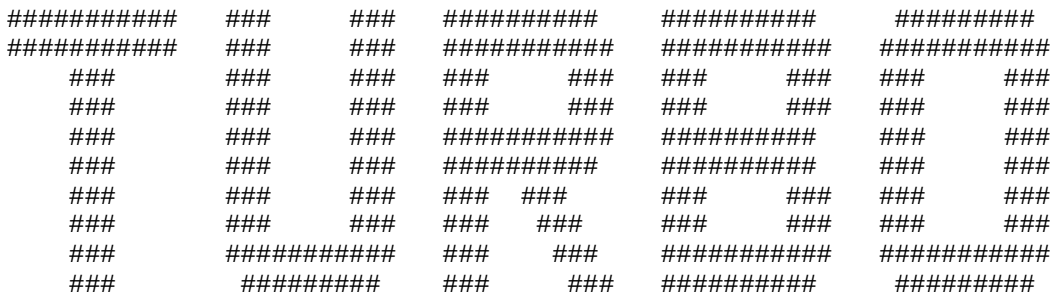
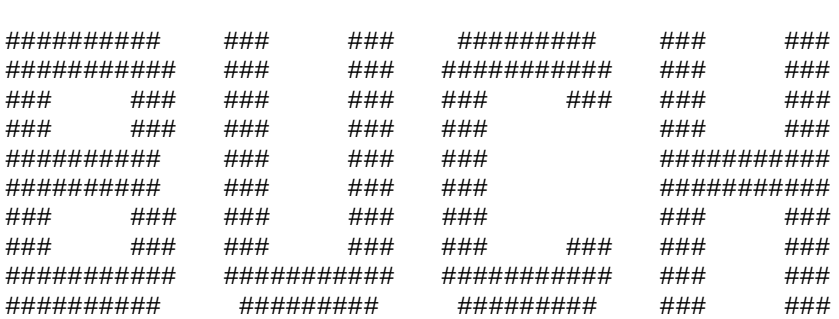
KARL-HERMANN ROLLKE





Karl-Hermann Rollke

DAS TURBO PASCAL BUCH

VEB Robotron - Vertrieb - Berlin
Werk Magdeburg
1986

Inhaltsverzeichnis

VORWORT

Der Kluge bemueht sich, alles richtig zu machen.
Der Weise bemueht sich, so wenig wie moeglich
falsch zu machen.

(Persisches Sprichwort)

Turbo Pascal ist ein neuer Pascal-Dialekt, der einige Vorzuege gegenueber seinen Konkurrenten aufzuweisen hat. Turbo Pascal ist schnell, und es ist komfortabel.
Hervorstechendes Merkmal beim Arbeiten mit Turbo Pascal sind die kurzen Zykluszeiten bei der Programmentwicklung. Das bedeutet, dass einerseits der Compiler die Textprogramme recht schnell uebersetzt und andererseits wenig Zeit benoetigt wird, um bei Auftreten eines Fehlers wieder in den Programmtext zu kommen, damit der Fehler korrigiert werden kann. Beim Programmieren schleichen sich recht haeufig Fehler ein, so dass der Anwender die Schnelligkeit des Turbo-Systems schaetzen lernt. Allerdings sollte der Anfaenger davor gewarnt werden, sogenannte Instant-Programme ohne vorherige Planung am Bildschirm zu entwickeln.
Gerade bei einem so schnellen System ist die Versuchung gross, etwas schlampig zu programmieren und z.B. fehlende Deklarationern erst spaeter einzufuegen.
Turbo Pascal ist auch in der Ausfuehrung von Programmen recht flott, und zwar deshalb, weil es direkt in die Maschinensprache des entsprechenden Prozessors uebersetzt wird und nicht in einem Zwischencode.
Die Arbeit des Programmerstellens und Uebersetzens wird auch durch keinen Linker beeintraechtigt: Die uebersetzten Programme sind sofort lauffaehig.
Mit Turbo Pascal steht dem Benutzer ein sehr komfortables Programmierwerkzeug zur Verfuegung. Es umfasst deutlich mehr Standardfunktionen und -prozeduren als Standard-Pascal und die meisten anderen Dialekte. Ausserdem arbeitet es mit einer grossen Rechengenauigkeit
und kennt einige zusaetzliche Typen und Konstanten.
Turbo Pascal ist auf allen CP/M-Rechnern lauffaehig sowie unter MS-DOS. Daher sind die Programme weitgehend ohne Veraenderungen uebertragbar, und Besonderheiten des Rechners spielen keine Rolle.
Allein kleine Unterschiede zwischen 8- und 16-Bit-Rechnern machen sich in den Sprachteilen bemerkbar, die besondere Kontakte zur tatsaechlichen Maschine herstellen.
Auf Besonderheiten bestimmter Rechnermodelle, wie insbesondere auf Grafikfaehigkeit, wurde im vorliegenden Buch ganz verzichtet, da die Allgemeingueltigkeit und die Transportabilitaet der Programme darunter leiden wuerde. Hier muss auf die Angebote des Softwaremarktes und auf vielfaeltige Artikel in der Fachpresse hingewiesen werden.
Alle Programme sind auf einem Rechner unter CP/M getestet worden und damit lauffaehig.
Das Buch ist als umfassende Programmieranleitung geschrieben und bietet einen Einstieg in Turbo Pascal fuer Anfaenger wie Umsteiger. Es bleibt jedoch nicht auf der Anfaengerebene, sondern behandelt alle Themen. Es ist bewusst enzyklopaedisch gehalten und der Sprache Pascal gemaess formal aufgebaut, um auch dem fortgeschrittenen Benutzer einen raschen Ueberblick und ein schnelles Finden bestimmter Themen zu ermoeglichen.

Ein besonderes Kapitel ist der Programmierhilfe Turbo Toolkit gewidmet, die vom gleichen Hersteller angeboten wird. Da davon auszugehen ist, dass der Turbo Pascal-Benutzer dieses komfortable und leistungsfaehige Software-Werkzeug frueher oder spaeter benutzen wird, soll es nicht unerwaehnt bleiben.
Der Anhang des Buches stellt eine kurze Uebersicht ueber die wichtigsten Begriffe, Befehle und Zusammenhaenge dar, die der Benutzer dieser Sprache staendig braucht.
Ich moechte an dieser Stelle dem Lektorat des Verlages und insbesondere Frau Wentges fuer die Ueberarbeitung des Buches und Syyylvia Krause und Suse Korn fuer die Bearbeitung danken.

Karl-Hermann Rollke

Unna, im September 1985

EINFUEHRUNG

1.1 Turbo Pascal im CP/M-Betriebssystem

Bevor wir uns naeher mit dem Turbo Pascal beschaeftigen, muessen wir einige grundlegende Dinge ueber das CP/M-Betriebssystem wissen. Hier sollen nur die wichtigsten Operationen in diesem Betriebssystem kurz beschrieben werden, damit der Einsteiger sie benutzen kann. Weitere Informationen sind dem CP/M-Handbuch oder der einschlaegigen Literatur zu entnehmen.
Die CP/M-Diskette mit dem Turbo-System wird in das Laufwerk gelegt, und der Rechner wird eingeschaltet. Nun laedt der Rechner das CP/M-Betriebssystem. Dieser Vorgang wird "Booten" genannt. Nach dem Booten erscheint das Copyright und ein ">"-Zeichen als Promptsymbol.

Es ist nuetzlich, die folgenden Befehle zu kennen:

- DIR <CR>: Listet das Inhaltsverzeichnis der Diskette (Directory) auf.
- A:<CR> oder B:<CR>: Schaltet auf ein anderes Laufwerk um. Die Laufwerke haben die Namen A:, B:, C: usw.
- ERA <Filename>: Loescht eine Datei (File) mit angegebenem Namen auf der Diskette.
- REN <Filename neu> = <Filename alt>: Benennt ein File um.

Auf der CP/M-Systemdiskette befinden sich drei unbedingt notwendige Programme, die wir sofort benutzen sollten. Damit die kostbare Turbo-Systemdiskette nicht versehentlich geloescht oder beschaedigt wird, legen wir als erstes eine Kopie an und bewahren das Original danach sicher auf. Dazu muessen wir zunaechst eine Diskette formatieren.

Wir legen die CP/M-Systemdiskette ins Laufwerk und tippen:

- FORMAT A: <CR>: Nun erscheint eine Begruessungszeile des Formatters, und wir muessen die Systemdiskette in Laufwerk A: durch die neue Diskette ersetzen. Dann druecken wir die <CR>-Taste, und die Diskette wird formatiert. Nach dem erfolgreichen Formatieren werden wir aufgefordert, die Systemdiskette wieder einzulegen und <CR> zu druecken.

Nun kopieren wir die Turbo-Systemdiskette. Mit der CP/M-Diskette in Laufwerk A: tippen wir:

- COPY B:=A:<CR>: Nun fordert uns das Kopierprogramm auf, das Original (d.h. die Turbo-Diskette) in Laufwerk A: zu legen und die Kopie (d.h. unsere neue Diskette) in Laufwerk B:. Mit <CR> wird der Kopiervorgang gestartet. Nach Beendigung des Kopierens werden wir gefragt, ob wir noch eine Kopie machen wollen. Mit N verlassen wir das Kopierprogramm.

  Hinweis: Manchmal ist es noetig, nur das Betriebssystem zu kopieren (weil z.B. eine besondere Druckeranpassung noetig ist). Dann tippen wir COPY

B:=A:/S <CR>. Alles andere ist dann wie oben.

Sollen nur bestimmte Files (Programme, Daten usw.) kopiert werden, so benutzen wir das PIP-Programm auf der CP/M-Systemdiskette.

- PIP <CR>: Bringt uns in das File-Kopierprogramm. Ein anderes Promptzeichen (*) erscheint auf dem Bildschirm. Mit folgender Zeile koennen wir nun ein File von einem Laufwerk auf ein anderes kopieren: B:<File neu>=A:<File alt><CR>. So wird ein File vom Laufwerk A: mit dem Namen <File alt> auf die Diskette im Laufwerk B: mit dem Namen <File neu> kopiert.
  Es gibt auch Abkuerzungen fuer die Filenamen, auf die hier aber nicht eingeganen werden soll (Siehe CP/M-Handbuch).
  Das PIP-Programm kopieren wir z.B. von Laufwerk A: nach Laufwerk B: mit der Anweisung B:PIP.COM=A:PIP.COM <CR>.
  Beendet wird PIP einfach mit <CR>.

Nun zu den Arten der Datein (Files) mit denen wir es in Verbindung mit Turbo Pascal zu tun haben. Alle Files enden mit dem Anhaengsel (Suffix), bestehend aus (meistens) 3 Zeichen angefuehrt durch einen Punkt.

Die wichtigsten Typen sind:

- .COM oder .CMD: Lauffaehiges Programm in Maschinencode
- .PAS : Pascal-Programmtext (Textfile)
- .BAK : Sicherheitskopie des Workfiles
- .OVR : Overlay
- .DTA : Daten

Oft ist auf den Disketten ein File mit dem Namen READ.ME zu finden. Dieses ist genauso zu behandeln, wie es heisst. Lesen Sie es. Dazu tippen Sie einfach

```
TYPE READ.ME <CR>
```

und dieser Text erscheint auf dem Bildschirm.

Kommen wir zum eigentlichen Thema: Turbo Pascal. Nachdem wir eine Sicherheitskopie der Originaldiskette gemacht haben, legen wir das Original sicher weg und starten die kopierte Turbo-Systemdiskette. Mit TURBO <CR> waehlen wir das Turbo-System.

Auf dem Bildschirm erscheint ein Titel mit dem Copyright und die Frage

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

In der Regel werden wir hier Y tippen, um die Fehlermeldungen im Klartext statt als Fehlercode zu bekommen.

Danach erscheint ein Menue:

```
Logged drive: A

Work file:
Main file:

Edit    Compile Run  Save
eXecute Dir     Quit compiler Options
```

```
  Text:     0 bytes
  Free: 62903 bytes
```

Im folgenden Kapitel werden die Moeglichkeiten des Menues naeher betrachtet. Hier wollen wir uns nur einmal anschauen, was alles zum Turbo-System an Files dazu gehoert. Mit D fuer Dir erhalten wir das Inhaltsverzeichnis der Diskette.

Die folgenden Files sollte die Diskette enthalten:

- TURBO.COM : Das Turbo Pascal.
- TURBO.OVR : Overlays, die noetig sind, wenn vom Turbo aus ein .COM-File gestartet werden soll.
- TURBO.MSG : Das Textfile, das die Fehlermeldungen enthaelt.
- TLIST.COM : Ein Programm zum eleganten Ausdrucken von Turbo-Programmtexten.
- TINST.DTA : Daten zu TINST.COM.
- READ.ME : Hinweise zum System.
- <Name>.PAS: Eventuell vorhandene Pascal-Programme.

Welche Teile umfasst das Turbo-System?

- Editor : Zum Schreiben und Veraendern von Texten (insbesondere Programmtexten).
- Compiler : Zum Uebersetzen der Programmtexte in lauffaehige Programme (in Maschinensprache).

Einen mit UCSD vergleichbaren Filer (zum Behandeln von Dateien und Disketten) stellt das CP/M-Betriebssystem selbst dar.

1.2 Das Menue

Mit TURBO <CR> waehlen wird das Turbo-System. Auf dem Bildschirm erscheint das Menue. Die Auswahlmoeglichkeiten des Menues werden durch einfaches Tippen der entsprechenden Buchstabentaste angewaehlt. Die Buchstaben, die das Wort abkuerzen, sind in der Bildschirmdarstellung hervorgehoben.

```
   Logged drive: A

   Work file:
   Main file:

   Edit    Compile Run  Save
   eXecute Dir     Quit compiler Options

   Text:     0 bytes
   Free: 62903 bytes
```

Die einzelnen Zeilen des Menues haben folgende Bedeutung:

- Logged drive: A
  Hier wird angegeben, auf welches Laufwerk standardmaessig zugegriffen werden soll. Durch druecken der L-Taste kann ein anderes Laufwerk gewaehlt werden.

- Work file:
  In dieser Zeile wird angezeigt, welches Workfile gerade bearbeitet wird, d.h. mit welchem Programmtext der Benutzer beschaeftigt ist. Durch Druecken der W-Taste kann der Benutzer ein Workfile in folgender Form definieren:
  <Filename>.<Typ>

  Beispiel: TEST.PAS
  Es kann auch .PAS weggelassen werden. In diesem Fall wird es vom System angefuegt. Wird jedoch z.B. TEST.BAK gewaehlt, so wird nichts angefuegt. Ist das Workfile nicht auf der Diskette, wird ein neues File erzeugt. Wenn ein Workfile im Rechner ist und der Benutzer einen neuen Namen als Workfile definiert, wird gefragt, ob das Workfile erst gespeichert werden soll (Antwort Y oder N).

- Main file:
  Wenn neben einem Workfile noch ein Mainfile genannt wird, ist es dieser Text, der vom Compiler uebersetzt wird (andernfalls das Workfile). So kann ein Programmtext benannt werden, der z.B. auf der Diskette steht und das zu uebersetzende Hauptprogramm enthaelt, waehrend nebenher ein anderer Text (der z.B. Textteile enthaelt, die eingefuegt werden sollen) im Texteditor bearbeitet wird.

- Edit
  Mit E wird der Editor aufgerufen (siehe folgendes Kapitel).

- Compile
  Mit C wird das Mainfile (oder das Workfile, wenn kein Mainfile existiert) in die Z80-Maschinensprache uebersetzt. Die Art der Uebersetzung haengt von den Compiler-Optionen ab.

- Compiler Options
  Mit O lassen sich Compiler-Optionen waehlen.
  * Memory
    Durch Druecken der M-Taste (das ist auch die Standardeinstellung) wird der Compiler veranlasst, den Text so zu uebersetzen, dass das lauffaehige Programm im Rechnerspeicher steht.

  * Com-file
    Mit der C-Taste koennen wir den Compiler veranlassen, das lauffaehige Programm als .COM-File (z.B. TEST.COM) auf der Diskette zu schreiben. Abhaengig vom Rechnertyp koennen noch Angaben ueber die genaue Plazierung des Maschinenprogramms im Speicher gemacht werden (siehe Handbuch).

  * cHn-file
    Mit der H-Taste sorgen wir dafuer, dass das Programm so uebersetzt wird, dass es nur von einem anderen Pascal-Programm aufgerufen werden kann (weil Teile des Pascal-Systems fehlen). Der Name wird durch .CHN (z.B. TEST.CHN) kenntlich gemacht (siehe ebenfalls Handbuch).

- Run
  Mit der R-Taste lassen wir den Rechner ein Programm abarbeiten. Entweder wird das Programm im Speicher oder (wenn die Compiler-Option C gewaehlt wurde) das entsprechende .COM-File abgearbeitet. Ist das Programm noch nicht uebersetzt, so wird dies erst getan.

- Save

Durch Druecken der S-Taste koennen wir das Workfile unter dem angegebenen Namen abspeichern. Es bleibt eine Kopie der letzten Version des Workfiles auf der Diskette mit dem Kuerzel .BAK (z.B. TEST.BAK).

- eXecute
  Mit der X-Taste koennen wir ein anderes Programm aufrufen.

- Dir
  Mit D wird das Inhaltsverzeichnis der angewaehlten Diskette ausgegeben.

- Quit
  Mit Q wird das Turbo-System verlassen (zurueck zum CP/M-Betriebssystem).

1.3 Der Turbo Editor

Der Turbo Editor ist ein Texteditor (verwandt mit dem bekannten Word Star), mit dem beliebige Texte bearbeitet werden koennen. Insbesondere werden wir natuerlich Programmtexte damit bearbeiten. Fuer WordStar-Benutzer entfaellt eine Einarbeitung voellig. Sie koennen diesen Editor (fast) wie gewohnt benutzen.
Neulinge auf diesem Gebiet sollten erst einmal zur Uebung mit dem Editor spielen, d.h. neue Texte erstellen und aendern sowie schon bestehende Texte aendern und in den Texten herumspringen.
Der Editor wird mit der Taste E aus dem Menue gewaehlt und durch Eingabe der Tastenfolge ^K^D verlassen

Hinweis: Die Bezeichnung ^K bedeutet, dass die Taste K zusammen mit der CTRL-Taste gedrueckt wird.

Hinweis fuer APPLE-Benutzer: Beim APPLE erzeugt ^K eine eckige Klammer. Daher ist fuer den APPLE jedes ^K durch ^O zu ersetzen (siehe auch Handbuch/Installationsanweisungen).

Die Editorbefehle lassen sich in folgende Gruppen aufteilen:
- Kontrollbefehle
- Cursorbewegungsbefehle
- Befehle zum Einfuehgen und Loeschen
- Blockbefehle
- Verschiedenes

1. Kontrollbefehle

- Beenden des Editors - ^K^D
- Einfuegen/Ueberschreiben - ^V
  Schaltet hin und her zwischen Einfuegen an der Cursorposition und Ueberschreiben an der Cursorposition.
- Automatischer Tabulator - ^Q^I
  Schaltet hin und her zwischen automatischem Tabulator (d.h. Sprung an die Spaltenposition in der folgenden Zeile an der die aktuelle Zeile beginnt) und Zeilenanfang am linken Rand.

2. Cursorbefehle

- Ein Zeichen nach links - ^S

- Ein Zeichen nach rechts - ^D
- Ein Zeichen nach oben - ^E
- Ein Zeichen nach unten - ^X

Hinweis: Die Cursortasten sind wie folgt angeordnet:

```
   E
S     D   (leicht zu merken)
   X
```

Neben diesen Zeichen sitzen die Zeichen zur wortweisen Rechts-/Linksbewegung und seitenweisen Auf-/Abbewegung.

- Ein Wort nach links - ^A
- Ein Wort nach rechts - ^F
- Eine Seite nach oben - ^R
- Eine Seite nach unten - ^C

Der ganze Bildschirm kann verschoben werden, ohne die Cursorposition zu aendern:

- Bildschirm eine Zeile nach oben - ^W
- Bildschirm eine Zeile nach unten - ^Z

Finden bestimmter Positionen im Text:

- Cursor in die oberste Bildschirmzeile - ^Q^E
- Cursor in die unterste Bildschirmzeile - ^Q^X
- Cursor an Textanfang - ^Q^R
- Cursor an Textende - ^Q^C
- Cursor an letzte Cursorposition - ^Q^P

3. Befehle zum Einfuegen und Loeschen

- Loeschen eines Zeichens links vom Cursor - DEL-Taste
- Loeschen eines Zeichens unter dem Cursor - ^G
- Loeschen eines Wortes rechts vom Cursor - ^T
- Loeschen einer Zeile - ^Y
- Loeschen bis Zeilenende - ^Q^Y
- Einfuegen einer Zeile - ^N

4. Blockbefehle

Unter einem Block ist ein zusammenhaengender Text zu verstehen, der durch eine Anfangs- und Endmarkierung (Block-Marker) gekennzeichnet ist. Ein solcher Block kann dann bewegt, geloescht oder von und nach Disketten kopiert werden.

- Anfangsmarkierung - ^K^B
- Endmarkierung - ^K^K
- Ein einzelnes Wort markieren - ^K^T
- Umschalten Blockanzeigen - ^K^H
  Schaltet zwischen normaler Darstellung und hervorgehobener Bildschirm-darstellng fuer den Block.

- Vorher markierter Block soll an Cursorposition kopiert werden - ^K^C
- Vorher markierter Block soll zu Cursorposition bewegt werden - ^K^V
- Loeschen eines Blocks - ^K^Y
- Block von Diskette lesen - ^K^R
- Block auf Diskette schreiben - ^K^W

5. Verschiedenes

- Tabulator an die Spalte des letzten Zeilenanfangs - ^I
- Urspruenglicher Inhalt der aktuellen Zeile - ^Q^L
- Abbruch eines Kommandos - ^U
- Eingabe eines "^"-Zeichens in den Text durch Voranstellen von - ^P
- Finden eines Strings bis 30 Zeichen Laenge - ^Q^F
  Der zu findene String wird eingegben und mit <CR> abgeschlossen (CTRL-Zeichen mit ^P einfuegen).
  Dann wird nach Option gefragt:
  -B: Rueckwaertssuchen ab Cursorposition (sonst vorwaerts).
  -G: Global suchen (d.h. im ganzen Text).
  -<Zahl> : Gibt an, das wievielte Auftreten des Textes gefunden werden soll.
  -U: Gross-/Kleinschreibung ignorieren.
  -W: Nur ganze Woerter finden (sonst auch Auftreten des Textteils in anderen Texten).

- Finden und Ersetzen - ^Q^A
  Wie finden. Mit zusaetzlicher Angabe des Ersatztextes. Optionen wie bei Finden. Zusaetzlich:
  - <Zahl> : Anzahl der zu ersetzenden Textmuster.
  - N: Ersetzen ohne Nachfrage (Y/N).
- Letzes Finden wiederholen - ^L

1.4 Ein einfaches Programm - der Compiler

Wir wollen nun ein einfaches Programm schreiben und uebersetzen sowie ausfuehren lassen, um zu erlernen, wie das Turbo Pascal-System zu benutzen ist.
Das Programm begruesst den Benutzer und gibt ihm an, wieviel Speicherplatz noch im Rechner vorhanden ist. MEMAVAIL ergibt unter CP/M 80 den freien Speicher in Byte (bei Werten ueber 32767 ein negativer Wert - in diesem Fall ist 65536.0 zu addieren). Unter CP/M 86 (MS-DOS) ergibt MEMAVAIL den freien Speicher in Paragraphen (ein Paragraph hat 16 Byte).
Nennen wie zunaechst das Workfile durch Druecken der Taste W aus dem Menue: SPEICHER. Dann waehlen wir den Editor (mit E aus dem Menue). Nun tippen wir den folgenden Text ein (jede Zeile wird mit <CR> abgeschlossen. Eingerueckt wird mit der Leertaste, ausgerueckt mit der BS-Taste oder mit **Pfeil-Links**).

```
PROGRAM Speicher;

VAR Mem : REAL;

BEGIN
```

```
    CLRSCR;
    WRITELN ('Guten Tag');
    WRITELN;
    WRITELN ('Der freie Speicher betraegt:');
    Mem:= MEMAVAIL;
    IF Mem < 0 THEN Mem := Mem + 65536.0;
    WRITELN (Mem : 5 : 0);
    WRITELN;
    WRITELN ('Ende mit <RETURN>');
    READLN
  END.
```

Nachdem der Text eingegeben ist, verlassen wir den Editor mit ^K^D. Mit S Speichern wir dieses Programm erst einmal ab. Nun gibt es zwei Moeglichkeiten, das Programm auszufuehren:

1. Mit R wird das Programm in den Rechnerspeicer uebersetzt und ausgefuehrt.

2. Mit O wird die Compiler-Option COM gewaehlt, dann das Compiler-Option-Menue verlassen und mit C das Programm auf die Disketten uebersetzt. Nun wird das Turbo-System verlassen (mit Q) und vom CP/M aus durch Eingabe des Namens SPEICHER das Programm (SPEICHER.COM) abgearbeitet.

Mit Version 2 ist offenbar der zur Verfuegung stehende freie Speicher erheblich groesser, weil auch der Speicher mit zu nutzen ist, den das Turbo-System in Version 1 einnimmt.

Wenn Fehler auftreten

Bei Eingabe eines Programms mit dem Editor koennen Fehler auftreten. Zwei Arten von Fehlern treten relativ haeufig auf: Syntax-Fehler (d.h. vergessenes Semikolon, falsch geschriebene reservierte Woerter usw.) oder Fehler im logischen Aufbau des Programms (z.B. Benutzung einer Variablen, die nicht erklaert wurde).
Beim Uebersetzen des Programms merkt und meldet der Compiler solche Fehler. Durch Druecken der ESC-Taste gelangen wir dann in den Editor zurueck und koennen den Fehler korrigieren.
Weiterhin koennen Fehler bei der Ausfuehrung des Programms auftreten. So koennte es z.B. passieren, dass Variable derartige Ergebnisse bekommen, dass durch Null dividiert wird. Dies ist sicher nicht korrekt und fuehrt zu einem Programmabbruch. Vom System wird dann die Speicherstelle angegeben, bei der der Abbruch auftrat, und im Programmtext nach der fehlerhaften Stelle gesucht. Wieder koennen wir durch Druecken ESC-Taste in den Editor zurueckgehen und den Fehler korrigieren.

2.1 Aufbau eines Programms

Man hoert haeufig, dass Pascal eine strukturierte Programmiersprache sei. Was ist damit gemeint? Das ganze Pascal-Programm hat eine bestimmte Struktur, auf die wir gleich zu sprechen kommen. Ausserdem hat ein jeder Teil des Programms eine sehr aehnliche Struktur. Die Strukturen, um die es sich hier handelt, werden oft Bloecke genannt und sind so geartet, dass sie eine recht einfache Loesung eines Programmierproblems zulassen.

Wir teilen das Programm in drei grosse Abschnitte ein:

1. Programmkopf
2. Deklarationsteil (mit Prozeduren/Funktionen)
3. Hauptprogramm

Zu 1.: Der Programmkopf besteht aus dem reservierten Wort PROGRAM und dem Programmnamen.

Zu 2.: Hier werden Konstanten, Datentypen und Variablen vereinbart, die im Programm vorkommen und gebraucht werden. Weiterhin folgen alle Prozeduren und Funktionen des Programms. Prozeduren sind Unterprogramme, die vom Hauptprogramm aufgerufen und dann abgearbeitet werden. Die Prozeduren haben die gleiche Form wie ein Hauptprogramm selbst, d.h. sie bestehen aus dem Prozedurkopf mit Deklarationsteil, evtl. aus weiteren Prozeduren und dem Prozedurrumpf mit den Befehlen zwischen BEGIN und END. Ausser den Prozeduren koennen noch Funktionen im Programm enthalten sein, die sich etwas in Form und Gebrauch von Prozeduren unterscheiden.
Besonderheiten von Turbo-Pascal: Alle Teile des Deklarationsteils koennen mehrfach auftreten (also auch Typ-, Variablen- und Konstantendeklarationen), und ihre Reihenfolge ist beliebig. Allerdings muessen Dinge, die benutzt werden sollen, vorher deklariert worden sein.

