprogrammen im Abschn. 2.14.5.

2.14.3. Gültigkeitsbereiche von Bezeichnern

Hinsichtlich der Gültigkeitsbereiche von Bezeichnern gibt es in TURBO-PASCAL folgende Regelung:

1. Bezeichner sind auf dem Blockniveau gültig, auf dem sie vereinbart wurden.
2. Eine Prozedur oder Funktionsdeklaration eröffnet ein neues Blockniveau. Auf diesem neuen Niveau können bereits vereinbarte Bezeichner neu vereinbart werden. Darunter ist zu verstehen, dass sie entweder als formale Parameter auftreten oder im Vereinbarungsteil des Blocks als Bezeichner von Konstanten, Typen, Variablen oder Routinen vereinbart werden. In einem solchen Falle sind die auf einem niedrigeren Niveau vereinbarten Bezeichner nicht mehr zu erreichen.
3. Bezeichner, die vor einer Prozedur- oder Funktionsdeklaration vereinbart wurden und in der Prozedur/Funktion nicht neu vereinbart werden, sind in der Prozedur/Funktion erreichbar. Marken gelten aber nur auf dem Blockniveau, auf dem sie vereinbart wurden. (Es ist also nicht möglich, eine Prozedur durch eine Sprunganweisung in das aufrufende Programm zu verlassen.)
4. Vordefinierte Bezeichner gelten als auf dem umfassendsten Niveau (Niveau 0) definiert. Sie stehen auf jedem weiteren Niveau zur Verfügung, können aber prinzipiell neu definiert werden (wovon allerdings abzuraten ist).

Bild 2.12. zeigt schematisch ein Programm, in das verschiedene Unterprogramme eingeschlossen sind. Es ist angegeben, welche Variablen auf den unterschiedlichen Niveaus erklärt sein sollen.

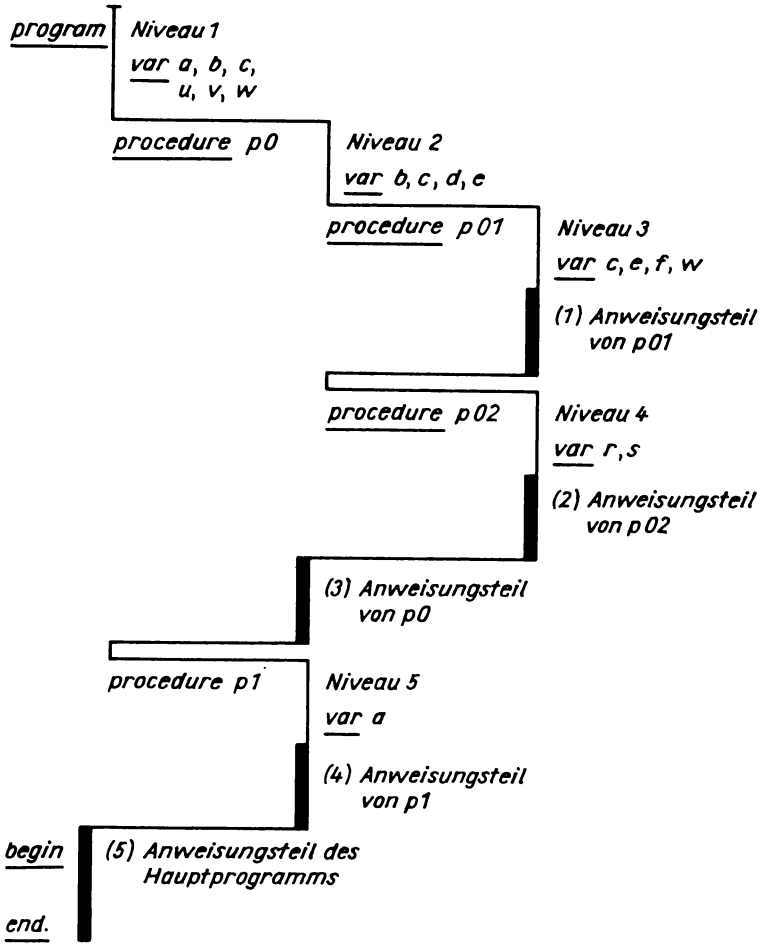


Bild 2.12. Blockstruktur und Gültigkeitsbereiche von Bezeichnern

An den markierten Stellen sind folgende
Bezeichner gültig:
(1) von Niveau 1: a,u,v,p0
von Niveau 2: b,d,p01
von Niveau 3: c,e,f,w

```
(2)  von Niveau 1: a,u,v,w,p0
     von Niveau 2: b,c,d,e,p01,p02
     von Niveau 3:  --
     von Niveau 4: r,s

(3)  von Niveau 1: a,u,v,w
     von Niveau 2: b,c,d,e,p01,p01
     von Niveau 3:  --
     von Niveau 4:  --

(4)  von Niveau 1: b,c,u,v,w,p0,p1
     von Niveau 2:  --
     von Niveau 3:  --
     von Niveau 4:  --
     von Niveau 5: a

(5)  von Niveau 1: a,b,c,u,v,w,p0,p1
     von Niveau 2:  --
     von Niveau 3:  --
     von Niveau 4:  --
     von Niveau 5:  --
```

2.14.4. Spezielle Standardfunktionen und Standardprozeduren

Im Zusammenhang mit der Darstellung von Formeln des Arbeitens mit Zeichenketten, mit dem Filekonzept und dem Zeigerkonzept wurden Standardfunktionen bzw. -prozeduren behandelt, die unmittelbar dazugehörten.
Darüber hinaus sind in TURBO-PASCAL einige spezielle Abläufe als Standardroutinen aufgenommen worden, die oftmals nützlich sind, sich aber nicht zwingend einem bisher behandelten Thema zuordnen lassen. Deshalb werden diese Routinen in einem besonderem Abschnitt erläutert. (Routinen zur grafischen Arbeit s. Abschn. 3.)

HALT
Die Abarbeitung dieser Prozedur bewirkt Programmunterbrechung und Rückkehr zum TURBO-PASCAL-System.

EXIT
Beenden der Abarbeitung des aktiven Blocks. Wird EXIT innerhalb einer Prozedur abgearbeitet, so bewirkt es das Verlassen der Prozedur, im Hauptprogramm das Beenden des Programms.

DELAY(ms)
ms bezeichnet einen ganzzahligen Wert. Das System generiert an Stelle der Prozedur einen Zyklus, der die Weiterführung des Programms um ms Millisekunden verzögert(bei einer Taktfrequenz von 4 MHz).

FILLCHAR(variable,anzahl,bc)
Darin bedeutet variable eine Variable beliebigen Typs, anzahl einen Ausdruck, der einen INTEGER-Wert liefert, bc einen Ausdruck des Typs CHAR oder BYTE. Die Wirkung der Prozedur besteht darin, anzahl Bytes der Variablen variable mit dem Wert von bc zu überschreiben. Dass es auf den Typ der Variable nicht ankommt, bedeutet, dass variable wie ein typfreier Parameter behandelt wird.

Beispiel:
```
FILLCHAR(u0,SIZEOF(u0),¤FF)
```

MOVE(quelle,ziel,anzahl)
Diese Standardprozedur überträgt anzahl Bytes byteweise von dem durch die Variable quelle spezifizierten Speicherplatz nach ziel. quelle und ziel werden als typfreie Parameter behandelt. Die Behandlung von Überlappungen der Bereiche quelle und ziel erfolgt im Sinne der byteweisen Übertragung.

RANDOMIZE
Die Prozedur initialisiert den Zufallszahlengenerator.

RANDOM
Diese parameterlose Funktion liefert eine Zufallszahl z, für die 0<=z<1 gilt.

RANDOM(i)
i symbolisiert einen Ausdruck, der einen positiven INTEGER-Wert liefert. Die Funktion liefert eine Zufallszahl z vom Typ INTEGER, für die gilt 0<=z<i.

SIZEOF(bezeichner)
bezeichner symbolisiert einen Variablennamen oder einen Typnamen. Als Ergebnis liefert die Funktion die Anzahl der Bytes, die im Hauptspeicher für das Objekt benötigt werden. Das Ergebnis ist vom Typ INTEGER.

HI(i)
i symbolisiert einen Ausdruck, der einen Wert vom Typ INTEGER liefert. Die Funktion liefert von den zwei Bytes dieses Wertes das

höherwertige. Im ganzzahligen Ergebnis steht dieses Byte auf der niederwertigen Stelle, die höherwertige Stelle ist mit 0 belegt.

LO(i)
i symbolisiert einen Ausdruck, der einen Wert vom Typ INTEGER liefert. Das niederwertige Byte dieses Wertes wird in das niederwertige Ergebnisbyte kopiert, das höherwertige Ergebnisbyte ist 0.

SWAP(i)
Das Argument i symbolisiert einen Ausdruck vom Typ INTEGER. Als Ergebnis wird eine ganze Zahl geliefert, die dadurch erzeugt wird, dass höherwertiges Byte und niederwertiges Byte vertauscht werden.

UPCASE(c)
c ist ein Wert vom Typ CHAR. Ergebnis ist das durch Tastaturumschaltung zugeordnete Zeichen. Wenn ein solches Zeichen nicht existiert, so ist der Funktionswert gleich dem Argumentwert.

KEYPRESSED
Diese logische Standardfunktion liefert den Wert TRUE, wenn ein Zeichen eingegeben wurde, sonst hat sie den Wert FALSE.

Eine Sonderstellung unter den Standardprozeduren bzw. -funktionen nehmen die Systemrufe ein. Um die BDOS- bzw. BIOS-Routinen des Betriebssystems CP/M aufrufen zu können, gilt es in TURBO-PASCAL die Standardprozeduren BDOS und BIOS und die Standardfunktionen BDOS, BDOSHL, BIOS und BIOSHL. In TURBO-PASCAL werden standardmässig alle Eingaben und Ausgaben über CP/M-BIOS ausgeführt.
Der Aufruf dieser Prozeduren/Funktionen geschieht folgendermassen:

```
BIOS(funktionsnummer[,parameter])
intvar:= BIOS(funktionsnummer[,parameter])
intvar:= BIOSHL(funktionsnummer[,parameter])
```

Die Angaben in eckigen Klammern bedeuten dabei, dass parameter auch fehlen darf.
An Stelle des Systemrufs BIOS kann auch der Systemruf BDOS verwendet werden.
In jedem Falle aktiviert der Ruf eine BDOS- bzw. BIOS-Routine. Als Funktion aufgerufen wird als Funktionswert der Wert zurückgegeben, den die BDOS- bzw. BIOS-Routine in das Register A einträgt; bei den Funktionsrufen BDOSHL bzw. BIOSHL erfolgt die Übergabe des Wertes aus dem Datenregister HL. Das Verwenden dieser Prozeduren/Funktionen kann nur dann erfolgen, wenn das CP/M-Betriebssystem bekannt ist oder entsprechende Spezialliteratur zugänglich ist /5/

/8/.
Für denjenigen, der mit CP/M vertraut ist, könnte folgende Zusammenstellung genügen:

Tafel 2.2. BIOS-Funktionen

Funktionsnummer	Operation	Parameterwert	Parameterreg.	Ergebniswert	Ergebnisreg.
0	Warmstart	-	-	-	-
1	Konsolstatus	-	-	FF/00	A
2	Konsoleingabe	-	-	Zeichen	A
3	Konsolausgabe	Zeichen	C	-	-
4	Listenausgabe	Zeichen	C	-	-
5	Stanzerausgabe	Zeichen	C	-	-
6	Lesereingabe	-	-	Zeichen	A
7	Spur 0 anwählen	-	-	-	-
8	Laufwerk anwählen	Nr. des Laufwerks	C	DPH-Adresse	HL
9	Spurnummer setzen	Nr. der Spur	BC	-	-
10	Sektornummer setzen	Nr. des Sektors	BC	-	-
11	Pufferadresse setzen	Adresse	BC	-	-
12	Sektor lesen	-	-	Fehlerkode	A
13	Sektor schreiben	-	-	Fehlerkode	A
14	Listenausgabestatus	-	-	0/1	A
15	Sektor konvertieren logisch -> physisch	logische Sektornr. Adr. der Transformationstab.	BC DE	physische Sektornr.	HL

Tafel 2.3. BDOS-Funktionen

Funktions-nummer	Operation	Para-meter-wert	Para-meter-reg.	Ergebnis-wert	Ergeb-nis-reg.
0	Warmstart	-	-	-	-
1	Konsoleingabe	-	-	Zeichen	A
2	Konsolausgabe	Zeichen	E	-	-
3	Lesereingabe	-	-	Zeichen	A
4	Stanzerausgabe	Zeichen	E	-	-
5	Druckerausgabe	Zeichen	E	-	-
6	Konsoleingabe/ -ausgabe direkt	FF(Eing.) oder Zeichen		0 (nicht bereit) oder Zeichen	A
7	IOBYTE lesen	-	-	Byte	A
8	IOBYTE setzen	Byte	E	-	-
9	Puffer ausgeben	Adresse	E	-	-
10	Puffer lesen	Adresse	E	-	-
11	Konsolstatus lesen	-	-	Byte	A (00/FF)
12	Versionsnr.lesen	-	-	Byte	HL
13	Reset Disk	-	-	-	-
14	Laufwerkanwahl	Nr. des Laufwerks	E	-	-
15	File eröffnen	Adresse FCB	DE	Fehlerkode	A
16	File abschliessen	Adresse FCB	DE	Fehlerkode	A
17	Erstes File suchen	Adresse FCB	DE	Fehlerkode	A
18	Nächstes File suchen	-	-	Fehlerkode	A
19	File löschen	Adresse FCB	DE	Fehlerkode	A
20	Sequentiell lesen	Adresse FCB	DE	Fehlerkode	A
21	Sequentiell schreiben	Adersse FCB	DE	Fehlerkode	A
22	File erzeugen	Adresse FCB	DE	Fehlerkode	A
23	File umbenennen	Adresse	DE	Fehlerkode	A

		FCB			
24	Aktuelle Laufwerke	-	-	Vektor	HL
25	Standardlaufwerk suchen	-	-	Laufwerk	A
26	DMA-Adresse setzen	DMA-Adr.	-	-	-
27	Belegungstab. lesen	-	-	Vektor	HL
28	Schreibschutz setzen	-	-	-	-
29	R/O-Laufwerke suchen	-	-	Vektor	HL
30	Fileattribute setzen	Adresse FCB	DE	-	-
31	Diskettenparameter-block lesen	-	-	Adresse	HL
32	Benutzernr. lesen	FF	E	INTEGER	A
	setzen	INTEGER	E	-	-
33	Direktes Lesen	Adresse FCB	DE	Fehlerkode	A
34	Direktes Schreiben	Adresse FCB	DE	Fehlerkode	A
35	Filegrössen lesen	Adresse FCB	DE	-	-
36	Komponentennr. setzen	Adresse FCB	DE	-	-
37	Reset Drive(s)	Drive Vector	DE	0	A

2.14.5. Beispiele und Übungen

1. Wert eines Polynoms an einer Stelle x0

```
{P(x)/x=x0}
function polynomwert( var a: koeffizientenvektor;
                          n: INTEGER;{Grad}
                          x0: REAL) :REAL;

var i :INTEGER;
    y :REAL;
begin
   y := 0.0;
   for i := n downto 0 do y := y * x0 + a[i];
   polynomwert := y;
end;
```

2. Suchen in einer verketteten Liste

Gegeben sei eine verkettete Liste mit Elementen, für die folgende Typdefinitionen gelten:

```
     type merkmal = string[10];
          ple = ^le;
          le  = recoord ...
                  kennz: merkmal;
                  nachfolger: ^ple
                  end;
```

Eine Pointer-Variable adressiere das erste Element. Die Funktion finde prüft, ob der Wert einer String-Variablen als Wert des Datenfeldes kennz in der Liste auftritt. Wird ein solcher Datensatz gefunden, so wird seine Adresse als Funktionswert übergeben, wird ein solcher Satz nicht gefunden, so ist der Funktionswert nil.

```
{finde}
{¤v-}
function finde (anf ple; aktuell:merkmal) :ple;
var hi :ple;
    gefunden :BOOLEAN;
begin gefunden := FALSE; hi := anf;
   while (hi <> nil) and not gefunden do
    if hi^.kennz = aktuell then gefunden := TRUE
                           else hi := hi^.nachfolger;
finde := hi;
end;
```

3. Numerische Integration

Nach der Simpsonschen Regel kann $I = \int_a^b f(x)\,dx$ näherungsweise durch die Formel

$$I = h/3 * (f(a) + 2(f(x_2) + f(x_4) + \ldots f(x_{n-2}) + 4(f(x_1) + f(x_3) + \ldots f(x_{n-1})) + f(b))$$

berechnet werden, wobei

$h = (b-a)/n$, n geradzahlig und x_i die Stützstellen a+ih sind, $x_0 = a$, $x_n = b$.

Die folgende Funktion berechnet I iterativ, indem, ausgehend von n = 4, die Anzahl der Stützstellen verdoppelt wird, bis die durch die Verdopplung erreichte Verbesserung vernachlässigt werden kann. Die in der Funktion simpson verwendete Funktion f ist global definiert. Das folgende Programm enthält die Deklarationen und einen Rahmen.

```
{simpson}
program numerischeIntegration;
var
function f( x:REAL):REAL;
begin {Hier erfolgt die Berechnung des Funktionswertes}
end;

function simpson (
                    a,b ,eps:REAL):REAL;
var x,s2,s4,y,c,h,i1,i2,d:REAL;
    q:BOOLEAN; {q steuert 1. oder i-te Integration}
    m,n:INTEGER;
begin
n:=4; m:=0; q:=TRUE;
s2:=0.0; x:=a;
c:=f(a)+f(b);
i1:=10e+10;
repeat
h:=(b-a)/n; s4:=0.0;
  if q then { Iterationsanfang}
       repeat
       m:=m+1; x:=x+h;
       y:=f(x);
       if odd(m) then s4:=s4+4*y
                 else s2:=s2+2*y;
       until m>n-1
       else
```

```
          begin
          x:=x-h; q:=TRUE;
          repeat
            m:=m+2; x:=x+2*h;
            s4:=s4+4*f(x);
          until m>n-2 {m beginnt mit -1}
          end;
        i2:=c+s2+s4; i2:=i2*h/3.0;
        d:=abs(i1-i2);              {Testvorbereitung}
        i1:=i2; n:=2*n; q:=FALSE;   {Vorbereitung der naechsten
                                     Iteration}
        m:=-1; x:=a; s2:=s2+0.5*s4;
until( d < eps) or (n>512);
simpson:=i2;
 if n>512 then WRITELN('Nach', n:3,' Schritten abgebrochen');
end{simpson};

begin
writeln (simpson(0,1,1.0e-14));
end.
```

4. Prozedur zur hexadezimalen Ausgabe eines Speicherbereichs

```
{dump}
program dump_bereich;
procedure dump(var bereich; n:INTEGER);
var bb:array[1..255] of BYTE {absoluter Bereich};
    i,j,m,k:INTEGER;
    c1,c2:CHAR;
    t:TEXT;
begin
ASSIGN(t, 'f:dump'); REWRITE(t);
j:=1;
for i:=1 to n do
    begin
    m := bb[i] div 16; k := bb[i] mod 16;
    if m in [0..9] then c1 := CHAR(¤30 + m)
                   else c1 := CHAR(¤61+m-10);
    if k in [0..9] then c2 := CHAR(¤30+k)
                   else c2 := CHAR(¤61+k-10);
    WRITE(t,c1,c2,'|');
    j:=j+1;
    if j=16 then begin WRITELN(t);
                       j:=1;
```

```
                end;
    end;
end;
begin
dump(MEM[¤80],90);
end.
```

```
Running

d2|71|cf|c3|e1|ce|e1|3d|c2|71|cf|cd|66|cc|cd|
5e|ca|21|f0|cf|e5|7e|32|cd|cf|3e|10|cd|60|ca|
e1|7e|32|dd|cf|af|32|ed|cf|11|5c|00|21|cd|cf|
06|21|cd|42|cc|21|08|c8|7e|b7|ca|3e|cf|fe|20|
ca|3e|cf|23|c3|30|cf|06|00|11|81|00|7e|12|b7|
ca|4f|cf|04|23|13|c3|43|cf|78|32|80|00|cd|98|
```

5. Hexadezimale Ausgabe eines INTEGER-Wertes

```
{int -> hex}
program int_hex;
procedure int_to_hex(m:INTEGER);
var ziffer: array [1..4] of INTEGER;
    i:INTEGER;
begin
  ziffer[1] := HI(m) div 16; ziffer[2] := HI(m) mod 16;
  ziffer[3] := LO(m) div 16; ziffer[4] := LO(m) mod 16;
  for i := 1 to 4 do
     if ziffer[i] in [0..9] then WRITE(ziffer[i])
                            else WRITE(CHR(ziffer[i] + 87));
WRITELN;
end;
begin
  int_to_hex(286);
  int_to_hex(-2);
end.
```

```
Running
011e
fffe
```

6.Lösung eines linearen Gleichungssystems

```
{lin. Gl.-Syst.}
program regulaeres_lineares_gleichungssystem;
const nz = 10;
      ns = 11;

type koeff = array [1..nz, 1..ns] of REAL;
     loesung = array [1..10] of REAL;
var l, m, n: INTEGER;
    fn: string[14];
    a, b: koeff;
    x: loesung;
    s, sum: REAL;
    koeff_file: TEXT;
    procedure gauss(var a:koeff; var x:loesung; n: INTEGER);
        var i, j, k, i1, j1, hj, n1:INTEGER;
            ha: REAL;
            pivotvektor: array[1..10] of INTEGER;
            v: loesung;
            procedure pivot(k :INTEGER;var i1,j1:INTEGER);
                var i, j: INTEGER;
                    h, h1: REAL;
                begin
                  h := 0;
                  for i := k to n do
                     for j := k to n do
                        begin
                          h1 := ABS(a[i,j]);
                          if h < h1 then begin h := h1;
                                               i1 := i;
                                               j1 := j;
                                         end;
                        end;
              end;
     begin
        n1 := n + 1;
        for k := 1 to n do pivotvektor[k] := k;
        for k := 1 to n-1 do
           begin pivot(k,i1,j1);
                 if i1 <> k then
                           for j := k to n1 do
                                  begin ha := a[k,j];
                                        a[k,j] := a[i1,j];
                                        a[i1,j] := ha;
                                  end;
```

```
                if j1 <> k then
                          for i := 1 to n do
                                 begin ha := a[i,k];
                                       a[i,k] := a[i,j1];
                                       a[i,j1]:= ha;
                                 end;
                hj := pivotvektor[k];
                pivotvektor[k] := pivotvektor[j1];
                pivotvektor[j1] := hj;
                for i := k+1 to n do
                      begin a[i,k] := a[i,k]/a[k,k];
                            for j := k+1 to n1 do
                              a[i,j] := a[i,j] - a[k,j]*a[i,k];
                      end;
                v[n] := a[n, n1]/a[n,n];
                for i := n-1 downto 1 do
                      begin v[i] := a[i,n1];
                            for j := i+1 to n do
                                  v[i] := v[i] - a[i,j]*v[j];
                            v[i] := v[i]/a[i,i];
                      end;
                for i := 1 to n do x[pivotvektor[i]]:= v[i];
      end;
 end; {gauss}
 begin
    WRITE('Name des Koeffizientenfiles eingeben:');
    READLN(fn);
    ASSIGN(koeff_file, fn); RESET(koeff_file);
    READLN(koeff_file,l);
    for m := 1 to l do
              for n := 1 to l+1 do
              begin
                 READ(koeff_file, a[m,n]); b[m,n] := a[m,n];
                 WRITELN(a[m,n]);
              end;
 gauss(a, x, l);
 for m := 1 to l do
     WRITELN('x[',m:2,']=',x[m]:18:6);

sum := 0;

for m := 1 to l do
   begin s := 0.0;
        for n := 1 to l do s := s + b[m,n] *x[n];
        s := s - b[m,l+1];
```

```
        sum := sum + s*s;
    end;
  WRITELN ('Norm |ax - b| :', sum);
end.
```

Im weiteren werden einige Aufgaben angegeben. Lösungen dafür findet man im Abschn. 4.

A2.14.5.-1: Nullstellen einer Funktion
Es ist eine Funktion zu schreiben, die von einer gegebenen Funktion f(x) eine Nullstelle ermittelt. In dem Intervall [a, b] gibt es mindestens eine Nullstelle, und es gilt, dass f(a) und f(b) unterschiedliches Vorzeichen haben.

A2.14.5.-2: Suchen des betragsmässig grössten Elements einer Matrix
Es ist eine Prozedur zu schreiben, die das betragsmässig grösste Element einer (m*n)-Matrix bestimmt und die Indizes dieses Elements übergibt.

A2.14.5.-3: Unendliche Reihe
Der Wert der Funktion

$$y = \sum_{n=0}^{\infty} (-1)^n \binom{0.5}{n} \frac{x^{2n}}{(2n+1)!}$$

ist an der Stelle x = x0 zu berechnen. Die (näherungsweise) Berechnung des Funktionswertes ist zu beenden, wenn der Betrag des letzten berechneten Summanden < 10e-7 ist.

A2.14.5.-4: Berechnung der Komponenten eines Vektors
Es ist eine Prozedur zu schreiben, die aus zwei Vektoren x und y mit den Komponenten x[0], x[1], ..., x[n-1], y entsprechend, die Komponenten a[j] eines Vektors a nach folgender Formel berechnet:

$$a_j = \sum_{i=o}^{n} \frac{y_i}{\prod_{\substack{k=0 \\ k \neq j}} (x_j - x_k)} \qquad , \; j = 0, 1, \ldots, n-1$$

2.14.6. Rekursive Routinen

Eine Prozedur oder Funktion kann rekursiv definiert werden. Darunter versteht man, dass im Anweisungsteil der Deklaration (in der Verbundanweisung) die Routine sich selbst aufruft. Prinzipiell ist Rekursion aber auch dadurch möglich, dass eine Routine p1 eine Routine p2, diese eine Routine p3 usw. aufruft, bis eine Routine pn wiederum p1 aufruft. Erfahrungsgemäss werden derartige Rekursionen selten benutzt, weil das Verfolgen der logischen Korrektheit und das Testen der Korrektheit mühevoll sind. Die Sprache gestattet aber solche Konstruktionen; im TURBO-PASCAL-System ist erforderlich, dass die Compilerdirektive {¤A-} kodiert ist (vgl. Abschn. 2.15.).

Beispiel: Fakultät

```
function fak(n:INTEGER) : INTEGER;
begin
if (n=0) or (n=1) then fak := 1
                  else fak := n*fak(n-1);
end.
```

(Diese Funktion hat leider den Nachteil, dass sie für n > 8 einen Wert liefert, der nicht mehr als ganze Zahl darstellbar ist.)
Nehmen wir an, dass die Rekursion nicht so direkt auftritt, sondern, dass eine Funktion f1 eine Funktion f2 aufruft, die ihrerseits die Funktion f1 aktiviert. Dann ergeben sich Notationsschwierigkeiten aus der Tatsache, dass Namen nur benutzt werden dürfen, wenn sie bereits vereinbart wurden. Das ist in dem Falle nicht mehr möglich, denn

```
function f1(...):typ1;
  ...
begin
  ...
     u:= f2(...);
  ...
end;
```

setzt voraus, dass der Funktionsbezeichner f2 existiert. Wollte man aber f2 davor deklarieren, etwa durch

```
function f2(...):typ2;
  ...
begin
  ...
     v:= f1(...);
  ...
end;
```

dann fehlte wiederum die Deklaration von f1.
Dieser Widerspruch wird in PASCAL durch eine sog. FORWARD-Deklaration gelöst. Das bedeutet, dass der gesamte Block einer Funktions- oder Prozedurvereinbarung durch das Basissymbol **forward** ersetzt werden kann. Der Compiler kann aus dem Routinenkopf alle Information über die Parameter (und eventuell über den Funktionstyp) entnehmen. Wird diese Routine aufgerufen, so ist die Parametervermittlung vollständig bekannt, entsprechende Anweisungen können in den Objektkode aufgenommen werden. Natürlich ist es erforderlich, den fehlenden Block, der ja die Algorithmen der Routine enthält, nachzutragen. Das muss auf dem gleichen Blockniveau erfolgen.
In dem oben skizzierten Fall sähe das so aus:

```
function f2(...):typ2; forward;
function f1(...):typ1;

  ...
begin
  ...
     u:= f2(...);
  ...
end;
function f2; {Hier duerfen keine Parameter wiederholt werden.}
  ...
begin
  ...
     v:=  f2(...);
  ...
end;
```

Für Prozeduren ist das völlig entsprechend.
Das folgende Beispiel zeigt solche mehrstufigen Rekursionen.

Beispiel: Zyklensuche
Die Beziehungen in einem gerichteten Graph werden in einer Matrix gespeichert. Eine Eintragung einer 0 in der Komponente a[i,j] besagt, dass vom Knoten i zum Knoten j keine Verbindung existiert. Eine 1 dagegen heisst, dass eine gerichtete Verbindung vom Knoten i zum Knoten j gibt. Wir geben nun ein Programm an, dass alle Zyklen in einem gerichteten Graphen sucht und legen für die Rahmenprozedur Bild 2.13. zugrunde.

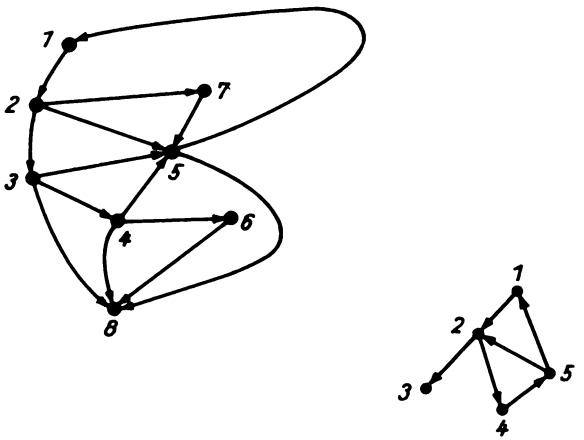


Bild 2.13. Beispielgraph für Zyklensuche

```
{¤A-}
program zyklen_in_graphen ;

const n = 16;
type  t_adj = array[1..n,1..n] of 0..1;
      a1 = array[1..n] of integer;
var   i, j, i1, j1, nn : integer;
      adj : t_adj;
      c : char;

procedure adj_eingabe(var adj:t_adj; var nn:integer);
  var i, j : integer;
  begin
    WRITE('Anzahl der Knoten:'); READLN(nn);
    for i:= 1 to n do
      for j:= 1 to n do adj[i,j]:= 0;
    WRITELN('Wertepaar eingeben');
    REPEAT
    READLN(i1,j1);
    adj[i1,j1]:= 1;
    WRITELN('Weitere Eingaben?  (n/j)');
    READLN(c);
    until c = 'n';
{Kontrolle, ob Matrix korrekt.}
   WRITELN(LST,'Belegung der Adjazenzmatrix'); WRITELN(LST);
   for i := 1 to n do
```

```
    begin for j := 1 to n do
      WRITE(LST,adj[i,j]:2);
      WRITELN(LST);
  end;
   for i := 1 to nn do
     for j := nn+1 to n do
       if adj[i,j] <> 0 then
         begin
           WRITELN(LST,'Bogen von ',i, ' nach ',j, ' falsch');
           halt
         end;
   for i := nn+1 to n do
     for j := 1 to n do
       begin
        if adj[i,j]<>0 then begin
                        WRITELN(LST,'Bogen von ',i, ' nach', j, ' falsch');
                        halt;end;
       end;
end    {adj_eingabe};

procedure result(laenge:integer; v:a1);
  var i : INTEGER;
  begin
    if laenge = 0 then WRITELN(LST,'Knoten gehoert keinem Zyklus an.')
                  else begin
                       for i := 1 to laenge do WRITE(LST,v[i]:4);
                       WRITELN(lst);
                       end;
  end        {result};

procedure zyklensuche(knoten, knotenanzahl : INTEGER;
                      var adj : t_adj);
  var i, k : INTEGER;
      nozyklus : BOOLEAN;
      zyklus : a1;
      vorh : array[1..n] of BOOLEAN;

  procedure zeile(var k:INTEGER);
  forward;{k ist Index im Vektor zyklus, bezeichnet dort den letzten
          verwendeten Knoten}

  procedure abspalten(var k, weiter:INTEGER);
    var i2, lk:INTEGER;
    begin
    lk := zyklus[k];
```

```
  for i2 := weiter to knotenanzahl do
    if lk <> i2 then
      if (adj[lk,i2] = 1) and not vorh[i2]  then
         begin
         vorh[i2] := TRUE;
         k := k+1;
         zyklus[k] := i2;
         zeile(k);
         i2 := n+1;
         end
                                          else
            if (adj[lk,i2] = 1) and (i2 = zyklus[1])
               then
              begin
              result (k,zyklus);end;
        if k> 1 then
        begin
         vorh[zyklus[k]] := FALSE;
         weiter := zyklus[k] +1;
         zyklus[k] := 0;
         k := k-1;
         abspalten(k, weiter);
         end;
end          {abspalten};

procedure zeile;
  var i1, nk, weiter : INTEGER;
  begin
  nk := zyklus[k];
  for i1 := 1 to knotenanzahl do
    if i1 <> nk then
      if (adj[nk,i1] =1) and not  vorh[i1]  then
         begin
         vorh[i1] := TRUE;
         k := k+1;
         zyklus[k] := i1;
         zeile(k);
         i1 := knotenanzahl+1;
         end
                                            else
            if (adj[nk,i1] = 1) and (i1 = zyklus[1]) then
            result(k,zyklus);
      if k > 1 then
         begin
         vorh[zyklus[k]] := FALSE;
```

```
            weiter := zyklus[k] +1;
            zyklus[k] := 0;
            k := k-1;
            abspalten(k, weiter);
            end;
    end          {zeile};

  begin     {Anweisungsteil von Zyklensuche}
  for i := 1 to knotenanzahl do
    begin
    zyklus[i] := 0;
    vorh[i] := FALSE;
    end;
  zyklus[1] := knoten;
  vorh[knoten] := TRUE;
  nozyklus := TRUE;
  i := 1;
  repeat
    if( i <> knoten) and
      (adj[knoten,i] = 1) then
        begin
        nozyklus := FALSE;
        vorh[i] := TRUE;
        zyklus[2] := i;
        k := 2;
        i := knotenanzahl+1;  {zum Verlassen der repeat-Anweisung}
        zeile(k);
        end;
  i := i + 1;
  until i > knotenanzahl;
   k := 0;
  if nozyklus then
              result(k,zyklus);
 end        {zyklensuche};


begin       {Hauptprogramm}
adj_eingabe(adj,nn);
i := 1 ;
repeat
  WRITELN(lst,'Zyklensuche mit Knoten',i:3,':');
  zyklensuche(i, nn, adj);
  i := i + 1;
  until i > nn;
end.
```

Es werden folgende Daten eingegeben:

```
8
1  2
2  3
2  5
2  7
3  4
3  5
3  8
4  5
4  6
4  8
5  1
6  8
7  5
```

Es folgt die Ausgabe der Ergebnisse:

Belegung der Adjazenzmatrix

```
 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0
 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0
 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0
 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
Zyklensuche mit Knoten  1:
   1   2   3   4   5
   1   2   3   5
   1   2   5
   1   2   7   5

Zyklensuche mit Knoten  2:
```

```
   2   3   4   5   1
   2   3   5   1
   2   5   1
   2   7   5   1

Zyklensuche mit Knoten  3:
   3   4   5   1   2
   3   5   1   2

Zyklensuche mit Knoten  4:
   4   5   1   2   3

Zyklensuche mit Knoten  5:
   5   1   2   3   4
   5   1   2   3
   5   1   2
   5   1   2   7

Zyklensuche mit Knoten  6:
Knoten gehoert keinem Zyklus an.

Zyklensuche mit Knoten  7:
   7   5   1   2

Zyklensuche mit Knoten  8:
Knoten gehoert keinem Zyklus an.
```

Es sei darauf hingewiesen, dass in rekursiven Routinen nicht lokale Werte auf der Position von Variablenparametern benutzt werden dürfen (vgl. Anhang F).

2.14.7. Überlagerungstechnik und externe Prozeduren

Es kann vorkommen, dass der vorhandene Hauptspeicherplatz nicht ausreicht, um ein Objektprogramm (.COM-File) vollständig zu laden. Diese Schwierigkeiten könnten dadurch behoben werden, dass abarbeitungsfähige Programmstücke während der Arbeit des Programms nachgeladen werden. Das wirft natürlich die Frage auf, wohin und wie nachgeladen wird. In diesem Abschnitt werden Sprachelemente für ein solches Nachladen bzw. für das Aktivieren von abgearbeiteten Programmkode behandelt.