Zu 3.: Das Hauptprogramm faengt mit BEGIN an und hoert mit END auf. Es besteht aus einzelnen Anweisungen und Bloecken von Anweisungen, die jeweils durch BEGIN und END zusammengehalten werden.

Merke: Kommentar wird in geschweifte Klammern geschrieben. Da viele Rechner diese Zeichen nicht darstellen koennen, wird oftmals als Ersatzzeichen (* Kommentar *) verwendet.

In manchen Programmiersprachen wird die Reihenfolge der Befehle durch Zeilennummern (z.B. in BASIC) oder Spruenge zu bestimmten Zeilennummern bestimmt. In Pascal ist das anders. Hier werden Befehle immer in der beschreibenen Reihenfolge abgearbeitet - fuer BASIC-Anwender erst ein wenig unverstaendlich.

Hinweis: Auch in Pascal gibt es den Sprungbefehl GOTO. Er kann aber in fast

allen Faellen vermieden und durch andere Strukturen ersetzt werden. In Kapitel 6.1 wird kurz auf GOTO eingegangen:

1. Alle Pascal-Befehle werden durch ein Semikolon (;) voneinander getrennt.
2. Pascal-Befehle werden immer in der Reihenfolge abgearbeitet, wie sie auftreten.
3. Die Anweisungen des Programms werden zwischen BEGIN und END geschachtelt.
4. Nach BEGIN steht kein Zeichen!
5. Vor END kann ein Semikolon stehen, sollte jedoch nicht.
   (Nur ein Semikolon allein erzeugt naemlich eine leere Anweisung.)
6. Nach END steht ein Punkt, wenn es das Ende des Hauptprogramms darstellt, sonst ein Semikolon.

Ein einfaches Programm, das nur aus einem Hauptprogramm besteht, d.h. ein Programm ohne Unterprogramme (Prozeduren/Funktionen), hat demnach folgende Form:

```
PROGRAM programmname;

CONST...;
TYPE...;
VAR...;

BEGIN
   ....
   ....
   Anweisungen (durch Semikolon getrennt)
   ....
   ....
END.
```

Ein Programm mit Prozeduren und Funktionen hat die Form:

```
PROGRAM programmname;

CONST...;
TYPE...;
VAR...;

PROCEDURE prozedurname;
   CONST...;
   VAR...;
       BEGIN
       ...
       END;

PROCEDURE prozedurname (variablendeklaration);
   CONST...;
   VAR...;
       BEGIN
       ...
       END;

   VAR...;
       BEGIN
       ...
       END;
```

```
PROCEDURE prozedurname (variablen, VAR variablen);
   CONST...;
   VAR...;
       BEGIN
       ...
       END;

FUNCTION funktionsname (variablendeklaration): typ;
   CONST...;
   VAR...;
       BEGIN
       ...
       END;

BEGIN {Hauptprogramm}
  ...
END.
```

2.2 Ein- und Ausgabe

Nun wird es aber bezueglich der Programmierung in Pascal ernst. Wir wollen, nachdem einige einfache Befehle geklaert sind, ein weiteres kleines Programm schreiben.
Um mit dem Computer ueber ein Programm in Kontakt treten zu koennen, brauchen wir sogenannte Ein- und Ausgabebefehle (die Ein- und Ausgabebefehle beziehen sich vorerst nur auf Eingaben mit der Tastatur und Ausgaben auf den Bildschirm):

WRITE und WRITELN:
Diese Befehle sind Ausgabebefehle, die uebersetzt soviel heissen wie "schreibe". WRITE wird immer dann gebraucht, wenn nach dem geschriebenen Wort oder der geschriebenen Zahl in der gleichen Zeile weitergeschrieben werden soll.
Wenn nach der Ausgabe ein Zeilenvorschub ("Carriage Return" oder kurz CR) folgen soll, d.h., wenn das Naechstgeschriebene in einer neuen Zeile anfangen soll, so wird WRITELN (sprich "writeline") verwendet.
Steht WRITELN allein, so wird nur eine Leerzeile geschrieben. Ansonsten folgen den Befehlen runde Klammern, in denen sich das zu Schreibende befindet.

Beispiele: WRITELN (Wert) schreibt den Dateninhalt von "Wert".
WRITELN (Wert1, Wert2, Wert3) wie oben, aber fuer drei Daten.
WRITELN ('Dies ist ein Beispiel') schreibt den Satz "Dies ist ein Beispiel" auf den Bildschirm.

Wenn mehrere Daten, Variablen, Woerter, Saetze usw. ausgegeben werden sollen, so werden sie in der Klammer jeweils durch ein Komma voneinander getrennt.
Wenn Text in der Klammer steht, so muss er stets durch Hochkomma (') angefuehrt und beendet werden. (Das Hochkomma wird natuerlich spaeter nicht mit ausgegeben!)

READ und READLN:
Dies sind Eingabebefehle, die uebersetzt soviel wie "lies" heissen.

READ/READLN (sprich "readline") ermoeglicht es, eine Zahl, ein Wort oder ein Zeichen einzugeben.
Der Name der einzugebenden Variablen steht wieder in einer Klammer hinter dem READ-/READLN-Befehl. Sollten mehrere Daten mit einem Befehl eingegeben werden, so sind sie in der Klammer durch Kommata zu trennen - hierbei aber Vorsicht: Mehrere einzugebende Daten in einem READLN-Befehl koennen spaeter im Programmlauf seltsame Effekte ergeben.
Die Eingabe wird mit der Return-Taste abgeschlossen. Wie bei den Ausgabebefehlen bewirkt READLN im Gegensatz zu READ, dass nach der Eingabe ein Zeilenvorschub ausgefuehrt wird.

Beispiel:
READLN (Wert) uebergibt der Variablen Wert eine einzugebende Zahl (vorausgesetzt, der Name Wert steht fuer eine Zahlenvariable).

Spaeter im Programmverlauf muss nach dem Eintippen der Zahl (oder des Wortes) die Return-Taste gedrueckt werden, um dem Rechner zu signalisieren, dass die Eingabe beendet ist.

Hinweis: Mit READ (KBD, <Wert>) kann ein Zeichen von der Tastatur KBD (Keyboard) eingegeben werden, ohne dass die <CR>-Taste zur Beendigung der Eingabe gedrueckt werden muss. Das Zeichen wird dann allerdings nicht auf dem Bildschirm dargestellt. Hierbei handelt es sich um eine Moeglichkeit, den Programmablauf durch einfachen Tastendruck zu steuern.

Wir wollen nun ein kleines Programm schreiben, das einige Saetze auf unseren Bildschirm bringt. Gehen Sie in den Editor und geben Sie folgenden Programmtext ein:

```
PROGRAM Saetze;

BEGIN
  WRITELN ('Dies ist ein kleines Programm,');
  WRITELN ('das ein paar Worte auf den Bildschirm');
  WRITELN ('bringen soll.')
END.
```

Beachten Sie: Vor dem END sollte kein Semikolon stehen, kann aber.

Sollten Sie keine Kleinbuchstaben zur Verfuegung haben, schreiben Sie einfach gross. Andererseits duerfen bei den meisten Systemen auch die Befehlswoerter klein geschrieben werden - in den folgenden Programmen werden Befehlswoerter von anderem Text optisch getrennt.
Nachdem Sie dieses Programm im Editor eingetippt haben, verlassen Sie ihn mit ^K^D. Dann rufen Sie den Compiler auf oder tippen einfach R, damit das Programm uebersetzt und gestartet wird.

Nun sollte der Rechner den folgenden Text auf den Bildschirm schreiben:

```
Die ist ein kleines Programm,
das ein paar Worte auf den Bildschirm
bringen soll.
```

Es koennte sein, dass dies nicht der einzige Text auf Ihrem Bildschirm ist, sondern sich noch die Kommandozeile oder irgendwelcher frueher geschriebener Text auf dem Bildschirm befindet.

Fuegen Sie einfach nach BEGIN eine Zeile ein, die so aussieht:

```
    CLRSCR;
```

Dieser Befehl bewirkt, dass der Bildschim geloescht wird (Clearscreen).

Wenn wir die Ausgabe an einer bestimmten Stelle des Bildschirms haben wollen, so verwenden wir

```
    GOTOXY(x,y);
```

mit x,y als ganzzahlige Koordinaten der entsprechenden Bildschirmstelle (die moeglichen Werte fuer x und y haengen natuerlich vom jeweiligen Bildschirm ab). Die obere linke Ecke hat die Koordinate (1,1).

Beispiel:

```
    PROGRAM  Bildschirmtest;

    BEGIN
      GOTOXY (10,20);
      WRITELN ('Noch ein ...');
      GOTOXY (15,30);
      WRITELN ('Testext.')
    END.
```

Im weiteren folgen noch einige Anmerkungen zu den Ausgabebefehlen, die Sie erst verwenden koennen, wenn Sie mit Zahlen und Variablen arbeiten:

Formatierte Ausgabe

Wenn ganze Zahlen mit einer bestimmten Stellenzahl ausgegeben werden sollen, so benutzt man die Form:

```
     WRITELN (Ganzezahl:Stellenzahl)
```

Beispiel:
WRITELN (Wert1:4, Wert2:5) bewirkt, dass Wert1 mit 4 Stellen, Wert2 mit 5 Stellen rechtsbuendig ausgegeben wird.

Sollen Dezimalzahlen mit einer bestimmtem Stellenzahl und einer bestimmten Nachkommastellenzahl ausgegeben werden, so schreibt man:

```
    WRITELN (Dezzahl:Stellenzahl:Nachkomma)
```

Beispiel:
WRITELN (Wert:7:2) bewirkt, deass Wert mit insgesamt 7 Stellen und davon 2 Stellen hinter dem Dezimalpunkt ausgegeben wird.
Achtung: Der Dezimalpunkt zaehlt bei der Stellenzahl mit!

Noch einige Beispiele:

Variable:	Inhalt:	Ausgabebefehl:	Ausgabe:
Wert1	123	WRITELN (Wert1:5)	123
Wert2	12	WRITELN (Wert2:2)	12
Wert3	12.4	WRITELN (Wert3:5:2)	12.40
Wert4	12.4	WRITELN (Wert4:7:3)	12.400

Sie sollten - als guter Programmierer - darauf achten, dass der Inhalt der Variablen die vorgegebene Stellenzahl nicht ueberschreitet, da die verschiedenen Rechner unterschiedlich darauf reagieren!

2.3 Konstanten und Variablen

```
PROGRAM Beispiel;

  CONST Pi = 3.1416;
  VAR   Radius, Flaeche : REAL;

  BEGIN
    WRITE ('Radius: ');
    READLN (Radius);
    Flaeche := Pi * Radius * Radius;
    WRITELN ('Flaeche=',Flaeche:7:2)
  END.
```

Das im letzten Abschnitt erstellte Programm bestand nur aus dem Programmkopf und dem Hauptprogramm zwischen BEGIN und END. Es hatte noch keine Konstanten und Variablen.

Variable: Variablen sind Speicherplaetze im Rechner, in denen Zahlen, Buchstabe, Woerter oder andere Objekte (die wir spaeter kennenlernen) gespeichert werden. Eine Variable hat einen Namen (Variablenname), einen sogenannten Datentyp und einen Dateninhalt.Mit der Variablen kann genauso gearbeitet werden wie mit den Daten selbst. Mit einer Zahlenvariablen kann z.B. genauso gerechnet werden wie mit anderen Zahlen. Der Name der Variablen und ihr Typ werden im Deklarationsteil des Programms festgelegt. Der Dateninhalt wird der Variablen aber erst im Programm zugewiesen und kann jederzeit geaendert werden.

Konstante: Eine Konstante ist aehnlich der Variablen ein Speicherplatz, der einen Namen und einen Dateninhalt hat. Jedoch laesst sich der Dateninhalt der Konstanten nicht mehr im Programm veraendern. Der Inhalt wird im Deklarationsteil festgelegt.

Merke: Wir unterscheiden fuer Konstanten und Variablen:

Name - Datentyp - Dateninhalt .

Bei Variablen werden Namen und Datentypen im Deklarationsteil festgelegt und die Dateninhalte im Programm zugewiesen.

Bei Konstanten werden Namen und Dateninhalte (und damit Datentypen) nur im Deklarationsteil festgelegt.

Global: Konstanten und Variablen ,die im Deklarationsteil des Programms festgelegt sind, heissen "global". Sie koennen ueberall im Programm verwendet werden (also auch in den Prozeduren und Funktionen). Spaeter werden wir noch lokale Konstanten und Variablen kennenlernen, die nur in der

jeweiligen Prozedur/Funktion gueltig sind.

Variablennamen: Die Variablennamen sind beliebig und koennen in der Regel bis zu 8 Zeichen enthalten. Das erste Zeichen muss ein Buchstabe sein. Sonderzeichen, die Pascal verwendet werden, sowie Namen von Pascal-Befehlen duerfen nicht verwendet werden (resevierte Woerter).

Beispiele fuer richtige und falsche Variablen- und Konstantennamen:

A	richtig
Test	richtig
x3	richtig
1a	falsch (1.Zeichen muss Buchstabe sein)
X-oben	falsch (keine Sonderzeichen)

CONST: Die Deklaration der Konstante beginnt mit dem Wort CONST. Danach werden, durch Semikolon getrennt die Konstanten aufgefuehrt, in dem den Konstantennamen durch ein Gleichheitszeichen ein Wert zugewiesen wird.

VAR: Die Variablendeklaration beginnt mit dem Wort VAR und zaehlt danach die Namen der Variablen und ihren Typ auf. Die Form der Deklaration ist die folgende: Variablenname, Doppelpunkt, Typ. Verschiedene Variablen gleichen Typs koennen vor dem Doppelpunkt, durch Komma getrennt, aufgezaehlt werden. Mehrere Variablendeklarationen werden durch Semikolon voneinander getrennt.

.Beispiel:

```
CONST Vier=4;
      Dez=5.7;
      Elf=11;
      Neun=10;

VAR Wert : INTEGER;
    Zahl,Volumen : REAL;
```

Damit wir mit der Variablendeklaration etwas anfangen koennen, werden wir kurz die sogenannten einfachen Datentypen darstellen (genaue Beschreibung folgt noch).

Einfache Datentypen

INTEGER: Ganze Zahlen. Der Zahlenbereich ist beschraenkt auf die Zahlen von -32768 bis +32767.

REAL: Dezimalzahl. Eine Dezimalzahl kann auf einem Computer ebenfalls nur mit einer beschraenkten Genauigkeit angegeben werden. Sollten bei sehr kleinen Zahlen die Stellen hinter dem Dezimalpunkt oder bei grossen Zahlen die Stellen davor nicht ausreichen, so werden die Zahlen in der Zehnerpotenzschreibweise (wie beim Taschenrechner) angegeben.
Beispiel: Statt 300 000 000 000 kann 3E11 (sprich: 3 mal 10 hoch 11) angegeben werden.

CHAR: Zeichen. Eine Variable dieses Typs kann ein beliebiges Zeichen aus dem Zeichensatz des Rechners enthalten.

STRING[n]: Zeichenkette. Eine Zeichenkette ist eine Folge von beliebigen Zeichen. Ein String kann z.B. ein Wort, ein Name oder ein Satz sein. In der eckigen Klammer wird die maximale Laenge des Strings angegeben.

BOOLEAN: Logische Variable. Dieser Variablentyp kennt nur zwei moegliche Werte: Wahr und Falsch. In Pascal heissen diese Werte:
TRUE - Wahr
FALSE - Falsch
Im Programm kann also einer Booleschen Variablen der Wert TRUE oder FALSE zugewiesen oder dieser Wert abgefragt werden.

Nun folgt ein kleines Beispielprogramm zur Uebung des Umgangs mit Konstanten und Variablen.
Damit der Programmtext etwas uebersichtlicher wird, rueckt man fuer gewoehnlich einige Textzeilen ein. Ausserdem sollte man moeglichst einige Kommentare in den Text einfuegen:

Kommentar: Kommentar laesst sich in geschweiften Klammern einfuegen. Er wird beim Uebersetzen des Programms vom Rechner nicht beruecksichtigt. Wenn keine geschweiften Klammern vorhanden sind, koennen Ersatzzeichen verwendet werden: (*Kommentar*).

```
PROGRAM Vartest;   { Ein Programm zur Demonstration einfacher Typen }

  CONST Drei=3;
        Elfkommadrei=11.3;

  VAR   Ganz1,Ganz2,Summe : INTEGER;
        Name : STRING[20];

 BEGIN
  CLRSCR;                                  { Bildschirm loeschen }
  WRITELN ('Dies ist ein Testprogramm:');  { Eing.von Zahlen/Worten}
  WRITELN;
  WRITE ('Geben Sie eine ganze Zahl ein:');
  READLN (Ganz1);
  WRITELN;
  WRITE ('Noch eine: ');
  READLN (Ganz2);
  WRITE ('Geben Sie Ihren Namen ein: ');
  READLN (Name);                           { Ende der Eingabe }
  Summe := Ganz1 + Ganz2;
  CLRSCR;                                  { Ausgabeteil: }
  WRITELN ('Lieber ',Name,'!');
  WRITELN;
  WRITELN ('Die Konstanten dieses Programms');
  WRITELN ('sind ',Drei,' und ',Elfkommadrei);
  WRITELN;
  WRITELN ('Sie haben folgende Zahlen eingegeben:');
  WRITELN (Ganz1,' und ',Ganz2);
  WRITELN ('Die Summe davon ist ', Summe);
  WRITELN;
```

```
        WRITELN ('...das war es.')
      END. { Programmende }
```

Tippen sie das Programm bitte in den Editor ein, und bringen sie es mit R zum Laufen. Experimentieren Sie mit verschiedenen Eingaben, und aendern Sie das Programm nach Ihren Wuenschen ein wenig. Aendern Sie z.B. die unbefriedigende Ausgabe der Konstanten, indem Sie formatisierte Ausgaben verwenden.

Zuweisung: Wird einer Variablen ein Wert zugewiesen, so benutzen wir das Zuweisungszeichen :=.

Beispiel:
```
a:=5;
a:=2*a;
```
(a erhaelt den doppelten Wert seines vorigen Inhalts)
Die Zuweisung ist logisch am besten von rechts nach links zu lesen.

Gleichheit: Das Gleichheitszeichen = ist nur dann zu benutzen, wenn die Werte zu seinen beiden Seiten tatsaechlich gleich sind (oder deren Gleichheit abgefragt wird).

2.4 Reservierte Woerter und Standardbezeichner

In der Namengebung fuer unsere Programme, Variablen, Prozeduren und Funktionen unterliegen wir einigen Beschraenkungen und Vorschriften. Wir duerfen unter Verwendung der folgenden Zeichen

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789
und _ (Unterstreichungszeichen)
```

Namen nach bestimmten Regeln, die aus den Syntaxdiagrammen zu ersehen sind, bilden.
Wir nennen diese Namen "benutzerdefinierte Bezeichner".
Diese duerfen jedoch nicht identisch sein mit sogenannten reservierten Woertern.

Reservierte Woerter: Dies sind Woerter, die in der Programmiersprache Pascal bestimmte Bedeutungen haben und dem Compiler eine korrekte Uebersetzung und Ausfuehrung des Programms ermoeglichen.
Selbstverstaendlich darf z.B. eine Variable nicht denselben Namen haben wie eine Pascal-Anweisung, da es sonst zu unloesbaren Konflikten kaeme.

Folgende reservierte Woerter sind vorhanden (es ist unerheblich, ob die Woerter gross oder klein geschrieben werden):

ABSOLUTE	EXTERNAL	NIL	SHR
AND	FILE	NOT	STRING
ARRAY	FOR	OF	THEN
BEGIN	FORWARD	OR	TO
CASE	FUNKTION	PACKED	TYPE
CONST	GOTO	PROCEDURE	UNTIL
DIV	IF	PROGRAM	VAR

DO	IN	RECORD	WHILE
DOWNTO	INLINE	REPEAT	WITH
ELSE	LABEL	SET	XOR
END	MOD	SHL	

Neben den reservierten Woertern, die nicht veraendert werden duerfen, gibt es noch eine Reihe von Standardbezeichnern fuer vordefinierte Funktionen, Prozeduren, Dateien und Datentypen. Diese duerfen zwar veraendert werden, jedoch tut der Anfaenger gut daran, diese Namen nicht als Bezeichner zu waehlen, damit nicht versehentlich Dinge umbenannt werden, die gar nicht umbenannt werden sollten.

ARCTAN	AUXINPTR	BLOCKWRITE	BYTE
ASSIGN	AUXOUTPTR	BOOLEAN	CHAIN
AUX	BLOCKREAD	BUFLEN	CHAR
CHR	FILESIZE	MEMAVAIL	ROUND
CLOSE	FILLCHAR	MOVE	SEEK
CLREOL	FLUSH	NEW	SEEKEOF
CLRSCR	FRAC	NORMVIDEO	SEEKEOLN
CON	GETMEN	ODD	SIN
CONINPTR	GOTOXY	ORD	SIZEOF
CONOUTPTR	HEAPPTR	OUTPUT	SQR
CONCAT	HI	OVRDRIVE	SQRT
COPY	INPUT	PARAMCOUNT	SUCC
COS	INSLINE	PARMASTR	SWAP
CRTEXIT	INSERT	PI	TEXT
CRTINIT	INT	PORT	TRM
DELLINE	INTEGER	POS	TRUE
DALAY	KBD	PRED	TRUNC
DELETE	KEYPRESSED	PTR	UPCASE
EOF	LENGTH	RANDOM	USR
EOLN	LN	RANDOMIZE	USRINPTR
ERASE	LO	READ	USROUTPTR
ERRORPTR	LOWVIDEO	READLN	VAL
EXECUTE	LST	REAL	WRITE
EXIT	LSTOUTPTR	RELEASE	WRITELN
EXP	MARK	RENAME	
FALSE	MAXINT	RESET	
FILEPOS	MEM	REWRITE	

Weiterhin verwendet Pascal folgende Symbole:

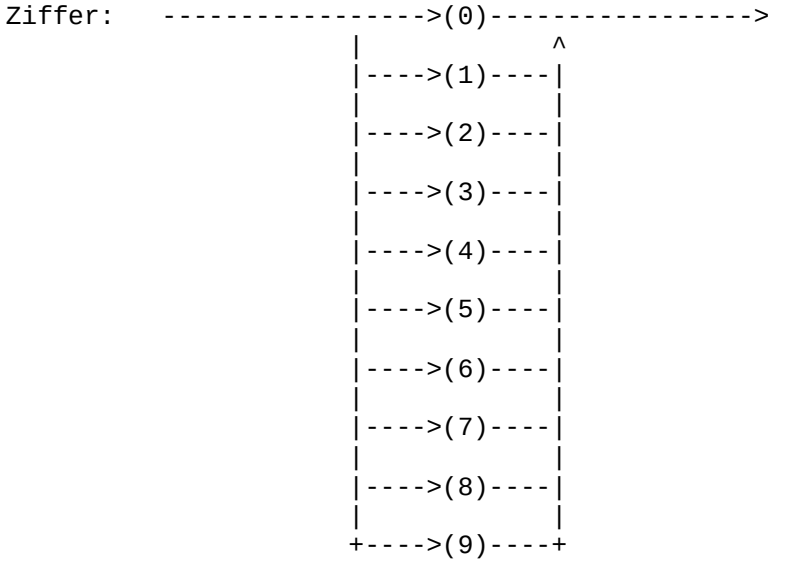
```
(   )
[   ]     Ersatzsymbol:  (.  .)
{   }     Ersatzsymbol:  (*  *)
'   '
, . ; :
+ - * /
= < >  < <= >=
:=
^
$  #
```

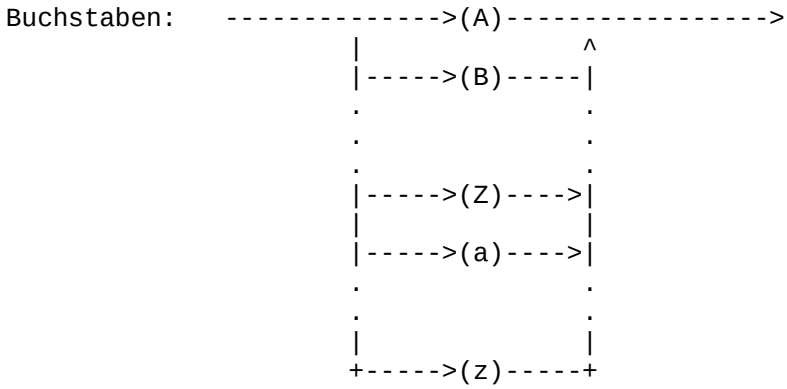
Merke: Benutzerdefinierte Bezeichner (Namen) duerfen niemals reservierten Woertern gleichen!

2.5 Syntaxdiagramme

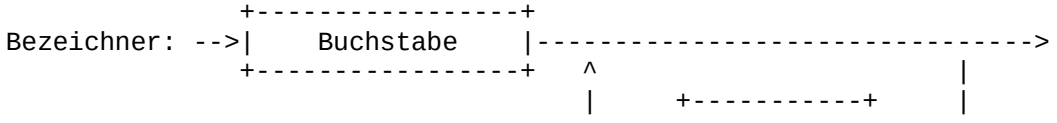
Jeder Konstruktion in Pascal, sei es eine bestimmte Anweisung, eine zusammengehoerige Folge von Anweisungen (Block) oder ein Bezeichner, liegen zugrunde, wie sie zu bilden ist. Diese Grammatik (Syntax) ist genau festgelegt, so dass es niemals Missverstaendnisse oder undefinierte Situationen zwischen Programmierer und Computer geben kann. Alles ist eindeutig festgelegt.

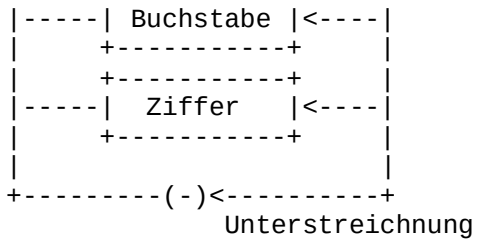
Syntaxdiagramme werden gelesen, indem man dem Pfeil folgt. Oft sind mehrere Wege moeglich. Ein anderes Mal nur bestimmte. Achten sie also auf den Pfeil. Ein Syntaxdiagramm sieht aus, wie das Gleissystem einer Modelleisenbahn.
In den abgerundeten Kaesten stehen fest definierte Symbole oder reservierte Woerter; in den eckigen Kaesten Bezeichner, die durch weitere Syntaxdiagramme erklaert werden.
Einige Beispiele fuer grundlegende Pascal-Konstruktionen:





Die Syntaxdiagrammme fuer "Ziffer" und "Buchstabe" lassen nur jeweils einen Weg zu,d.h. eine Ziffer (ein Buchstabe) besteht aus genau einem der aufgefuehrten Symbole.





Im Weg durch das Syntaxdiagramm "Bezeichner" muessen wir als erstes einen Buchstaben benutzen. Danach sind wir entweder fertig oder koennen noch eine(n) oder mehrere Ziffern (Buchstaben) oder Unterstreichungszeichen folgen lassen.

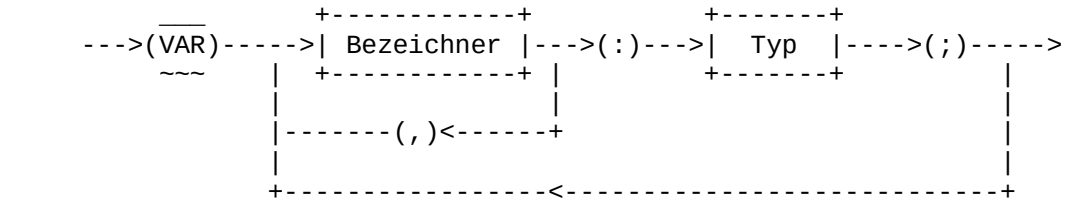
Folgende Bezeichner sind also syntaktisch korrekt:

```
Bezeichner
BEGIN
ANFANG
x1
Drei45Sechs
Was_sonst
```

Falsch dagegen sind:

```
4Teile
Was-sonst
*Test*
```

Variablendeklaration:



Bemerkung: Typ muss natuerlich als Syntaxdiagramm vorher erklaert sein

Die folgenden Variablendeklarationen sind also syntaktisch korrekt:

```
    VAR A, Test, Wert : INTEGER;
oder
    VAR X : REAL;
    Y,Z : CHAR;
    Drucker : INTERACTIVE;
```

Falsch dagegen sind:

```
    VARIABLES X, Y : STRING;
oder
    VAR: A, B, C : CHAR;
oder
    VAR X = REAL
```

Wir werden in den folgenden Kapiteln noch einige Syntaxdiagramme kennenlernen.

3.1 Ganze Zahlen - INTEGER - Hexadezimal - Byte

Wir kennen alle aus unserem Anfangsmathematikunterricht die Division ganzer Zahlen mit Rest. Dazu wollen wir ein kleines Programm schreiben: Eingegeben werden zwei ganze Zahlen, der Dividend und der Divisor. Ausgegeben wird das ganzzahlige Ergebnis und der Rest.