2.14.7.1. EXECUTE und CHAIN

Im Abschnitt 1.1. wurden die Compileroptionen erklärt. Wird in dem O-Menü C eingegeben, so wird das erzeugte File auf der Diskette abgelegt und erhält den Namen des Quelltextfiles, worin jedoch der Namenszusatz .PAS durch .COM ersetzt worden ist. Dieses .COM-File enthält den Objektkode des Programms und die Laufzeitbibliothek des TURBO-PASCAL-System. Es kann vom Niveau des TURBO-PASCAL-System mit dem Kommando EXECUTE gestartet werden. Innerhalb der Programmiersprache TURBO-PASCAL gibt es die Standardprozedur EXECUTE. Diese Prozedur kann durch

EXECUTE(filevariable)

aufgerufen werden, der Aufruf ist aber nicht beim Arbeiten im Hauptspeichermodus erlaubt(Compileroption M). Der Aufruf hat zur Folge, dass das File - Zuweisung eines .COM-Files und Eröffnen des Files vorraussetzt - in den Hauptspeicher geladen wird, dort das rufende Programm überschreibt und gestartet wird. Ein Rücksprung ins aufrufende Programm wird nicht ausgeführt. Ebenso gibt es keine Vorkehrungen für eine Parametervermittlung.
Sollen Parameter des aufrufenden Programms an das mit EXECUTE aktivierte Programm übergeben werden, so sind zwei Wege gangbar.
1. Die zu übergebenden Variablen werden in beiden Programmen in gleicher Reihenfolge deklariert, so dass ihnen bei der Compilation gleiche Speicheradressen zugewiesen würden. Da das TURBO-PASCAL-System nicht automatisch die Variablen initialisiert, können sie von dem gerufenen Programm verwendet werden. Ein mit EXECUTE gerufenes Programm kann seinerseits völlig entsprechend wiederum ein .COM-File aufrufen. Auf dem Speicherplatz mit der Adresse ¤80 befindet sich Information, ob das gerade aktive Programm vom System direkt gestartet wurde (Wert des Bytes <=127) oder mit EXECUTE (Wert des Bytes >127).
2. Parameter können mit Hilfe absolut adressierter Variablen vermittelt werden (vgl. Abschn. 2.13.3.). Dazu ist jedoch erforderlich, dass bei der Programmniederschrift Klarheit über die Speicherbereiche herrscht, die nicht anderweitig verwendet werden. Diese Information kann bei der Compilation bereitgestellt werden. Wird nämlich mit der Compileroption C oder H übersetzt, so erscheinen auf dem Bildschirm folgende Zeilen:

```
Start  address :XXXX (min YYYY)
End    address :XXXX (max YYYY)
```

Die hexadezimale Startadresse ist die Adresse des ersten Bytes des

Objektkodes, davor ist die Laufzeitbibliothek gespeichert. Entsprechend ist die Endadresse zu interpretieren, die maximale Adresse ist die Anfangsadresse des BDOS minus 1.
Es ist nun möglich, durch Eingabe von S oder/und E diese Adresse zu verändern. Sinnvoll ist es, die Startadresse zu erhöhen, um den Speicherbereich zwischen der Laufzeitbibliothek und der aktuellen Startadresse für absolut adressierte Variablen zu benutzen, d.h., über diesen Bereich die Parametervermittlung zu organisieren.

Ebenso ist mit der Startprozedur

CHAIN(filevariable)

der Aufruf eines weiteren TURBO-PASCAL-Programms möglich. Allerdings muss das dem File-Namen filevariable zugewiesene Programm mit der Compileroption H übersetzt worden sein. Der Objektkode so übersetzter Programme hat den Namenszusatz .CHN . Das Nachladen von .CHN-Files unterscheidet sich nur dadurch, dass das Laufzeitsystem des aufrufenden Programms verwendet wird.

2.14.7.2. Überlagerung von Prozeduren und Funktionen

Das TURBO-PASCAL-System schliesst ein System zur Erzeugung von Überlagerungsstrukturen ein. Prozedur- und Funktionsdeklarationen können durch das Voranstellen des Basissymbols **overlay** gekennzeichnet werden. Im Quellprogramm direkt aufeinanderfolgende Überlagerungsroutinen werden als eine Gruppe behandelt. Der Compiler übersetzt diese Routinen, erzeugt ein File und hat Informationen über den Speicherbedarf für die Routine maximaler Länge. Dieser Speicherplatz wird in dem Hauptprogramm reserviert. (Eine Prozedur als Überlagerungsprozedur zu benutzen, ist also sinnlos.) Jede der Routinen einer solchen Überlagerungsgruppe wird an den Anfang des reservierten Überlagerungsbereichs geladen. Daraus folgt, dass sich Routinen der gleichen Überlagerungsgruppe nicht aufrufen können. Im Quelltext werden Überlagerungsgruppen durch Nicht-Überlagerungsroutinen getrennt. Eventuell müssen zur Trennung leere Prozeduren verwendet werden. Die Organisation wird im Bild 2.14. veranschaulicht.

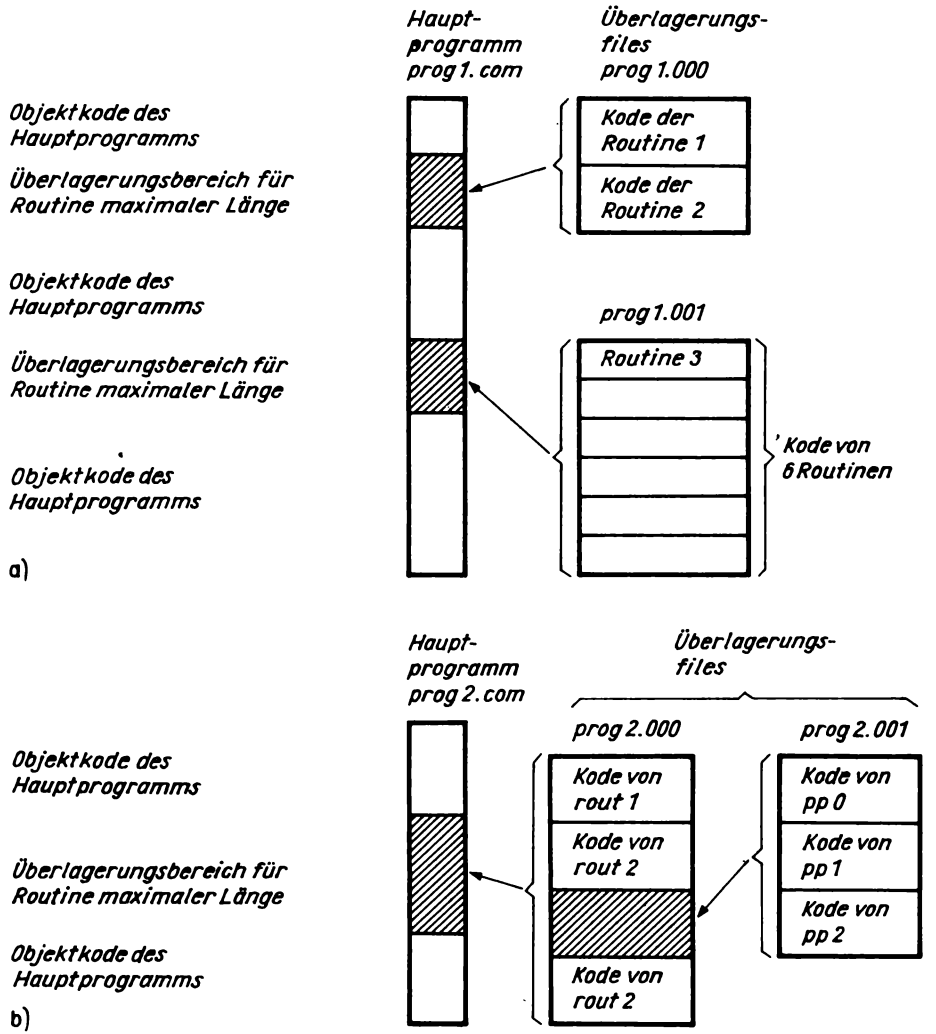


Bild 2.14. Überlagerung von Routinen
a) einfach, b)verschachtelt

Die Struktur des Bildes 2.14b entspricht folgender Programmorganisation:

```
program prog2;
```

```
overlay procedure rout1(...);                          ┐
                                                       │
overlay procedure rout2(...);                          │
                                                       │
          procedure pp(...);                           │
             ...                                       │
          overlay procedure pp0(...);   ┐              ├ Gruppe 1
             ...                        │              │
          overlay procedure pp1(...);   ├ Gruppe 2     │
             ...                        │              │
          overlay procedure pp2(...);   ┘              │
             ...                                       │
  ...; {Ende von rout2}                                ┘

begin

   .
end.
```

Die in einer Überlagerungsgruppe zusammengefassten Routinen werden als ein File auf dem Laufwerk gespeichert, auf dem das Objektprogramm abgelegt ist. Sie erhalten den File-Namen des Objektprogramms, wobei der Namenszusatz durch fortlaufende Numerierung .000, .001 usw. bestimmt wird.
Bild 2.14b zeigt, dass auch die Verschachtelung von Überlagerungsroutinen erlaubt ist, die Numerierung des Files entspricht auch dann der Reihenfolge der auftretenden Überlagerungsgruppen. Da der Überlagerungsbereich so gross sein muss, dass er den Kode der grössten Routine einer Überlagerungsgruppe aufnehmen kann, bleibt beim Nachladen kleiner Routinen ein Teil des Überlagerungsbereichs ungenutzt. Effektive Organisation liegt in der Hand des Programmierers.
Beim Auftreten von Laufzeitfehlern in einer aktivierten Überlagerungsroutine ist die Fehlersuche kompliziert. Prinzipiell kann im O-Menü durch Eingabe von F (Find run-time error) der Compiler veranlasst werden, Massnahmen für Fehleranalyse zu generieren und beim Auftreten eines Fehlers durch eine Ausschrift der Form (vgl. Anhang E)

```
Run-time error xx  PC = XXXX
Program aborted
```

die Fehlersuche zu unterstützen. Wird mit

```
Enter PC : XXXX
```

die Abbruchstelle aus obiger Mitteilung eingegeben, so wird der dazugehörige Quelltext zur Korrektur auf dem Bildschirm angeboten. Tritt der Fehler aber im Überlagerungsbereich auf, so kann nicht festgestellt werden, welche Routine der Überlagerungsgruppe aktiv ist; die Quelltextbereitstellung ist nicht möglich. Es ist deshalb zu empfehlen, Routinen erst dann mit dem Attribut **overlay** zu versehen, wenn sie hinreichend getestet sind.

2.14.7.3. Externe Unterprogramme

Externe Unterprogramme sind Programmbausteine, deren Kode unabhängig von der Compilation des Hauptprogramms erzeugt wurde. Meist handelt es sich um Prozeduren oder Funktionen in Maschinenkode (vgl. auch Abschn. 2.14.8.). Formal bestehen externe Unterprogramme aus einem Prozedur- oder Funktionskopf, dem statt des Blocks das Basissymbol **external** und eine ganzzahlige Konstante folgen.

Beispiel:

```
procedure p0(x,y : INTEGER); external ¤9000;
```

Beim Aufruf wird zu dieser Stelle verzweigt. Die Parameter werden im Kellerspeicher übergeben und müssen von dem aktiven externen Unterprogramm aus dem Keller übernommen werden, Ergebnisparameter werden ebenfalls über den Kellerspeicher vermittelt. Die Arbeit mit dem Kellerspeicher erfolgt über PUSH- bzw.POP-Befehle des Maschinenkodes. Für die Nutzung externer Unterprogramme ist also (u.a.) erforderlich zu wissen, wie die Parameter beim Aufruf im Keller hinterlegt werden. Dafür gelten folgende Festlegungen :

1. Werte von Aufzählungstypen belegen im Kellerspeicher ein Wort (zwei Bytes). Das höherwertige Byte ist mit 0 belegt, wenn es sich um Variablen handelt, die nur ein Byte benötigen.
2. REAL-Werte belegen 6 Bytes im Kellerspeicher entsprechend der Darstellung von Gleitkommazahlen (vgl. /6/).
3. Zeigerwerte enthalten die Adresse der dynamischen Variablen.
4. Von Feldern und Records wird die Adresse des ersten Bytes im Wortformat übergeben.
5. Werte von einem String-Typ werden so übergeben, dass an oberster

Stelle die Länge des Strings steht (ein Byte). Die folgenden Plätze des Kellerspeichers enthalten die Zeichen des Strings.
6. Set-Variablen werden in 32 Bytes im Kellerspeicher übergeben.

Neben der Schwierigkeit, mit der Parametervermittlung detailliert vertraut zu sein, gibt es eine weitere, nämlich, das externe Unterprogramm auf den Speicherbereich zu schreiben, der in der Deklaration angegeben wird. Auf verschiedene Möglichkeiten wird im Abschn. 2.14.8. eingegangen. Hier soll nur ein mögliches Vorgehen angedeutet werden: Man liest den Maschinenkode des externen Unterprogramms mit der BLOCKREAD-Prozedur ein und weist ihn dabei einer Variablen als Wert zu, die mit dem Attribut **absolute** an den Speicherplatz gebunden wurde, der auch in der Deklaration des externen Unterprogramms als Speicheradresse angegeben ist. Es muss sichergestellt sein, dass der reservierte Speicherplatz für die Aufnahme des Maschinenkodes ausreicht.

2.14.8. Prozeduren mit Maschinenkode

2.14.8.1. Warum Maschinenkode?

Die Programmiersprache PASCAL bietet sprachliche Mittel, mit denen man eigentlich alle Algorithmen ausdrücken können sollte. Darüber hinaus bietet TURBO-PASCAL mit dem MEM- und dem PORT-Feld Zugriff zu den wichtigsten Hardwareteilen des Rechners. Wozu benötigt man also die Möglichkeit, in PASCAL-Programme Kodestücke in Maschinensprache einzufügen? Dafür gibt es zwei vernünftige Gründe:

1. Zur Erhöhung der Effektivität und
2. zur Formulierung spezieller Abläufe.

Wenn ein TURBO-PASCAL-Programm ein Laufzeitverhalten hat, das den Ansprüchen nicht gerecht wird, also zu langsam arbeitet, dann sollte man zuerst überlegen, ob die Wahl eines anderen Algorithmus das Programm nicht schneller macht. Meist ist so ein entscheidender Effektivitätsgewinn zu erreichen. Erst wenn diese Möglichkeiten alle ausgeschöpft sind, kann man versuchen - nach gründlicher Abwägung von Aufwand und Nutzen -, durch Umsetzung bestimmter Programmteile von TURBO-PASCAL in Maschinensprache das Laufzeitverhalten zu verbessern. Dazu muss zuerst ermittelt werden, in welchen Teilen das Programm den wesentlichen Teil der Zeit verbringt. Nur durch Umkodierung dieser Teile kann eine Verbesserung erwartet werden. Das sollte man sich immer vor Augen halten, denn auch ein noch so aufwendiges Umprogrammieren in Maschinenkode verbessert die

Gesamtlaufzeit unwesentlich, wenn nur Programmteile, die unbedeutend zur Laufzeit beitragen, berührt werden.

Aber effektives Arbeiten besteht nicht nur im Entwickeln von Programmen mit gutem Laufzeitverhalten, sondern auch darin, die Programme schnell zu entwickeln. Das erreicht man durch Verwendung von Maschinenkode, was allerdings gute Kenntnisse in maschinennaher Programmierung voraussetzt. Gewisse Operationen lassen sich in Maschinensprache viel kürzer und durchsichtiger aufschreiben als in TURBO-PASCAL, wie das folgende Beispiel zeigt. Es soll ein 16-Bit-Wort "kode" zyklisch um ein Bit nach links verschoben werden (Bild 2.15.).

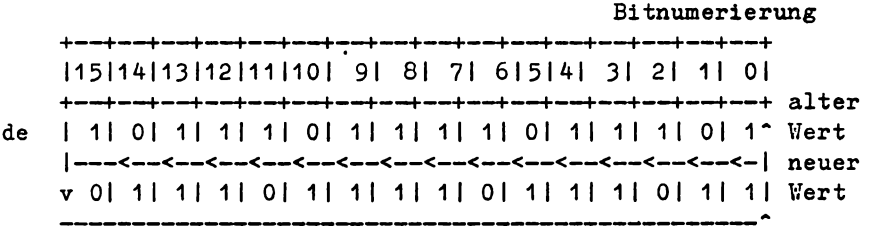


Bild 2.15. Skizze zum Rotationsproblem

Das Programmstück in Maschinensprache könnte wie folgt aussehen:

```
    ld   hl,(kode)
    add  hl,hl
    jr   nc,m1
    inc  hl
m1: ld   (kode),hl
```

TURBO-PASCAL-Äquivalente dazu sind

```
   if (kode >= 0) and (kode < $4000) then
     kode := kode+kode
   else if (kode >= 0) then
     kode := $8000+(kode-$4000)+(kode-$4000)
   else if (kode >= $c000) then
     kode := $8000+(kode+$4000)+(kode+$4000)+1
   else
     kode := (1+kode+$7fff)+(1+kode+$7fff)+1;
und
```

```
neg := kode < 0;
kode := kode shl 1;
if  neg then kode := kode or 1;
```

Man erkennt an der ersten Form, dass das TURBO-PASCAL-Programm viel mehr Uberlegung erfordert, um alle Fälle, die zur Bereichsüberschreitung des INTEGER-Types führen würden, auszuschalten, als dies bei Verwendung der Maschinensprache notwendig ist.
Dem Leser wird der Trockentest obiger Programmzeilen empfohlen.)
Der zweite Grund, weshalb es manchmal notwendig wird, Maschinenkodeteile in TURBO-PASCAL-Programme einzufügen, ist, dass sich bestimmte Aktionen nicht anders ausdrücken lassen. Dafür einige Beispiele:

- Es ist möglich, in einem TURBO-PASCAL-Programm die Bedienung eines Interruptes einer Prozedur zu übertragen. Dazu wird die Adresse der Prozedur in den Interruptvektor eingetragen. Die Prozedur muss dann aber sichern, dass alle Register beim Verlassen der Routine den gleichen Inhalt haben, wie bei ihrem Aufruf. Die dafür notwendigen Anweisungen zum Retten und Wieder herstellen der Registerinhalte lassen sich jedoch nicht in TURBO-PASCAL formulieren, sondern nur in Maschinensprache. Ebenso kann die spezielle Anweisung "RETI" zum Verlassen der Interruptroutine nur in Maschinensprache formuliert werden.
- In Prozeduren, in denen Variable geändert werden sollen, die auch in Interruptroutinen beeinflusst werden, ist es notwendig, Interrupts zeitweilig zu verbieten. Dafür bietet PASCAL keine Möglichkeit.
- Um bestimmte Hardwarefunktionen mancher Rechner zu erreichen, ist die Abarbeitung ganz bestimmter Maschinenbefehle notwendig, die TURBO-PASCAL nicht erzeugen würde. So gibt es zum Beispiel beim U880- bzw.Z80-Prozessor zwei Arten von Ausgabebefehlen; einen mit der Portadresse im Befehl selbst und weitere, bei denen die Portadresse im Register C stehen muss. Mit letzteren werden zusätzliche acht Bit Information aus dem B-Register der Peripherie zur Verfügung gestellt, die diese auswerten kann. Zum Beispiel könnte das Setzen eines Punktes in einer Grafikdisplay-Ansteuerung wie folgt gelöst sein:

```
ld  a,x     ; A = x-Koordinate des Punktes
ld  b,y     ; B = y-Koordinate des Punktes
ld  c,pon   ; C = Portadresse zum Setzen eines Punktes
out (c),a   ; Setzen des Punktes (x,y).
```

Nachdem dargelegt wurde, wozu Maschinenkode in Programmen manchmal benötigt wird, soll gezeigt werden, wie in TURBO-PASCAL die Einbindung von Kodeteilen in Maschinensprache erfolgt.

2.14.8.2. Die INLINE-Anweisung

TURBO-PASCAL übersetzt Programme in direkten Maschinenkode. Es bietet sich daher an, an den Stellen im Programm, an denen die Einfügung von Maschinenkode notwendig ist, diesen direkt an Ort und Stelle einzufügen. Dazu dient die INLINE-Anweisung. Sie besteht aus dem Basissymbol **inline** und einer in runden Klammern eingeschlossenen Liste von Bytes und Worten. Die Bytes und Worte werden durch Schrägstrich "/" voneinander getrennt. Die Liste kann auf mehreren Programmzeilen verteilt sein und darf auch Kommentare enthalten, die in Kommentarklammern einzuschliessen sind. Zum Beispiel:

Beispiele:

```
  inline ( ¤F3 );
  inline ( ¤F5 / ¤C5 / ¤D5 / ¤E5 );
  inline ( ¤21 / ¤FF00 /     (* ld  hl,0ff00h *)
           ¤11 / ¤FE00 /     (* ld  de,0fe00h *)
           ¤01 / ¤100  /     (* ld  bc,100h   *)
           ¤ED / ¤B0 );      (* ldir          *)
```

Bei der Abarbeitung der INLINE-Anweisung werden die Bytes und Worte als Maschinenbefehle behandelt und ausgeführt. Man beachte, dass bei Worten zuerst der niederwertige Teil, dann der höherwertige Teil abgelegt wird. Die Anweisung

```
  inline ( ¤3E26 );
```

stellt also nicht den Befehl "ld a,26h", sondern den Befehl "ld h,3eh" dar.

Ob ein Wert in der INLINE-Anweisung ein Byte oder ein Wort ist, d. h., ob er ein oder zwei Zeichen Kode liefert, hängt von der Grösse des Wertes ab. Werte im Bereich 0 bis 255 werden als Byte behandelt und liefern ein Zeichen; alle anderen sind Worte und ergeben zwei Zeichen Kode.
Es gibt allerdings die Möglichkeit, selbst festzulegen, wieviel Zeichen Kode aus einem Wert in der INLINE-Anweisung erzeugt werden soll. Wird dem Wert ein Kleinerzeichen '<' vorangestellt, so wird genau ein Zeichen erzeugt. Bei Werten grösser als 255 werden die niederwertigen acht Bits des Wertes verwendet. Ein vorangestelltes Grösserzeichen '>' bewirkt die Ausgabe von zwei Zeichen.

Beispiele:

```
inline ( ¤21 / >5 );          (* ld hl,5           *)
inline ( ¤3E / <intproz );    (* ld a,low intproz  *)
```

In der INLINE-Anweisung können nicht nur Direktwerte stehen, sondern auch Bezeichner des TURBO-PASCAL-Programms. In der INLINE-Anweisung repräsentieren sie die Adresse des Objektes, welches der Bezeichner benennt (bei Variablen, Prozeduren und Funktionen) oder den Wert (bei deklarierten Konstanten). Damit ist es möglich, aus der INLINE-Anweisung heraus Variablenwerte zu benutzen und zu verändern, sowie Prozeduren und Funktionen aufzurufen.

Beispiele:

```
inline ( ¤2A / kode );        (* ld hl,(kode)     *)
inline ( ¤CD / kopf );        (* call kopf         *)
```

Ausser Direktwerten und Bezeichnern kann man noch das Zeichen "*" als Wert benutzen. Es repräsentiert die Adresse, auf welcher der Wert nach der Compilation im Speicher steht. Man beachte dabei den Unterschied zur der in Assemblersprachen üblichen Bedeutung, wo es die Adresse des Befehles bedeutet, in dem das Zeichen "*" verwendet wird.

Beispiel:

```
inline ( c3 / *+4 /           (*  jp m1          *)
         01 / 02 /            (*  db 1,2         *)
         ...       );         (* m1:             *)
```

2.14.8.3. Erzeugung von INLINE-Anweisungen

Um INLINE-Anweisungen für TURBO-PASCAL-Programme zu erzeugen, kann man sich mehrerer Methoden bedienen. In diesem Abschnitt werden davon folgende vorgestellt und besprochen:

- Kodieren der INLINE-Anweisungen von Hand
- Erzeugen der INLINE-Anweisungen mit Hilfe eines speziellen Assemblers aus Quelltext in der Sprache dieses Assemblers
- Erzeugen der INLINE-Anweisung mit Hilfe eines speziellen Linkers aus dem mit dem Assembler M80 gewonnenen Objektkode
- Erzeugen der INLINE-Anweisung mit Hilfe eines speziellen Programmes aus der mit dem Assembler M80 gewonnenen Druckdatei

Kodieren von Hand

Beim Kodieren von Hand werden die Befehlskodes der gewünschten Maschinenanweisungen selbst in den TURBO-PASCAL-Quelltext eingetragen. Man wird sich dazu in der Regel einer Befehlsliste bedienen, aus der man die Kodeierung einzelner Befehle entnehmen kann. Diese Methode ist nur für sehr kurze Programmstücke zu empfehlen, denn sie birgt eine Reihe Fehlermöglichkeiten. Die häufigsten Fehler, die begangen werden, sind:

- Schreibfehler, beim Eingeben der Befehlskodes; z. B. statt ¤ED / ¤B0 wird ¤EB / ¤D0 eingegeben
- Verwechseln von Befehlen mit ähnlichen Befehlen beim Suchen in der Befehlsliste; z. B. Verwechslung von "ld hl,nnnn" mit "ld hl,(nnnn)"
- Nichtbeachtung der Regeln für Bytes und Worte in INLINE-Anweisungen; z. B.

```
inline ( ¤CD / 5 )      statt   inline ( ¤CD / >5 )
inline ( ¤3E / intp )   statt   inline ( ¤3E / <intp)
```

Alle diese Fehler führen dazu, dass die INLINE-Anweisung bei der Abarbeitung anders als beabsichtigt ausgeführt wird. Unvorhersagbare Reaktionen des Programmes sind die Folge.
Schreibfehler und Verwechseln von Befehlen kann man durch Einführung und Verwendung von privaten Mnemoniken vermeiden. Man definiert sich dazu in einer Konstantenvereinbarung des PASCAL-Programms Bezeichner für die benötigten Maschinenbefehle und verwendet dann diese Bezeichner in der INLINE-Anweisung.

Beispiel:

```
const
    ldir   =  ¤B0ED;  {gespeichert als EDB0}
    ldab   =  ¤78;
    jp     =  ¤C3;
    jrnz   =  ¤20;
    ...

inline ( ldir / ldab / jp / >5 );
```

Legt man die Konstantendefinitionen der privaten Mnemoniken in einer separaten Datei ab, die bei Bedarf mit dem INCLUDE-Mechanismus (I-Direktive, vgl. Abschn. 2.15.) des TURBO-PASCAL-Systems in das Programm eingefügt wird, so hat man Schreibfehler und Verwechslungen fast ausgeschlossen.

Der INLASS-Assembler

Der INLASS-Assembler ist ein Programmierwerkzeug, welches aus einer Datei mit Assemblerquelltext eine Datei mit einer INLINE-Anweisung erzeugt.

Die Notierung der Maschinenbefehle im INLASS-Quelltext ist identisch mit der für den am häufigsten benutzten Assembler M80, so dass für die meisten Programmierer kein zusätzlicher Lernaufwand entsteht. Einige Pseudoanweisungen des M80 sind entfallen, da sie für INLASS nicht benötigt werden (z. B. cseg, dseg, .phase, .dephase u. ä.). Einige Operatoren fehlen in INLASS (z. B. low und high). Deklarationen externer Symbole (nämlich der in der INLINE-Anweisung verwendeten Bezeichner aus dem TURBO-PASCAL-Quellprogramm) erfolgen für INLASS anders als für M80:

```
INLASS:                    M80:
Seite  ext                   extrn Seite,Zeile
Zeile  ext
```

Soll ein grösseres Programmstück in Maschinensprache in ein TURBO-PASCAL-Programm eingefügt werden, so kann man mit dem TURBO-Editor den Quelltext erstellen und in einer Datei mit der Namenserweiterung .SOU ablegen. Aus dem Grundmenü des TURBO-PASCAL-Systems wird dann der INLASS-Assembler aufgerufen. Dieser erzeugt eine Datei mit einer INLINE-Anweisung. Diese Datei mit der Namenserweiterung .INL kann dann nach Rückkehr in das TURBO-System mit dem Editorkommando ^Kr in den Quelltext eingefügt werden, oder sie kann mit dem INCLUDE-Mechanismus (Compilerdirektive i) bei der Übersetzung in das Programm eingefügt werden. Letzteres ist für die Programmentwicklung günstiger, da nach Veränderung am Assemblerquelltext nur INLASS aufgerufen werden muss und ohne Änderung am Quelltext sofort neu übersetzt werden kann.
Der Quelltext des Programmes INLASS sowie eine ausführliche Anleitung findet man in /9/.

Beispiel: INLASS-Quelltext:

```
pasbez  ext

bdos    equ     5
funf    equ     5
```

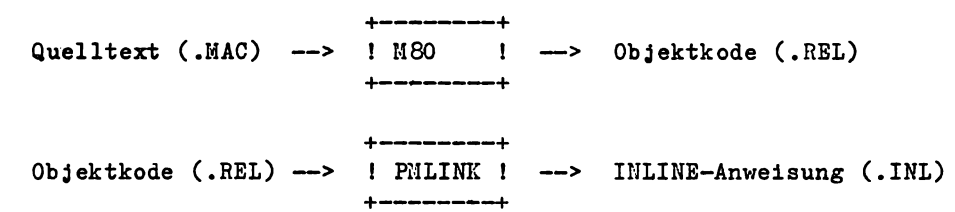
```
tausend equ     1000

        ld      hl,funf
        call    pasbez
        call    bdos
        ld      a,(tausend)
        end
```

Von INLASS erzeugte INLINE-Anweisung:

```
INLINE(
                        (* BDOS     EQU 5                *)
                        (* FUNF     EQU 5                *)
                        (* TAUSEND  EQU 1000             *)
                        (* PASBEZ   EXT                  *)
¤21/¤05/¤00             (*          LD HL,FUNF           *)
/¤CD/PASBEZ             (*          CALL PASBEZ          *)
/¤CD/¤05/¤00            (*          CALL BDOS            *)
/¤3A/¤E8/¤03            (*          LD A,(TAUSEND)       *)
                        (*          END                  *)
)
```

Der PMLINK-Linker

PMLINK ist ein Programmierwerkzeug, welches aus einer Objektdatei im M80-Format eine INLINE-Anweisung erzeugt.



PMLINK verarbeitet den vom Assembler M80 erzeugten Objektkode vollständig. Ab Version 3.4 verfügt M80 über zusätzliche Möglichkeiten (Verwendung von externen Daten und verschieblichen Werten der Länge 8 Bit). Diese können von PMLINK nicht verarbeitet werden und führen zu einer Fehlermeldung.

Beispiel: M80-Quelltext:

```
funf    equ     5
tausend equ     1000
```

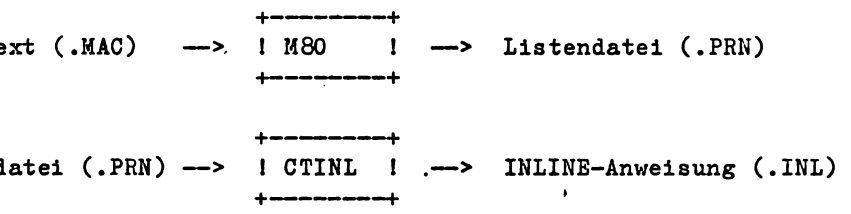
```
        extrn   pasbez
        ld      hl,funf
        call    pasbez
        ld      a,low tausend
        end
```

Von PMLINK erzeugte INLINE-Anweisung:

```
  begin (* Modul TPM *)
    inline (
    (*0000*)    ¤21/¤05/¤00/¤CD/PASBEZ /¤3E/¤E8)
  end;  (* TPM *)
```

Das Werkzeug CTINL

CTINL ist ein Programmierwerkzeug, welches aus einer M80-Assemblerliste eine INLINE-Anweisung erzeugt.



CTINL verarbeitet im Gegensatz zu PMLINK externe Werte der Länge 8 Bit korrekt.

Beispiel: M80-Quelltext:

```
        extrn   pasbez
        extrn   iport

funf    equ     5
tausend equ     1000

        ld      hl,funf
        call    pasbez
        ld      a,low pasbez
        ld      a,low tausend
        in      a,(iport)
        end
```

M80-Listendatei:

```
        MACRO-80                PAGE    1
                                        extrn   pasbez
                                        extrn   iport

  0005                          funf    equ     5
  03E8                          tausend equ     1000

  0000'   21 0005                       ld      hl,funf
  0003'   CD 0000*                      call    pasbez
  0006'   3E 00*                        ld      a,low pasbez
  0008'   3E E8                         ld      a,low tausend
  000A'   DB 00*                        in      a,(iport)
                                        end

        MACRO-80                PAGE    S

Macros:
Symbols:
0005    FUNF            0000*   IPORT           0004*   PASBEZ
03E8    TAUSEND

No Fatal error(s)
```

Von CTINL erzeugte INLINE-Anweisung:

```
INLINE  (
¤21/¤05/¤00/      (*        ld      hl,funf         *)
¤CD/ > PASBEZ/    (*        call    pasbez          *)
¤3E/ < PASBEZ/    (*        ld      a,low pasbez    *)
¤3E/¤E8/          (*        ld      a,low tausend   *)
¤DB/ < IPORT/     (*        in      a,(iport)       *)
¤00 );
```

2.14.8.4. Nachladen von Maschinenkode

Es gibt noch weitere Möglichkeiten, Programmteile in Maschinenkode in ein TURBO-PASCAL-Programm einzubeziehen. Eine davon ist das Nachladen von Prozeduren und Funktionen, die in Maschinensprache geschrieben sind. Dazu kann man sich der Eigenschaft des TURBO-PASCAL-Systems bedienen, die Startadresse des Programmes im Optionsmenü verändern zu können. Diese ist für TURBO-PASCAL (Version

3.0) standardmässig die Adresse 20E2H. Vergrössert man sie auf z. B. 3000H, so ist der Bereich 20E2H bis 2FFFH frei für Programmteile in Maschinensprache. Mit einem beliebigen Assembler kann man nun Unterprogramme erstellen, die in dem von TURBO-PASCAL-System freigehaltenen Speicherbereich ablauffähig sind. Anschliessend muss dafür gesorgt werden, das die Programmteile zur Abarbeitungszeit des Programms auch im Speicher stehen und dass der TURBO-PASCAL-Compiler weiss, wo sie stehen. Das erste kann auf zwei Arten gelöst werden. Man kan vom TURBO-PASCAL-Programm eine .COM-Datei erzeugen und in diese dann mit Hilfe eines geeigneten Testhilfeprogramms, wie z. B. ZSID, das Assemblerprogrammstück in den vom Compiler erzeugten Maschinenkode einfügen. Dies ist jedoch besonders in der Phase des Programmtests ein mühsames Unterfangen. Leichter ist es, das Assemblerprogrammstück vom TURBO-PASCAL-Programm selbst nach dem Programmstart in den Speicher laden zu lassen. Dazu dient die Funtion "loadprog", die beim Aufruf den Dateinamen der zu ladenden Datei erhält und die einen INTEGER-Wert zurückliefert, der aussagt, ob die Datei geladen werden konnte. Das Programm "loaddemo" demonstriert die Verwendung dieser Funktion.

Beispiel:

```
{Loaddemo.pas}
(* Dieses Programm demonstriert den Gebrauch der Funktion
   "loadprog" zum Nachladen von Assemblerunterprogrammen  *)

program loaddemo;
type fname = string[20];
var  ok : BOOLEAN;

procedure up1;          (* Deklaration der 1. nachzuladenden *)
  external ¤2100;       (* Prozedur *)
procedure up2;          (* Deklaration der 2. nachzuladenden *)
  external ¤2103;       (* Prozedur *)

(*¤Iloadprog.inc*)

begin
  ok := FALSE;
  case loadprog('test.com') of
    0 : ok := TRUE;
    1 : WRITELN('File ist zu lang, um geladen werden zu koennen');
    -1: WRITELN('File existiert nicht')
  end;
  if ok then begin
    up1;
```

```
    up2
            end
end.

function loadprog(f:fname):INTEGER;
const fcb = ¤5c;
var t : file;
    noeof : BOOLEAN;
    adr : INTEGER;
    ende: INTEGER absolute ¤101;
begin
  ASSIGN(t,f);
  if BDOS(15,fcb) = 255 then
    loadprog := -1
  else begin
    adr := ¤2100;
    noeof:=TRUE;
    while (adr+128<=ende) and noeof do
   begin
      BDOS(26,adr);
      noeof := BDOS(20,fcb)<>1;
      adr := adr+128
    end;
    if noeof then
      loadprog:=1
    else
      loadprog:=0
  end
end;
```

2.15. Compilerdirektiven

Der Compiler bietet gewisse Möglichkeiten, die beim Programmieren genutzt werden können oder nicht. Im Programm können darüber Festlegungen getroffen werden durch sogenannte Pseudokommentare. Ein Pseudokommentar besteht aus den einschliessenden geschweiften Klammern, wobei der öffnenden Klammer unmittelbar das Zeichen ¤ folgt. Dem ¤-Zeichen folgt eine Liste von Direktiven, die jeweils aus einem Buchstaben und einem Plus- oder Minuszeichen bzw. einer natürlichen Zahl besteht. Mehrere Direktiven werden durch Kommas getrennt.

Beispiel: {¤A+,U-}

Die Compilerdirektiven - mit Ausnahme der B- und C-Direktive - sind lokal in dem File.
Für alle Compilerdirektiven sind Standardwerte festgelegt unter dem Gesichtspunkt, schnelle und speicherplatzeffektive Programme zu erzeugen.
Im weiteren werden die Compilerdirektiven erläutert, die im TURBO-PASCAL-System für CP/M-80 zur Verfügung stehen.

A Absoluter Objektkode
Standardwert: A+
Durch die A-Direktive wird die Erzeugung von absolutem Maschinenkode gesteuert. Wird die A-Direktive durch {¤A-} ausgeschaltet, so sind rekursive Aufrufe im Maschinenprogramm gestattet.

B Auswahl des Einabe/Ausgabe-Modus
Standardwert: B+
Standardmässig wird die Konsole, also das durch CON: bezeichnete Gerät (gepufferte Eingabe/Ausgabe), für die Standardfiles OUTPUT und INPUT verwendet. {¤B-} aktiviert statt dessen das logische Gerät TRM:.
Die B-Direktive gilt für das vollständige Programm, sie kann in einem Programm nicht neu gesetzt werden.

C Programmunterbrechung bei Datenanforderung
Standardwert: C+
Bei Datenanforderung an der Konsole kann ^C eingetastet werden. Dieses Steuerzeichen führt dann zum Abbruch des Programms. Die Direktive {¤C-} verhindert das Interpretieren von Steuerzeichen. Ausser ^C ist auch die Eingabe von ^S möglich mit der Wirkung, dass die Bildschirmausgabe aus- bzw. eingeschaltet wird.
Die C-Direktive kann innerhalb des Programms nicht verändert werden.