```
PROGRAM Teile;
 VAR Dividend,Divisor,Ergebnis,Rest : INTEGER;

  BEGIN
    WRITE ('Eingabe des Dividenden: ');
    READLN (Dividend);
    WRITE ('Eingabe des Divisors: ');
    READLN (Divisor);
    Ergebnis := Dividend DIV Divisor;
    Rest := Dividend MOD Divisor;
    WRITELN (Dividend:5,':',Divisor:5,'=',Ergebnis:5,' Rest',Rest:5);
  END.
```

Bei einer Eingabe von 29 als Dividend und 3 als Divisor antwortet das Programm:

```
29:3=9 Rest 2
```

Alle Variablen des Programms sollen ganze Zahlen sein. Wir nennen diesen Datentyp in Pascal INTEGER.
Eine ganze Zahl zeichnet sich dadurch aus, dass sie einen Nachfolger und einen Vorgaenger hat, der durch Addition (Subtraktion) mit 1 ermittelt wird. Aufgrund der Darstellung im Rechner ist allerdings darauf hinzuweisen, dass es nur einen begrenzten Bereich von ganzen Zahlen gibt. Die groesste ganze Zahl ist durch die Konstante MAXINT und die kleinste durch -MAXINT-1 bestimmt. MAXINT ist eine vordefinierte Konstante.

Im Turbo Pascal ist MAXINT = (2 hoch 15) - 1 = 32767.

Folgende Darstellungen sind gueltige INTEGER-Werte:

```
12345
+24
-24
0
3
MAXINT
-MAXINT
```

Fehlerhafte Werte fuer INTEGER sind:

```
2.56     (kein Dezimalpunkt erlaubt)
3,456    (kein Komma erlaubt)
3E20     (keine Zehnerpotenzdarstellung erlaubt)
120340   (groesser als MAXINT)
```

Nachdem wir nun wissen, welche Form INTEGER-Variablen (und natuerlich auch Konstanten, siehe Kap. 2.3) haben, brauchen wir noch Rechenoperationen:

```
i + j    Addition
i - j    Subtraktion
i * j    Multiplikation
i DIV j  Division fuer ganze Zahlen mit ganzzahligem Ergebnis.
         Der moegliche Nachkommawert wird abgeschnitten (nicht gerundet !).
i MOD j  Rest (modulo). Ergibt den Rest bei der Division.
```

Und fuer Anwender, die haeufig mit Bitkombinationen zu tun haben:

```
i AND j  Bitweise Und-Verknuepfung zweier ganzer Zahlen.
i OR  j  Bitweise Oder-Verknuepfung zweier ganzer Zahlen.
i XOR j  Bitweise Entweder-Oder-Verknuepfung zweier ganzer Zahlen.
NOT i    Bitweise Nicht-Operation auf einer ganzen Zahl.
i SHL j  Ganze Zahl i bitweise j Stellen nach links schieben.
i SHR j  Ganze Zahl i bitweise j Stellen nach rechts schieben.
```

Beispiele:

```
3 + 5 ergibt 8
8 - 5 ergibt 3
12 - 20 ergibt -8
2 * 13 ergibt 26
7 DIV 2 ergibt 3
7 MOD 2 ergibt 1
13 MOD 5 ergibt 3
32 SHR 3 ergibt 4 (denn 32 ist binaer 100000, und 4 ist binaer 100)
36 XOR 5 ergibt 33
255 AND 15 ergibt 15
NOT 20 ergibt 11
```

Wir koennen Zahlen miteinander vergleichen. Hierzu verwenden wir die folgenden Vergleichsoperatoren:

```
>      groesser als
>=     groesser oder gleich
<      kleiner als
<=     kleiner oder gleich
=      gleich
<>     ungleich
```

Im Zusammenhang mit dem Datentyp INTEGER sind noch einige Funktionen zu nennen, die verwendet werden koennen. Hinter einer Funktion stecken eine Reihe von Operationen, die mit dem Argument der Funktion (der Wert in der Klammer hinter dem Funktionsnamen) durchgefuehrt werden. Eine Funktion hat stets ein Ergebnis.

Im folgenden ist i eine INTEGER-Variable und r eine Variable vom Typ REAL (Dezimalzahl).

```
ABS(i)      Absolutwert einer Zahl i. Ergebnis vom Typ INTEGER.
            Beispiel: ABS(-10) ergibt 10
                      ABS(+10) ergibt 10

SQR(i)      Quadrat einer Zahl i. Ergebnis vom Typ INTEGER.
            Beispiel: SQR(3) ergibt 9
```

TRUNC(r) Ganzzahliger Anteil einer Dezimalzahl (abgeschnitten-nicht gerundet). Ergebnis vom Typ INTEGER.
Beispiel: TRUNC(3.6) ergibt 3
TRUNC(-20.2) ergibt -20

ROUND(r) Gerundeter ganzzahliger Teil einer Dezimalzahl. Ergebnis vom Typ INTEGER.
Beispiel: ROUND(3.6) ergibt 4

Weitere Standardprozeduren und -funktionen siehe Kap. 6.

In Turbo Pascal gibt es noch eine weitere Darstellungsart von Konstanten vom Datentyp INTEGER, naemlich die hexadezimale Darstellung. Das Hexadezimalsystem ist ein Zahlensystem mit 16 Ziffern:

0 1 2 3 4 5 6 7 8 9 A B C D E F

Wenn wir in diesem System zaehlen, so kommt nach F die Zahl 10 (dezimal 16). Zahlen im Hexadezimalsystem werden in Turbo Pascal durch Voranstellen eines Dollarszeichen (bei BC/PC = $) kenntlich gemacht.

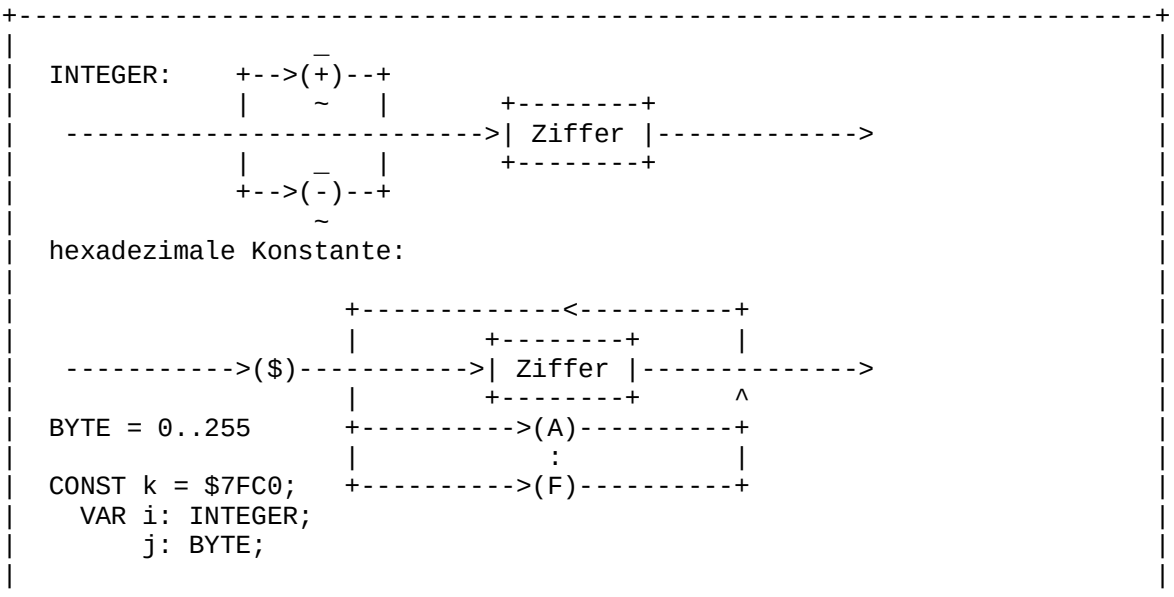
$10 = 16
$FF = 255
$100 = 256
usw.

Der Zahlenbereich ist dann beschraenkt von $0000 bis $ffff.

Ausserdem gibt es in Turbo Pascal einen Datentyp, der insbesondere fuer Operationen mit Speicherstellen geeignet ist: BYTE. Der Datentyp BYTE umfasst den Zahlenbereich 0..255.

BYTE = 0..255

Im Gegensatz zu den Daten vom Typ INTEGER, die zwei Byte Speicherplatz einnehmen, braucht der Typ BYTE nur 1 Byte Speicher.



```
|  Variablen vom Datentyp INTEGER koennen die Werte -(MAXINT+1),....,-3,  |
|  -2,-1, 0, 1, 2, 3,..., MAXINT enthalten.                                |
|  Konstanten aus dieser Wertemenge sind ebenfalls vom Typ INTEGER.        |
|  MAXINT ist eine vordefinierte Konstante (32767).                        |
|  Die Rechenoperationen sind: + (Addition), - (Subtraktion),              |
|  * (Multiplikation), DIV (Division), MOD (Rest).                         |
|  Binaere Operationen sind: AND, OR, XOR, NOT, SHL, SHR.                  |
|  Vergleichsoperationen sind: <, <=, >, >=, =, <>.                        |
|  Standardfunktionen  mit  einem  Ergebnis vom Typ INTEGER sind: ABS(i),  |
|  SQR(i), TRUNC(r), ROUND(r).                                             |
|  Hexadezimale  Konstanten werden durch Voranstellen eines Dollerzeichens|
|  ($) kenntlich gemacht.                                                  |
|  BYTE ist ein Unterbereich von INTEGER, der die Zahlen 0..255 umfasst.   |
|                                                                          |
+--------------------------------------------------------------------------+
```

Ausgabeformate

Wenn man folgendes deklariert:
VAR i,n:INTEGER;
ergibt sich bei:
WRITELN(i); Ausgabe von i mit der tatsaechlichen Stellenzahl von i.
WRITELN(i:n); Ausgabe von i mit n Stellen.

3.2 Dezimalzahlen - REAL

Im folgenden Programm wollen wir Oberflaechen und Volumen einer Kugel berechnen.Dazu kennen wir aus der Geometrie die Formeln

```
O = 4 * PI * r * r   und   V = 4 / 3 * PI * r * r * r
```

Turbo Pascal verfuegt ueber eine vordefinierte Konstante PI, so dass die Programmzeile

```
CONST PI = 3.1415926536;
```

entfallen kann.
Nun folgt unser kleines Programm:

```
PROGRAM Kugel;

   VAR Radius, Oberflaeche, Volumen : REAL;

   BEGIN
     WRITE ('Eingabe des Radius: ');
     READLN (Radius);
     Oberflaeche := 4 * PI * Radius * Radius;
     Volumen := 4/3 * PI * Radius * Radius * Radius;
     WRITELN ('Die Kugel mit dem Radius ',Radius:6:2,' hat');
     WRITELN ('ein Volumen von ',Volumen:9:2);
     WRITELN ('und eine Oberflaeche von ',Oberflaeche:7:2)
   END.
```

Die Variablen in diesem Programm sollten Dezimalzahlen aufnehmen. Sie sind vom Datentyp REAL. Auch dieser Datentyp hat einen beschraenkten Wertebereich, der von 1E-38 bis 1E+38 geht. Die Genauigkeit betraegt insgesamt 11 Ziffern.

Allerdings hat auch diese Genauigkeit ihre Grenzen. So laesst sich 1/3 nicht durch 0.33333333333333... darstellen, da die Ziffer 3 noch unendlich oft auftreten wuerde. Ab einer bestimmten Stelle muss also gerundet werden. Dies fuehrt bei vielen Rechenschritten moeglicherweise zu erheblichen Rundungsfehlern.
Die Ergebnisse des obigen Programms werden formatiert ausgegeben. Dabei gibt die erste Zahl hinter dem Doppelpunkt die Groesse des Feldes an, in dem das Ergebnis ausgegeben wird, und die zweite Zahl gibt die Anzahl der Nachkommastellen an.

Achtung! Um unnoetigen Aerger zu vermeiden, sollten wir folgende Hinweise beim arbeiten mit REAL-Variablen beachten:

- Niemals die Gleichheit von Rechenergebnissen abfragen, sondern ueberpruefen, ob der Absolutwert der Differenz der Ergebnisse einen bestimmten Wert unterschreitet.

- Die Subtraktion fast gleich grosser Zahlen vermeiden, wo es moglich ist.

- Moeglichst wenig Rechenschritte vorsehen, um den Rundungsfehler klein zu halten.

Wie sehen nun Zahlen des Types REAL aus ?
Moeglicherweise angefuehrt von einem Vorzeichen, schreiben wir Ziffern vor und hinter dem Dezimalpunkt (wie bei Taschenrechnern). Ausserdem ist die Darstellung mit Zehnerpotenzen erlaubt. So kann man 3400 darstellen als 3.4 mal 10 hoch 3: also schreiben als 3.4E3 (ebenfalls wie bei Taschenrechnern).

Gueltige REAL-Zahlen sind z.B.:
```
   12.
   12.0
   +3.67 oder 3.67
   -9.03
   0.45
   4.67E+3 oder 4.67E3
   1.4E-4
```

Die Rechenoperationen fuer REAL sind:

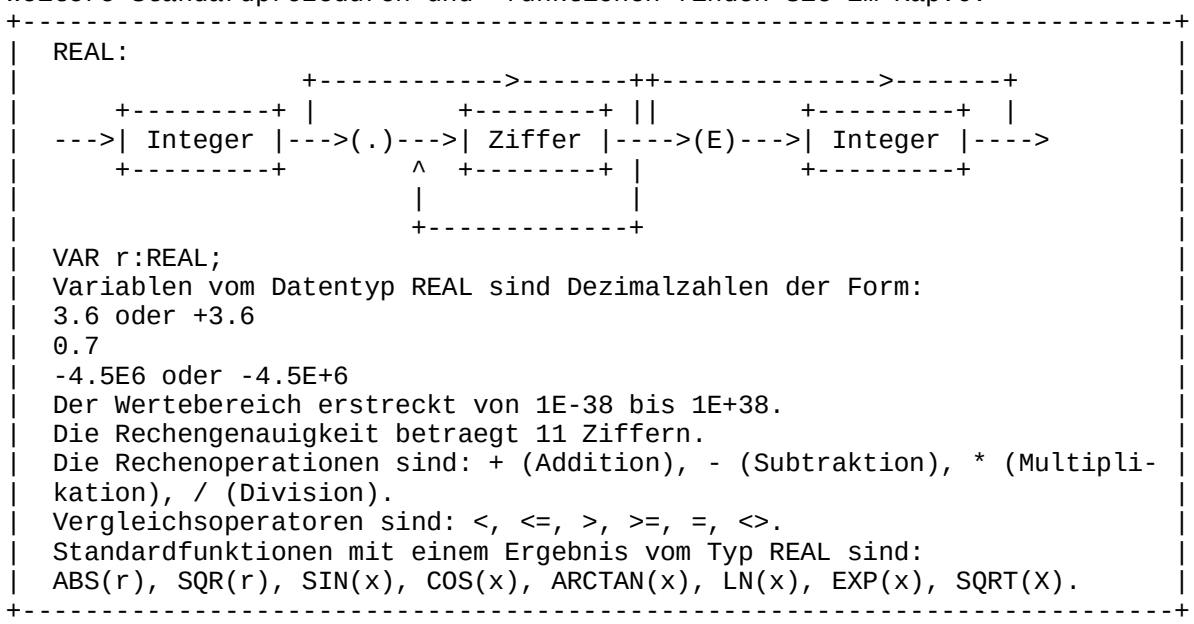
```
   + Addition
   - Subtraktion
   * Multiplikation
   / Division mit Ergebnis vom Typ REAL
```

Ausserdem sind die schon bekannten Vergleichsoperatoren >, >=, <, <=, =, <> (siehe INTEGER, Kap.3.1) zu verwenden.
Auch in diesem Programm wurde die formatierte Ausgabe gewaehlt. Die ganze Zahl hinter dem ersten Doppelpunkt gibt die gesamte Anzahl der Stellen an (einschliesslich Dezimalpunkt !). Die ganze Zahl hinter dem zweiten Doppelpunkt gibt die Zahl der Stellen hinter dem Dezimalpunkt an.
Hier sind einige Standardfunktionen fuer den Datentyp REAL (r steht fuer eine Variable vom Typ REAL, i fuer INTEGER und x fuer INTEGER oder REAL):

```
   ABS(r)         Absolutwert einer Zahl r. Ergebnis REAL.
   SQR(r)         Quadrat von r. Ergebnis REAL.
   SIN(x)         Sinus von x (x im Bogenmass). Ergebnis REAL.
   COS(x)         Cosinus von x (x im Bogenmass). Ergebnis REAL.
   ARCTAN(x)      Arcustangens von x (x im Bogenmass). Ergebnis REAL.
```

```
   LN(x)          Natuerlicher Logarithmus von x. Ergebnis REAL.
   EXP(x)         Exponentialfunktion e hoch x. Ergebnis REAL.
   SQRT(x)        Quadratwurzel von x. Ergebnis REAL.
```

Weitere Standardprozeduren und -funktionen finden Sie im Kap.6.



Unterschied INTEGER <----> REAL

Bei der Variablendeklaration

```
   VAR x,y : REAL;
         i : INTEGER;
```

ergeben die nachstehenden Programmzeilen einen Fehler:

```
   x := 1.5;
   y := 2;
   i := x*y;(*hier fehlerhafte Zuweisung*)
```

Hier liegt ein sogenannter Typkonflikt vor. Opwohl in diesem Fall das Ergebnis der Multiplikation als ganze Zahl dargestellt werden kann, ist es doch von der Typdeklaration her REAL. x*y ergibt naemlich nicht 3, sondern 3.0. Einen Ausweg aus diesem Dilemma bieten die Uebergangsoperationen von REAL und INTEGER:

```
   TRUNC(r) und ROUND(r) (siehe Kap. 3.1).
```

Ausgabeformate

Wenn man folgendes deklariert:

```
   VAR r : REAL;
       n,m : INTEGER;
```

ergibt sich bei:

`WRITELN(r);`	Ausgabe von r mit groesster Genauigkeit (in der Regel in Zehnerpotenzschreibweise).
`WRITELN(r:n:m);`	Ausgabe von r mit insgesamt n Stellen (einschliesslich Dezimalpunkt!),davon m Stellen hinter dem Dezimalpunkt.

3.3 Zeichen - CHAR

Unser Computer kann nicht nur Zahlen (numerische Ausdruecke) verarbeiten, sondern auch Zeichen. Unter Zeichen verstehen wir Buchstaben, Ziffern, Satzzeichen, Zeichen fuer Rechenoperationen und sogenannte Steuerzeichen (z.B. fuer den Drucker).
Der Datentyp, der fuer die alphanumerischen Zeichen verwendet wird, heisst CHAR. Welche Zeichen zum Zeichensatz gehoeren, haengt vom Rechnertyp ab. Sehr viele Rechner verwenden Zeichen nach dem sogenannten ASCII-Zeichensatz (American Standard Code for Information Interchange). Dieser Zeichensatz umfasst in der Regel 128 Zeichen (manche Rechner haben einen erweiterten Zeichensatz von 256 Zeichen).

Aufgabe: Sehen Sie sich den Zeichensatz im Anhang an !

Die Zeichen sind durchnummeriert von 0 bis 255. Wir werden gleich auf die Bedeutung der Nummern zu sprechen kommen.
Zu Beachten ist, dass die Zeichen von Nummer 0 bis 31 nur sogenannte Steuerzeichen sind, also Zeichen, die eine bestimmte Funktion ausueben. So ist z.B. Nummer 12 das Zeichen FF (Form-Feed) - das ist ein Seitenvorschub (auf dem Bildschirm oder dem Drucker). Die restlichen Zeichen mit den Nummern 32 bis 126 sind sichtbare Zeichen. Nur Nummer 127 hat noch eine Steuerfunktion (Loeschzeichen).

Die Nummern der Zeichen haben eine grosse Bedeutung. Durch sie ist der Zeichensatz geordnet. So ist z.B. das Zeichen "A" kleiner als das Zeichen "B".
In Pascal werden Zeichen immer zwischen einfache Anfuehrungsstriche (Hochkommata) gesetzt. Will mann einen Anfuehrungsstrich als Zeichen verwenden, wird er doppelt zwischen zwei Hochkommata geschrieben, also so:''''.
Wir wollen uns einige Beispiele ansehen:

```
PROGRAM Testzeichen;

   CONST a = 'A';
         b = 'B';
   VAR  ch : CHAR;

   BEGIN
      READ (KBD,ch);
      WRITELN (ch:4);
      READLN (ch);
      WRITELN (ch);
      WRITELN (a,b);
      ch := 'C';
      WRITELN (ch);
      ch := a;
      WRITELN (ch)
   END.
```

Bei der ersten Eingabe (mit READ) reicht es, wenn der Benutzer eine Taste drueckt (ohne RETURN-Taste). Die zweite Eingabe (mit READLN) muss mit der RETURN-Taste abgeschlossen werden.
In der formatierten Ausgabe gibt die ganze Zahl hinter dem Doppelpunkt die Anzahl der Stellen an, die das Zeichen beansprucht. Ist die Stellenzahl groesser als 1, werden Leerzeichen vorangestellt.

Achtung ! Die Zuweisung ch := 'AB' ist falsch, da 'AB' aus mehr als einem Zeichen besteht (es ist vom Typ STRING, siehe Kap. 3.4).

Mit dem Typ CHAR lassen sich natuerlich keine arithmetischen Operationen ausfuehren. Trotzdem gibt es einige Standardfunktionen (c ist vom Typ CHAR und i vom Typ INTEGER):

ORD(c) Ordnungsnummer von c, d.h. die Nummer des Zeichens in der Codiertabelle (z.B. ASCII). Ergebnis vom Typ INTEGER.
Beispiel: ORD('A') ergibt 65

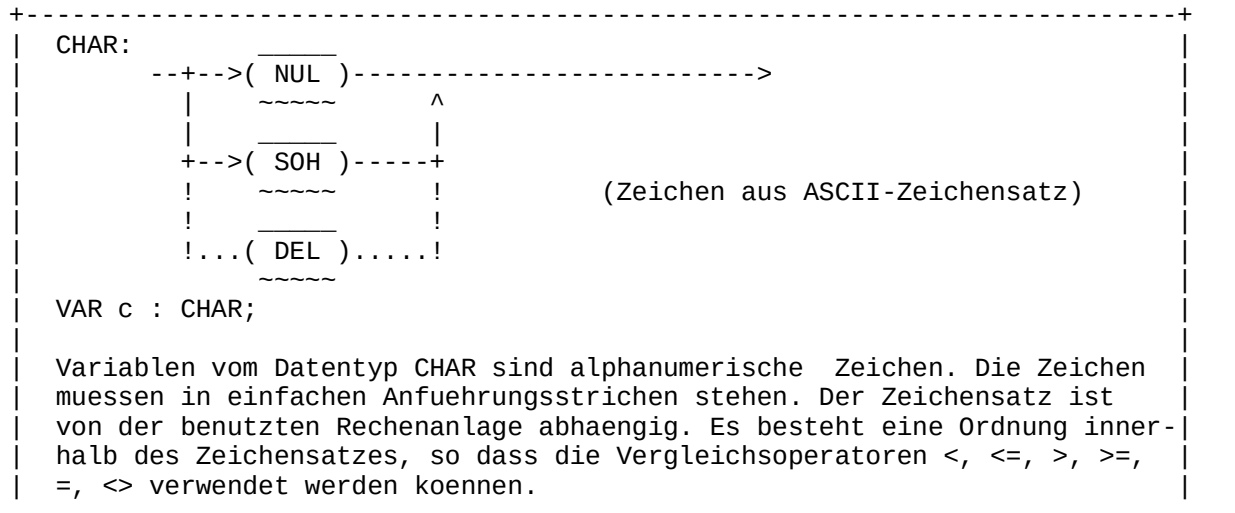
CHR(i) Zeichenfunktion. Liefert das Zeichen mit der nummer i in der Codiertabelle (z.B. ASCII). Ergebnis vom Typ CHAR.
Beispiel: CHR(65) ergibt A

PRED(c) Vorgaengerfunktion. Liefert das dem Zeichen c in der Codierungstabelle vorangehende Zeichen. Ergebnis vom Typ CHAR.
Beispiel: PRED('E') ergibt D

SUCC(c) Nachfolgefunktion. Liefert das dem Zeichen c in der Codierungstabelle nachfolgende Zeichen. Ergebnis vom Typ CHAR.
Beispiel: SUCC('E') ergibt F

Ausserdem gelten die schon bekannten Vergleichsoperatoren:
<, <=, >, >=, =, <>.

Dabei ist zu beachten, dass die Elemente des Zeichensatzes in der Reihenfolge der Codierungstabelle angeordnet sind. Im ASCII-Zeichensatz gilt z.B.: ...''<'!'<''''<'#'<...<'/'<'1'<'2'<...<'A'<...<'Z'<...
Dabei kann es zu Sortierproblemen kommen, denn einige Sonderzeichen kommen in der Reihenfolge vor den Buchstaben. Insbesondere das Leerzeichen hat mit 32 die kleinste Ordnungsnummer der sichtbaren Zeichen.



```
|  Standardfunktionen mit einem Ergebnis vom Typ CHAR sind:                |
|  CHR(i), PRED(c), SUCC(c).                                               |
|  Standardfunktionen mit einem Ergebnis vom INTEGER ist: ORD(c).          |
+--------------------------------------------------------------------------+
```

Ein-/ und Ausgabeformate

Wenn man folgendes deklariert:

```
   VAR c:CHAR;
       n:INTEGER;
```

ergibt sich bei:

```
   READ(c);       Eingabe eines Zeichens ohne Zeilenvorschub.
   READLN(C);     Eingabe eines Zeichens mit Zeilenvorschub.
   READ(KBD,c);   Eingabe eines Zeichens ohne Return-Taste.
   WRITELN(c);    Ausgabe mit einer Stelle.
   WRITELN(c:n);  Ausgabe mit n-Stellen. Die fuehrenden Stellen werden mit
                  Leerzeichen aufgefuellt.
```

3.4 Zeichenketten - STRING

Standard-Pascal sieht den Datentyp STRING an sich nicht vor. Viele Pascal-Versionen besitzen ihn trotzdem. Hier werden die Standardfunktionen im Turbo Pascal vorgestellt.
Der Datentyp STRING wird fuer Zeichenketten verwendet. Eine Zeichenkette ist eine Aneinanderreihung von Elementen des Typs CHAR.
Daher kann man sich den Datentyp STRING selbst definieren, wenn er in der benutzten Pascal-Version nicht vorgesehen ist. Die Form ist dann:

```
   TYPE STRING = ARRAY[1..n] OF CHAR;
```

wobei n die maximale Anzahl der Zeichen einer Zeichenkette ist.

Der Turbo Pascal-Benutzer braucht STRING nicht zu definieren,es ist schon vordefiniert. Dem Wort STRING wird eine Zahl zwischen 1 und 255 in eckigen Klammern angehaengt, die angibt, wie gross die maximale Laenge der STRING-Variablen ist.

```
   VAR s:STRING[15];
```

Die maximale Laenge der Zeichenkette s betraegt dann 15 Zeichen.
Als Beispielprogramm zum Datentyp STRING wollen wir ein kleines Programm schreiben, das die Buchstaben eines eingegebenen Wortes untereinander schreibt. Dazu brauchen wir eine sogenannte Schleife (siehe Kap.4.1), die von 1 bis zur Anzahl der Zeichen des Wortes zaehlt:

```
   FOR i := 1 TO LENGTH(Wort);
```

So sieht das Programm aus:

```
   PROGRAM Worttest;

      VAR Wort : STRING[20];
             i : INTEGER;  (* Laufvariable *)
```

```
BEGIN
   WRITELN ('Geben Sie ein Wort ein: ');
   READLN (Wort);
   FOR i := 1 TO LENGTH(Wort) DO
      WRITELN (Wort[i])
END.
```

In diesem Programm sehen wir, dass es moeglich ist, ein Zeichen aus einer Zeichenkette herauszunehmen, indem wir die Nummer des Zeichens in eine eckige Klammer hinter den Namen der Zeichenkette schreiben.

Beispiel:

```
Wort := 'Computer';
Wort[3] ergibt m
```

Achtung ! Eine Zeichenkette, die ein Zeichen enthaelt, ist deshalb noch lange nicht vom Typ CHAR. Eine Zuweisung oder ein Vergleich der Datentypen STRING und CHAR fuehrt immer zu einem Typkonflikt. Statt dessen schreiben wir:

```
VAR c: CHAR;
    s: STRING[20];
    n: INTEGER;
```

```
und die Zuweisung   c := s[n];     (wobei n die Nummer des Zeichens im
                                    STRING s ist).
```

Der grundsaetzliche Unterschied zwischen einem ein Zeichen langen String und einer Variablen vom Typ CHAR liegt darin, dass im (nicht sichtbaren) nullten Zeichen des Strings die Laenge codiert ist. ORD(s[0]) ist die Laenge des Strings s.
Zeichenketten duerfen auch nicht druckbare Steuerzeichen oder Control-Zeichen enthalten. Dazu wird (ohne zusaetzliche Leerstelle) die ASCII-Codierung des Zeichens, angefuehrt durch das #-Zeichen, eingegeben. Auch CTRL-Zeichen koennen direkt durch Voranstellen eines ^-Zeichens eingefuegt werden.

Beispiele:

```
#10   : Zeilenvorschub (Line Feed)
#$A   : Ebenfalls Zeilenvorschub
#65   : A
#$1b  : ESC
^P    : CTRL-P
^G    : CTRL-G (Bell, Piepston)
```

oder

```
WRITELN(^G^G^G'Aufwachen !');
WRITELN(#12'Neue Seite');
```

Als Standardfunktion mit Zeichenketten haben wir im Programm LENGTH(s) benutzt.. Es gibt noch weitere Funktionen (s, s1, s2... sind vom Datentyp STRING und n und m vom Typ INTEGER):

```
LENGTH(s)         Laenge der Zeichenkette s, d.h. Anzahl der Buchstaben.
```

Ergebnis vom Typ INTEGER.
Beispiel: LENGTH('Wort') ergibt 4.

POS(s1,s) Position des erstmaligen Auftretens der Zeichenkette s1 in der Zeichenkette s. Ergebnis vom Typ INTEGER.
Beispiel: POS('buch','Handbuch') ergibt 5.
POS('ball','Handbuch') ergibt 0.
POS('ei','Weinstein') ergibt 2.

CONCAT(s1,s2,s3,...,sn)
Verkettung mehrerer Zeichenketten. Ergebnis vom Typ STRING.
Beispiel: CONCAT('Hand','buch') ergibt Handbuch.

s1+s2+...+sn Statt mit CONCAT koennen mehrere STRINGS auch mit dem Zeichen + verknuepft werden.
Beispiel: 'Hand'+'buch' ergibt Handbuch.

COPY(s,n,m) Herausnehmen eines Teils aus der Zeichenkette s ab der Stelle n mit der Laenge m. Ergebnis vom Typ STRING.
Beispiel: COPY('Computer',4,3) ergibt put.

Neben den Standardfunktionen gibt es noch die Standardprozeduren mit Zeichenketten. Im Gegensatz zu den Funktionen, die immer ein Ergebnis haben, das einer Variablen zugewiesen werden muss, werden Prozeduren nur aufgerufen und koennen dann Variablen aendern, die ihnen beim Prozeduraufruf mitgegeben werden (siehe auch Kap. 6.1).

INSERT(s1,s,n) Einfuegen einer Zeichenkette s1 in die Zeichenkette s an der Stelle n.
Beispiel: Sei s:='Comer';
INSERT('put',s,4) ergibt Computer fuer s.

DELETE(s,n,m) Loeschen von m Zeichen ab Stelle n in der Zeichenkette s.
Beispiel: Sei s:='Buchstaben';
DELETE(s,3,5) ergibt Buben fuer s.

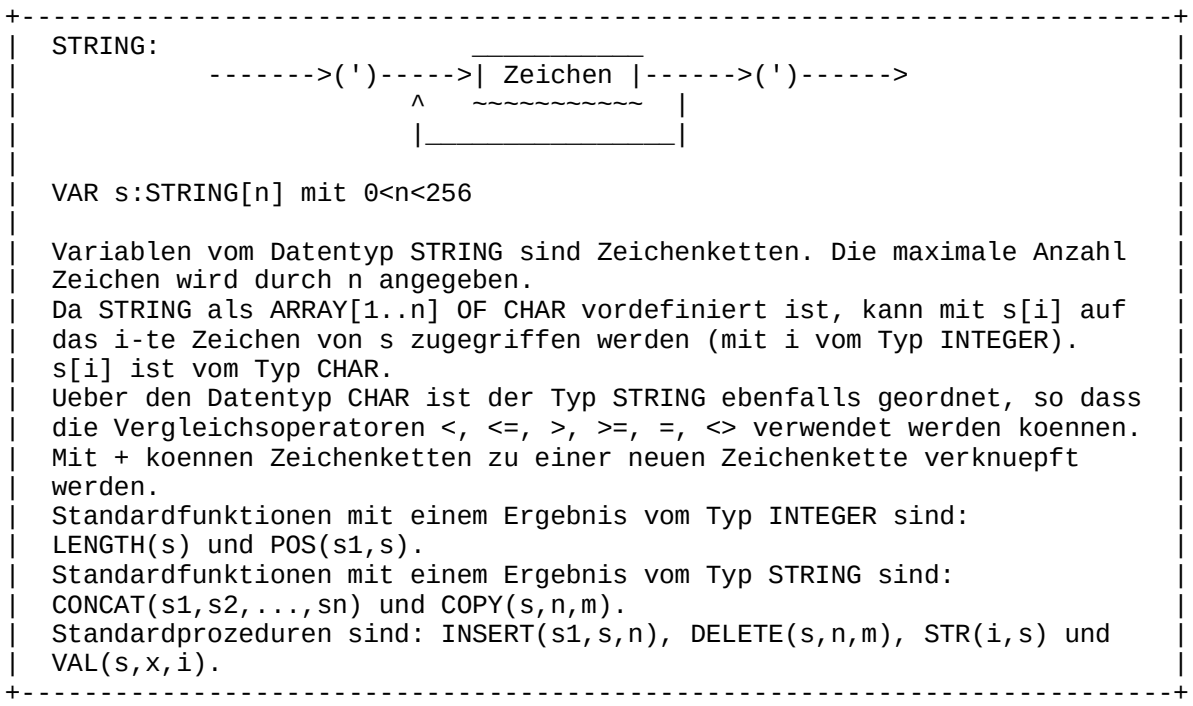
STR(i,s) Wandelt eine Zahl i vom Typ INTEGER in einen STRING s um.

VAL(s,x,i) Wandelt einen String s, der eine Zahl vom Typ REAL oder INTEGER enthaelt, in eine Zahl x des entsprechenden Typs um. i ist eine INTEGER-Variable, die die Stelle im String s markiert, an der ein Fehler bei der Umwandlung passiert. Ohne Fehler ist i=0.

Da an dieser Stelle die Anwendung von Funktionen und Prozeduren noch nicht bekannt ist, ein kleines Programm, das die Benutzung der o.g. Funktionen und Prozeduren verdeutlichen soll:

```
PROGRAM Stringdemo;

   VAR Wort, Teil : STRING[30];
       Stelle, Laenge : Integer;

   BEGIN
      Wort := 'Donauschiff';
```

```
        Laenge := LENGTH(Wort);
        WRITELN (Laenge);              (* Ausgabe: 11 *)
        Teil := 'aus';
        Stelle := POS(Teil,Wort);
        WRITELN (Stelle);              (* Ausgabe 4 *)
        Wort := CONCAT(Wort,'skapitaen');
        WRITELN (Wort);                (* Ausgabe: Donauschiffskapitaen *)
        Teil := COPY(wort,6,6);
        WRITEL (Teil);                 (* Ausgabe: Schiff *)
        INSERT('dampf',Wort,6);
        WRITELN (Wort);           (* Ausgabe: Donaudampfschiffskapitaen *)
        Stelle := 1;
        Laenge := 17;
        DELETE(Wort,Stelle,Laenge);
        WRITELN (Wort);           (* Ausgabe: kapitaen *)
      END.
```



Ausgabeformate

Wenn mann folgendes deklariert:

```
   VAR s : STRING[20];
       n : INTEGER;
```

ergibt sich bei:

```
   WRITLN(s):     Ausgabe von s mit LENGTH(s) Stellen.
   WRITELN(s:n);  Ausgabe von s mit n Stellen. Wenn n>LENGTH(s) ist, so wer-
                  den die fuehrenden Stellen mit Leerzeichen aufgefuellt.
```

3.5 Wahheitswerte - BOOLEAN

DER UNTERE SATZ IST WAHR
DER OBERE SATZ IST GELOGEN

Was ist denn nun wahr ?

Aus der Aussagelogik kennen wir sogenannte logische Aussagen. Sie koennen die Werte "wahr" oder "falsch" einnehmen.

Beispiel: "Paris ist die Hauptstadt von England" ist falsch.
"Paris ist die Hauptstadt von Frankreich" ist wahr.
"Dieses Buch ist leicht verstaendlich" ist objektiv nicht entscheidbar.

Alle drei Aussagen sind allerdings fuer unsere Arbeit mit dem Computer unbrauchbar. Insbesondere darf es niemals eine unentscheidbare Situation geben. Vielmehr haben wir es mit Aussagen folgenden Typs zu tun:

```
5 = 4 ist falsch
5 > 4 ist wahr
```

In Pascal haben die Wahrheitswerte folgende Namen:

```
TRUE  - wahr
FALSE - falsch
```

Ausserdem kann man einen Wahrheitswert einer Variablen zuordnen, die dann vom Typ BOOLEAN ist.

Wenn die Variable b vom Datentyp BOOLEAN ist, dann gilt fuer die folgenden Zuweisungen:

```
b := TRUE;              richtig
b := FALSE;             richtig
b := b = FALSE;         richtig
b := 17 < 3;            richtig
b := i <> j;            richtig (mit i,j vom Typ INTEGER)
b := (12>2)AND(7<2);    richtig
b := 12+3;              falsch
```

Wir wollen ein Programm schreiben, das eine Wahrheitstabelle fuer die "Und-Verknuepfung" (AND) angibt.
So sieht das Programm aus:

```
PROGRAM Wahr;

   VAR a,b,c : BOOLEAN;

   PROCEDURE Ausgabe;
   BEGIN
      c := a AND b;
      WRITELN (a:6,' AND ',b:6,' = ',c:6)
   END; (* von Ausgabe *)

   BEGIN
      a := FALSE; b:= FALSE; Ausgabe;
      a := FALSE; b := TRUE; Ausgabe;
      a := TRUE; b := FALSE; Ausgabe;
      a := TRUE; b := TRUE;  Ausgabe
   END.
```

In diesem Programm wurde eine einfache Prozedure verwendet. Das ist ein Unterprogramm (Teilprogramm), das einfach durch Nennung seines Namens aufgerufen und abgearbeitet wird.
Werden Daten durch die bekannten Operatoren <,<=, >, >=, =, <> verglichen, so ist das Ergebnis vom Typ BOOLEAN und koennte einer entsprechenden Variablen zugewiesen werden.

Ausdruecke vom Typ BOOLEAN koennen benutzt werden (mit a,b vom Typ Boolean):

```
a AND b   logisches "UND"
a OR  b   logisches "ODER"
a XOR b   logisches "ENTWEDER ODER"
NOT a     Negation von a
```

Weiterhin gilt: FALSE < TRUE.

Mit zwei FOR-Schleifen laesst sich das obige Programm noch eleganter schreiben:

```
PROGRAM Wahr2;

   VAR a,b :BOOLEAN;

   BEGIN
      FOR a := FALSE to TRUE DO
         FOR b := FALSE TO TRUE DO
            WRITELN (a:6,' AND ',b:6,' = ',a AND b :6)
   END.
```

Wahrheitstabellen fuer die logischen Operatoren:

a	NOT a
FALSE	TRUE
TRUE	FALSE

a	b	aANDb	a OR b	a = b	aXORb
FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	FALSE	TRUE
TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
TRUE	TRUE	TRUE	TRUE	TRUE	FALSE

ausserdem gibt es eine Funktion mit einem Ergebnis vom Typ BOOLEAN:

ODD(i) ergibt TRUE, wenn die ganze Zahl i ungerade ist.

```
BOOLEAN:

VAR b:BOOLEAN;
```

```
|                                                                          |
|  Variablen vom Datentyp BOOLEAN sind logische Variablen mit Dateninhalt  |
|  TRUE oder FALSE.                                                        |
|  Logische Operatoren sind: AND, OR, XOR, NOT, = .                        |
|  Eine Standardfunktion mit einem Ergebnis vom Typ BOOLEAN ist ODD(i).    |
+--------------------------------------------------------------------------+
```

Hinweis: Ein "logischer Schalter" ist der Ausdruck:
a:= a = FALSE; (mit VAR a:BOOLEAN;)
Wenn a FALSE ist, erhaelt es den Wert TRUE und umgekehrt.

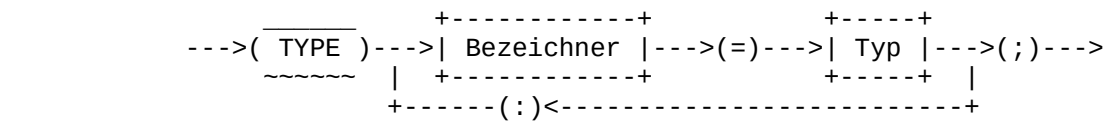
Logische Ausdruecke und Variablen werden besonders im Zusammenhang mit Schleifen (Kap.4) und mit Entscheidungen (Kap.5) gebraucht.

Ein- und Ausgabeformate

READLN(b)	Einlesen einer Zeichenkette TRUE oder FALSE.
WRITELN(b:n)	Ausgabe des Wahrheitswertes der Variablen b als Zeichenkette TRUE oder FALSE in einem Feld von n Stellen Laenge.

3.6 Die TYPE-Deklaration - Aufzaehlungen - Unterbereiche

Mit Hilfe des reservierten Wortes TYPE koennen wir im Programmkopf eigene Datentypen erklaeren. Dazu wird der (von uns erfundene) Name des neuen Datentypes nach einem Gleichheitszeichen erklaert.



Beispiele:

```
TYPE Dezimal = REAL;
     Feld = ARRAY[0..5] OF INTEGER;
```

und

```
VAR x, y:Dezimal;
       f:Feld;
```

Achtung: Haeufig passieren Fehler durch Verwechslungen von Variablennamen mit Datentypen. Feld koennte sicher auch ein Variablenname sein. Im o.g. Fall jedoch ist es als Datentyp deklariert. Eine Variable mit Namen Feld darf es also nicht geben!

Sicher gibt es nicht nur die Moeglichkeit, schon bekannte Datentypen mit neuen Namen zu versehen. Wir koennen auch ganz neue Datentypen erzeugen.

Aufzaehlungstypen

Nehmen wir an, wir haetten haeufig mit Farben im Programmen zu tun. Dann koennten wir folgenden Typ durch eine Aufzaehlung deklarieren:

```
TYPE Farbe = (rot,gelb,gruen,braun,weiss,schwarz,violett);
```

Achtung: Bei den Farben handelt es sich nicht um STRINGs !
Nun erklaeren wir:

```
VAR Ampel,Bildfarbe:Farbe;
```

und

```
PROCEDURE fahren;...
PROCEDURE Zeichne;...
```

Im Programm sind folgende Anweisungen moeglich:

```
IF Ampel = gruen THEN fahren;

FOR Bildfarbe := rot TO violett DO Zeichne;

Ampel := rot;
Ampel := SUCC(Ampel);(*Ergebnis:gelb*)
```

Allerdings lassen sich Variablen, deren Typen durch Aufzaehlungen entstanden sind, nicht so einfach ein- und ausgeben.

```
WRITELN(Ampel); ist nicht moeglich!
```

Fuer Ein- und Ausgaben muessen etwas umfangreichere Hilfsprozeduren verwendet werden.

Merke: Aufzaehlungstypen sind durch die Reihenfolge der Aufzaehlung in der TYPE-Deklaration geordnet (z.B. gilt weiss > braun), kennen Vorgaenger (PRED) und Nachfolger (SUCC) und lassen sich daher z.B. auch in FOR-Schleifen verwenden.

Ausserdem ist die Funktion ORD auf Aufzaehlungstypen anzuwenden (z.B. gilt ORD(braun) = 3).Die Zaehlung der ORD-Funktionen beginnt bei 0 fuer das 1.Element.

Unterbereiche

Bei der oben verwendeten Variablen Ampel ist der Datentyp Farbe sicher nicht gluecklich gewaehlt. Besser waere es, nur die Farben Rot, Gelb, Gruen zuzulassen.
Da aus Gruenden der Eindeutigkeit Bezeichner nicht doppelt verwendet werden duerfen, koennen wir nicht einen weiteren Farbtyp deklarieren. Allerdings gibt es die Moeglichkeit in Pascal, von jedem aufzaehlbaren Datentyp Unterbereiche zu verwenden.

Beispiele:

```
TYPE Farbe = (rot,gelb,gruen,braun,weiss,schwarz,violett);
     Ampelfarbe = rot..gruen;
     SWFarbe = weiss..schwarz;
     Ziffern = 0..9;
     Buchstaben = 'a'..'z';
```

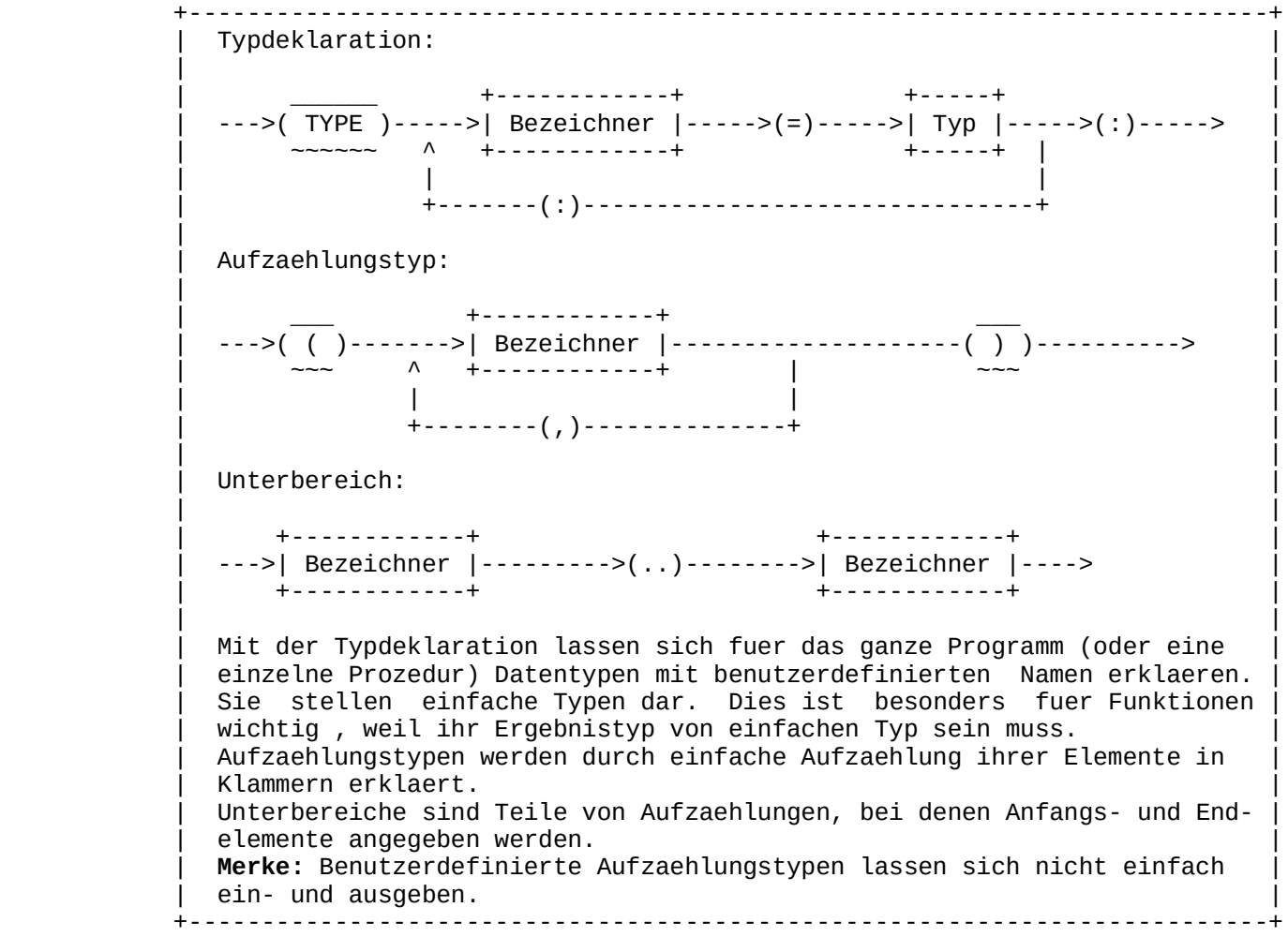
Ein Unterbereich wird dadurch angegeben, dass Anfangs- und Endelement des Bereichs durch zwei (!) Punkte getrennt aufgeschrieben werden.
Wir wollen uns noch ein paar Beispiele fuer Aufzaehlungen und Unterbereiche ansehen:

```
TYPE Tag = (Mo,Di,Mi,Do,Fr,Sa,So);
     Monat = (Jan,Feb,Mar,Apr,Mai,Jun,Jul,Aug,Sep,Okt,Nov,Dez);
     Autotyp = (PKW,Kombi,LKW,Transporter,Bus);
     Klassen = (Sexta,Quinta,Quarta,Untertertia,Obertertia,Untersekunda,
     Obersekunda,Unterprima,Oberprima);

     Wochenende = Sa..So;
     Arbeitstag = Mo..Fr;
     Sommer = Jun..Sep;
     Oberstufe = Obersekunda..Oberprima;
     Posint = 1..MAXINT;
```

Nun folgt ein kleines Programm zur Verdeutlichung des Gebrauchs von Aufzahlungen und Unterbereichen. Zunaechst sind jedoch einige Hinweise noetig, die auf spaetere Themen vorgreifen.
Eine Prozedure (Unterprogramm) kann mit einem Wert aufgerufen werden. Dazu wird eine entsprechende Variable hinter dem Prozedurennamen erklaert. Weiterhin wird eine Fallunterscheidung mit CASE benutzt, da die einzelnen Werte eines Aufzaehlungstyps nicht direkt ausgegeben werden koennen. Nach dem Wort CASE steht eine Variable und nach dem Wort OF eine Liste der Werte, die die Variable annehmen kann. Immer dann, wenn die Variable einen Wert aus der Liste annimmt, wird die dazugehoerige Anweisung ausgefuehrt.

```
PROGRAM Tiere;

   TYPE Alle = (See,  Land,  Flug,  Borsten, Loewe, Adler, Schwein, Ente,
               Hund, Elefand);
        Vor  = See .. Borsten;
        Nach = Loewe .. Elefant;

   VAR Erst : Vor;
       Tier : Nach;

   PROCEDURE Drucke (Wort : Alle);
      BEGIN
         CASE Wort OF
               See : Write ('    See');   Loewe : WRITE ('loewe  ');
              Land : Write ('   Land');   Adler : WRITE ('adler  ');
              Flug : Write ('   Flug'); Schwein : WRITE ('schwein');
           Borsten : Write ('Borsten');    Ente : Write ('ente   ');
                                           Hund : WRITE ('hund   ');
                                        Elefant : WRITE ('elefant')
         END (* von Case *)
      END; (* von Drucke *)

   BEGIN (* Hauptprogramm *)
      WRITELN ('Tiernamengenerator:');
      WRITELN;
      FOR Tier := Loewe TO Elefant DO BEGIN
         WRITELN;
         FOR Erst := See to Borsten DO BEGIN
            Drucke (Erst);
            Drucke (Tier);
            WRITE (' ')
         END (* von Erst *)
      END (* von Tier *)
   END.
```

Sicher haette man das Programm auch mit anderen Datentypen (STING z.B.) schreiben koennen. Insbesondere faellt auf, dass die Ausgabe recht aufwendig ist, da jeder Wert vom Typ Alle in eine Zeichenkette uebersetzt werden muss. Wozu braucht man dann ueberhaupt Aufzaehlungstypen und Unterbereiche ? Ein Programm mit einer Schleife "FOR Mon := Januar TO April DO..." ist sicher lesbarer als eines mit einer Schleife "FOR i := 1 TO 4 DO...". Ausserdem eignen sich Aufzaehlungstypen (insbesondere in grossen Programmen) zur Geschwindigkeitssteigerung und zum Sparen von Speicherplatz, da sie nur ein Byte Speicherplatz brauchen (STRING so viele Bytes wie Zeichen). Auch der Gebrauch von Unterbereichen macht Programme sicherer eleganter und uebersichtlicher.



3.7 Typumwandlungen - Absolute Speicheradressen

In Turbo Pascal lassen sich Daten verschiedener skalarer Typen sehr einfach in andere skalare Typen umwandeln.
In Standard-Pascal stehen dazu nur die Standardfunktionen CHR und ORD zur Verfuegung, um eine eingeschraenkte Typumwandlung vorzunehmen. Beispielsweise ergibt ORD('A') = 65.

Dies laesst sich in Turbo auch folgendermassen formulieren:

```
   INTEGER('A') = 65
```

Die Typumwandlung geschieht also dadurch, dass nach dem Zieltyp in der Klammer als Argument ein Wert aus dem umzuwandelnden Typ steht, z.B.:

```
   TYPE Farbe = (rot,gruen,blau,gelb);
        Tag   = (Mon,Die,Mit,Don,Fre,Sam,Son);
        Gross = 'A'..'Z';
        Klein = 'a'..'z';
```

dann ist:

```
   Gross('d') = 'D';
   INTEGER(blau) = 2;     (Achtung: Zaehlung bei 0 beginnen)
   Farbe(Mit) = blau;
   Tag(4) = Fre;
```

Hinweis: Es duerfen natuerlich nur aufzaehlbare Typen verwendet werden. REAL ist daher nicht zulaessig.

Normalerweise legt Turbo Pascal Variablen in einem dafuer vorgesehenen Speicherbereich ab. Wenn aber aus irgend einem Grund gewuenscht wird, dass Variablen ganz bestimmte Plaetze im Speicher einnehmen sollen, so kann dies in Turbo Pascal durch absolute Variablendeklarationen geschehen.
Hierbei ist ein kleiner Unterschied zwischen 8- und 16-Bit-Systemen zu beachten.

Beispiele:

8-Bit-System:

```
   VARIOByte : BYTE ABSOLUTE $0003;
      Textseite: ARRAY[0..959] OF CHAR ABSOLUTE $0400;
```

16-Bit-System:

```
   VAR Def: INTEGER ABSOLUTE $0000:$00fe;
```

Bei 8-Bit-Systemen wird hinter dem Datentyp das reservierte ABSOLUTE und eine hexadezimale Speicherstelle angegeben, ab der die Variable abgespeichert werden soll. Bei 16-Bit-Systemen ist dies geringfuegig anders. Vor der Speicherstelle steht noch - mit einem Doppelpunkt von der Speicherstelle abgetrennt - die Segmentnummer.
Die absulute Adressierung kann auch von einer anderen Variablen abhaengig gemacht werden.