I Behandlung von Eingabe/Ausgabe-Fehlern
Standardwert: I+
Standardmässig werden alle mit Programmaktivitäten, die Eingabe/Ausgabe betreffen, auf korrekte Ausführung geprüft und führen im Fehlerfall zu Programmabbruch und Fehlermeldung. Durch {¤I-} kann der Programmierer solchen Abbruch unterbinden. Das System hinterlegt eine Nachricht in der vordefinierten Variablen IORESULT;

die Auswertung ihres Wertes kann zur Steuerung der Programmfortsetzung benutzt werden.
Beispiel: Im Programm wird die Eingabe eines Filenamens verlangt, dieses File soll anschliessend zum Lesen eröffnet werden. Existiert das File nicht, so würde im Standardfall Programmabbruch erfolgen. Mittels der I-Direktive kann zur Wiederholung der Eingabe aufgefordert werden.
Dazu muss man wissen, dass das TURBO-PASCAL-System eine Standardvariable IORESULT verwaltet, die den Wert 0 erhält, wenn eine Eingabe/Ausgabe-Aktivität korrekt ausgeführt wurde, sonst einen von 0 verschiedenen Wert (vgl. Abschn. 2.15.). Damit lässt sich z.B. nach der Eingabe eines unzulässigen Filenamen die Eingabe wie folgt wiederholen:

```
WRITE('Filename eingeben: ');
repeat
READLN(fn); {var fn:string[14]}
ASSIGN(akt_file,fn);
{¤I-} RESET(akt_file); {¤I+}
if IORESULT <>0 then
     WRITELN('Filename unzulaessig; Eingabe wiederholen');
until IORESULT=0;
```

I Filesubstitution (include file)
I gefolgt von einem Leerzeichen und dem Namen eines Textfiles bewirkt, dass das Textfile an der durch die Direktive bezeichneten Stelle eingefügt, d.h. dem Compiler zur Verfügung gestellt wird. Beim Fehlen eines Namenszusatzes wird .PAS angefügt. Erkennt in dem eingefügten Text der Compiler einen syntaktischen Fehler, so wird nach dem Übergang in den Edit-Modus das eingefügte File zur Korrektur angeboten; nach der Korrektur kann die Compilation sofort wiederholt werden.
Diese I-Direktive muss als eigene Programmzeile notiert sein.
Innerhalb von Files, die mittels der I-Direktive eingefügt werden, dürfen keine weiteren I-Direktiven enthalten sein (keine Verschachtelung).
Beispiel:

```
program verwaltung;
{i Vorspann.dek}
{i Proz1.pas}
begin
end.
```

R Indexprüfung
Standardwert: R-
Durch {¤R+} fügt der Compiler in den Objektkode Anweisungen ein, die bei Zugriff zu einem Feldelement prüfen, auch die Indexwerte in den entsprechenden Definitionsbereichen liegen. Ebenso wird bei allen Zuweisungen zu Variablen eines Teilbereichstyps und eines Skalartyps geprüft, ob der Wert im Definitionsbereich liegt. Beim Auftreten von Fehlern erfolgt Programmunterbrechung. Es sind mittels Escape-Taste der Übergang in den Edit-Modus und sofortige Korrektur des Quelltextes möglich. Die Abarbeitungszeit für Programme, die mittels {¤R+} übersetzt sind, ist logischerweise grösser als die Zeit für Standardübersetzungen. Für die Phase des Austestens der logischen Korrektheit eines Programms ist die R-Direktive eine sehr gute Hilfe. Ausgetestete Programme sollten danachmit dem Standardwert noch einmal übersetzt werden.

U Programmunterbrechung durch Nutzer
Standardwert: U-
Durch {¤U+} wird erreicht, dass von der Tastatur durch Eingabe von ^C jederzeit das Programm abgebrochen werden kann. Die Direktive {¤U+} vergrössert die Abarbeitungszeit.

V Typkontrolle für Referenzparameter
Standardwert: +
Werden Zeichenketten über Referenzparameter übergeben (Attribut **var**), so wird standardmässig geprüft, ob die Länge des aktuellen Strings der des formalen entspricht. {¤V-} schaltet diese Prüfung aus.

W Verschachtelung von WITH-Anweisungen
Standardwert: W2
Durch die W-Direktive wird die Verschachtelung von With-Anweisungen auf die Tiefe beschränkt, die durch die natürliche Zahl angegeben wird. Es sind die Verschachtelungstiefen 1 bis 9 erlaubt.

X Organisation von Feldern
Standardwert: X+
Im Standardfall werden Felder so organisiert, dass die Zugriffszeiten zu Feldkomponenten minimal werden und damit die Abarbeitungszeit kurz wird. Im Falle {¤X-} ist das Ziel, den Kode zu minimieren.

3. Bildschirmprogrammierung – ein Überblick

3.1. Bildschirmsteuerung

Unter dem Bertriebssystem CPM/80 (oder kompatiblen Systemen) gibt es in TURBO-PASCAL folgende Standardprozeduren zur Bildschirmsteuerung:

CLRSCR
Der Bildschirm wird gelöscht, der Kursor wird in die linke obere Ecke gesetzt (Ecke mit den kartesischen Koordinaten (1,1)).

CLREOLN
Löscht die Zeichen von der Stellung des Kursors bis zum Zeilenende. Die Position des Kursors wird dabei nicht verändert.

DELLINE
Es wird die Zeile, in der sich der Kursor befindet, gelöscht. Alle folgenden Zeilen werden nach oben verschoben.

INSLINE
Es wird an der Stelle, an der sich der Kursor befindet, eine Leerzeile eingefügt. Alle folgenden Zeilen werden um eine Zeile nach unten verschoben.

GOTOXY(xkoord,ykoord)
Der Kursor wird an die durch die ganzzahligen Werte xkoord und ykoord bezeichnete Stelle bewegt.
Darüber hinaus gibt es drei Prozeduren zur Steuerung der Bildschirmhelligkeit, nämlich

```
NORMVIDEO
LOWVIDEO
HIGHVIDEO
```

Ob sie installiert werden, hängt von den technischen Gegebenheiten des jeweiligen Rechnersystems ab.
Mit diesen Prozeduren und der WRITE-Prozedur lassen sich einfache Grafiken aufbauen.

3.2. Erweiterter Grafikmodus

Für 16-Bit-Rechner und die dafür geschaffenen Betriebssysteme gibt es in TURBO-PASCAL einen erweiterten Grafikmodus. Es ist nicht beabsichtigt, die Unterstützung des Einsatzes grafischer Bildschirme in unterschiedlichen Rechnerstrukturen durch TURBO-PASCAL-Erweiterungen hier zu besprechen. In einem kurzen Überblick soll jedoch dargelegt werden, was die durch TURBO-PASCAL angebotenen Prozeduren prinzipiell leisten. Hardwaremässig wird ein Bildschirmterminal mit einem bestimmten Auflösungsvermögen vorrausgesetzt, mit dem in unterschiedlichen Modi gearbeitet werden kann. Typische Daten sind

25 Zeilen mit 40 oder 80 Zeichen	für Textmodus
320 * 200 Bildpunkte für Schwarzweiss-Grafik	für Grafikmodus
320 * 200 Bildpunkte für Farbwiedergabe	für Farbgrafikmodus

oder Bildschirmterminal mit dem höheren Auflösungsvermögen von 640 * 200 Bildpunkten.
Das System bietet nun Bezeichner für Farben als vordefinierte Konstanten eines Aufzählungstyps an (16 Bezeichner) und einen Satz von Standardprozeduren. Die Prozeduren erlauben die Definition der Hintergrundfarbe und das Setzen von Punkten, komplexere Prozeduren füllen Bereiche des Schirms mit farbigen Mustern. In bezug auf das Zeichnen von Figuren werden zwei Arbeitsweisen angeboten. Die eine besteht darin, dass die Bildpunkte durch kartesische Koordinaten adressiert werden. Koordinaten können berechnet und punktweise in den zugelassenen Farben gesetzt werden. Für die Erzeugung einfacher geometrischer Linien gibt es Standardprozeduren (so zeichnet z.B.

```
DRAW(x1, y2, x2, y2, farbe)
```

eine Gerade, die die Punkte (x1,y1), (x2,y2) mit der Farbangabe farbe verbindet).

Durch Angabe der beiden Eckpunkte der Diagonalen eines Rechtecks wird ein Rechteck bestimmt, in dem so angezeichneten Bildschirmteil kann weitergearbeitet werden (Window-Technik, window = Fenster). Die Koordinaten sind dann relative Koordinaten in dem Fenster.

Die zweite Arbeitsweise ist die sog. Turtle-Grafik (turtle = Schildkröte). Um grafische Darstellungen einfach erzeugen zu können, wird in der Mitte des Schirms eine Schildkröte gesetzt (sie kann sichtbar oder versteckt sein), die nun auf dem Schirm bewegt werden kann. Die Ausgangsposition der Turtle-Koordinaten ist (0,0), das ist der Mittelpunkt des Schirms oder des aktiven Fensters. Die Schildkröte kann nach oben und unten, nach links und rechts bewegt

werden. Das entspricht in diesen Koordinaten den Wertebereichen
-159 <= x <= 160 und -99 <= y <= 100 (320 * 200 Bildpunkte)
bzw. -319 <= x <= 320 und -99 <= y <= 100 (640 * 200 Bildpunkte).
Die Schildkröte ist mit einer "Feder" zum Zeichnen ausgestattet, durch PENDOWN oder PENUP erzeugt sie auf ihrem Weg einen Strich oder nicht. Richtungsänderungen werden durch Standardprozeduren bewirkt, deren Argumente als Winkelangaben interpretiert werden. Die Turtle-Grafik ist mit der Window-Technik kombiniert. Zusammen mit den allgemeinen Möglichkeiten der Algorithmendarstellung mittels TURBO-PASCAL lassen sich damit Computergrafiken aufbauen.

4. Lösungen zu den Übungen

A2.8.-1: Erzeugen einer Zahl

```
{Loesung zu A2.8.-1}
program   Erzeugen_einer_Zahl;
var  h: INTEGER;
          c: CHAR;
begin
   h:= 0;
   READ(c);
while  (c>='0')  and  (c<='9')  do
      begin  h:= h+10+INTEGER(c)-INTEGER('0');
                  READ(c);
      end  ;
   WRITELN(h);
end
```

A2.8.-2: Tabelle des Werteverlaufs einer Funktion

```
{Loesung zu A2.8.-2}
program Werteverlauf;
var t, dt, y, a, b: REAL;
    i: INTEGER;
    c: CHAR;
begin
WRITE('Intervallgrenzen eingeben');
READLN(a,b);
WRITE('Schrittweite eingeben');
READLN(dt);
t:= a;  i:= 0;
repeat
y:= (0.4*EXP(-0.2*t)+0.1)*(2.0*COS(t)+0.5*COS(2.0*t));
WRITELN(t:8:2,y);
f:= i+1;
if i=10 then begin i:= 0;
                   READLN(c);
             end;
t:= t+dt;
until t>b+0.001*dt+
end.
```

A2.8.-3: Grösster gemeinsamer Teiler

Das Diagramm zeigt die Vorgehensweise nach dem sogenannten Euklidischen Algorithmus. Danach wird a dargestellt als ein Vielfaches von b und einen Rest r, der kleiner als b ist:
a=q*b+r
Entsprechend wird b als ein Vielfaches von r und einen Rest r1 dargestellt, wobei r1<b ist. Setzt man das so fort, so erhält man letztlich den Rest 0. Der letzte von 0 verschiedene Rest ist der grösste gemeinsame Teiler von a und b.
Nach dem Diagramm lässt sich unmittelbar kodieren.

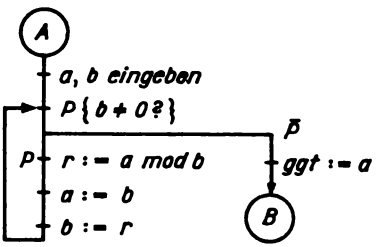


Bild 4.1. Euklidischer Algorithmus

```
 {Loesung zu A2.8.-3}
program ggT;
var a, b, r: INTEGER;
begin
WRITE('1. Wert eingeben:');
READLN(a);
while a<>0 do
      begin WRITE('2. Wert eingeben:');
            READLN(b);
            WRITE('a=',a,',b=',b);
            repeat
            r:=a mod b;
            a:=b; b:=r;
            until r=0;
            WRITELN('ggT(a,b)=',a);
      WRITE('1. Wert des naechsten Paares eingeben: ');
      READLN(a);
      end;
end.
```

A2.8.-4: Berechnungen eines Funktionswertes
Schreibt man die ersten Summanden explizit hin, so erhält man

$$y = \frac{1}{1.2} - \frac{x}{2.3} + \frac{x^2}{3.4} - +$$

Bezeichnet man mit su den 2. Summanden, mit s die Partialsumme und führt für den Index n ein, so ergibt sich nebenstehen- d e s D i a g r a m m. Aus dem Diagramm (Bild 4.2.) folgt unmittelbar folgendes Programm.

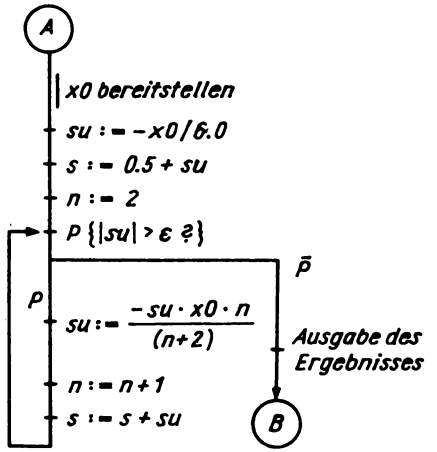


Bild 4.2. Berechnung eines Funktionswertes

```
{Loesung zu 2.8.-4}
program Funktionswert;
const eps>1.0e-10 ;
var n, s, su, x0 :REAL;

begin
WRITE('Argument eingeben:'); READLN(x0);
su:=-x0/6.0;
s :=0.5+su;
n :=2.0;
while ABS(su)>eps do
      begin su:=-su*x0*n/(n+2.0);
            n :=n+1.0;
            s :=s+su;
      end;
WRITELN('X0=',x0,'y=',s);
{Programmfortsetzung}
end.
```

A2.9.2.-1: Zeilensummen und Spaltensummen einer Matrix

```
{Zeilensummen}
     for i := 1 to m do
          begin
          h := 0;
          for j := 1 to n do h := h + a[i,j];
```

```
          z[i] := h;
          end;
{Spaltensummen}
     for j := 1 to n do
          begin
          h := 0;
          for i := 1 to m do h := h + a[i,j];
          s[j] := h;
          end;
```

A2.9.2.-2: Maximale Komponente eines Vektors

```
     maxkomp := a[1];  index_nr := 1;
     for i := 2 to n do
          if maxkomp < a[i] then begin maxkomp := a[i];
                                       index_nr := i;
                                 end;
     WRITELN('Maximum = ', maxkomp);
     WRITELN('Index   = ', index_nr);
```

A2.9.2.-3: Summe der Diagonalelemente einer quadratischen Matrix

```
     s1 := 0;  s2 := 0;   {Initialisierung}
     for i := 1 to n do begin s1 := s1 + a[i,i];
                              s2 := s2 + a[i,n+1-i];
                        end;
```

A2.9.2.-4: Summe der Elemente oberhalb der Hauptdiagonalen

In der i-ten Zeile sind die Elemente

```
     a[i,i+1]  a[i,i+2]  ...   a[i,n]
```

zu addieren. Das ist für jede Zeile i, 1 <= i < n, durchzuführen. Die Addition in der i-ten Zeile lässt sich schreiben als

```
     for j := i + 1 to n do s := s + a[i,j].
```

Für alle Zeilen heisst das also

```
     s := 0;
     for i := 1 to n-1 do
          for j := i + 1 to n do s := s + a[i,j];
```

Andere Vorgehensweisen wären

```
     s := 0;
     for j := n downto 2 do
          for i := 1 to j-1 do s := s + a[i,j];
```

oder

```
     s := 0;
     for j := 2 to n do
          for i := 1 to n+1-j do s := s + a[i,i-1+j];
```

Man mache sich klar, in welcher Reihenfolge in jedem Fall die Feldelemente addiert werden!

A2.9.3.-1: Zugriff zu den Komponenten strukturierter Variabler
Es können nur Vereinbarungen angegeben werden, die für die verwendeten Variablen ausreichend wären. Selbstverständlich sind auch andere Vereinbarungen möglich.

```
(1)  type matrix = array[n1..m1,n2..m2] of REAL;
          satz1 = record
                    ...
                    y: matrix;
                    ...
                    end;
     var x: satz1;

(2)  var x1: record
               ...
               x2: record
                   ...
                   q: array[1..8] of BOOLEAN;
                   ...
                   end;
                ...
                end;

(3)  var v1: array[0..5] of record
                            ...
                            v2: array[0..9] of REAL;
                            ...
                            end;
(4)  type string4 = string[4];
               r0 = record
                      ...
                      r: string4;
                      ...
                      end;
             index = mm..n;
           vektor = array[index] of r0;
     var tk: vektor;

(5)  var s: array[0..127] of tau;
                           {tau bedeute einen Aufzaehlungstyp}
         d: record
            ...
            u: array[tau] of k0..k1;
            ...
            end;
```

A2.12.1.-1: Bild 2.10a

```
     ...
```

```
type string7 = string[7];
     pstring7 = ^string7;
var vektor: array[1..3] of pstring7;
...
begin
     NEW(vektor[1]);  vektor[1]^ := 'BERLIN';
     NEW(vektor[2]);  vektor[2]^ := 'POTSDAM';
     NEW(vektor[3]);  vektor[3]^ := 'ROSTOCK';
...
end;
```

A2.12.1.-2: Bild 2.10b

```
type objekt2 = {beliebiger Typ} record ... end;
     pobjekt2 = ^objekt2;
     objekt1 = record
               {beliebige Datenfelder}
               verweise: array[1..3] of pobjekt2;
var w: ^objekt1;
...
begin
...
NEW(w);
NEW(w^.verweise[1]);  NEW(w^.verweise[3]);
w^.verweise[2] := nil;
...
end.
```

A2.12.1.-3: Bild 2.10c

In dem Bild ist für das unterste Niveau der Typ INTEGER angegeben. Also müssen die darüberliegenden Variablen vom Typ ^INTEGER sein. Da diese Variablen nur durch Verweise erreicht werden, handelt es sich ebenfalls um dynamische Variablen. Lediglich die beiden Variablen auf dem oberen Niveau sind statisch, sie müssen als Variablen deklariert werden, die auf (dynamische) Variablen verweisen, die vom Typ ^INTEGER sind. Dieser Sachverhalt lässt sich folgendermassen darstellen:

```
type pint = ^INTEGER;
     ppint = ^pint;
var p1, p2: ppint;
...
begin
     NEW(p1);  NEW(p2);
     NEW(p1^);
     p2^ := p1^;
```

Bild 4.3. zeigt die mit diesen Bezeichnungen vervollständigte Struktur.