```
   VAR Str : STRING[80];
      StrLaenge : BYTE ABSOLUTE Str;
```

Das heisst, dass eine Variable bei derselben Speicherstelle beginnt wie eine andere. Damit teilen sich mehrere Variablen ein und denselben Speicherplatz.

Die Standardfunktion

```
   ADDR(Variablenname)
```

gibt bei 8-Bit-Systemen die Anfangsspeicherstelle der angegebenen Variablen an. Bei 16-Bit-Systemen wird entsprechend das Segment mit angegeben.

3.8___Mengen

Finden Sie den Unterschied !

Erste Version:

```
   WRITELN('Waehlen Sie:');
   WRITELN(' M(enue     ');
   WRITELN(' S(ortieren ');
   WRITELN(' A(endern   ');
   WRITELN(' F(inden    ');
   WRITELN(' E(nde      ');
   REPEAT
      READ(ch)
   UNTIL ((((ch='M') OR (ch='m')) OR ((ch='S') OR (ch='s'))) OR (((ch='A')
         OR (ch='a')) OR ((ch='F') or (ch='f')))) OR ((ch='E') OR (ch='e'));
```

Zweite Version:

```
   WRITELN('Waehlen Sie: ');
   WRITELN(' M(enue      ');
   WRITELN(' S(ortieren  ');
   WRITELN(' A(endern    ');
   WRITELN(' F(inden     ');
   WRITELN(' E(nde       ');
   REPEAT
      READ(ch)
   UNTIL ch in ['M','m','S','s','A','a','F','f','E','e'];
```

Richtig ! Bei der zweiten Version ist eine Menge verwendet worden. In der UNTIL-Abfrage wird getestet, ob das Zeichen ch Element der Menge der angegebenen Zeichen ist (IN).
Es ist in Pascal tatsaechlich moeglich, mit Mengen zu operieren. Der dazu noetige Datentyp heisst SET.

Beispiele:

```
   TYPE Menge = SET OF BYTE;
        Letter = SET OF CHAR;
        Klein = SET OF [1..7];
        Farbm = SET OF (rot,gruen,gelb,blau);

   VAR a,b,m : Menge
       l,k   : Letter;
```

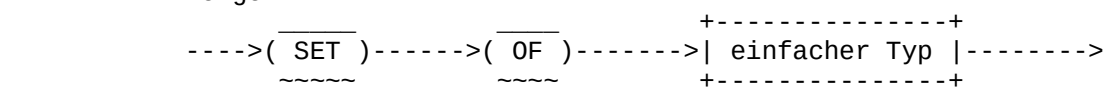
Folgende Zuweisungen sind dann richtig:

```
   a := [1,2,3,4,5,6,7,8];
   b := [1..8];
   l := ['a'..'z','A'..'Z'];
   k := ['+',':','*'];
```

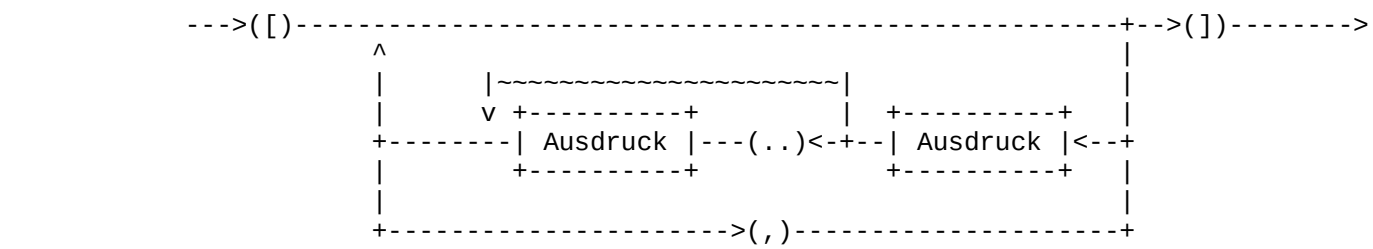
Mengen koennen durch Aufzaehlen der Elemente erstellt werden. Die Elemente werden in eckigen Klammern, durch Kommata getrennt, aufgezaehlt. Auch Unterbereiche koennen in den Klammern auftreten. Die Elemente duerfen nur

von einfachen Datentypen sein (BYTE, CHAR, BOOLEAN, Aufzaehlungen, Unterbereiche).

Menge:



Darstellung einer Menge im Programm:



Aus dem Syntaxdiagramm ersehen wir, dass auch

```
a := [];
```

korrekt ist.

Wenn zwischen den eckigen Klammern nichts steht, so handelt es sich um die leere Menge.

Welche Operatioen sind mit Mengen definiert ?

Vereinigung:	+
Differenz:	-
Durchschnitt:	*
Gleichheit:	=
Ungleichheit:	<>
Ist Teilmenge von:	<=
Ist echte Teilmenge:	<
Ist Obermenge von:	>=
Ist echte Obermenge:	>
Ist Element von:	IN

Beispiele:

```
[1,2,3,4] + [3,4,8,19]  ergibt [1,2,3,4,8,19]
[1,2,3,4] - [3,4,8,19]  ergibt [1,2]
[1,2,3,4] * [3,4,8,19]  ergibt [3,4]
[1,2,3] = [2,1,3]       ist wahr
5 IN [1,4,5,6]          ist wahr
['a'..'z'] > ['c'..'f'] ist wahr
```

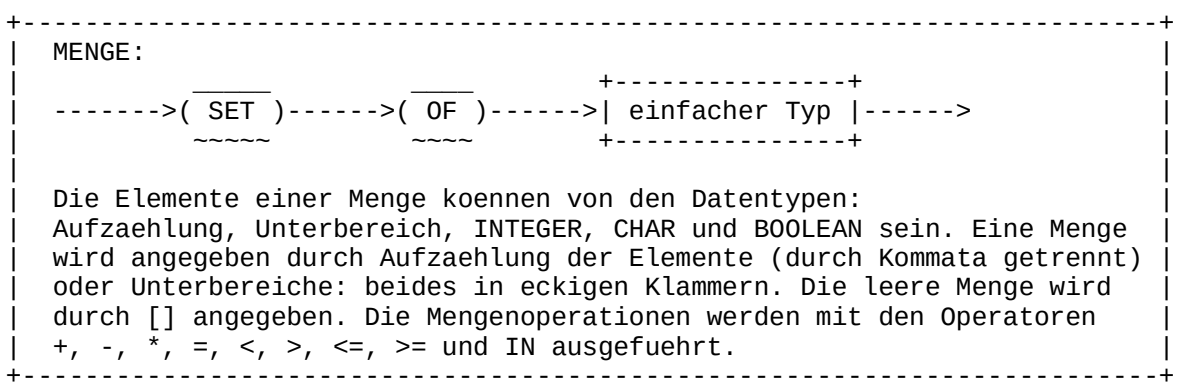
Nun folgt ein kleines Programmbeispiel zu den Mengenoperationen.
Das Programm bestimmt die Anzahl der Ziffern, Buchstaben und Sonderzeichen in einem eingegebenen Text. Ausserdem werden die vorhandenen Vokale, Konstanten und Ziffern ausgegeben. In der Ausgabeprozedur ist ein kleiner Trick vorhanden. Da die Elemente von Mengen nicht direkt ausgegeben werden koennen, muss der ganze Zeichensatz mit einer Schleife durchlaufen werden und das Element immer dann ausgegeben werden, wenn in der Menge ist.

```
PROGRAM Textmengen;
   TYPE Mengen = SET OF CHAR;

   VAR Ziffern,Buchstaben,Sonstige : Mengen;
       Vokale,Konstante,Urmenge    : Mengen;
       Zi,Bu,So,i                  : INTEGER;
       Satz                        : STRING[80];
       ch                          : CHAR;

   BEGIN (* Hauptprogramm *)
      CLRSCR;
      WRITELN ('Geben Sie einen Satz ein:');
      READLN (Satz);
      Bu := 0; Zi := 0; So := 0;
      Ziffern := ['0'..'9'];
      Buchstaben := ['A'..'Z'] + ['a'..'z'];
      Sonstige := [' '..'ue'] - (Ziffern + Buchstaben);
         (* ue = Zeichen mit hoechsten ASCII - Code *)
      Vokale := ['a','e','i','o','u','A','E','I','O','U'];
      Konstanten := Buchstaben - Vokale;
      Urmenge := [ ];

      FOR i := 1 TO LENGTH(Satz) DO BEGIN
         IF Satz [i] IN Buchstaben THEN Bu := Bu + 1 ELSE
         IF Satz [i] IN Ziffern THEN Zi := Zi + 1 ELSE So := So + 1;
         Urmenge := Urmenge + [Satz [i]]
      END;

      Vokale := Vokale * Urmenge;
      Konstanten := Konstaten * Urmenge;
      Ziffern := Ziffern * Urmenge;

      WRITELN ('Der Satz:');
      WRITELN (Satz);
      WRITELN ('enthaelt ',Bu:3,' Buchstaben,');
      WRITELN ('         ',Zi:3,' Ziffern,');
      WRITELN ('     und ',So:3,' Sonstige Zeichen,');
      WRITELN ('folgende Vokale: ');
      FOR ch := ' ' TO 'ue' DO (* ue siehe oben *)
         IF ch IN Vokale THEN WRITE(ch);
      WRITELN;
      WRITELN ('folgende Konsonanten: ');
      FOR ch := ' ' TO  'ue' DO (* ue siehe oben *)
         IF ch IN Konsonanten THEN WRITE(ch);
      WRITELN;
      WRITELN ('folgende Ziffern: ');
      FOR ch := ' ' TO 'ue' DO (* ue siehe oben *)
         IF ch IN Ziffern THEN WRITE(ch);
      WRITELN
   END.
```

Ausgabe des Programmes:

```
   Der Satz:
   Dies ist ein (Beispiel-) Satz, um das Programm zu testen!
   enthaelt  43 Buchstaben,
              0 Ziffern,
        und  16 sonstige Zeichen,
```

folgende Vokale:
aeiou
folgende Konsonanten:
BDPSdglmnprstz
folgende Ziffern:



Einschraenkung in Turbo Pascal:

- Eine Menge darf hoechstens 256 Elemente haben.
- Die Elemente einer Menge duerfen nur aus aufgezaehlten Datentypen sein, die nicht mehr als 256 Elemente enthalten.

4.1 Die FOR-Schleife

In Pascal gibt es eine Anweisung die es erlaubt, festzulegen, wie oft andere Anweisungen ausgefuehrt werden sollen. Diese Anweisung nennt man FOR-Schleife. Dazu wird eine sogenannte Laufvariable auf einen bestimmten Anfangswert gesetzt und bei jedem Schleifendurchgang auf ihren Nachfolger erhoeht, bis die Laufvariable den Endwert erreicht oder ueberschritten hat. Die Form ist:

```
FOR Zaehlvariable:= Anfangswert TO Endwert DO Anweisung;
FOR Zaehlvariable:= Anfangswert TO Endwert DO BEGIN Anweisungen END;
```

Es ist ausserdem moeglich, rueckwaerts zu zaehlen. In diesem Fall muss in der obigen Anweisung "TO" durch "DOWNTO" ersetzt werden:

```
FOR Zaehlvariable:= Anfangswert DOWNTO Endwert DO Anweisung;
FOR Zaehlvariable:= Anfangswert DOWNTO Endwert DO BEGIN Anweisungen END;
```

Die Angabe einer Schrittweite - wie in manchen anderen Programmiersprachen - ist in Pascal nicht moeglich. Die Zaehlvariable wird also immer auf ihren Nachfolger (d.h. bei Zaehlvariablen vom Typ INTEGER um Eins) erhoeht oder auf den Vorgaenger vermindert.

Dazu sehen wir uns ein kleines Beispiel an:

In Kap.3.3 haben wir den ASCII-Zeichensatz kennengelernt. Hier ist nun ein Programm, das alle druckbaren Zeichen dieses Zeichensatzes vorwaerts und rueckwaerts ausgibt.

```
PROGRAM Ascii;

VAR i,zeit : INTEGER;

BEGIN
  CLRSCR;                  (* Loescht der Bildschirm *)
  WRITELN;
  WRITE ('Der ASCII-Zeichensatz  ');
  WRITELN ('- vorwaerts:');
  WRITELN;
  FOR i:=32 TO 126 DO WRITE(i:5,':',chr(i):2);
  FOR zeit:=1 TO 30000 DO;    (* Verzoegerungsschleife *)

  CLRSCR;                  (* Loescht den Bildschirm *)
  WRITE('Der ASCII-Zeichensatz  ');
  WRITELN('- und rueckwaerts:');
  WRITELN;
  FOR i:=126 DOWNTO 32 DO WRITE(i:5,':',chr(i):2);
  FOR zeit:=30000 DOWNTO 1 DO;    (* Verzoegerungsschleife *)
  CLRSCR
END.
```

Beachtenswert ist der Einsatz von FOR-Anweisungen als Verzoegerungsschleife mit einer "leeren" Anweisung nach dem DO. Nach der Deklaration von

```
    VAR ch:CHAR;(* CHAR=Buchstabe *)
```

kann man die entsprechenden FOR-Anweisungen wie folgt aendern:

```
    FOR ch:=''TO'-'DO bzw.FOR ch:='-'DOWNTO''DO
```

Die Ausgabe bleibt gleich. Wie an diesem Beispiel zu sehen ist, kann man als Zaehlvariable auch andere Datentypen als INTEGER wie z.B.CHAR verwenden. Die Datentypen muessen allerdings genau einen Nachfolger oder Vorgaenger haben. Daher kommen zunaechst nur INTEGER, CHAR und BOOLEAN (nur zwei Werte!) sowie Aufzaehlungen und Unterbereiche als Zaehlvariable in Frage. In einem Programm koennen somit durchaus die folgenden Anweisungen auftreten:

```
     FOR ch:='A'TO'Z'DO WRITE(ch);      (* gibt alle Grossbuchstaben aus *)
     FOR ch:='z'DOWNTO'a'DO WRITE(ch)   (* gibt alle Kleinbuchstaben aus *)
```

Wodurch kann nun erreicht werden, dass ein Programm - z.B.mit einer der o.a. FOR-Schleifen - nach der Ausgabe jedes einzelnen Buchstabens kurz anhaehlt und dann fortfaehrt? Offensichtlich kann eine Verzoegerungsschleife innerhalb des Anweisungsblocks einer FOR-Schleife dies bewirken. In einem solchen Fall spricht man von Schachtelung. Dabei ist zu beachten , dass die innere Schleife jeweils vollstaendig abgearbeitet wird, bevor die aeussere Schleife fortgefuehrt wird.
Verdeutlichen wir uns dies anhand eines weiteren Programms.
Es soll ein Dreieck, zusammengesetzt durch den wiederholten Ausdruck des Zeichens *, ausgegeben werden.

```
   PROGRAMM Dreieck;

   VAR Sterne_pro_Zeile,
       Aussen, Innen, Leer,
       Anzahl_der_Zeilen: INTEGER;
    BEGIN
      CLRSCR;
      WRITE ('Wie viele Zeilen soll das Dreieck haben? ');
      READLN (Anzahl_der_Zeilen);
      Sterne_pro_Zeile:=1               (* fuer erste Zeile *)

      FOR  Aussen := Anzahl_der_Zeilen DOWNTO 1 DO
              BEGIN          (* von Aussen *)
                FOR Leer:=Aussen DOWNTO 1 DO WRITE (' '); (* Gibt ' ' aus *)
                FOR Innen:=1 TO Sterne_pro_Zeile DO WRITE ('*');
                Sterne_pro_Zeile := Stern_pro_Zeile + 2;
                WRITELN                         (* naechste Zeile *)
              END                               (* von Aussen *)
    END.
```

Innerhalb der "Aussen"-Schleife, die einen Anweisungsblock umfasst, sind zwei weitere FOR-Schleifen untergebracht, die jeweils nur eine Anweisung beinhalten. Ferner wird die Zaehlvariable "Aussen" in der "Leer"-Schleife als Anfangswert eingesetzt. Lassen wir das Programm einmal mit Anzahl_der_Zeilen:=6 ablaufen, so ergibt sich folgender Ausdruck:

```
                         *
                        ***
                       *****
                      *******
                     *********
```

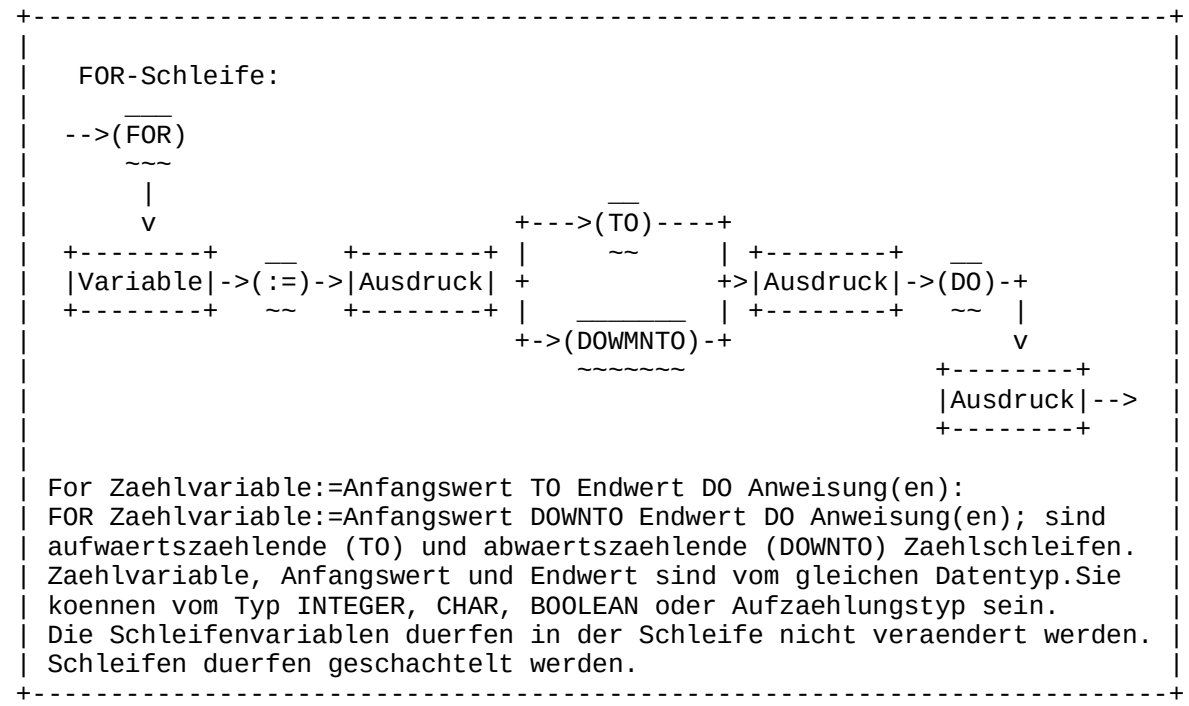
****

Wir halten fest:
Die FOR-Schleife zaehlt eine Zaehlvariable von einem Anfangswert bis zu einem Endwert (herauf mit TO, herunter mit DOWNTO).
- Anfangs- und Endwert duerfen Konstanten oder Variablen sein.
- Anfangs-, Endwert und Zaehlvariable muessen vom gleichen Typ sein.
- Als Schleifentyp kann jeder Datentyp verwendet werden, der genau einen Nachfolger (bzw.Vorgaenger) hat (INTEGER,CHAR,BOOLEAN und Aufzaehlungstypen).
- Eine Schleife wird einmal abgearbeitet, wenn Anfangs- und Endwert uebereinstimmen.

- Eine Schleife wird keinmal abgearbeitet, wenn bei Verwendung von TO der Anfangswert groesser als der Endwert, bei Verwendung von DOWNTO der Anfangswert kleiner als der Endwert ist.
- Schleifen koennen geschachtelt werden.
- Schleifen mit einer leeren Anweisung koennen als Verzoegerungsschleifen eingesetzt werden.

Darueber hinaus
- kann eine Zaehlvariable zur Berechnung anderer Variablen verwendet werden;
- sollten Zaehlvariable, Anfangs- und Endwert in einer Schleife nicht veraendert werden.

Hinweis: In einigen Pascal-Versionen ist das Aendern von Zaehlvariablen, sowie von Anfangs- und Endwert innerhalb der Schleife zwar moeglich, sollte aber aus Gruenden eines "sauberen" Programmierstiles unterbleiben.



Hinweis:
Eine Schleife der Form FOR Zaehlvariable:= Anfangswert TO Endwert DO; ist ebenfalls moeglich. Das Semikolon hinter dem Wort DO stellt eine leere Anweisung dar. Diese Form kann z.B. als Verzoegerungsschleife angewandt

werden, da der Rechner in dieser Schleife lediglich zaehlt.

4.2 Die REPEAT-Schleife

Neben der im vorangegangenen Paragraphen besprochenen FOR-Anweisung gibt es in Pascal zwei weitere Moeglichkeiten, einen Programmteil wiederholt vom Rechner ausfuehren zu lassen. Beide unterscheiden sich von der FOR-Anweisung dadurch, dass die Anzahl der Schleifendurchlaeufe (=Wiederholungen) nicht von vornherein durch die Angabe eines Endwertes festgelegt werden muss. Veilmehr wird zur Beendigung der Schleife eine Bedingung - die sogenannte Abbruchbedingung - herangezogen.
Die REPEAT-Schleife ist eine der beiden Moeglichkeiten. Sie hat die Form

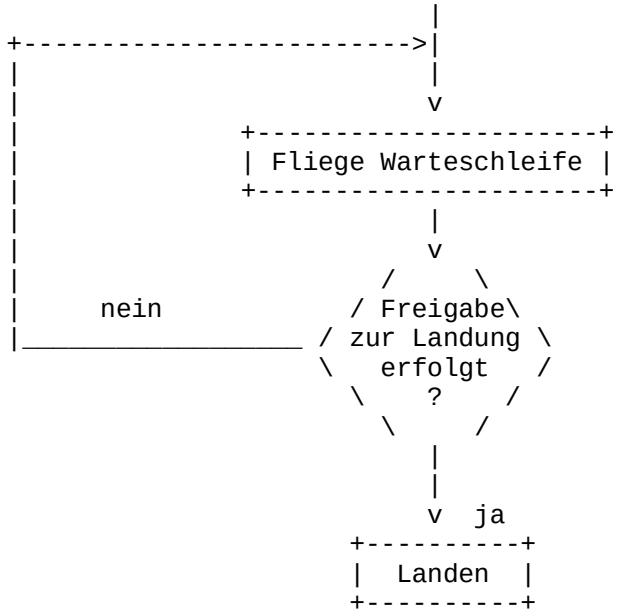
REPEAT Anweisung(en) UNTIL Bedingung

auf deutsch: WIEDERHOLE Anweisung(en) BIS Bedingung (erfuellt).

Verdeutlichen wir uns die arbeitsweise einer solchen Schleife einmal an folgendem Modell:
Flugzeuge die einen Flughafen zur Landung anfliegen, werden gelegentlich angewiesen, eine Warteposition einzunehmen, d.h.Schleifen zu fliegen, bis die Freigabe zur Landung erfolgt. In unsrer algorithmischen Sprache muss das Flugzeug die Anweisung

WIEDERHOLE Warteschleifen fliegen BIS Freigabe zur Landung erfolgt

befolgen. Als Flussdiagramm ergibt sich fuer unser Modell der REPEAT-Schleife folgedes Bild:



Wenn die Freigabe zur Landung nicht erfolgt, so stuertzt unser Flugzeug ab - genau wie unser Programm. Die (Warte-)Schleife wird nur dann verlassen, wenn die Abbruchbedingung erfuellt ist, d.h.den Wert TRUE annimmt.
Sehen wir uns unter diesem Aspekt einmal die folgende Anweisung etwas genauer an:

```
REPEAT WRITE ('Drucker einschalten') UNTIL Drucker eingeschaltet
```

Ist der Drucker nicht eingeschaltet, so hat unsere Abbruchbedingung, die

sich in Pascal leider nicht so einfach darstellen laesst,den Wert FALSE. Dies hat eine Wiederholung der WRITE-Anweisung zur Folge. Erst, wenn der Drucker eingeschaltet wird, kann der Rechner die Schleife beenden.
Ist kein Drucker vorhanden, so rettet uns nur noch die <CR>-Taste vor der Wiederholungswut des Rechners.
Es ist also peinlich genau darauf zu achten, dass die Abbruchbedingung den Wert TRUE ueberhaupt annehmen kann (Problem der Terminiertheit). Dies ist natuerlich nur dann moeglich, wenn die Voraussetzungen fuer diese Bedingung in der Schleife geaendert werden.
Ein lauffaehiges Beispiel fuer die Verwendung der REPEAT-Schleife ist das folgende kleine Programm.

```
PROGRAMM Wiederholung;

VAR ch : CHAR;
   Zeit : INTEGER;

BEGIN
  REPEAT                                          (* aeussere Schleife *)
    REPEAT                                         (* innere Schleife  *)
      CLRSCR;                                  *( loescht den Bildschirm *)
      GOTOXY(3,5); (* setzt den Cursor auf die 3.Spalte in der 5.Zeile *)
      WRITE ('Soll die Schleife verlassen werden? (j/n)');
      READLN (ch)
    UNTIL (ch='j') OR ( (ch='n');                   (* innere Schleife *)
    WRITELN; WRITELN ('Gleich geht es weiter');
    FOR Zeit:=25000 DOWNTO 1 DO;                     (* leere Schleife *)
  UNTIL ch='j';                                   (* aeussere Schleife *)
  CLRSCR;
  GOTOXY(10,15);
  WRITELN ('ENDE')
END.
```

Zunaechst faellt auf, dass REPEAT-Schleifen genau wie FOR-Schleifen geschachtelt werden koennen.
Was leistet nun das Programm? Wird ein von j oder n verschiedener Buchstabe eingegeben, so wird der Bildschirm geloescht, der Cursor auf die vorgegebene Stelle gesetzt, die Frage ausgegeben und erneut ein Zeichen eingelesen. Man beachte dabei, dass der Rechner sehr wohl zwischen j und J unterscheidet. Die innere Schleife wird also erst verlassen, wenn ein ganz bestimmtes Zeichen (hier:j bzw.n) eingelesen wurde. Hat die Variable ch den Wert n, wird die innere Schleife verlassen und das Programm beendet.
Bemerkenswert ist ferner, dass die Anweisungen (!) zwischen REPEAT..UNTIL nicht durch BEGIN und END zu einem Anweisungsblock zusammengefasst sind.
Die reservierten Woerter REPEAT..UNTIL uebernehmen hier die klammernde Funktion, die sonst BEGIN und END vorbehalten ist. Auch das Semikolon vor UNTIL kann entfallen. Ist es vorhanden, so wird es - wie auch vor END - als leere Anweisung behandelt.
Wir wollen nun noch ein kleines Problem aus der Mathematik loesen: die Berechnung der Eulerschen Zahl e. Dazu benutzen wir das von den Mathematikern bereitgestellte Wissen, dass sich e ueber eine Produktreihe berechnen laesst:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \ldots + \frac{1}{n!}$$

$$n! = 1 * 2 * 3 * 4 * \ldots * n$$

Die Zahl e soll mit der groessten dem Rechner moeglichen Genaugkeit berechnet werden. Dazu brauchen wir eine Schleife, die die oben dargestellten Brueche so lange addiert, bis sich keine Aenderung des Ergebnisses mehr zeigt.
Als Abbruchbedingung koennte man den zuletzt berechneten Wert fuer e mit dem neuen Wert vergleichen. Durch Rundungsfehler bei REAL-Zahlen(siehe Kap.3.2) fuehrt dies jedoch selten zu einem Abbruch, denn gleiche Zahlen sind fuer den Rechner oft nicht gleich. Wir wuerden damit eine unendlich lange Schleife konstruieren.
Statt dessen benutzen wir eine Hilfsvariable, die den jeweils letzten Wert von e bekommt, subtrahieren die Hilfsvariable vom neuen Wert fuer e und vergleichen diese Differenz mit einer sehr kleinen Zahl (abhaengig von der Rechengenauigkeit des verwendeten Rechners - nicht Null !!).
Nun folgt das Programm:

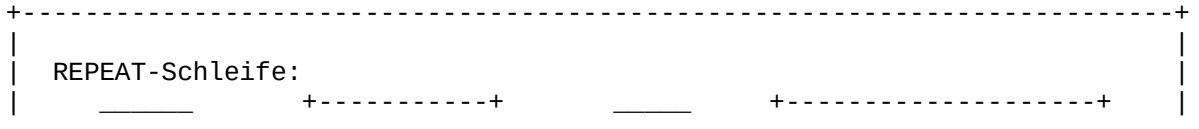
```
PROGRAM Euler;

CONST Delta=1E-10;
VAR   e, Hilf, n, Nenner, Differenz : REAL;

BEGIN
  e:=1;
  Nenner:=1;
  n:=1;
  REPEAT
    Hilf:= e;
    e:= e + 1 / Nenner;
    n:= n + 1;
    Nenner:=Nenner * n;
    Differenz:=ABS(Hilf - e)
  UNTIL Differenz < Delta;
  WRITELN ('Nach  ',n:4:0,' Durchlaeufen ist e =',e:12:10)
END.
```

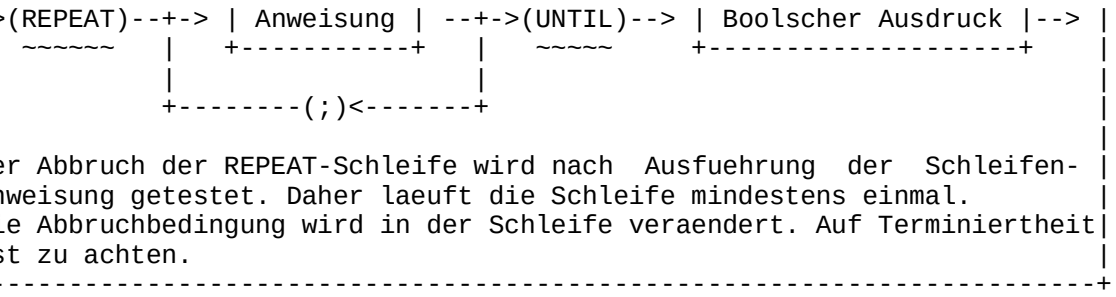
Merke:
- Die Abbruchbedingung der REPEAT-Schleife muss ein Ausdruck oder eine Variable vom Typ BOOLEAN sein (siehe auch Kap.3.5).
- Die Abbruchbedingung muss in der Schleife veraendert werden, da die Schleife sonst unendlich lange laeuft.
- Die Schleife wird beendet wenn die Abbruchbedingung wahr ist.

Die besondere Eigenschaft der REPEAT-Schleife liegt darin, dass die Abbruchbedingung am Ende der Schleife geprueft wird. Daher laeuft eine REPEAT-Schleife mindestens einmal.

Terminiertheit

Es ist stets genau darauf zu achten, dass die Schleife auch tatsaechlich terminiert (beendet) ist. Wir muessen dazu folgendes ueberpruefen:
- Ist in der Abbruchbedingung ueberhaupt eine Variable vorhanden, die der Bedingung den Wert TRUE geben kann?
- Wird die Abbruchbedingung jemals erreicht?





4.3 Die WHILE-Schleife

Beim Anblick der Museumseisenbahn - besonders wenn sie von einer Dampflokomotive gezogen wird - kommen haeufig nostalgische Gefuehle auf. Benutzen wir einen solchen Zug einmal zur Veranschaulichung des dritten Typs der in Pascal moeglichen Schleifen. Dazu lassen wir ihn, wie bei den entsprechenden touristischen Attraktionen ueblich, eine ringfoermig angelegte Strecke befahren. Nur zum Auffuellen von Wasser und Brennstoff und zum Abstellen des Zuges ueber Nacht wird die Hauptstrecke verlassen und das (einzige) Abstellgleis aufgesucht. Die Betriebsgesellschaft einer solchen Museumseisenbahn stellte nun eines Tages einen arg beschraenkten, aber trotzdem sehr diensteifrigen Lokomotivfuehrer ein. Dieser verliess trotz mannigfaltiger Hinweise und Erlaeuterungen jedesmal die Hauptstrecke, wenn der Zug an der Weiche zum Abstellgleis angelangt war. Zum Ausgleich stellte er ihn des oefteren nachts auf der Hauptstrecke ab. Daraufhin erhielt der Lokfuehrer die untenstehende "Dienstanweisung". Da er sich peinlich genau an sie hielt, waren somit alle Unregelmaessigkeiten beseitigt.

```
BEGINNE                                  (* Tagewerk *)
Zug vom Abstellgleis fahren;

WIEDERHOLE
   SOLANGE noch Kohlen und Wasser vorhanden
   TUE     Strecke befahren;
   Kohlen und Wasser fassen;
BIS es Nacht ist;

Zug auf Abstellgleis fahren
ENDE.                          (* Feierabend *)
```

Eingeweihte wissen, dass es bei dieser "Dienstanweisung" um die umgangssprachliche Formulierung eines Algotithmus handelt. Untersuchen wir ihn einmal auf seine Struktur: Die uns bereits bekannte Schleife WIEDERHOLE..BIS enthaelt hier die Anweisung

```
SOLANGE noch Kohlen und Wasser vorhanden TUE Strecke befahren;
```

Anweisungen dieses Typs werden wir nunmehr als dritte Moeglichkeit der Wiederholung kennenlernen. In unserem Beispiel folgt dem reservierten Wort SOLANGE (engl.WHILE) eine Bedingung. Ist sie wahr, so wird die Anweisung bzw.der in BEGIN und END gefasste Anweisungsblock ausgefuehrt, der dem reservierten Wort TUE (engl.DO) folgt. Der Lokomotivfuehrer unserer Museumseisenbahn hat also, bevor er die Hauptstrecke befaehrt, zu pruefen, ob noch genuegend Brennstoff und Wasser vorhanden sind. Erst wenn dies der Fall ist, legt unser Zug eine weite Runde auf der Hauptstrecke zurueck. Die Abzweigung zum Abstellgleis wird erst dann benutzt, wenn es an einem von

beidem fehlt; d.h. die Bedingung falsch wird.

Zusammengefasst hat die WHILE-Schleife folgende Form:

```
WHILE Bedingung DO Anweisung/-sblock
```

In einem weiteren kleinen Programmbeispiel wollen wir die Quersumme einer ganzen Zahl berechnen und ausgeben.
Dazu wird eine ganze Zahl eingegeben und folgende Berechnung angestellt: Solange die Zahl noch groesser als Null ist, wird der Rest beim Teilen durch 10 (mit MOD), d.h. die jeweils letzte Ziffer, aufaddiert und die Zahl durch 10 geteilt.

Zur Loesung des Problems benutzen wir die WHILE-Schleife. Sie hat die Form: WHILE Bedingung DO Anweisung(en) oder auf deutsch: SOLANGE Bedingung TUE Anweisung(en).
Das Programm sieht dazu folgendermassen aus:

```
PROGRAM Quersumme;

  VAR n, Summe : INTEGER;

  BEGIN
    WRITE ('Eingabe der Zahl: ');
    READLN (n);
    Summe := 0;
    WHILE n > 0 DO BEGIN
      Summe := Summe + n MOD 10;
      n := n DIV 10
    END; (* der WHILE-Schleife *)
    WRITELN ('Die Quersumme ist: ', Summe:4)
  END.
```

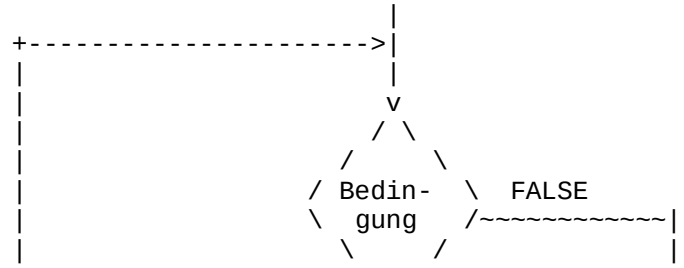
Solange die Bedingung den Wahrheitswert TRUE hat, laeuft die Schleife. Folgendes muss dabei beachtet werden:

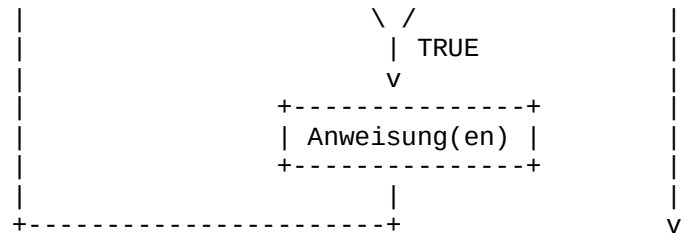
- Die Bedingung der WHILE-Schleife muss ein Ausdruck oder eine Variable vom Typ BOOLEAN sein (siehe auch Kap.3.5.)

- Die Bedingung muss in der Schleife veraendert werden, da die Schleife sonst unendlich lange laeuft.

- Die Schleife wird beendet, wenn die Bedingung falsch ist.

Die besondere Eigenschaft einer WHILE-Schleife liegt darin, dass die Bedingung am Anfang der Schleife geprueft wird. Daher laeuft eine WHILE-Schleife moeglicherweise keinmal. In einem Flussdiagramm laesst sich das wie folgt darstellen:

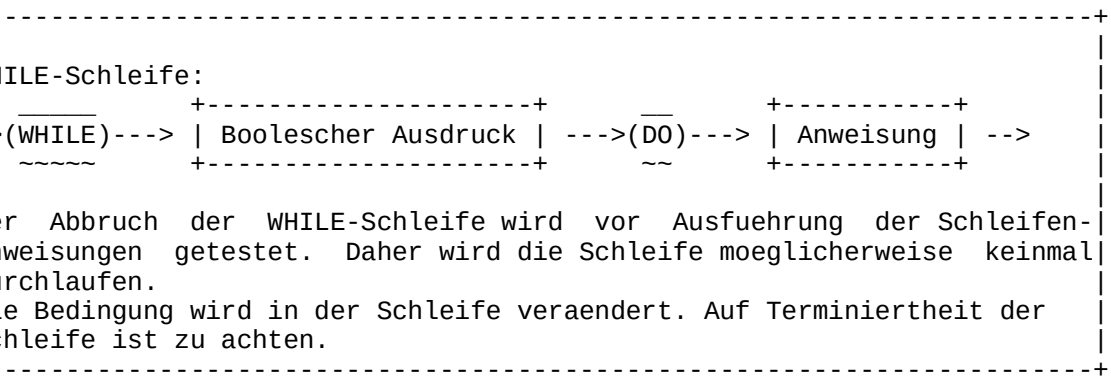




Terminiertheit

Es ist peinlich genau darauf zu achten, dass die Schleife auch tatsaechlich terminiert (beendet) ist. Wir muessen dazu folgendes ueberpruefen:

- Ist in der Bedingung ueberhaupt eine Variable vorhanden, die der Bedingung den Wert TRUE geben kann ?
- Wird die Bedingung jemals erreicht ?



Unterschied REPEAT - WHILE

An dieser Stelle sei noch einmal auf die Unterschiede der Schleifenarten hingewiesen. Bei der REPEAT-Schleife wird die Bedingung nach Ausfuehrung der Schleifenanweisungen auf den Wahrheitswert TRUE ueberprueft, bei der WHILE-Schleife vorher.
Die REPEAT-Schleife laeuft, waehrend die Bedingung FALSE zutrifft (d.h. bis sie TRUE ist) - die WHILE-Schleife laeuft, solange die Bedingung den Wert TRUE hat.
Man kann beide Schleifenformen durch die jeweils andere ersetzen. Dazu betrachten wir unser Programm QUERSUMME mit einer REPEAT-Schleife:

```
PROGRAM Quersumme;

  VAR n, Summe : INTEGER;

  BEGIN
    WRITE ('Eingabe der Zahl: ');
    READL (n);
    Summe := 0;
    REPEAT
      Summe := Summe + n MOD 10;
      n := n DIV 10
    UNTIL n  <= 0;
```

```
   WRITELN ('Die Quersumme ist: ', Summe:4)
END.
```

5.1 Entscheidungen mit IF

Entscheidungen gibt's...

```
+-------------------------------------------------------------------------+
|                                                                         |
|   Soll ich mir einen Computer kaufen oder nicht ?                       |
|   Diese Frage kann ich mir ganz einfach beantworten:                    |
|   Wenn                                                                  |
|     ich genug Feld habe und  einen Computer besitzen moechte,           |
|   dann                                                                  |
|     gehe ich in einen Computer-Shop und kaufe einen.                    |
|   Was ist aber, wenn die Bedingungen nicht zutreffen ?                  |
|   Nun, dann geht das Leben eben weiter.                                 |
|                                                                         |
+-------------------------------------------------------------------------+
```

In diesem Monolog (den manch ein Computerfan kennt) kommt eine Entscheidung vor, die von zwei Bedingungen abhaengt. Beide Bedingungen muessen zutrefffen, was durch das Wort " und " bestimmt wird. Wir kennen schon Ausdruecke und Verknuepfungen dieser Art. Sie sind vom Typ BOOLEAN, denn sie koennen die Wahrheitswerte " wahr " oder " falsch " annehmen. Wenn also beide Bedingungen " wahr " sind, dann habe ich gruenes Licht fuer den Computerkauf. Sollte das nicht der Fall sein, so geht mein Leben programmgemaess weiter.
Auch in der Programmiersprache Pascal gibt es diese Situation. Oft sollen eine oder mehrere Anweisungen nur dann ausgefuehrt werden, wenn eine Bedingung oder eine Kombination von Bedingungen erfuellt ist. Dazu uebersetzen wir einfach WENN..DANN ins Englische und erhalten IF..THEN. Dieses sind genau die reservierten Woerter fuer eine bedingte Ausfuehrung von Aweisungen.
Die Form ist: **IF logischer Ausdruck THEN Anweisung(en);**

Wenn mehrere Anweisungen ausgefuehrt werden sollen, so werden sie wieder mit BEGIN..END zusammengefasst. Der logische Ausdruck muss den Wahrheitswert TRUE haben, damit die Anweisung(en) ausgefuehrt wird (werden), anderenfalls wird das Programm weiter fortgefuehrt. Aus Kap.3.5 kennen wir schon logische Ausdruecke. Beispiele sind:

```
B            : wobei B vom Datentyp BOOLEAN ist
B=TRUE
B=FALSE
X < Y        : wobei X und Y vom gleichen Typ sind
(X < Y) AND (X > Z)
NOT (A = B)
```

Hinweis: Wenn mehrere logische Ausdruecke durch Verknuepfungsoperatoren (AND, OR, NOT) zusammengefasst werden, so ist auf korrekte Klammerung zu achten !
Wie wollen uns nun ein kleines Beispiel anschauen:

```
PROGRAM Waswohl;
```

```
VAR I : INTEGER;

BEGIN
  FOR I:=1 TO 99 DO BEGIN
    WRITE (I:3);
    IF I MOD 9 = 0 THEN WRITELN
  END (* der FOR-Schleife *)
END.
```

Frage: Was bewirkt dieses kleine Programm ?

Einerseits koennten wir es in den Rechner tippen und ausprobieren, andererseits laesst sich die Funktion auch auf dem Papier ermittteln:
Im Programm laeuft eine Schleife von 1 bis 99. Bei jedem Schleifendurchlauf wird die Schleifenvariable ohne Zeilenvorschub in einem Feld von 3 Zeichen geschrieben. Wenn sich allerdings die Variable durch 9 teilen laesst (I MOD 9 = 0), dann wird ein Zeilenvorschub gemacht, so dass die Zahlen in Neunerkolonnen geschrieben werden:

```
 1   2   3   4   5   6   7   8   9
10  11  12  13  14  15  16  17  18
19  20  21  22  23  24  25  26  27
28  29  30  31  32  33  34  35  36
37  38  39  40  41  42  43  44  45
46  47  48  49  50  51  52  53  54
55  56  57  58  59  60  61  62  63
64  65  66  67  68  69  70  71  72
73  74  75  76  77  78  79  80  81
82  83  84  85  86  87  88  89  90
91  92  93  94  95  96  97  98  99
```

In dem vorangegangenen Problem handelte es sich um eine einseitige Entscheidung, denn fuer den Fall, dass die Bedingung das Ergebnis FALSE hat, wurde mit der naechsten Anweisung fortgefahren.
Nun sind in Pascal aber auch zweiseitige Entscheidungen vorgesehen. Das Woertchen ELSE (andernfalls) gibt es uns die Moeglichkeit, bei negativem Ausgang (FALSE) der Entscheidung einen anderen Anweisungsteil ausfuehren zu lassen.
Die Form ist: **IF logischer Ausdruck THEN Anweisung(en) ELSE Anweisung(en);**

Achtung: Vor ELSE darf kein Semikolon stehen !

Wir wollen nun ein Beispielprogramm entwerfen: Zahlenraten. Der Rechner erzeugt eine Zufallszahl zwischen 0 und 99. Der Benutzer darf 7mal eine Zahl raten. Wenn die Zahl groesser oder kleiner ist als die Zufallszahl, dann wird dies dem Benutzer mitgeteilt. Bei richtiger Eingabe der Zahl begluekwuenscht der Rechner den Benutzer.

Bemerkung: Die Erzeugung von Zufallszahlen zwischen 0 und 99 geschieht in Turbo Pascal durch die Funktion RANDOM (99). Allgemein erzeugt RANDOM (n) eine ganze Zahl als Zufallszahl zwischen 0 und n.

Soll bei jedem neuen Durchlauf des Programms eine neue Zufallszahl erzeugt werden, so geben wir RANDOMIZE ein. (siehe auch Kap.6)
Das Programm sieht so aus:

```
PROGRAM Zahlenraten;
```

```
VAR Ratezahl, Zufallszahl, I : INTEGER;
    Geraten : BOOLEAN;

BEGIN
  WRITELN ('Zahlenraten zwischen 0 und 99');
  WRITELN;
  RANDOMIZE;     (* erzeugt neue Zufallszahl *)
  Zufallszahl := RANDOM (99);   (* Zahl zwischen 0 und 99 *)
  Geraten := FALSE;
  I := 1;
  REPEAT
    WRITE ('Ratezahl: ');
    READLN (Ratezahl);
    IF Ratezahl = Zufallszahl THEN Geraten := TRUE
    ELSE BEGIN
      IF Ratezahl > Zufallszahl THEN WRITELN ('zu gross')
      ELSE WRITELN ('zu klein')
    END; (* von ELSE *)
    I := I + 1
  UNTIL Geraten OR (I > 7);
  IF Geraten THEN BEGIN
    WRITELN ('Herzlichen Glueckwunsch');
    IF I < 3 THEN WRITELN ('Das war Spitze!')
  END (* von IF *)
  ELSE WRITELN ('Die Zahl war: ',Zufallszahl:3)
END.
```

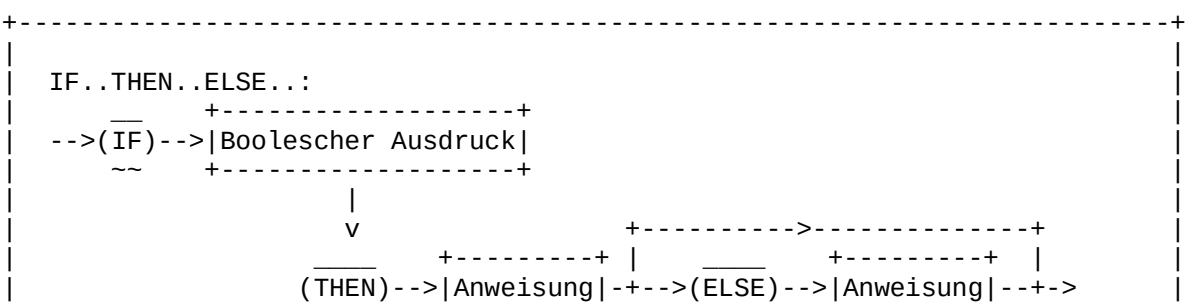
Aufgabe: Veraendern Sie das Programm so, dass der Benutzer entscheiden kann, ob er das Spiel noch einmal spielen will.

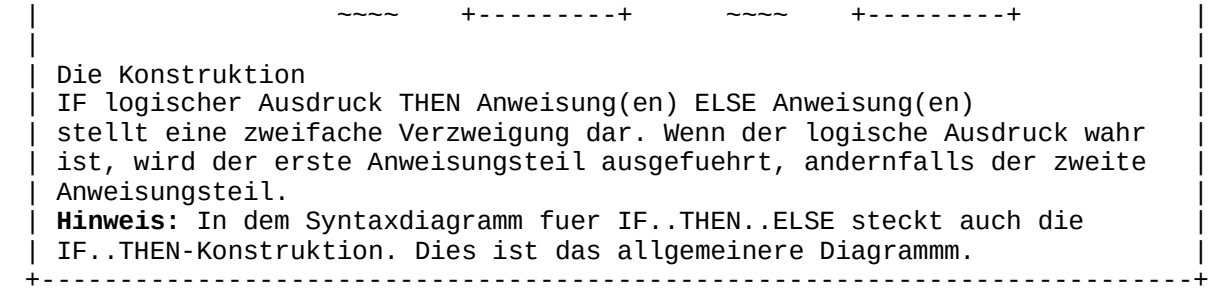
In dem vorangegangenen Programm wurden auch geschaltete Entscheidungen verwendet. Dies ist selbstverstaendlich bei beiden IF-Konstruktionen moeglich. Dabei ist zu beachten, dass eine IF-Abfrage, die von einer anderen abhaengt, nur dann ausgefuehrt wird, wenn die erste Aufgabe ein wahres Ergebnis hat.

Beispiel:

```
IF Tag=13 THEN IF Wochentag=Freitag  THEN WRITELN ('Vorsicht heute!');
IF (Tag=13) and (Wochentag=Freitag)THEN WRITELN ('Vorsicht heute!');
```

Beide Konstruktionen haben den gleichen Effekt. Nur wenn die Variable Tag gleich 13 ist **und** die Variable Wochentag gleich Freitag ist, wird der Text geschrieben. Die erste (geschaltete) Konstruktion hat aber den Vorteil, dass **nur**, wenn Tag gleich 13 ist, der Wochentag auch noch ueberprueft wird. Dies spart natuerlich Rechenzeit, da in der zweiten Konstruktion beide Variablen immer geprueft werden.





Fehlerquellen: Bei geschachtelten Verzweigungen kann es insbesondere mit der IF..THEN..ELSE-Konstruktion leicht zu Fehlern kommen, wenn man nicht sorgfaeltig plant.
Wir wollen uns ein Beispiel einer Schachtelung ansehen:

```
IF Spannung >= 2 THEN
  IF Spannung > 20 THEN
    IF Spannung > 100 THEN
      WRITELN('Bereich ueberschritten')
    ELSE WRITELN('grosser Bereich')
  ELSE WRITELN('normaler Bereich')
ELSE WRITELN('Spannung unter " V');
```

Diese Messbereichsauswahl fuer ein Spannungsmessgeraet hat folgende Funktion:

Bei Spannung groesser als 100V:	Messbereichsueberschreitung.
Bei Spannungen zwischen 20V und 100V:	grosser Bereich.
Bei Spannungen zwischen 2V und 20V:	normaler Bereich.
Bei Spannungen unter 2V:	Messbereich unter 2V.

5.2 Entscheidungen mit CASE

Wuerden sie folgende Programmsequenz in ihrem Programm dulden ?

```
IF Tag=0 THEN WRITELN('Sonntag')
  ELSE IF Tag=1 THEN WRITELN('Montag')
    ELSE IF Tag=2 THEN WRITELN('Dienstag')
      ELSE IF Tag=3 THEN WRITELN('Mittwoch')
        ELSE IF Tag=4 THEN WRITELN('Donnerstag')
          ELSE IF Tag=5 THEN WRITELN('Freitag')
            ELSE WRITELN('Samstag');
```

Furchtbar unuebersichtlich und schreibintensiv. Als Alternative zu dieser Mehrfachentscheidung mit IF..THEN..ELSE bietet Pascal die Mehrfachentscheidung mit CASE:

```
CASE Tag OF
 0:WRITELN('Sonntag');
 1:WRITELN('Montag');
 2:WRITELN('Dienstag');
 3:WRITELN('Mittwoch');
 4:WRITELN('Donnerstag');
 5:WRITELN('Freitag')
ELSE
 WRITELN('Samstag')
```

```
   END;
```

Wir haben es hier mit einer Fallunterscheidung zu tun. Fuer den Fall (CASE), dass Tag von (OF) der Form einer der folgenden Faelle ist, wird ein Anweisungsteil ausgefuehrt. Nach der Aufzaehlung der moeglichen Faelle kann nach ELSE ein Anweisungsteil folgen, der ausgefuehrt wird, wenn keiner der Faelle zutrifft. Die ganze Fallunterscheidung wird mit END abgeschlossen (vor dem END braucht kein Semikolon zu stehen).
Vor den Doppelpunkten der CASE-Anweisung koennen auch mehrere Werte stehen, die dann durch Kommata getrennt werden.

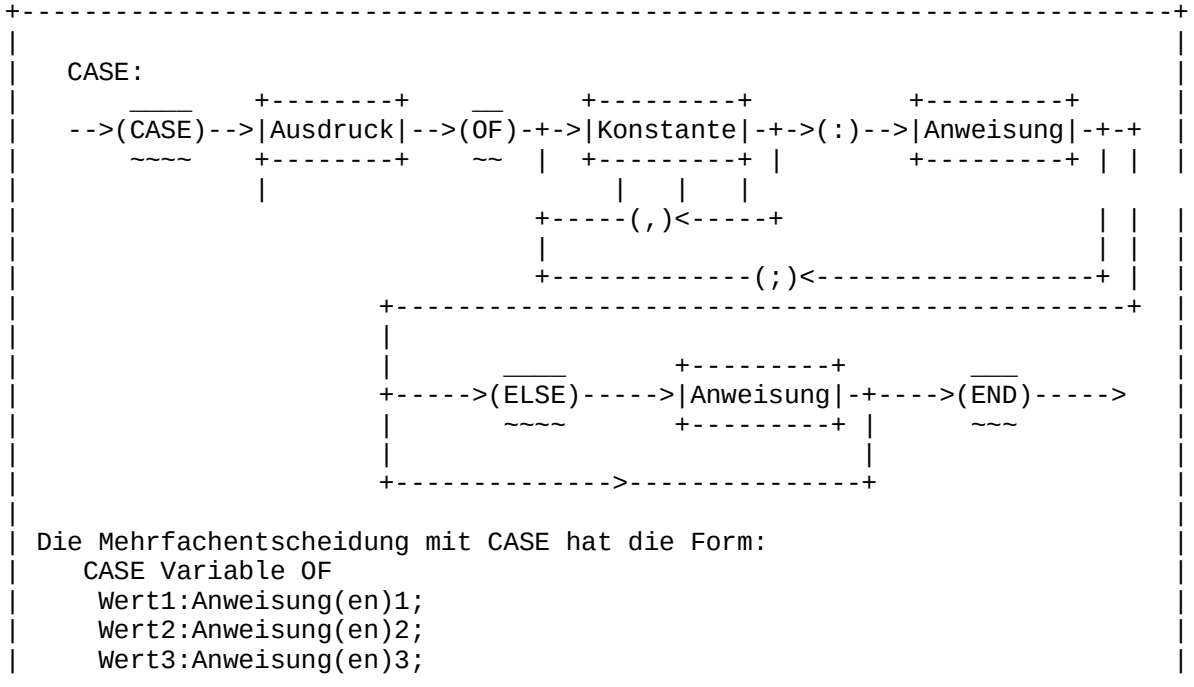
Beispiel:

```
CASE Frage OF
     'E','e':Eingabe;
     'A','a':Ausgabe;
     'S','s':Sortieren;
     'D','d':Drucken;
     'F','f':Finden;
     'Z','z':Ende
    END;
```

Eine solche CASE-Amweisung koennte aus einem Menue stammen. Hierbei wird dem Benutzer z.B. auf dem Bildschirm angeboten:

```
   Datenverarbeitung, waehlen sie:
    E(ingabe von Daten
    A(usgabe von Daten
    S(ortieren
    D(rucken
    F(inden nach Kriterien
    Z(um Schluss
```

Der Benutzer braucht nur den ersten Buchstaben (Eingabe z.B. mit READ oder READ(KBD...)) seiner Wahl einzutippen, und das Programm fuehrt entsprechende Anweisungen (evtl. ganze Prozeduren) aus.