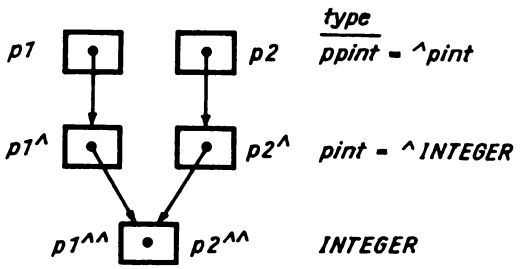


Bild 4.3.
Struktur zu A2.12.1.-3.

A2.14.5.-1: Nullstelle einer Funktion
Es wird das Verfahren der Intervallschachtelung benutzt. Die Iteration wird abgebrochen, wenn die Intervallänge < 10e-6 ist.

```
{nullstelle}
     function nullstelle (a,b: REAL): REAL;
     {f ist eine zuvor deklarierte Funktion}
     var y, ya, yb, mitte: REAL;
     begin
          ya := f(a);  yb := f(b);
          repeat
          mitte := (a + b) * 0.5;
          y := f(mitte);
          if y * ya > 0 then a := mitte;
                        else b := mitte;
          until ABS(b-a) < 10e-6;
          nullstelle := (a + b) * 0.5;
     end;
```

A2.14.5.-2: Suchen des betragsmässig grössten Elements einer Matrix

```
     procedure max_el (var a:  matrix;  m,  n:  INTEGER;
                                  zeile, spalte: INTEGER);
     var i, j, ii, jj: INTEGER;
          el: REAL;
     begin
          el := ABS(a[1,1]);  ii := 1;  jj := 1;
          for i := 1 to m do
               for j := 1 to n do if ABS(a[i,j]) > el then
                    begin  el := ABS(a[i,j]);
                           ii := i;
                           jj := j;
```

```
                    end;
      zeile := ii;  spalte := jj;
      end;
```

A2.14.5.-3: Unendliche Reihe

```
{Reihe}
program reihe;
const eps = 1.0e-6; {Genauigkeitsschranke fuer Approximation}
var
   x: REAL;
   function f(x,eps:REAL):REAL;
   var xx, s1, s, n:REAL;
   begin
     xx := sqr(x);
     n := 1.0;
     s1 := -xx/12.0;
     s := 1.0 + s1;
     while ABS(s1) > eps do
        begin
        s1 := -s1 * xx * (0.5-n)/((n+1)*(2.0*n+2.0)*(2.0*n+3.0));
        n := n + 1.0;
        s := s + s1;
        end;
     f:= s;
  end {f};
begin
   WRITELN('Werteverlauf der Funktion im Intervall -0.5<=x<=0.5');
   WRITELN;
   x := -0.5;
   while x <= 0.5 do
      begin WRITELN(x:5:1, f(x,eps));
            x := x +0.1;
      end;
end.
```

A2.14.5.-4: Berechnung der Komponenten eines Vektors

```
{Koeffizienten}
program Koeff_eines_Polynoms;
const pn = 10;
type
   stuetzstellen = array[1..pn] of REAL;
   koeffvektor = array[0..pn] of REAL;
var
   n,i:INTEGER;
   a: koeffvektor;
   x, y: stuetzstellen;
```

```
   procedure koeff ( n:INTEGER; x,y: stuetzstellen;
                     var a: koeffvektor);
   var
      j, k: INTEGER;
      c, p:REAL;
   begin
     c := 0.0;
     for j := 1 to n do c := c + y[i];
     for j := 1 to n do
      begin
        p := 1.0;
        for k := 1 to n do
                 if k<>j then p := p*(x[j] - x[k]);
        a[j-1] := c/p;
     end;
     end{koeff};
begin
  WRITELN('Anzahl der Stuetzstellen:'); READLN(n);
  if n in [3..10] then
                  begin
                    for i := 1 to n do READLN(x[i],y[i]);
                    koeff(n, x, y, a);
                    WRITELN('Koeffizienten des Polynoms:');
                    for  i := n-1 downto 0 do
                             WRITELN('a[',i:2,']= ',a[i]:12:4);
                 end
                  else
                  WRITELN('Wert fuer n falsch.');
{ ... }
end.
```

Anhänge

A ASCII

00	0	^@	NUL	20	32	blank	40	64	@	60	96		
01	1	^A	SOH	21	33	!	41	65	A	61	97	a
02	2	^B	STX	22	34	"	42	66	B	62	98	b
03	3	^C	ETX	23	35	#	43	67	C	63	99	c
04	4	^D	EOT	24	36	¤	44	68	D	64	100	d
05	5	^E	ENQ	25	37	%	45	69	E	65	101	e
06	6	^F	ACK	26	38	&	46	70	F	66	102	f
07	7	^G	BEL	27	39	'	47	71	G	67	103	g
08	8	^H	BS	28	40	(	48	72	H	68	104	h
09	9	^I	HT	29	41	)	49	73	I	69	105	i
0A	10	^J	LF	2A	42	*	4A	74	J	6A	106	j
0B	11	^K	VT	2B	43	+	4B	75	K	6B	107	k
0C	12	^L	FF	2C	44	,	4C	76	L	6C	108	l
0D	13	^M	CR	2D	45	-	4D	77	M	6D	109	m
0E	14	^N	SO	2E	46	.	4E	78	N	6E	110	n
0F	15	^O	SI	2F	47	/	4F	79	O	6F	111	o
10	16	^P	DLE	30	48	0	50	80	P	70	112	p
11	17	^Q	DC1	31	49	1	51	81	Q	71	113	q
12	18	^R	DC2	32	50	2	52	82	R	72	114	r
13	19	^S	DC3	33	51	3	53	83	S	73	115	s
14	20	^T	DC4	34	52	4	54	84	T	74	116	t
15	21	^U	NAK	35	53	5	55	85	U	75	117	u
16	22	^V	SYN	36	54	6	56	86	V	76	118	v
17	23	^W	ETB	37	55	7	57	87	W	77	119	w
18	24	^X	CAN	38	56	8	58	88	X	78	120	x
19	25	^Y	EM	39	57	9	59	89	Y	79	121	y
1A	26	^Z	SUB	3A	58	:	5A	90	Z	7A	122	z
1B	27	^[	ESC	3B	59	;	5B	91	[	7B	123	{
1C	28	^\	FS	3C	60	<	5C	92	\	7C	124	|
1D	29	^]	GS	3D	61	=	5D	93	]	7D	125	}
1E	30	^^	RS	3E	62	>	5E	94	^	7E	126	~
1F	31	^_	US	3F	63	?	5F	95	_	7F	127	DEL

Die Steuerzeichen haben folgende Bedeutung:

NUL	NIL	DC1	device control 1
SOH	start of heading	DC2	device control 2
STX	start of text	DC3	device control 3
ETX	end of text	DC4	device control 4
EOT	end of transmission	NAK	negative acknowledge
ENQ	enquiry	SYN	synchronisationck
ACK	acknowledge	ETB	end of block
BEL	bell	CAN	cancel
BS	backspace	EM	end of medium
HT	horizontal tabulator	SUB	substitution
LF	line feed	ESC	escape
VT	vertical tabulator	FS	form seperator
FF	form feed	GS	group seperator
CR	carriage return	RS	seperator
SO	shift out	US	seperator
SI	shift in		
DLE	data link escape		

B Syntax von TURBO-PASCAL

1. Darstellung in Backus-Naur-Form (BNF)

1.1. Metasymbole

Zur Notation der Syntax werden folgende Metasymbole benutzt:

Symbol	Bedeutung
::=	"ist definiert als"
|	"oder"
"text"	Metasprachliche Variablen. Der Wertebereich einer solchen Variablen wird in einer Definition mittels "::=" beschrieben.

Alle anderen in der Syntaxdarstellung auftretenden Zeichen sind Teile der beschriebenen Sprache, gehören also zur Symbolmenge von TURBO-PASCAL.
Halbfett gedruckte Wörter sind Basissymbole der Sprache. Sie werden in Programmen durch die Buchstabenfolge dargestellt, dürfen aber nur in dem durch die Syntax definierten Zusammenhang benutzt werden (Schlüsselwörter, reserved words).

1.2. Grundbegriffe

Folgende metasprachliche Variablen werden im weiteren ohne Definition benutzt:

"ziffer"	Zeichen 0...9 .
"ziffern"	Aus den Ziffernzeichen 0...9 gebildete Folge .
"buchstabe"	Zeichen a...z und A...Z und das Unterstreichungszeichen _ .
"hexaziffer"	"ziffer" oder eines der Zeichen a...f oder A...F .
"hexaziffern"	Aus "hexaziffern" gebildete Folge .
"vergleichsoperator"	Die Zeichen < | > | = und die Zeichenpaare <= | >= | <> .
"additionsoperator"	Die Zeichen + | - | **or** | **xor** .
"mult-operator"	Die Zeichen * | / | **div** | **mod** | **and** | **shl** | **shr** .

"bezeichner"	Jede aus "buchstabe" und "ziffer" gebildete Folge, die mit einem Buchstaben beginnt .
"zeichen"	Elemente von ASCII, die nicht "steuerzeichen" sind .
"steuerzeichen"	^"zeichen"|#"ziffern"|@"ziffern" .
"zeichenkette"	"steuerzeichen" und/oder in Apostrophe eingeschlossene Folge von "zeichen" können zu einer Kette zusammengefügt werden . Beispiel:'DO'^G'ING'

1.3. Syntax des TURBO-PASCAL-Programms

```
"programm"::="programmkopf""block" .
  "programmkopf"::="leer"|program "bezeichner""programmparameter";
    "leer"::=
    "programmparameter"::="leer"|("filebezeichnerliste")
      "filebezeichnerliste"::="bezeichner"|
                              "filebezeichnerliste","bezeichner"
  "block"::="vereinbarungsteil""verbundanweisung"

    "vereinbarungsteil"::="leer"|
                          "vereinbarungsabschnitt"|
                          "vereinbarungsteil""vereinbarungsabschnitt"
      "vereinbarungsabschnitt"::="markendefinitionsteil"|
                                 "konstantendefinitionsteil"|
                                 "variablendeklarationsteil"|
                                 "typendefinitionsteil"|
                                 "routinendeklarationsteil"
        "markendefinitionsteil"::=label "markenliste";
          "markenliste"::="marke"|"markenliste","marke"
          "marke"::="buchstabe"|"ziffer"|"marke""marke"

        "konstantendefinitionsteil"::=const "konstantenliste";
          "konstantenliste"::="konstantendefinition"|
                              "konstantenliste";"konstantendefinition"
            "konstantendefinition"::="konstantenbezeichner"="konstante"|
                  "konstantenbezeichner":="typbezeichner"="typkonstante"

              "konstantenbezeichner"::="bezeichner"
              "konstante"::="numerische konstante"|
                            "zeichenkettenkonstante"|
                            "konstantenbezeichner"
                "numerische konstante"::="vorzeichenlose zahl"|
                                         "vorzeichen""vorzeichenlose zahl"|
                                         "konstantenbezeichner"|
```

```
                              "vorzeichen""konstantenbezeichner"
          "vorzeichenlose zahl"::="ziffern"|"hexaziffern"|
                                   "positive zahl"
            "positive zahl"::="dezimalzahl"|
                              "ziffern""exponententeil"|
                              "dezimalzahl""exponententeil"
              "dezimalzahl"::="ziffern"."ziffern"
              "exponententeil"::=E"ziffern"|e"ziffern"|
                                 E"vorzeichen""ziffern"|
                                 e"vorzeichen""ziffern"
          "vorzeichen"::=+|-

          "zeichenkettenkonstante"::="zeichenkette"|
                                     "konstantenbezeichner"

        "typbezeichner"::="bezeichner"
        "typkonstante"::="konstante"|"aggregat"
          "aggregat"::="feldkonstante"|"recordkonstante"|
                       "mengenkonstante"
            "feldkonstante"::=("feldkomponentenliste")
              "feldkomponentenliste"::="konstante"|"aggregat"|
                 "feldkomponentenliste","feldkomponentenliste"
            "recordkonstante"::=("recordkomponentenliste")
              "recordkomponentenliste"::=
                            "datenfeldbezeichner":"konstante"|
                               "datenfeldbezeichner":"aggregat"|
               "recordkomponentenliste";"recordkomponentenliste"
                "datenfeldbezeichner"::="bezeichner"

            "mengenkonstante"::=[] | ["mengenelement"]
              "mengenelement"::="konstante"|
                                "konstante".."konstante"|
                                "mengenelement","mengenelement"

"variablendeklarationsteil"::=var"variablenliste";
  "variablenliste"::="variablendeklaration"|
                     "variablenliste";"variablendeklaration"
    "variablendeklaration"::="bezeichnerliste":"typ";|
                             "bezeichner":"typ" absolute "adresse"
      "bezeichnerliste"::="bezeichner"|
                          "bezeichnerliste","bezeichner"
      "adresse"::="ziffern"|"hexazahl"|"variablenbezeichner"
        "variablenbezeichner"::="bezeichner"

"typendefinitionsteil"::=type"typenliste";
  "typenliste"::="typdefinition"|
                 "typenliste";"typdefinition"
    "typdefinition"::="bezeichner"="typ"
```

```
"typ"::="einfacher typ"|"pointertyp"|
        "strukturierter typ"|"gepackter typ"

  "einfacher typ"::="aufzählungstyp"|"typbezeichner"
    "aufzählungtyp"::="standardaufzählungstyp"|
                      "skalartyp"|"teilbereichstyp"|
                      "typbezeichner"
      "standardaufzählungstyp"::="typbezeichner"
      "skalartyp"::=("bezeichnerliste")
      "teilbereichstyp"::="konstante".."konstante"

  "pointertyp"::=^"typbezeichner"

  "strukturierter typ"::="zeichenkettentyp"|"feldtyp"|
                         "mengentyp"|"filetyp"|"recordtyp"
    "zeichenkettentyp"::=string["ziffern"]|
                         string["hexaziffern"]|
                         string["konstantenbezeichner"]|
                         "typbezeichner"
    "feldtyp"::=array["indextypliste"] of
                                        "komponententyp"
      "indextypliste"::="indextyp"|
                        "indextypliste","indextyp"
        "indextyp"::="aufzählungstyp"
      "komponententyp"::="typ"

    "mengentyp"::=set of "aufzählungstyp"

    "filetyp"::=file of "typ"

    "recordtyp"::=record "datenfeldliste" end
      "datenfeldliste"::="fester teil"|"variantenteil"|
                         "fester teil";"variantenteil"
        "fester teil"::="datenfeldbezeichnerliste":
                                              "typ"|
                        "fester teil";fester teil"
          "datenfeldbezeichnerliste"::=
                              "datenfeldbezeichner"|
                          "datenfeldbezeichnerliste",
                               "datenfeldbezeichner"
            "datenfeldbezeichner"::="bezeichner"

        "variantenteil"::=case "kennzeichen" of
                                          "variante"|
                          "variantenteil";"variante"
          "kennzeichen"::="aufzählungstyp"|
                          "bezeichner":"aufzählungstyp"
          "variante"::="konstante":("datenfeldliste")|
```

```
                                    "konstante","variante"

          "gepackter typ"::=packed "strukturierter typ"


    "routinendeklarationsteil"::="routine"|
                                    "routinendeklarationsteil";"routine"
      "routine"::="prozedurdeklaration"|"funktionsdeklaration"
      "prozedurdeklaration"::="prozedurkopf""block";|
                                    "prozedurkopf" forward;
        "prozedurkopf"::=procedure "prozedurbezeichner"
                                              "formalparameterteil";
         "prozedurbezeichner"::"bezeichner"
         "formalparameterteil"::="leer"|("formalparameterliste")

           "formalparameterliste"::="formalparameter"|
                              "formalparameterliste";"formalparameter"
             "formalparameter"::="bezeichnerliste":"typbezeichner"|
                        var "bezeichnerliste":"typbezeichner"

     "funktionsdeklaration"::="funktionskopf""block";|
                                    "funktionskopf" forward;

       "funktionskopf"::=function "funktionsbezeichner"
                            "formalparameterteil":"typbezeichner"|
                          function "funktionsbezeichner"
         "funktionsbezeichner"::="bezeichner"


"verbundanweisung"::=begin "verbundanhang"|
                     "marke": begin "verbundanhang"
  "verbundanhang"::="anweisung" end|
                    "anweisung""verbundanhang"
    "anweisung"::="einfache anweisung"|"marke""einfache anweisung"|
                  "strukturierte anweisung"|
                  "marke""strukturierte anweisung"
      "einfache anweisung"::="leeranweisung"|
                              "wertzuweisung"|"sprunganweisung"|
                              "prozeduranweisung"|"inlineanweisung"
        "leeranweisung"::="leer"
        "wertzuweisung"::="variable":="ausdruck"|
                           "funktionsbezeichner":="ausdruck"
        "sprunganweisung"::= goto "marke"
        "prozeduranweisung"::=
                        "prozedurbezeichner""aktualparameterteil"
          "aktualparameterteil"::="leer"|
                                  ("aktualparameterliste")
```

```
"aktualparameterliste"::="aktualparameter"|
              "aktualparameterliste","aktualparameter"
  "aktualparameter"::="ausdruck"|"variable"
    "ausdruck"::="einfacher ausdruck"|
       "einfacher ausdruck""vergleichsoperator"
                              "einfacher ausdruck"
      "einfacher ausdruck"::="term"|
                             "additionsoperator""term"|
        "einfacher ausdruck""additionsoperator""term"

        "term"::="komplementierter faktor"|
                 "term""mult-operator""komplementierter
                                          faktor"
          "komplementierter faktor"::="faktor"|
                                      not "faktor"


            "faktor"::="numerische konstante"|
                       "konstantenbezeichner"|
                       "funktionsaufruf"|"menge"|
                       ("ausdruck")
    "variable"::="vollständige variable"|
                "variablenkomponente"|
                "dynamische variable"
      "vollständige variable"::="variablenbezeichner"|
                                 "typkonstantenbezeichner"
        "variablenbezeichner"::="bezeichner"
        "typkonstantenbezeichner"::=
                                   "konstantenbezeichner"

      "variablenkomponente"::"indizierte variable"|
                             "qualifizierte variable"|
                             "datenfeldbezeichner"
        "indizierte variable"::="feldvariable""index"
          "feldvariable"::="variable"
          "index"::=["indexliste"]|
                    "index""index"
            "indexliste"::="ausdruck"|
                           "indexliste","ausdruck"

        "qualifizierte variable"::=
             "recordvariable"."datenfeldbezeichner"
          "recordvariable"::="variable"

      "dynamische variable"::="pointervariable"^
        "pointervariable"::="variable"

"funktionsaufruf"::="funktionsbezeichner"
```

```
                                                "aktualparameter"
      "menge"::="mengenvariable"|"mengenkonstruktion"
        "mengenvariable"::="variable"
        "mengenkonstruktion"::="leer"|"mengenelement"
     "inlineanweisung" := inline ( "maschinenkode")
            ("maschinenkode" bedeutet eine durch Schraegstriche
            getrennte Folge von Bytes oder formalen Parametern)

"strukturierte anweisung"::="verbundanweisung"|
                            "bedingte anweisung"|
                            "with-anweisung"|
                            "zyklusanweisung"
  "bedingte anweisung"::="if-anweisung"|"case-anweisung"

    "if-anweisung"::="if-klausel""anweisung"|
                "if-klausel""anweisung" else "anweisung"
      "if-klausel"::= if "ausdruck" then
    "case-anweisung"::=case "ausdruck" of "case-liste"
                                        "case-anhang" end
      "case-liste"::="kennzeichenliste":"anweisung"|
                     "case-liste" ; "case-liste"
        "kennzeichenliste"::="konstante"|
                             "konstante".."konstante"
      "case-anhang"::="leer"|
                      ; else "anweisungssequenz"
        "anweisungssequenz"::="anweisung"|
                              "anweisungssequenz";"anweisung"

     "with-anweisung"::= with "recordvariablenliste" do
                                                  "anweisung"
       "recordvariablenliste"::="recordvariable"|
              "recordvariablenliste","recordvariable"
         "recordvariable"::="variable"
     "zyklusanweisung"::="for-anweisung"|"while-anweisung"|
                         "repeat-anweisung"

      "for-anweisung"::=for "variablenbezeichner":="ausdruck"
                          to "ausdruck" do "anweisung"|
                        for "variablenbezeichner":="ausdruck"
                         downto "ausdruck" do "anweisung"

      "while-anweisung"::= while "ausdruck" do "anweisung"

      "repeat-anweisung"::= repeat "anweisungssequenz" until
                                                   "ausdruck"
```

C Standardfunktionen und Standardprozeduren

1. Standardfunktionen von TURBO-PASCAL (alphabetisch geordnet)

Aufruf	Parameter	Ergebnis-typ	Erläuterung
ABS(i)	i:INTEGER	INTEGER	|i|
ABS(x)	x:REAL	REAL	|x|
ADDR(v)	varv: bel. Typ	INTEGER	absolute Adr.
ARCTAN(x)	x:REAL	REAL	arctanx (Hauptw.)
BDOS(m,n)	m,n:INTEGER	INTEGER	BDOS-Funktion setzen
BDOSHL(m)	m:INTEGER	INTEGER	BDOS-Funktion; Übergabe über HL
BIOS(m,n)	m,n:INTEGER	INTEGER	BIOS-Funktion setzen
BIOSHL(m)	m:INTEGER	INTEGER	BIOS=Funktion; Übergabe in HL
CHR(m)	m:INTEGER	CHAR	Konvertiert m in ASCII-Zeichen
CONCAT(s1, ..., sn)	si:string	string	Verketten der Strings; Länge(erg)<=255
COPY(s,m,n)	s:string; m,n:INTEGER	string	Aus s werden m Zeichen ab Pos. m als Ergebnis geliefert. Es gilt: Fehler, wenn p nicht in [1..255]
COS(x)	x:REAL oder x:INTEGER	REAL	cos x, x im Bogenmass
EOF(f)	var f: file of bel. Typ	BOOLEAN	TRUE, wenn Filepointer auf Ende des Files zeigt, FALSE sonst
EOLN(f)	var f: TEXT	BOOLEAN	TRUE, wenn Filepointer auf eoln-Zeichen zeigt oder EOF(f)=TRUE,

			FALSE sonst
EXP(x)	x:REAL oder x:INTEGER	REAL	e hoch x
FILEPOS(f)	**var f: file of nicht_text**	INTEGER	Nummer der Komponente, auf die der Filepointer zeigt, Zählung beginnt mit 0
FILESIZE(f)	**var f : file of nicht_text**	INTEGER	Anzahl der Komponenten des Files
FRAC	x:REAL oder x:INTEGER	REAL	Gebrochener Teil von x
HI(i)	i:INTEGER	INTEGER	Höerwertiges Byte (high) von i
INT(x)	x:REAL oder x:INTEGER	REAL	Ganzer Teil von x
IORESULT	-	INTEGER	Wert ist nach E/A-Operation der Fehlerkode(s. Abschn. 2.15.)
KEYPRESSED	-	BOOLEAN	TRUE, wenn eine Taste gedrückt wurde
LENGTH(s)	s: **string**	INTEGER	Zeichenanzahl von s
LN(x)	x:REAL oder x:INTEGER	REAL	ln x
LO(i)	i:INTEGER	INTEGER	Niederwertiges Byte (low) von i
MAXAVAIL	-	INTEGER	Länge des grössten zusammenhängenden Speicherplatzes auf der Halde
MEMAVAIL	-	INTEGER	Anzahl der ungenutzten Bytes auf der Halde
ODD(i)	i:INTEGER	BOOLEAN	TRUE, wenn i ungerade, FALSE sonst
ORD(a)	a: Aufzählungstyp	INTEGER	Wert von a in dem Aufzählungstyp
POS(s1, s2)	s1,s2: **string**	INTEGER	Index der 1. Pos. des Teilstrings s1 in s2, 0, falls s1 nicht in s2
PRED(a)	a: Aufzählungstyp	Aufz.-T.	Vorgänger von a
RANDOM	-	REAL	Zufallszahl aus

RANDOM(i)	i:INTEGER	INTEGER	[0, 1) Zufallszahl aus [0, i)
ROUND(x)	x:REAL	INTEGER	ROUND(x)= TRUNC(x+0.5), x >= 0, ROUND(x)= TRUNC(x-0.5), x<0
SEEKEOF(f)	**var** f: TEXT	BOOLEAN	TRUE, wenn eof-Zeichen gefunden wird, FALSE sonst
SIN(x)	x:REAL oder x:INTEGER	REAL	sin x
SIZEOF(d)	d:bel. Typ	INTEGER	Anzahl der Bytes von d
SQR(i)	i:INTEGER	INTEGER	i hoch 2
SQR(x)	x:REAL	REAL	x hoch 2
SQRT(x)	x:REAL oder x:INTEGER	REAL	Quadratwurzel
SUCC(a)	a:Aufzählungstyp	Aufz.-T.	Nachfolger von a
SWAP(i)	i:INTEGER	INTEGER	Vertauschen der Bytes von i
TRUNC(x)	x:REAL	INTEGER	Ganzer Teil von x (entspr. INT)
UPCASE(c)	c:CHAR	CHAR	Grossbuchstabe von c

2. Standardprozeduren von TURBO-PASCAL (alphabetisch geordnet)

Aufruf	Parameter	Wirkung
ASSIGN(f,s)	**var** f: bel. File-Typ s: **string**	Das logische File f wird mit dem phys. File s identifiziert
BDOS(m,n)	**var** : m,n: INTEGER	BDOS-Prozedur m wird mit Parameter n aktiviert
BIOS(m,n)	**var** m,n: INTEGER	BIOS-Prozedur m wird mit Parameter n aktiviert
BLOCKREAD(f, ziel, anzahl)		
	var f : **file**; **var** ziel:bel. Typ; anzahl: INTEGER	Vom File f werden anzahl Sätze mit je 128 Bytes gelesen und der Variablen quelle zugewiesen
BLOCKWRITE(f, quelle, anzahl)		
	var f: **file**; **var** quelle: bel. Typ; anzahl:INTEGER	Von der Variablen quelle werden anzahl Sätze mit je 128 Bytes in das **file** f übertragen
CHAIN(f)	**var** f: **file**;	f (typfreie File-Variable mit

		Endung .CHN) wird als Programm aufgerufen. f muss durch ASSIGN-Anweisung ein physisches File zugewiesen sein.
CLOSE(f)	**var** f: bel. Typ	Abschliessen des Files
CLREOL	-	Zeilen von Kursorposition bis Ende löschen
CLSCR	-	Löschen des Bildschirms
DELETE(s,m,n)	**var** s: **string**; m,n: INTEGER	Ab Position m werden n Bytes im String s gelöscht
DELLINE	-	Löschen der Zeile, in der sich der Kursor befindet
DISPOSE(pv)	pv: piontertyp	Speicherplatz der dynamischen Variablen pv^ wird freigegeben
ERASE(f)	**var** f: bel. File-Typ	File f wird gelöscht
EXECUTE(f)	**var** f: **file**;	f (typfreie File-Variable, Objektkodefile) wird als Programm aufgerufen
EXIT		Die Abarbeitung entspricht einem Sprungbefehl an das Ende des aktiven Blockes
FILLCHAR(d,n,c)	d:bel. Typ; n:INTEGER c*CHAR	n Bytes der Variablen d werden mit dem Wert des Parameters c beschrieben
FLUSH(f)	**var** f: bel.File-Typ	Pufferinhalt wird nach f übertragen
FREEMEM(pv,i)	pv:pointertyp; i:INTEGER	Korrespondierend zum vorhergehenden GETMEM-Aufruf werden i Bytes der Halde freigegeben
GETMEM(pv,i)	pv:pointertyp; i:INTEGER	Es werden i Bytes der Halde bereitgestellt; die Anfangsadresse dieses Bereichs wird der Pointer-Variablen als Wert zugewiesen
GOTOXY(m,n)	m,n:INTEGER	Kursor wird in Zeile m und Spalte n positioniert (obere linke Ecke hat Koordinaten (1, 1))
HALT	-	Programmabbruch
INSERT(s1,s2,n)	s1:**string**; **var** s2: **string**; n: INTEGER	Einfügen der Zeichenkette s1 in die Stringvaraiable s2 ab Position n
MARK(pv)	pv:pointertyp	Fixieren des Wertes der Pointer-Variablen pv. Dieser Wert der Speicherfreigabe durch RELEASE
MOVE(quelle,ziel,n)	n:INTEGER;	n Bytes von quelle werden nach

	quelle,ziel:bel. Typ	ziel übertragen
NEW(pv)	pv:pointertyp	Bereitstellung von Speicherplatz für eine dynamische Variable pv^. Die Pointer-Variable pv erhält die Adresse dieses Speicherbereichs als Wert zugewiesen
RANDOMIZE -		Initialisierung des Zufallszahlengenerators
READ(f,v1, ... , vn)	**var** f: **file of** bel. Typ; **var** v1, ..., vn: bel. Typ	Aus dem File f werden n Komponenten übertragen und den Variablen v1, ..., vn als Werte zugewiesen
READLN(f)	**var** f: TEXT	Positionierung des File-Pointers auf das erste Zeichen der nächsten Zeile
RELEASE(pv)	pv:pointertyp	Freigabe des Haldenspeichers bis zu dem Speicherplatz, der durch MARK(pv) fixiert worden ist
RESET(f)	**var** f: **file of** bel. Typ	Filepointer wird auf Komponente 0 eines existierenden Files gesetzt
REWRITE(f)	**var** f: **file of** bel. Typ	Filepointer verweist auf Komponente 0; existiert das File, so wird es gelöscht
SEEK(f,m)	**var** f: **file of** nicht_text; m:INTEGER	Filepointer wird auf n-te Komponente gestellt
STR(e,s)	e:REAL; oder e:INTEGER; **var** s: **string**	e symbolisiert einen arithmetischen Ausdruck, wie er bei Ausgabe auf ein Textfile verwendet werden kann, d. h., es können zusätzliche Formatierungsangaben gemacht werden. Gemäss dieser Konvertierung wird eine Zeichenkette erzeugt und der Stringvariablen s als Wert zugewiesen
VAL(s,x,i)	s: **string**; **var** x: REAL; oder **var** x:INTEGER; i:INTEGER	Der Wert von s muss eine ganze oder reelle Zahl darstellen entsprechend der Syntax von Eingabewerten. Dieser Wert wird in eine Zahlendarstellung konvertiert und x zugewiesen. i erhält den Wert 0, wenn die Zahlendarstellung in s korrekt war, sonst erhält i als Wert den Index der fehlerhaften

		Stelle
WRITE(f,k1,...,kn)		
	var f: **file of** bel. Typ;	Auf das File f werden n Komponenten übertragen (vgl. auch WRITELN)
	ki:komponententyp	
WRITELN(f,k1, ..., kn)		
	var f: TEXT;	Die ki symbolisieren Ausdrücke,
	ki: Ausdrücke von einem der folgenden Typen: REAL, BYTE, INTEGER, CHAR, **string**, BOOLEAN, **array**[..] **of** CHAR	zusätzlich können Formatierungsangaben gemacht werden, und zwar in der Form ki:l:d bei reellen Werten, ki:l bei allen (erlaubten) Typen.
		Auf das File werden n Werte ausgegeben, zusätzlich wird ein end-of-line-Zeichen ausgegeben
WRITELN(f)	**var** f: TEXT	Auf das File f wird ein end-of-line-Zeichen ausgegeben

D Fehlernachrichten des Compilers

Die Fehlernachrichten des Compilers befinden sich in dem File TURBO.MSG. Beim Starten des TURBO-PASCAL-System wird gefragt, ob dieses File geladen werden soll:

Include error messages(Y/N)?

Wird mit Y geantwortet, so erhält man während der Übersetzung beim Auftreten eines Fehlers eine der folgenden Mitteilungen; wird mit N geantwortet, so erscheint nur die Fehlernummer. Verzichtet man auf die vollständige Fehlernachricht, so gewinnt man ungefähr 1.5 KByte Hauptspeicherbereich.

Im grossen und ganzen erklären die Fehlerausschriften den aufgetretenen Verstoss gegen die Syntax der Sprache. Allerdings muss man beachten, dass der Fehler vor der Stelle liegen kann - und oftmals liegt - an der er angezeigt wird.
In den Fällen, in denen die Fehlermeldung nicht unmittelbar verständlich ist, sind in die folgende Zusammenstellung Bemerkungen eingefügt worden.
Die wichtigsten Vokablen zum Verständnis der Fehlernachrichten sollen vorangestellt werden:

allow	erlaubt, zugelassen	identifier	Bezeichner, Name
assignment	Zuweisung	match	übereinstimmen
bound	Grenze, Begrenzung	nested	verschachtelt
case selector type	Typ der Fallvariablen	overflow	Überlauf
create	erzeugen	overlay	Überlagerung
exceed	überschreiten	pointer	Zeiger
expect	erwarten	range	(Werte-) Bereich
expression	Ausdruck	subrange	Teilbereich

```
01  ';'      expected
02  ':'      expected
03  ','      expected
04  '('      expected
05  ')'      expected
06  '='      expected
```

```
07  ':='      expected
08  '['       expected
09  ']'       expected
10  '.'       expected
11  '..'      expected
12  begin     expected
13  do        expected
14  end       expected
15  of        expected
16  procedur or funktion expected
17  then      expected
18  to or downto expected

20  Boolean expression expected
     (logischer Ausdruck erwartet)
21  File variable expected
22  Integer constant expected
23  Integer expression expected
     (Ausdruck vom Typ Integer erwartet)
24  Integer variable expected
25  Integer or real constant expected
26  Integer or real expression expected
27  Integer or real variable expected
28  Pointer variable expected
29  Record variable expected
30  Simple type expected
     (INTEGER, BYTE, BOOLEAN, CHAR, Teilbereichstyp, Skalartyp)
31  Simple expression expected
     (Ausdruck darf Vergleichoperatoren nicht enthalten.)
32  String constant expected
33  String expression expected
34  String variable expected
35  Textfile expected
36  Type identifier expected
37  Untyped file expected
40  Undefined label
41  Unknown identifier or syntax error
     (Es ist ein nicht definierter Bezeichner gefunden worden oder
     einnicht zu präzisierender Verstoss  gegen die Syntax.
     Dieser Fehler tritt z.B. auf, wenn innerhalb  einer Zei-
     chenkette nur ein Apostroph geschrieben wurde.)
42  Undefined pointer type in preceding type definitions
43  Displicate identifier or label
     (Bezeichner oder Marke mehrfach definiert.)
44  Tvpe mismatch
     (Dieser Fehler tritt in folgenden Fällen auf:
      1. Typen der linken und rechten Seite der Wertzuweisung sind
         nicht verträglich.
```

2. Typ eines Indexausdrucks ist nicht verträglich mit dem Indextyp in der Array-Vereinbarung.
3. Die Typen der Operanden eines Ausdrucks sind nicht verträglich. Dieser Fehler tritt z.B. dann auf, wenn Vergleichsausdrücke durch logische Operatoren verknüpft werden und nicht beachtet wurde, dass die logischen Operatoren stärker binden als die Vergleichsoperatoren.
4. Typ des aktuellen Parameters ist nicht verträglich mit dem korrespondierenden formalen Parameter.)

45 Constant out of range
46 Constant and case selector type does not match
47 Operand type(s) does not match operator
48 Invalid result type
(Ergebnistyp von Funktionen sind alle Aufzählungstypen, REAL, Stringtypen und Pointertypen.)
49 Invalid string length
(Die Länge von Zeichenketten ist auf 255 beschränkt.)
50 String constant length does not match type
51 Invalid subrange base type
(Teilbereichstypen können nur von Aufzählungstypen gebildet werden.)
52 Lower bound > upper bound
(In Teilbereichstypen muss der Wert der unteren Grenze kleiner oder gleich dem Wert der oberen Grenze sein.)
53 Reserved word
(Ein Wortsymbol der Basissymbole von TURBO-PASCAL wird als Bezeichner verwendet.)
54 Illegal assignment
55 String constant exceeds line
(Eine Zeichenkettenkonstante darf ein end-of-line-Zeichen nicht enthalten.)
56 Error in integer constant
(Konstante ist entweder syntaktisch falsch oder liegt nicht in dem Zahlenbereich des Rahmens.)
57 Error in real constant
(vgl. Bemerkung zu Fehler 56)
58 Illegal character in identifier
60 Constants are not allowed here
(z.B. auf linken Seiten von Wertzuweisungen oder als aktuelle Parameter auf der Position von Referenzparametern.)
61 Files and pointers are not allowed here
62 Structured variables are not allowed here
63 Textfiles are not allowed here
64 Textfiles and untyped files are not allowed here
65 Untyped files are not allowed here
66 I/O not allowed here
(In der Eingabeliste/Ausgabeliste treten Variablen oder Ausdrücke auf, die einen Typ haben, für den die Ein-

gabe/Ausgabe-Operation nicht erklärt ist.)

67 Files must be var parameters

68 File components may not be files

69 Invalid ordering of fields

70 Set base type out of range
(Als Basistyp von Mengen sind nur solche Aufzählungstypen erlaubt, die dem Teilbereichstyp m..n entsprechen, wobei m>=0 und n<=255 ist.)

71 Invalid goto

72 Label not within current block
(Sprunganweisungen dürfen nicht aus dem Block, in dem sie kodiert sind, hinausführen.)

73 Undefined forward procedure(s)
(Der Block einer mittels forward deklarierten Routine fehlt.)

74 Inline error

75 Illegal use of absolute
(Dieser Fehler tritt in folgenden Fällen auf:
1. Es treten mehrere Bezeichner in der Deklaration auf.
2. Das Attribut absolute wurde innerhalb eines Record-Typs benutzt.)