```
|     ...                                                                  |
|     Wertn:Anweisung(en)n                                                 |
|    ELSE                                                                  |
|     Anweisung(en)                                                        |
|    END;                                                                  |
|                                                                          |
| Wenn ein Anweisungsteil von mehreren Werten abhaengen soll, werden die   |
| Werte durch Kommata getrennt.                                            |
| Die Variable muss von einem skalaren Datentyp, nicht aber vom Typ REAL   |
| sein, d.h.Aufzaehlungstyp, Unterbereich, INTEGER, CHAR, BOOLEAN.         |
+--------------------------------------------------------------------------+
```

6.1 Prozeduren

Machen wir noch einen kleinen Ausflug in die Mathematik. Es gibt eine Reihe von Taschenrechnern, die die Bruchrechnung beherrschen. Schade, dass unser Computer das nicht kann, oder doch?
Eigentlich ist die Sache doch gar nicht so schwer. Wir wollen zuerst einmal ein kleines Programm schreiben, das einen Bruch kuerzt. Hier ist es schon:

```
PROGRAMM Bruch;

 VAR Z1,Z2,N1,N2 : INTEGER; (* fuer Zaehler und Nenner *)

 BEGIN
  Eingabe;
  Kuerze;
  Ausgabe
 END.
```

Es handelt sich bisher nur um den Programmkopf und das Hauptprogramm. Die scheinbar neuen Befehle Eingabe, Kuerze und Ausgabe muessen noch zum Leben erweckt werden. Des Raetsels Loesung liegt darin, dass wir Prozeduren benutzen.
Eine Prozedur ist ein Programmteil (auch Unterprogramm genannt), der unter einem Namen aufgerufen werden kann.
Die Prozedur wird in der Form

```
PROCEDURE <Name>;
 CONST...;
 VAR...;
 BEGIN
  <Prozedurtext>
 END;
```

geschrieben.
Die Prozedurtexte sind Teile des Deklarationsteils des Programms. Vom Programm aus wird die Prozedur aufgerufen, indem ihr Name in einer Anweisungszeile genannt wird.

- Prozeduren koennen auch von anderen Prozeduren aus aufgerufen werden.
- Prozeduren koennen mehrmals aufgerufen werden.
- Es ist zu beachten, dass eine bestimmte Reihenfolge eingehalten wird; eine Prozedur, die von einer anderen aufgerufen wird, muss auch vor dieser stehen.

Sie haben sicher bemerkt, dass Prozeduren auch eigene Variablen und Konstanten haben koennen. Dabei gelten folgende Regeln:

- Variablen und Konstanten des Hauptprogramms (vor den Prozeduren deklariert) nennen wir "global". Sie haben im gesamten Programm Gueltigkeit, d.h. auch in den Prozeduren.
- Variablen und Konstanten einer Prozedur werden "lokal" genannt. Sie haben nur in dieser Prozedur Gueltigkeit.

- Sollten Variablen und Konstanten von Hauptprogramm und Prozeduren denselben Namen tragen (was erlaubt ist), so gilt immer der lokale Name.

Machen Sie moeglichst oft Gebrauch von lokalen Variablen. Durch sie ist die Prozedur nicht mehr so stark (oder gar nicht) abhaengig von Gegebenheiten des Hauptprogramms, und so kann die Prozedur eventuell leicht in anderen Programmen verwendet werden.
Nun aber zurueck zu unserem Mathematikproblem. Wir koennen einen Bruch in sehr einfacher Weise kuerzen, indem wir versuchen, Zaehler und Nenner mit Eins angefangen durch immer groesser werdende Zahlen zu teilen. Die groesste Zahl, durch die sowohl Zaehler als auch Nenner teilbar ist, stellt den groessten gemeinsamen Teiler (ggT) dar. Mit ihm kuerzen wir dann den Bruch. Die groesste Zahl, die wir finden, kann hoechstens das Minimum von Zaehler und Nenner sein. Die kleinste Zahl ist sicher Eins, weil sich beide durch Eins teilen lassen.

```
PROCEDURE Kuerze;
 VAR I, Teiler, Min : INTEGER;
 BEGIN
  IF Z1<N1 THEN Min:=Z1
           ELSE Min:=Z2;
  FOR I:=1 TO Min DO
   IF (Z1 MOD I =0) AND (N1 MOD I =0)
     THEN Teiler:=I;
  Z2:=Z1 DIV Teiler;
  N2:=N1 DIV Teiler
 END;
```

Die beiden anderen Prozeduren sind so einfach, dass wir sie sofort im gesamten Programm erstellen koennen.

```
PROGRAMM Bruch;

VAR Z1,Z2,N1,N2 : INTEGER;
  PROCEDURE Eingabe;
   BEGIN
    WRITE ('Zaehler: ');
    READLN (Z1);
    WRITE ('Nenner: ');
    READLN (N1)
   END;

  PROCEDURE Kuerze;
   VAR I, Teiler, Min : Integer;
   BEGIN;
    IF Z1<N1 THEN Min:=Z1
             ELSE Min:=Z2;
    FOR I:=1 TO Min DO
     IF (Z1 MOD I =0) AND (N1 MOD I =0)
       THEN Teiler:=I;
    Z2:=Z1 DIV Teiler;
    N2:=N1 DIV Teiler
   END;

  PROCEDURE Ausgabe;
   BEGIN
    WRITELN (Z1:5,'   ',Z2:5);
    WRITELN ('----- = -----');
```

```
       WRITELN (N1:5,'   ',N2:5)
      END;

     BEGIN  (* Hauptprogramm *)
      Eingabe;
      Kuerze;
      Ausgabe
     END.
```

Nun kann unser Computer endlich kuerzen. Allerdings sollten wir uns damit noch nicht begnuegen. Die Prozedur Kuerze hat immer noch einen kleinen Mangel. Sie braucht, so wie wir sie geschrieben haben, unbedingt die globalen Variablen Z1,Z2,N1,N2. Dieser Mangel laesst sich dadurch beheben, dass wir eine andere Form von Prozeduren verwenden.

Prozeduren mit Variablenuebergabe

In Pascal gibt es die Moeglichkeit, einer Prozedur eine oder mehrere Variablen zu uebergeben und diese dann von der Prozedur veraendert zurueckzubekommen. Die Form sieht dann folgendermassen aus:

```
   PROCEDURE <Name> (VAR .....);
    CONST...;
    VAR...;
    BEGIN
     ...
    END;
```

In der Klammer hinter dem Prozedurnamen steht die sogenannte Parameterliste. Hier werden die Variablen aufgefuehrt, die von der Prozedur veraendert werden sollen.

Beispiele:

```
   PROCEDURE Test (VAR I: INTEGER);
 oder
   PROCEDURE Mehr (VAR A,B : CHAR; VAR C : REAL);
```

Nach diesen Beispielen muss die Prozedur Test mit genau einer Variablen vom Typ INTEGER aufgerufen werden. Die Prozedur Mehr muss dagegen mit zwei Variablen vom Typ CHAR und einer Variablen vom Typ REAL aufgerufen werden - und zwar genau in dieser Reihenfolge. Die Variablen werden durch Kommata voneinander getrennt.
Unsere Prozedur Kuerze koennte nun so aussehen:

```
   PROCEDURE Kuerze (VAR A,B : INTEGER);
    VAR I, Teiler, Min : Integer;
    BEGIN
     IF A<B THEN Min:=A
            ELSE Min:=Z2;
     FOR I:=1 TO Min DO
      IF (A MOD I =0) AND (B MOD I =0)
        THEN Teiler:=I;
     A:=A DIV Teiler;
     B:=B DIV Teiler
    END;
```

Der Aufruf der Prozedur mit den beiden INTEGER-Variablen Z und N

```
    Kuerze (Z,N);
```
hat dann den Effekt, dass nach dem Abarbeiten dieser Prozedur die Variablen Z und N gekuerzt sind.

Prozeduren mit Wertuebergabe

Es gibt in Pascal auch noch eine andere Art der Parameteruebergabe an Prozeduren, die Wertuebergabe. Bei dieser Art wird der Prozedur nur ein Wert uebergeben, mit dem die Prozedur arbeitet, der aber nicht veraendert wird. Die Form ist fast identisch mit der Variablenuebergabe, mit der kleinen Aenderung, dass hier das Woertchen VAR in der Parameterliste entfaellt.
Schreiben wir die Prozedur Ausgabe z.B. in dieser Art:

```
   PROCEDURE Ausgabe (A,B,A1,B1 : INTEGER);
    BEGIN
     WRITELN (A:5,'   ',A1:5);
     WRITELN ('----- = -----');
     WRITELN (B:5,'   ',B2:5)
    END;
```

Ein Aufruf der Prozedur `Ausgabe (12,24,1,2);` haette dann die Ausgabe

```
12    1
-- = --
24    2
```

zur Folge. Bemerkenswert ist, dass wir die Prozedur Ausgabe auch mit vier Variablen vom Typ INTEGER oder mit einer Mischung aus Variablen und Konstanten haetten aufrufen koennen. Es muessen halt nur vier Werte (Variablen oder Konstanten) des richtigen Typs in der richtigen Reihenfolge uebergeben werden.
Unser Programm koennte nun folgendermassen aussehen:

```
   PROGRAMM Bruch;

   VAR Z1,Z2,N1,N2 : INTEGER;

    PROCEDURE Eingabe (VAR Z,N : INTEGER);
     BEGIN
      WRITE ('Zaehler: ');
      READLN (Z);
      WRITE ('Nenner: ');
      READLN (N)
     END;

   PROCEDURE Kuerze (VAR A,B : INTEGER);
    VAR I, Teiler, Min : INTEGER;
    BEGIN
     IF A<B THEN Min:=A
            ELSE Min:=B;
     FOR I:=1 TO Min DO
      IF (A MOD I =0) AND (B MOD I =0)
        THEN Teiler:=I;
     A:=A DIV Teiler;
     B:=B DIV Teiler
    END;

   PROCEDURE Ausgabe (A,B,A1,B1 : INTEGER);
```

```
    BEGIN
     WRITELN (A:5,'   ',A1:5);
     WRITELN ('----- = -----');
     WRITELN (B:5,'   ',B1:5)
    END;

   BEGIN  (* Hauptprogramm *)
    Eingabe(Z1,N1);
    Z2:=Z1;  N2:=N1;
    Kuerze(Z2,N2);
    Ausgabe(Z1,N1,Z2,N2)
   END.
```

Fassen wir noch einmal zusammen: Prozeduren sind Unterprogramme, die einen Namen haben, unter dem sie von anderen Programmteilen aus aufgerufen werden. Sie koennen die Variablen und Konstanten des Hauptprogramms benutzen (globale Variable und Konstante) oder eigene Variable oder Konstante (lokale Variable und Konstante) haben, die im Deklarationsteil der Prozedur erklaert werden.
Ausserdem koennen wir einer Prozedur Werte oder Variable uebergeben, mit denen dann in der Prozedur gearbeitet wird.
Werden der Prozedur Werte (ohne VAR) uebergeben, wird diese Uebergabeart auch "call-by-value" genannt.
Wenn der Prozedur Variablen (mit VAR) uebergeben werden, so werden diese von der Prozedur aufgenommen und nach Abarbeitung wieder geaendert zurueckgegeben. Diese Art wird "call-by-reference" oder "call-by-variable" genannt.
Machen wir uns diesen Unterschied hier noch einmal ganz klar:

```
   PROGRAM Unterschied;
    VAR x : Integer;

    PROCEDURE Aenderenichts (a : INTEGER);
     BEGIN
      a:=a*2;
      WRITELN (a)
     END;

    PROCEDURE Aenderewas (VAR a: INTEGER);
     BEGIN
      a:=a*2;
      WRITELN (a)
     END;

    BEGIN  (* Hauptprogramm *)
     x := 5;
     WRITELN (x);
     Aenderenichts (x);
     WRITELN (x);
     Aenderewas (x);
     WRITELN (x)
    END;
```

Das Programm gibt folgende Zahlenreihe aus:

```
  5
 10
  5
 10
```

Warum ?
5 : x hat den Wert 5 bekommen, der mit WRITELN (x) ausgegeben wird.
10 : Der Wert von x wird an die Prozedur Aenderenicht uebergeben, hier mit 2 multipliziert und mit WRITELN (a) ausgegeben.
5 : Nach Durchlaufen der Prozedur hat sich die Variable x nicht geaendert. Mit WRITELN (x) wird sie geschrieben.
10 : Die Variable x wird der Prozedur Aenderewas uebergeben, hier verdoppelt, mit WRITELN (a) ausgegeben und der Wert von a wird im Hauptprogramm der Variablen x wieder zurueckgegeben.
10 : Nun hat x also den Wert 10, der mit WRITELN (x) ausgegeben wird.

Fehlerquelle: Folgender Aufruf der Prozedur Aenderewas waere voellig falsch:
Aenderewas(5); falsch !
Denn der Prozedur muss eine Variable ("call-by-variable") uebergeben werden, damit der Wert dieser Variablen nach Durchlaufen der Prozedur veraendert werden kann. Bei der Konstanten 5 waere dies nicht moeglich.
In diesem kleinen Beispiel haben Sie gesehen, dass die Uebergabeparameter der Prozeduren andere Namen haben als die Variablen im Hauptprogramm. Dies ist nicht unbedingt noetig. Haetten wir die Variable des Hauptprogramms auch x genannt, so waere sie eine andere Variable (ein anderer Speicherplatz) als in den Prozeduren!
Ein weiteres kleines Programmbeispiel soll verdeutlichen, dass wir einer Prozedur auch mehrere Parameter uebergeben koennen, die nicht von der gleichen Uebergabeart sind.
Es handelt sich um ein Programm, das zu einem einzugebenen Datum den dazugehoerigen Wochentag ermittelt. Dazu wird in einer Prozedur Berechne, der wir die Werte Tag, Monat und Jahr uebergeben, nach einer hier nicht weiter ausgefuehrten Formel eine Zahl zwischen 0 und 6 ermittelt, die einer vierten Variablen ("call-by-variable") wieder zurueckgegeben wird.

```
PROGRAM Datum;
VAR Tag,Monat,Jahr,Tagnummer : INTEGER;

PROCEDURE Eingabe (VAR Tag,Monat,Jahr : INTEGER);
 VAR Februar : INTEGER;
     Erfolg  : BOOLEAN;

 BEGIN
   WRITELN ('Das Programm ermittelt zu einem beliebigen Datum innerhalb');
   WRITELN ('des Zeitraumes 1701 bis 2099 den dazugehoerigen Wochentag.');
   WRITELN ('Geben Sie bitte das Datum ein: ');
   WRITELN;
   Erfolg := TRUE;
   REPEAT
     IF NOT Erfolg THEN WRITELN ('Falsche Eingabe, neu eingeben: ');
     WRITE ('Geben Sie den Tag ein: ');
     READLN (Tag);
     WRITE ('Geben Sie den Monat ein: ');
     READLN (Monat);
     WRITE ('Geben Sie das Jahr ein: ');
     READLN (Jahr);
     IF Jahr <100 THEN Jahr:=1900+Jahr; (* fuer Schreibfaule *)
     IF (Jahr <1701) OR (Jahr >2099) THEN Erfolg:=FALSE;
     IF Monat >12 THEN Erfolg:=FALSE;
     IF (Jahr MOD 4 =0) THEN Februar:=29 ELSE Februar:=28; (* Sch.jahr*)
     IF (Jahr MOD 100 =0) THEN Februar:=28; (* kein Schaltjahr *)
```

```
      IF (Jahr MOD 400 =0) THEN Februar:=29; (* Schaltjahr *)
      IF Tag >31 THEN Erfolg:=False;
      CASE Monat OF
       2 : Erfolg:=(Tag<=Februar);
       4 : Erfolg:=(Tag<=30);
       6 : Erfolg:=(Tag<=30);
       9 : Erfolg:=(Tag<=30);
      11 : Erfolg:=(Tag<=30);
      END;      (* von Case *)
    UNTIL Erfolg;
  END;    (* von Eingabe *)

PROCEDURE Berechne (T, M, J : INTEGER; VAR Wtag : INTEGER);
 VAR X,Y,Z : INTEGER;

 BEGIN
  IF M>2 THEN M:=M-2
         ELSE BEGIN
          M:=M+10;
          J:=J-1
         END;   (* Else *)
  X:=J MOD 100;
  Z:=J DIV 100;
  Y:=(13*M-1) DIV 5 + X DIV  4 + Z DIV 4;
  Wtag:=(X+Y+T-2*Z) MOD 7
 END; (* Brechne *)

 PROCEDURE Ausgabe (Tagnummer : INTEGER);

  BEGIN
    CLRSCR;
    WRITELN; WRITELN; WRITELN;
    IF (Tagnummer=5) AND (Tag=13) THEN BEGIN
        WRITELN ('An Ihrer Stelle wuerde ich mich Vorsehen, denn der');
        WRITELN (' 13.',Monat,'.',Jahr,' ist ein Feiertag.')
      END
    ELSE
      BEGIN
        WRITE ('Der ',Tag,'.',Monat,'.',Jahr,' ist ein ');
        CASE Tagnummer OF
          0 : WRITELN ('Sonntag');
          1 : WRITELN ('Montag');
          2 : WRITELN ('Dienstag');
          3 : WRITELN ('Mittwoch');
          4 : WRITELN ('Donnerstag');
          5 : WRITELN ('Freitag');
          6 : WRITELN ('Samstag')
        END (* von Case *)
      END  (* von Else *)
   END; (* von Ausgabe *)

  BEGIN (* Hauptprogramm *)
    REPEAT
      CLRSCR;
      Eingabe (Tag,Monat,Jahr);
      Berechne(Tag,Monat,Jahr,Tagnummer);
      Ausgabe(Tagnummer);
      WRITELN;
```

```
      WRITE ('Wuenschen Sie eine erneute Berechnung (J/N)? ');
      READ (Frage)
    UNTIL (Frage = 'N') OR (Frage = 'n')
  END.
```

Zu Beginn dieses Kapitels war die Rede von einem Bruchrechenprogramm, das unserem Computer die gleichen Faehigkeiten geben soll, wie sie ein komfortabler Taschenrechner auch besitzt. Weil wir die Prozeduren Eingabe und Kuerze schon universell verwendbar geschrieben haben, arbeiten wir sie gleich in das Programm ein.
Die Prozedur Ausgabe muss geringfuegig geaendert werden. Ausserdem sind vier Prozeduren fuer die vier Grundrechenarten zu erstellen. Die mathematischen Hintergruende sind sehr einfach und direkt aus den Prozeduren zu verstehen.

Der Einfachheit halber sind die Prozeduren fuer die Grundrechenarten nicht mit Uebergabeparametern geschrieben. Dies wuerde das ganze Programm recht aufwendig machen und steht wahrscheinlich in keinem Verhaeltnis zum Nutzen.

```
  PROGRAM Bruchrechnung;
  VAR Z1,Z2,N1,N2,ZE,NE : INTEGER;
      OP : CHAR;

  PROCEDURE Eingabe (VAR Z,N : INTEGER);
   BEGIN
    WRITE ('Zaehler: ');
    READLN (Z);
    WRITE ('Nenner: ');
    READLN (N)
   END;

  PROCEDURE Kuerze (VAR A,B : INTEGER);
  VAR I, Teiler, Min : INTEGER;
   BEGIN
    IF A<B THEN Min:=A
           ELSE Min:=B;
    FOR I:=1 TO Min DO
     IF (A MOD I =0) AND (B MOD I =0)
       THEN Teiler:=I;
    A:=A DIV Teiler;
    B:=B DIV Teiler
   END;

  PROCEDURE  Ausgabe (Z1,N1,Z2,N2,ZE,NE : INTEGER; OP : CHAR);
   BEGIN
    WRITELN;
    WRITELN (Z1:4,'   ',Z2:4,'     ',ZE:4);
    WRITELN ('---- ',OP,' ---- = ----');
    WRITELN (N1:4,'   ',N2:4,'     ',NE:4)
   END;

  PROCEDURE Plus;
   BEGIN
    ZE := Z1*N2 + Z2*N1;
    NE := N1*N2
   END;

  PROCEDURE Minus;
   BEDIN
```

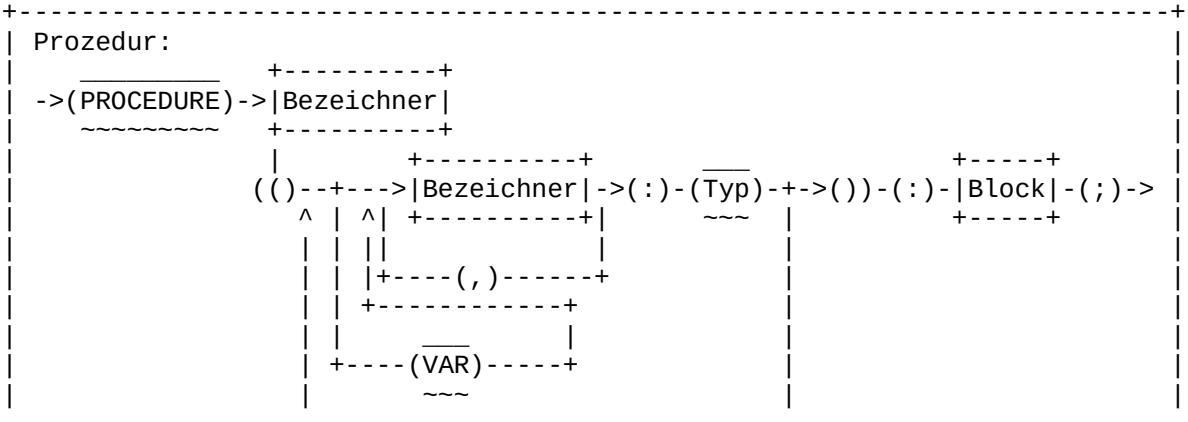
```
     ZE := Z1*N2 - Z2*N1;
     NE := N1*N2
    END;

   PROCEDURE Mal;
    BEGIN
     ZE := Z1*Z2;
     NE := N1*N2
    END;

   PROCEDURE Durch;
    BEGIN
     ZE := Z1*N2;
     ZE := Z2*N1
    END;

   BEGIN  (* Hauptprogramm *)
    WRITELN ('1. Bruch: ');
    Eingabe (Z1,N1);
    WRITELN ('2. Bruch: ');
    Eingabe (Z2,N2);
    WRITE ('Operator (+-*/): ');
    REPEAT READ (OP) UNTIL OP IN ['+','-','*','/'];
    CASE OP OF
     '+' : Plus;
     '-' : Minus;
     '*' : Mal;
     '/' : Durch
    END;
    Kuerze(ZE,NE);
    Ausgabe(Z1,N1,Z2,N2,OP);
   END.
```

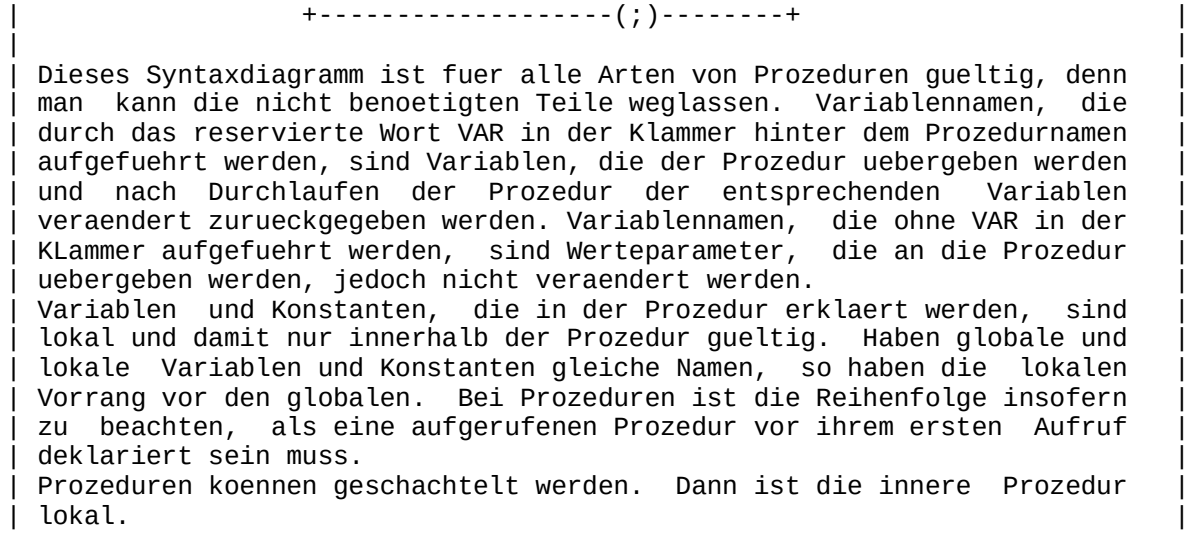
Die leistungsstarken Moeglichkeiten, die Pascal mit seinen Prozeduren bietet, insbesondere durch die Unabhaengigkeit der Prozedur vom Hauptprogramm, sollten wir nicht ungenutzt lassen.
Immerhin liegt hier der groesste Vorteil gegenueber BASIC. In BASIC gibt es zwar auch Unterprogramme, diese arbeiten aber stets mit globalen Variablen (d.h. mit den Variablen des Hauptprogramms). So sind die BASIC-Unterprogramme nur dann transportabel (d.h. in anderen Programmen zu verwenden), wenn die Variablen dieselben Namen haben.
Das ist anders in Pascal.
Hier koennen wir uns eine sogenannte Prozedurbibliothek aufbauen, in der Prozeduren stehen, die wir haeufiger brauchen. Mit den Block-Kopier-Befehlen des Editors lassen sich solche Hilfsroutinen dann sehr einfach ins Programm einfuegen.
Noch ein letztes Beispiel fuer Prozeduren:

```
   RROGRAM Wenigerleer;

    TYPE Stg = STRING[80];
    VAR  Satz : Stg;
         Altleer, Neuleer : INTEGER;

    PROCEDUR Eingabe (VAR S : Stg);
     BEGIN
      WRITELN ('Geben Sie einen Satz ein,');
      WRITELN ('der von ueberfluessigen Leerstellen');
```

```
      WRITELN ('befreit werden soll.');
      READLN (S);
     END;

    PROCEDURE Leerraus (VAR S : Stg);
     VAR i, Ende : INTEGER;
     BEGIN
      Ende := LENGTH(S);
      i := 1;
      WHILE i < Ende DO BEGIN
       IF (S[i]=' ') AND (S[i+1]=' ' THEN BEGIN
         DELETE (S,i+1,1);
         Ende := Ende-1
       END;
       ELSE i := i+1;
      END; (* von WHILE *)
     END;

    PROCEDURE Zaehle (S : Stg; VAR L : INTEGER);
     VAR i : INTEGER;
     BEGIN
      L:=1;
      FOR i:=1 TO LENGTH(S) DO IF S[i]=' ' THEN L:=L+1
     END;

    PROCEDURE Ausgabe (S : Stg; A,N : INTEGER);
     BEGIN
      WRITELN ('Der bereinigte Satz ist:');
      WRITELN (S);
      WRITELN ('Vorher enthielt er ',A:2,' Leerstellen');
      WRITELN ('nachher ',N:2,' Leerstellen.')
     END;

   BEGIN  (* Hauptprogramm *)
    Eingabe (Satz);
    Zaehle (Satz, Altleer);
    Leerraus (Satz);
    Zaehle (Satz, Neuleer);
    Ausgabe (Satz, Altleer, Neuleer)
   END.
```

Aufgabe: Welche Funktion hat das Programm Wenigerleer? Welche Prozeduren sind vom Typ "call-by-variable" und welche vom Typ "call-by-value"?





Hinweise zum komfortablen Umgang mit Prozeduren

FORWARD: Wird es aus einem programmtechnischen Grund heraus noetig, dass eine aufrufende Prozedur erst nach der Stelle des Aufrufs im Programmtext stehen soll, so kann das Problem mit der FORWARD-Deklaration geloest werden. Der Prozedurkopf wird dann mit dem reservierten Wort FORWARD versehen mit der Parameterliste zusammen vorgezogen. Erst spaeter folgt der Prozedurrumpf (mit dem einfachen Prozedurkopf).

Beispiel:

```
PROCEDURE Eingabe (VAR Wert1, Wert2 : INTEGER);
FORWARD; (*hier also mit Parameterliste*)

PROCEDURE Berechnen;
  BEGIN
   ...
   Eingabe (a,b);
   ...
  END;

PROCEDUR Eingabe; (*hier keine Parameterliste*)
  BEGIN
   ...
   ...
  END;
```

Exit: (siehe Kap.6.2) In Turbo Pascal mit Versionsnummern kleiner als 3.0 gibt es keine Standardprozeduren zum vorzeitigen Verlassen einer Prozedur. In diesem Fall muss sich der Benutzer mit der GOTO-Anweisung helfen. Dazu wird im Deklarationsteil der Prozedur ein Label definiert und an geeigneter Stelle zu diesem Label, das sinnvollerweise am Ende der Prozedur steht, gesprungen.
Achtung: Sprung ist nur innerhalb der Prozedur moeglich!