76 Overlays can not be forwarded

77 Overlays not in direct mode
(Bei Programmübersetzung mit der Compileroption M sind Überlagerungsstrukturen nicht möglich.)

90 File not found
(Der Fehler tritt beim Einschluss von Quelltexten mit der I-Direktive auf.)

91 Unexpected end of source
(Das physische Ende des Programms tritt vor dem logischem Ende auf; begin-end-Klammerung fehlerhaft.)

92 Unable to create overlay file

93 Invalid compiler directive

97 Too many nested withs
(Standardmässig dürfen nur zwei WITH-Anweisungen verschachtelt sein, sonst ist die W-Direktive erforderlich.)

98 Memory overflow
(Es wird mehr Speicherplatz für Variablen angefordert, als in der Anlage zur Verfügung steht. Abhilfe - wenn überhaupt möglich - durch Umstrukturierung des Quelltextes.)

99 Compiler overflow
(Abhilfe: Zerlegen des Quellprogramms und Zusammenfügen der Teile mit der I-Compilerdirektive.)

E Fehlernachrichten in der Laufzeit

Während der Abarbeitung eines korrekt übersetzten Programms können Fehler auftreten, die zum Abbruch des Programms führen. In einem solchen Falle wird die Nummer eines Laufzeitfehlers angegeben und die Adresse des Befehls, der den Abbruch verursachte (program counter address=PC).
Informationen über die Ursache des Abbruchs enthält die folgende Zusammenstellung.
Durch Drücken der ESCAPE-Taste (oder ^[) geht das TURBO-PASCAL-System in den Edierzustand, es wird die Quelltextstelle angezeigt, die den Fehler verursachte.
Die Bedeutung der Fehlernummern ist

01 Gleitkommaüberlauf.
02 Division durch 0.
03 Argument der Standardfunktion SQRT negativ.
04 Argument der Standardfunktion LN negativ oder 0.
10 Fehlerhafte Länge eines String-Ausdrucks.
1. String-Länge > 255 oder
2. Nur Zeichenketten der Länge 1 können in CHAR-Werte umgewandelt werden.
11 Index bei Zugriff zu einer String-Variablen fehlerhaft.
Der Fehler kann bei Aufruf der Prozeduren COPY, DELETE, und INSERT auftreten.
90 Index nicht im Wertebereich.
91 In Wertzuweisung ist der zuzuweisende Wert nicht im Definitionsbereich der Zielvariablen.
92 INTEGER-Wert liegt nicht im zulässigen Zahlenbereich.
Der Fehler tritt z.B. auf, wenn die Standardfunktion ROUND aufgerufen wird.
F0 Überlagerungsfile nicht gefunden.
FF Speicheranforderung an Halde (heaps) oder Rekursionskeller (recursion stack) kann nicht mehr befriedigt werden.

Die im Zusammenhang it Eingabe/Ausgabe-Aktivitäten auftretenden Fehler sind:

01 File existiert nicht.
02 File wurde nicht für die Eingabe eröffnet.
03 File wurde nicht für die Ausgabe eröffnet.
04 File wurde nicht eröffnet.
Der Fehler tritt beim Zugriff mit BLOCKREAD oder BLOCKWRITE auf.

10 Numerisches Format fehlerhaft.
20 Die Operation ist für logische Geräte nicht erklärt.
21 Operation nicht im Direktmode erlaubt.
EXECUTE und CHAIN können im Compilermodus M nicht abgearbeitet werden.
22 ASSIGN ist für Standardfiles nicht erlaubt.
90 Unverträgliche Record-Längen.
Die Komponentenlänge einer File-Variablen stimmt nicht mit der Komponentenlänge des aktuellen Files überein.
91 Suchen hinter File-Ende.
99 File-Ende nicht erwartet.
F0 Fehler beim Schreiben auf Diskette.
F1 Disketten-Fileverzeichnis voll.
F2 Zulässige File-Grösse überschritten.
F3 Zu viele eröffnete Files.
FF File ist verschwunden.
Fehlerursache kann Diskettenwechsel sein.

F Zusammenstellung der Abweichungen von Standard-PASCAL

1. Einschränkungen gegenüber Standard-PASCAL

(1) Die Standardprozeduren GET und PUT gibt es nicht. Dafür sind READ und WRITE so erweitert worden, dass die Funktionen von GET und PUT ausgeführt werden können.

(2) Die implizit mit den File-Variablen eingeführten Puffervariablen gibt es nicht. Quelle bzw. Ziel von Datenübertragungen müssen in der WRITE- bzw. READ-Anweisung angegeben werden.

(3) Die Sprunganweisung GOTO darf nicht aus dem kleinsten hinausführen, in dem sie benutzt wird. Es ist also nicht möglich, eine Prozedur durch eine Sprunganweisung zu verlassen.

(4) Die in der Standardprozedur NEW verlangte Pointervariable darf nicht an einen Record-Typ mit Variantenteil gebunden sein.

(5) Auf der Position von formalen Parametern sind Prozedurköpfe und Funktionsköpfe (formale Prozeduren bzw. formale Funktionen) verboten.

(6) Es ist gestattet, Routinen rekursiv zu definieren, jedoch sind lokale Variablen beim rekursiven Aufruf nicht auf der Position von Referenzparametern zu benutzen. Man beachte, dass Wertparameter wie lokale Variablen behandelt werden.

(7) Die Standardprozedur PAGE gibt es nicht (das Betriebssystem CP/M verwendet kein Form-feed-Zeichen).

(8) Die Standardprozeduren PACK und UNPACK sind nicht implementiert. Das Basissymbol **packed** wird ignoriert. Packen wird vom System automatisch durchgeführt.

2.Erweiterungen von Standard-PASCAL

Während man einerseits die Einschränkungen von Standard-PASCAL als unerheblich betrachten kann, sind andererseits die Erweiterungen, die in TURBO-PASCAL vorgenommen wurden, erheblich. Sie werden hier nur mit Verweisen auf Textstellen zusammengestellt.

(1) Die Standardtypen sind durch BYTE und **string** erweitert. Mit Hilfe des String-Typs ist ein flexiblerer Umgang mit Zeichenketten möglich als mit dem Typ **array of** CHAR. Für die String-Verarbeitung wird eine Reihe von Standardprozeduren und Standardfunktionen angeboten (vgl. Abschn. 2.7.).

(2) Der logische Operator **xor** ist hinzugekommen. Logische Operatoren sind auch auf Ausdrücke vom Typ INTEGER (oder Teilbereiche davon) anwendbar (vgl. Abschn. 2.6.2.).

(3) Steuerzeichen des ASCII dürfen in Quellprogrammen benutzt

werden. Sie werden nicht in Apostrophe eingeschlossen. Folgen von Steuerzeichen sind zulässig (vgl. Abschn. 2.7.).

(4) Es ist möglich, einfache und strukturierte Variablen bei ihrer Deklaration mit Anfangswerten zu belegen (Typkonstanten, vgl. Abschn. 2.10.).

(5) Die numerischen Standardfunktionen sind erweitert (vgl. Abschn. 2.6.1. und Anhang C).

(6) Typbezeichner für einfache Typen können als Funktionsbezeichner zur Wertkonvertierung benutzt werden (vgl. Abschn. 2.9.6.).

(7) Die strenge Reihenfolge der Teile des Vereinbarungsteils in PASCAL wird in TURBO-PASCAL aufgehoben (vgl. Abschn. 2.10.).

(8) Die Anzahl der vordefinierten Textfiles wurde erweitert (vgl. Abschn. 2.11.2.).

(9) Es sind Sprachelemente für die Fälle der Filemanipulation vorgesehen, bei denen es auf den Komponententyp des Files nicht ankommt (vgl. Abschnitte 2.11.3. und 2.11.4.).

(10) Es gibt unterschiedliche Möglichkeiten für den direkten Zugriff zum Hauptspeicher (vgl. Abschnitte 2.13. und 2.14.8.).

(11) Die CASE-Anweisung ist durch einen ELSE-Teil erweitert worden (vgl. Abschn. 2.8.5.).

(12) Es ist möglich, Programmbausteine in Assemblersprache zu kodieren (vgl. Abschn. 2.14.8.).

(13) Selbständig übersetzte Programme können von Programmen aufgerufen werden. Parametervermittlung ist über absolute Adressen möglich (vgl. Abschn. 2.14.7.).

(14) Der Aufbau von Überlagerungsstrukturen ist möglich (vgl. Abschn. 2.14.7.).

(15) Systemrufe sind vom Programm aus möglich (vgl. Abschn. 2.14.4.).

(16) Es gibt Standardprozeduren für die Bildschirmsteuerung (vgl. Abschn. 3.).

(17) Es gibt einen Zufallszahlengenerator.

G Edierkommandos

^KD beendet die Arbeit mit dem Editor.
Unter "Wort" wird eine Zeichenfolge verstanden, die durch eines der folgenden Zeichen begrenzt wird:
Leerzeichen , . ; ' ^ ( ) [ ] + - * / ¤ < >

1.Positionierung des Kursors

Kursorbewegung	Steuerzeichen	Kursorbewegung	Steuerzeichen
Eine Zeile nach oben	^E	Zeilenanfang	^QS
Eine Zeile nach unten	^X	Zeilenende	^QD
Ein Zeichen nach links	^S	Bildschirmanfang	^QF
Ein Zeichen nach rechts	^D	Fileanfang	^QR
Seitenanfang davor	^R	Fileende	^QC
Seitenanfang danach	^C	Blockanfang	^QB
Ein Wort nach links	^A	Blockende	^QK
Ein Wort nach rechts	^F	Letzte Kursorposition	^QP
Rückwärtsrollen des Bildschirms	^W		
Vorwärtsrollen des Bildschirms	^Z		

2.Einfügen und Löschen

Wirkung	Steuerzeichen	Wirkung	Steuerzeichen
Schalter für Einfügemodus	^V	Löschen des linken Zeichens	DEL
Einfügen Zeilenendezeichen	^N	Löschen des Zeichens auf Kursorposition	^G
		Löschen des rechten Wortes	^T
		Löschen von Kursor bis Zeilenende	^QY
		Löschen der Zeile	^Y

3. Arbeit mit Blöcken

Wirkung	
Blockbeginnmarke setzen/löschen	^KB
Blockendemarke setzen/löschen	^KK
Blockmarkierungsanzeige ein/aus	^KH

Verschieben eines Blockes auf die Kursorposition	^KV
Einfügen der Kopie eines Blockes an Kursorposition	^KC
Löschen eines Blockes	^KY
Einsetzen eines Files von Diskette an Kursorposition (Filename vollständig angeben)	^KR
Speichern eines Blockes als File auf Diskette	^KW

4. Spezielle Edieranweisungen

Wirkung	
Abbruch eines Kommandos bei Datenanforderung	^U
Tabulatorposition an Kursorstelle fixieren	TAB/^
Automatische Zeilenverzahnung schalten (indent)	^QI
Rückspeichern des unveränderten Zeileninhalts	^QL
Suchen einer Zeichenfolge	^QF

Folgende Optionen können zusätzlich genannt werden:

- B Rückwärtssuchen
- U Kleinbuchstaben und Grossbuchstaben werden nicht unterschieden
- W Nur vollständige Wörter suchen

Suchen einer Zeichenfolge und Ersetzen ^QA

Folgende Optionen können zusätzlich genannt werden:

- B Suchen und Ersetzen erfolgen von der Kursorposition zum Fileanfang
- G Suchen und Ersetzen von Fileanfang bis Fileende. Das Ersetzen erfolgt nur nach Bestätigung über die Eingabetastatur (y eingeben)
- N Ersetzen erfolgt ohne Anfrage und Bestätigung
- U Kleinbuchstaben ud Grossbuchstaben werden nicht unterschieden
- W Nur vollständige Wörter werden gesucht und ersetzt

Ausserdem kann durch Anfügen einer positiven Zahl festgelegt werden, wie oft das Ersetzen einer Zeichenfolge ausgeführt werden soll.

Beispiele:

GN bedeutet, dass die Ersetzung an jeder Stelle im File erfolgen soll.

N5 bedeutet, dass ab Kursorposition die nächsten 5 Zeichenfolgen ohne Rückfrage ersetzt werden sollen, die als Suchkennzeichen angegeben wurden.

Wiederholen des letzten ^QF- oder ^QA-Kommandos	^L
Einfügen von Steuerzeichen (Nach dem Eintasten von ^P das Steuerzeichen eintasten.)	^P

H Beispiel

```
program Fettdruck {von Basissymbolen};
const bufsize  = 80;
      kommentar=' kann nicht eroeffnet werden';
      controlb =^B;
type  string127=string[127];
var   filename      :  string[14];
      a             string127;
      fin           file;
      fout          TEXT;
      rest,
      blocknr       INTEGER;
      c             CHAR;
      res,
      letter,
      digit,
      eofile    :   BOOLEAN;
      buffer    :   array[1..bufsize,1..128]of CHAR;
procedure work(var buff0;size:INTEGER);
var buffer:array[0..20480]of CHAR absolute buff0;
    pos   :INTEGER;
function reserviert(a:string127):BOOLEAN;
const inum =44;
      symbols  :array[1..inum]of string[10]= (
                  {reservierte Worte von Turbo-Pascal}
        'ABSOLUTE',     'AND',          'ARRAY',      'BEGIN',
        'CASE',         'CONST',        'DIV',        'DO',
        'DOWNTO',       'ELSE',         'END',        'EXTERNAL',
        'FILE',         'FOR',          'FORWARD',    'FUNCTION',
        'GOTO',         'IF',           'IN',         'INLINE',
        'LABEL',        'MOD',          'NIL',        'NOT',
        'OF',           'OR',           'OVERLAY',    'PACKED',
        'PROCEDURE',    'PROGRAM',      'RECORD',     'REPEAT',
        'SET',          'SHL',          'SHR',        'STRING',
        'THEN',         'TO',           'TYPE',       'UNTIL',
        'VAR',          'WHILE',        'WITH',       'XOR');
var   res  :BOOLEAN;
      i    :INTEGER;
procedure upstring(var s);
begin
  inline(¤2A/s/             {_LD   HL,<S>}
         ¤46/               { LD   B,<HL>}  {a[0] in B}
         ¤23/               { INC  HL    }
         ¤7E/               { LD   A,<HL>}
         ¤FE/¤61/           { CP   'a'   }
```

```
        ¤38/¤07/          { JR   C,9    }
        ¤FE/¤7B/          { CP   'z'+1  }
        ¤30/¤03/          { JR   NC,5   }
        ¤D6/¤20/          { SUB  20H    }
        ¤77/              { LD   <HL>,A }
        ¤10/¤F1)          { DJNZ -13    }  {wird mit B gesteuert}
end;
begin
  upstring(a); {aktueller Parameter entspricht globalem a}
  res:=FALSE;
  i:=1;
  repeat
    res:= a=symbols[i];
    i:=i+1
  until (i>inum) or res;
  reserviert:=res
end; {reserviert}
begin (* work *)
  pos:=0;
  while not eofile and (pos shr 7 < size) do
    begin
      c:=buffer[pos];
      letter:=  c in ['A'..'Z', 'a'    'z'];
      digit:= c in ['0'    '9'];
      if a<>'' then
        begin
          if letter or digit then a:=a+c
          else
            begin
              res:=reserviert(a);
              if not res then WRITE (fout,a)
                         else WRITE(fout,
                                    controlb+a+controlb);
              a:='';{loeschen}
              WRITE(fout,c)
            end
        end
              else
        if letter then
          begin
            a:=c;
          end
        else
          WRITE(fout,c);
      eofile:= c=^Z;
      pos:=succ(pos);
      if pos and ¤7F = 0 then WRITE('*');{zum Verfolgen der Arbeit
                                          auf dem Bildschirm}
```

```
    end;
end;   (* work *)

begin
  CLRSCR;
  repeat
    WRITELN;
    WRITE('Name der Quelldatei oder <CR> fuer Ende:');
    READLN(filename);
    if filename<>'' then
      begin
        ASSIGN(fin,filename);                          {¤i- }
        RESET(fin);                                    {¤i+ }
        if IORESULT<>0 then WRITELN(filename,kommentar)
        else
          begin
            WRITE('Name der Zieldatei:');
            READLN(filename);
            ASSIGN(fout,filename);                     {¤i- }
            REWRITE(fout);                             {¤i+ }
            if IORESULT<>0 then WRITELN(filename,kommentar)
            else
              begin
                rest:=FILESIZE(fin);
                WRITELN('Filesize:',rest,' Bloecke zu 128 Byte');
                a:=''; {Initialisierung}
                eofile:=FALSE;
                res:=TRUE;
                while rest>0 do
                  begin
                    blocknr:=rest;
                    if blocknr>=bufsize then blocknr:=bufsize;
                    BLOCKREAD(fin,buffer,blocknr);
                    work(buffer,blocknr);
                    rest:=rest-blocknr
                  end
              end;
            CLOSE(fout)
          end;
        CLOSE(fin)
      end
  until filename=''
end.
```

Literaturverzeichnis

[1] Wirth, N.: Revidierter Bericht über die Programmiersprache PASCAL. Berlin: Akademie-Verlag 1976
[2] Paulin, G., Schiemangk,H.: Programmieren mit PASCAL. 2. Auflage. Berlin: Akademie-Verlag 1986
[3] Borland Int. (Herausgeber): Turbo Pascal. Version 3.0. Reference Manual. Scotts Valley, California: 1985
[4] Joepgen, H.-G.: TURBO-PASCAL. Das Kompendium für die Programmierpraxis. München,Wien: Carl Hanser Verlag 1985
[5] Plate, J.: Betriebssystem CP/M. Vom Monitorprogramm zum Mehrbenutzersystem. München: Franzis-Verlag 1984
[6] Classen, L., Oefler, U.: Wissensspeicher Mikroprogrammierung. 2. Auflage. Berlin: VEB Verlag Technik 1986
[7] Wirth, N.: A Collection of PASCAL Programs. Berichte des Instituts für Informatik der ETH Zürich 1979
[8] Bernert, J., Hanisch, C., Burow, M.: CP/M in der Praxis. Berlin: Verlag Technik 1988
[9] CHIP - Sonderheft. Anwenderprogramme in TURBO-PASCAL, 1985
[10] Systemunterlagendokumentation MOS K1520. Anwenderdokumentation. VEB Robotron-Buchungsmaschinenwerk Karl-Marx-Stadt 1984

Sachwörterverzeichnis

TURBO-PASCAL ist ein leistungsfähiges System zum Entwickeln von Programmen. Kern des Systems ist eine erheblich erweiterte Version der Sprache PASCAL. Zur Quelltextbereitstellung dient eine adaptierte Fassung des Textverarbeitungssystems WordStar. Das Auffinden und Korrigieren syntaktischer und logischer Fehler wird durch Rückkehr zum Quelltext unterstützt.
Neben BASIC ist TURBO-PASCAL die am häufigsten verwendete Programmiersprache für die Entwicklung von Anwenderprogrammen für unterschiedliche Rechnerarchitekturen.

ISBN 3-341-00521-8
ISSN 0863-0860 Technische Informatik