```
PROCEDURE Beispiel;
```

```
LABEL Exit;
VAR...;
BEGIN
 ...
 ...
 IF NOT Weiter THEN GOTO Exit;
 ...
Exit: END; (*Ende der Prozedur*)
```

Schachtelung von Prozeduren: Prozeduren duerfen geschachtelt werden. Das heisst, im Deklarationsteil der Prozedur duerfen weitere Prozeduren deklariert werden. Diese sind dann jedoch lokal zu der Prozedur, in der sie erklaert werden. Lokale Prozeduren duerfen also nur von der Prozedur aus aufgerufen werden, zu der sie lokal sind.

```
PROCEDURE Eins;
  VAR...;

  PROCEDURE Zwei;(* lokal zu Eins *)
    VAR...;
    BEGIN
     ...
    END;(* von Zwei *)

BEGIN (* von Eins *)
  ...
  Zwei;
  ...
END;(* von Eins *)
```

Immer dann, wenn aus einem ehemaligen Hauptprogramm eine Prozedur gemacht wird, werden sicher geschachtelte Prozeduren verwendet.

6.2 Standardprozeduren

Turbo Pascal stellt eine Reihe von Prozeduren bereit, die schon fertig sind und nur vom Benutzer aufgerufen zu werden brauchen: sogenannte Standard-prozeduren.
Die Namen dieser Prozeduren sind keine reservierten Woerter, sondern koennen vom Benutzer auch als Bezeichner verwendet werden. In dem Fall ist jedoch die entsprechende Prozedur nicht mehr zugaenglich, da ihr Name dann ander-weitig verwendet wird.

CLREOL;
Diese Prozedur loescht alle Zeichen von der Cursorposition an bis zum Zeilenende. Die Cursorposition wird nicht veraendert.
Beispiel: CLREOL;

CLRSCR;
Loescht den Bildschirm und setzt den Kursor in die linke obere Ecke.
Beispiel: CLRSCR;

CRTINIT;
Sendet den Terminal-Installations-String an den Bildschirm. Dieser ist durch die Installation des Turbo Pascal bestimmt.
Beispiel: CRTINIT;

CRTEXIT;
Sendet den Terminal-Reset-String an den Bildschirm. Dieser ist ebenfalls durch die Installation bestimmt.
Beispiel: CRTEXIT;

DELAY(Zeit);
Eine Verzoegerungsprozedur, die den Rechner ungefaehr soviel Millisekunden warten laesst, wie der Parameter Zeit vom Typ INTEGER angibt.
Beispiel: DELAY(5000); wartet ca. 5 Sekunden.

DELLINE;
Loescht die Zeile, in der der Kursor steht, und schiebt alle folgenden Zeilen nach.
Beispiel: DELLINE;

EXIT;
Ab Version 3.0 verfuegbare Prozedur ohne Parameter, die dafuer sorgt, dass der entsprechende Programmteil (Prozedur, Funktion, Hauptprogramm), in dem sich die EXIT-Anweisung befindet, vorzeitig abgebrochen wird.
Beispiel: EXIT;

INSLINE;
Fuegt an der Kursorposition eine Leerzeile ein und laesst alle folgenden Zeilen nach unten wandern.
Beispiel: INSLINE;

GOTOXY(x,y);
Positioniert den Cursor entsprechend den Bildschirmkoordinaten x,y (beide vom Typ INTEGER). x ist die Nummer der Spalte, y die Nummer der Zeile. Die Koordinate 1,1 ist die linke obere Ecke. Die dem GOTOXY-Befehl folgenden Ein- oder Ausgabe findet an der Cursorposition statt.
Beispiel: GOTOXY (10,15);
WRITELN ('Test');
Das Wort Test wird in der 15.Zeile und darin an der 10.Stelle geschrieben.

HALT;
Standardprozedur ohne Parameter, die dafuer sorgt, dass das aktuelle Programm abgebrochen wird.
Beispiel: HALT;

LOWVIDEO;
Schaltet den Bildschirm auf das Low-Video-Attribut, das in der Installation von Turbo Pascal vereinbart wurde.
Beispiel: LOWVIDEO;

NORMVIDEO;
Schaltet den Bildschirm auf das Normal-Video-Attribut, das in der Installation vereinbart wurde.
Beispiel: NORMVIDEO;

RANDOMIZE;
Sorgt dafuer, dass der Zufallsgenerator eine neue Zufallszahl erzeugt. Wird RANDOMIZE nicht verwendet, so gibt es bei jedem Programmdurchlauf gleiche Zufallszahlen.
Beispiel: RADOMIZE;

MOVE (Var1, Var2, Anzahl);

Bewegt eine ganzzahlige Anzahl von Bytes im Speicher von der Variablen Var1 zur Variablen Var2. Die Variablen koennen von beliebigem Typ sein.
Beispiel: VAR a,b : ARRAY [1..20] OF INTEGER;
```
...
MOVE(a,b,20);
```
bewegt die Haelfte der Arrays a zum Array b (denn der Datentyp INTEGER benoetigt 2 Bytes Speicherplatz).

FILLCHAR (Var,Anzahl,Wert);
Fuellt den Speicherbereich angefangen bei der ersten Speicherstelle, die von der Variablen Var eingenommen wird, mit einer Anzahl von Werten vom Typ BYTE oder CHAR. Var ist von beliebigem Typ. Anzahl vom Typ INTEGER.
Beispiel: VAR a : CHAR ABSOLUTE $3000;
```
...
FILLCHAR (a,1024*8,CHR(255));
```
fuellt den Speicherbereich ab hexadezimal $3000 bis $4FFF (d.h. 8 KByte = 8*1024) mit dem Zeichen CHR(255). Beim APPLE unter Turbo Pascal wuerde das bedeuten, dass der Grafikbildschirm geloescht wird.

Weitere Standardprozeduren sind die STRING-Prozeduren (siehe Kap. 3.4), die Ein-/Ausgabeprozeduren, die Dateiprozeduren (siehe Kap. 8.1) und die Zeigerprozeduren (siehe Kap. 8.2).

6.3 Funktionen

Eine andere Form von Unterprogrammen neben den Prozeduren sind in Pascal die Funktionen. Sie aehneln den Prozeduren sehr. Auch sie werden im Deklarationsteil aufgeschrieben und dann von anderen Programmteilen aus (die spaeter folgen) aufgerufen.
Der Hauptunterschied liegt aber darin, dass Funktionen nicht nur einfach mit Namen aufgerufen, sondern einer Variablen zugewiesen werden. Funktionen liefern naemlich stets ein Ergebnis, wenn sie abgearbeitet worden sind.
Die Form dieser Funktion ist:

```
FUNCTION <Name> (<Parameterliste>) : <Ergebnistyp> ;
 CONST...;
 VAR   ;
 BEGIN
 ...
 <Name> := <Ergebnis>;
 ...
 END;
```

Die Parameterliste hat die gleiche Form wie bei den Prozeduren. Auch hier koennen die Parameter wieder als Wert ("call-by-value") oder Variable ("call-by-variable") uebergeben werden. Ausserdem wird ein Ergebnis verlangt. Dies ist der Datentyp des Ergebnisses der Funktion.
Im Anweisungsteil der Funktion muss natuerlich irgendwann einmal ein Ergebnis erlangt und dem Namen der Funktion zugewiesen werden.

Beispiel: Eine Funktion, die die Potenz einer Dezimalzahl mit ganzzahligem Exponenten berechnet. Dabei werden auch negative Exponenten zugelassen.

```
FUNKTION Potenz (Basis: REAL; Exponent: INTEGER): REAL;
 VAR i : INTEGER;
```

```
      p : REAL;
      positiv : BOOLEAN;
  BEGIN
    p := 1;
    positiv := (Exponent = ABS (Exponent));
    Exponent := ABS (Exponent);
    FOR i := 1 TO Exponent DO p := p * Basis;
    IF positiv THEN Potenz := p
               ELSE Potenz := 1/p
    END; (* von Potenz *)
```

Die Variable positiv gibt an, ob der Exponent positiv ist.

Hinweise: Die Funktion Potenz wird aufgerufen mit zwei Werten, von denen der erste vom Typ REAL und der zweite vom Typ INTEGER sein muss. Das Ergebnis ist vom Typ REAL.

Man ist im Anweisungsteil moeglicherweise geneigt, ohne die Hilfsvariable p zu arbeiten und zu schreiben

```
FOR i:=1 TO Exponent DO Potenz:=Potenz*Basis;  (*falsch!*).
```

Dies ist falsch, da auf der rechten Seite des Zuweisungszeichens die Funktion mit Parametern aufgerufen werden muss.
(Wenn sich eine Funktion in ihrem Anweisungsteil selbst aufruft, so entsteht eine Rekursion; siehe Kap. 6.5)
Die Funktion laesst sich nun folgendermassen aufrufen:

```
X:=Potenz(5.3,4);   berechnet 5.3 hoch 4
X:=Potenz(A,B);     berechnet A (REAL) hoch B (INTEGER)
X:=Potenz(A,2);     berechnet A (REAL) hoch 2
```

Eigentlich sind uns Funktionen gar nicht so unbekannt. Denken wir z.B. an die trigonometrischen Funktionen. Hier heisst es im Funktionsaufruf z.B.

```
X:=SIN(3.4);
X:=SIN(A);
```

Niemals jedoch darf der Name der Funktion allein genannt werden. Es wird stets ein Ergebnis ermittelt und ausgegeben oder zugewiesen.
Anders bei den Prozeduren. Diese "machen" etwas, ermitteln jedoch kein Ergebnis (ausser in den Uebergabevariablen). Zum Beispiel

```
CLRSCR;
```

ist eine solche vordefinierte Prozedur. Sie loescht den Bildschirm.
Wir wollen nun ein paar Funktionen erstellen, um die Struktur dieser Konstruktion besser kennenzulernen und zu ueben.

Das Minimum zweier ganzer Zahlen

Die Funktion Min ermittelt das Minimum zweier Zahlen vom Datentyp INTEGER. Das Ergebnis ist natuerlich ebenfalls vom Typ INTEGER.

```
FUNCTION Min (a,b : INTEGER) : INTEGER;
  BEGIN
    IF a < b THEN Min := a
             ELSE Min := b
    END; (* von Min *)
```

Groesster gemeinsamer Teiler ggT

Die Funktion ggT soll den groessten gemeinsamen Teiler zweier Zahlen vom Datentyp INTEGER ermitteln. Das Ergebnis ist selbstverstaendlich ebenfalls

vom Datentyp INTEGER.
Dazu zaehlen wir eine Variable namens Teiler von 1 bis zum Minimum der beiden Zahlen,denn der gemeinsame Teiler kann hoechstens so gross sein wie die kleinere Zahl. Immer dann, wenn sich beide Zahlen durch Teiler teilen lassen, ist ggT gleich diesem Teiler
Fuer das Minimum benutzen wir gleich die schon geschriebene Funktion.

```
FUNCTION ggT (a,b : INTEGER) : INTEGER;
  VAR Teiler : INTEGER;

  FUNCTION Min (a,b : INTEGER) : INTEGER;
    BEGIN
      IF a < b THEN Min :- a
               ELSE Min := b
    END; (* von Min *)

  BEGIN (* von ggT *)
    FOR Teiler := 1 TO Min(a,b) DO
      IF a MOD Teiler = 0 THEN IF b MOD Teiler = 0 THEN ggT := Teiler
  END; (* von ggT *)
```

Hinweis: Wir haben es hier mit geschachtelten Funktionen zu tun. Die Funktion MIN ist lokal zur Funktion ggT.

Pruefen einer Zahl als Primzahl

Die Funktion Prim ermittelt, ob eine Zahl vom Typ INTEGER eine Primzahl ist oder nicht. Das Ergebnis ist vom Datentyp BOOLEAN, also wahr oder falsch.
Eine Zahl kleiner als 2 ist keine Primzahl. In einer Schleife wird geprueft, ob sich die Zahl durch einen Teiler zwischen 2 und der Quadratwurzel der Zahl teilen laesst.

```
FUNCTION Prim (Zahl : INTEGER) :BOOLEAN;
  VAR Teiler : INTEGER;

  BEGIN
   Prim := TRUE;
   FOR Teiler:=2 TO ROUND(SORT(Zahl)) DO
    IF (Zahl MOD Teiler =0) AND (Zahl>3) THEN Prim:=FALSE
   END; (* von Prim *)
```

Weitermachen (J/N)?

Die Funktion Weiter fragt den Benutzer, ob er im Programm weiter fortfahren moechte oder nicht. Das Ergebnis ist ebenfalls wieder vom Datentyp BOOLEAN. Die Funktion eignet sich fuer Programme, in denen oft nach einer Fortfuehrung des Programms gefragt wird.

In Schleifen koennte dann die Funktion zum Einsatz kommen mit:

```
            WHILE Weiter DO ...;
oder mit:
            REPEAT
             ...
            UNTIL NOT Weiter;
```

```
   FUNCTION Weiter : BOOLEAN;
     VAR ch : CHAR;
     BEGIN
       WRITE ('Wollen Sie weitermachen (J/N)? ');
       READ (KBD,ch);
       Weiter := (ch='J') OR (ch='j')
     END; (* von Weiter *)
```

Lieszeichen

Die folgende Funktion Lieszeichen liest ein Zeichen aus der in der Parameterliste angegebenen Menge von der Tastatur ein. Wird eine Taste mit einem Zeichen gedrueckt, das nicht in der Menge ist, so ertoent ein Ton. Andernfalls wird das Zeichen angenommen.
Diese Funktion hat gegenueber der Eingabe mit READ erhebliche Vorteile, da der moegliche Eingabebereich eingeschraenkt wird.
Im Hauptprogramm muss deklariert sein:

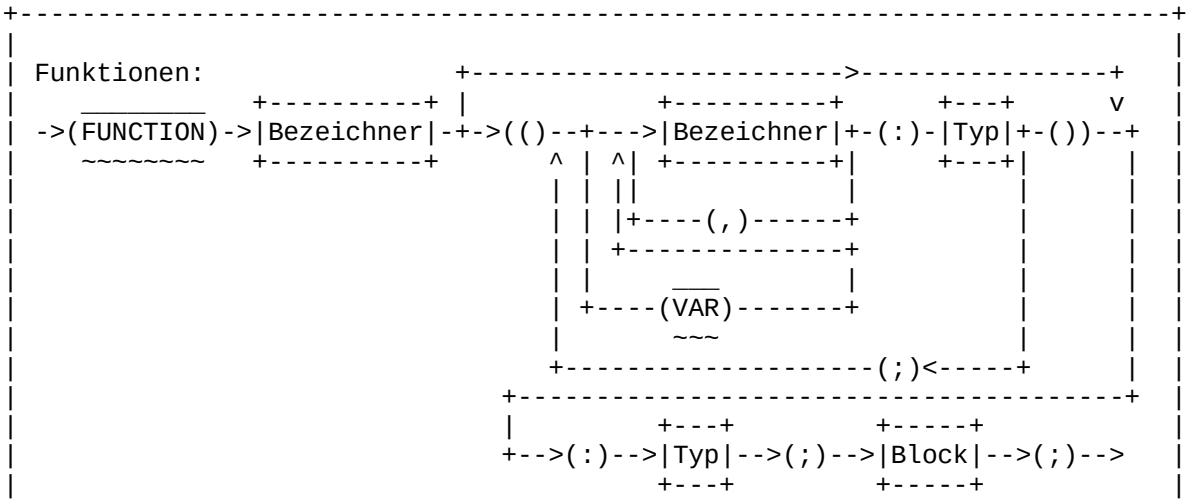
```
   TYPE Setofchar = SET OF CHAR;

   FUNCTION Lieszeichen (m : Setofchar) : CHAR;
    VAR ch : CHAR;
        OK : BOOLEAN;
    BEGIN
     REPEAT
      READ (KBD, ch);                  (* Lies Zeichen ohne Echo *)
      IF EOLN (KBD) THEN ch:=CHR(13);        (* <Return>-Taste *)
      OK :=ch IN m;
      IF NOT OK THEN WRITE (CHR(7))                      (* Bell *)
                ELSE IF ch IN [' '..CHR(126)]   (* druckbare Zeichen *)
                        THEN WRITE (ch)
     UNTIL OK;
     Lieszeichen := ch
    END; (* von Lieszeichen *)
```

Im folgenden Programm wollen wir die oben aufgefuehrten Funktionen auf ihre Richtigkeit hin testen und anwenden:

```
PROGRAM Funtest;
TYPE Setofchar = SET OF CHAR;
VAR ch:CHAR;
     r:REAL;
   a,b:INTEGER;

FUNCTION Potenz (Basis : REAL; Exponent : INTEGER) : REAL;
   VAR i : INTEGER;
       p : REAL;
       positiv : BOOLEAN;

   BEGIN
     p := 1;
     positiv := (Exponent = ABS (Exponent));
     Exponent := ABS (Exponent);
     FOR i := 1 TO Exponent DO p := p * Basis;
     IF positiv THEN Potenz := p
```

```
                ELSE Potenz := 1 / p
   END; (* von Potenz *)

FUNCTION ggT (a,b : INTEGER) : INTEGER;
   VAR Teiler : INTEGER;

   FUNCTION Min (a,b : INTEGER) : INTEGER;
     BEGIN
     IF a < b THEN Min := a
              ELSE Min := b
     END; (* von Min *)

   BEGIN
     FOR Teiler := 1 TO Min(a,b) DO
       IF a MOD Teiler = 0 THEN IF b MOD Teiler = 0 THEN ggT := Teiler
   END; (* von ggT *)

FUNCTION Prim (Zahl : INTEGER) : BOOLEAN;
   VAR Teiler : INTEGER;

   BEGIN
    Prim := TRUE;
    FOR Teiler:=2 TO  ROUND(SQRT(Zahl)) DO
     IF (Zahl MOD Teiler = 0) AND (Zahl>3) THEN Prim:=FALSE
    END; (* von Prim *)

FUNCTION Weiter : BOOLEAN;
   VAR ch : CHAR;
   BEGIN
     WRITE ('Wollen Sie weitermachen (Y/N) ?');
     READ (KBD,ch);
     Weiter := (ch='Y') OR (ch='y')
   END; (* von Weiter *)

FUNCTION Lieszeichen (m : Setofchar) : CHAR;
   VAR ch : CHAR;
       OK : BOOLEAN;
   BEGIN
    REPEAT
     READ (KBD, ch);      (* Lies Zeichen ohne Echo *)
     IF EOLN (KBD) THEN ch:=CHR(13);    (* <CR>-Taste *)
     OK := ch IN m;
     IF NOT OK THEN WRITE (CHR(7))      (* Bell *)
               ELSE IF ch IN [' '..CHR(126)]   (* druckbare Zeichen *)
                       THEN WRITE (ch)
    UNTIL OK;
    Lieszeichen := ch
   END; (* von Lieszeichen *)

   BEGIN (* Hauptprogramm *)
    REPEAT
     CLRSCR;
     WRITELN ('Waehlen Sie:');
     WRITELN;
     WRITELN (' G(gt ');
     WRITELN (' P(rimzahlen');
     WRITELN (' R(echnen mit Potenzen');
     ch:=Lieszeichen (['g','G','p','P','r','R']);
```

```
    WRITELN; WRITELN;
    CASE ch OF
    'g','G' : BEGIN
               WRITE ('1. Zahl: '); READLN (a);
               WRITE ('2. Zahl: '); READLN (b);
               WRITELN ('ggt: ',ggt(a,b))
              END;
    'p','P' : BEGIN  WRITE ('Zahl: '); READLN (a);
                     WRITE (a,' Ist ');
                     IF NOT Prim(a) THEN WRITE ('keine ');
                     WRITELN ('Primzahl.')
              END;
    'r','R' : BEGIN  WRITE ('Basis: '); READLN (r);
                     WRITE ('Exponent: '); READLN (b);
                     WRITELN (r:7:2,' hoch ',b,' ist ',Potenz(r,b):11:5)
              END
   END
   UNTIL NOT Weiter
  END.
```




```
  FUNCTION name (Parameterliste) : Ergebnistyp;
    CONST...;
    VAR...;
    BEGIN
      ...
      name := ...;
      ...
    END;
```

Eine Funktion hat im Gegensatz zu einer Prozedur stets ein Ergebnis. Funktionen koennen mit oder ohne Parameter aufgerufen werden. Die Parameter koennen mit "call-by-value" und "call-by-variable" uebergeben werden. Im Anweisungsteil der Funktion wird dem Funktionsnamen (links vom Zuweisungszeichen) ein Ergebnis vom Ergebnistyp zugewiesen. Beim Funktionsaufruf (rechts vom Zuweisungszeichen) muessen die Parameter der Funktion uebergeben werden.
Die Parameter und das Ergebnis muessen von einfachen Datentypen sein.
Funktionen koennen wie Prozeduren geschachtelt werden. Bei der Reihenfolge gelten dieselben Regeln wie bei Prozeduren: Eine aufgerufene Funktion/Prozedur muss im Text vor der aufrufenden Funktion/Prozedur

| stehen. |
+---+

Hinweise: Die Hinweise zur FORWARD-Deklaration und zum vorzeitigen Abbruch einer Funktion gelten analog zu den Prozeduren (siehe Kap.6.1).

6.4 Standardfunktionen

Turbo Pascal stellt eine Reihe von Funktionen bereit, die schon fertig sind und nur vom Benutzer aufgerufen zu werden brauchen: sogenannte Standardfunktionen.
Die Namen dieser Funktionen sind keine reservierten Woerter, sondern koennen auch vom Benutzer als Bezeichner verwendet werden. In diesem Falle ist jedoch die entsprechende Funktion nicht mehr zugaenglich, da ihr Name dann anderweitig verwendet wird.

Arithmetische Funktionen

ABS (Zahl);
Absolutwert einer Zahl. Das Argument ist entweder REAL oder INTEGER. Das Ergebnis ist vom Typ des Arguments.
Beispiel: x:=ABS(-3.7); dann hat x den Wert 3.7.

ARCTAN (Zahl);
Arcustangens einer Zahl. Der Winkel wird in Bogenmass angegeben. Zahl ist vom Typ REAL oder INTEGER. Das Ergebnis ist vom Typ REAL.
Beispiel: x:=ARCTAN(1); dann hat x den Wert Pi/4.

COS (Zahl);
Cosinus einer Zahl. Der Winkel wird in Bogenmass angegeben. Zahl ist vom Typ REAL oder INTEGER. Das Ergebnis ist vom Typ REAL.
Beispiel: x:=COS(PI/2); dann hat x den Wert 0.

EXP (Zahl);
Exponentialfunktion zur Basis e, d.h.e^Zahl. Zahl ist vom Typ REAL oder INTEGER. Das Ergebnis ist vom Typ REAL.
Beispiel: x:=EXP(1); dann hat x den Wert 2.718281828.

FRAC (Zahl);
Ergibt den gebrochenen Teil einer Zahl (d.h. Nachkommawert). Zahl kann vom Typ REAL oder INTEGER sein. Das Ergebnis ist vom Typ REAL.
Beispiel: x:=FRAC(3.7); dann hat x den Wert 0.7.

INT (Zahl);
Ergibt den ganzzahligen Anteil einer Zahl. Zahl kann vom Typ REAL oder INTEGER sein.Das Ergebnis ist vom Tyo REAL.
Beispiel: x:=INT(3.7); dann hat x den Wert 3.0.

LN (Zahl);
Natuerlicher Logarithmus einer Zahl (d.h. zur Basis e). Zahl ist vom Typ REAL oder INTEGER. Das Ergebnis ist vom Typ REAL.
Beispiel: x:=LN(2); dann hat x den Wert 6.931471806E-01.

SIN (Zahl);
Sinus einer Zahl. Der Winkel wird in Bogenmass angegeben. Zahl ist vom Typ REAL oder INTEGER. Das Ergebnis ist vom Typ REAL.
Beispiel: x:=SIN(PI/2);dann hat x den Wert 1.

SQR (Zahl);
Quadrat einer Zahl. Zahl ist vom Typ REAL oder INTEGER. Das Ergebnis ist vom Typ des Arguments.
Beispiel: x:=SQR(2.5);dann hat x den Wert 6.25.

SQRT (Zahl);
Quadratwurzel einer Zahl. Zahl ist vom Typ REAL oder INTEGER. Das Ergebnis ist vom Typ REAL.
Beispiel: x:=SQRT(9);dann hat x den Wert 3.0.

Skalare Funktionen

PRED (Argument);
Vorgaenger des Arguments. Argument und Ergebnis sind vom gleichen skalaren, aufzaehlbaren Typ.
Beispiel: x:=PRED('B');dann hat x den Wert A.

SUCC (Argument);
Nachfolger des Arguments. Argument und Ergebnis sind vom gleichen skalaren, aufzaehlbaren Typ.
Beispiel: x:=SUCC(15);dann hat x den Wert 16.

ODD (Zahl);
Ist eine Funktion mit Ergebnistyp BOOLEAN. Erhaelt den Wert TRUE, wenn Zahl eine ungerade Zahl ist, sonst FALSE. Zahl muss vom Typ INTEGER sein.
Beispiel: x:=ODD(16);dann hat x den Wert FALSE.

Uebergang zwischen verschiedenen Datentypen

CHR (Zahl);
Ergibt den zu einer Zahl gehoerenden ASCII-Wert. Zahl ist vom Typ INTEGER. Das Ergebnis ist vom Typ CHAR.
Beispiel: x:=CHR(66);dann hat x den Wert B.

ORD (Wert);
Ergibt die Ordnungsnummer eines Wertes aus einer Aufzaehlung. Dabei wird bei 0 angefangen zu zaehlen. Das Ergebnis ist vom Typ INTEGER. Siehe auch KAP. 3.7.
Beispiel: x:=ORD('A');dann hat x den Wert 65.

ROUND (Zahl);
Rundet eine Dezimalzahl zu einer ganzen Zahl. Fuer Zahl>0 ergibt sich der ganzzahlige Anteil von Zahl+0.5. Fuer Zahl<0 ergibt sich der ganzzahlige Anteil von Zahl-0.5.
Beispiel: x:=ROUND(3.7);dannhat x den Wert 4.

TRUNC (Zahl);
Ergibt den ganzzahligen Anteil einer Dezimalzahl. Das Ergebnis ist vom Typ INTEGER.
Beispiel: x:=TRUNC(3.7);dann hat x den Wert 3.

Sonstige Funktionen

HI (Argument);

Das Ergebnis vom Typ INTEGER erhaelt als niederwertiges Byte das hoeherwertige Byte des Arguments. Das hoeherwertige Byte wird auf Null gesetzt.
Beispiel: x:=HI(8446);dann hat x den Wert 32
oder x:=HI($20FE);hat das glieche Ergebnis ($20=32).

KEYPRESSED;
Diese Funktion hat kein Argument und ein Ergebnis vom Typ BOOLEAN. Das Ergebnis ist TRUE,wenn eine Taste gedrueckt wurde, sonst FALSE.
Beispiel: REPEAT...UNTIL KEYPRESSED.

LO (Argument);
Das Ergebnis vom Typ INTEGER erhaelt als niederwertiges Byte das niederwertige Byte des Arguments. Das hoeherwertige Byte wird auf Null gesetzt.
Beispiel: x:=LO(8446);dann hat x den Wert 254
oder x:=LO($20FE);hat das gleiche Ergebnis ($FE=254).

MEMAVAIL;
Das Ergebnis vom Typ INTEGER gibt die Groesse des freien Speicherplatzes an. Bei 8-Bit-Systemen in Byte; bei 16-Bit-Systemen in Paragraphen pro 16 Byte. Ist das Ergebnis negativ, so ist 65536 dazuzuaddieren.
Beispiel: x:=MEMAVAIL;

PARAMCOUNT;
AB Version 3.0 verfuegbare Funktion ohne Parameter mit Ergebnistyp INTEGER, die die Anzahl der uebergebenden Parameter angibt. Siehe auch PARAMSTR.
Beispiel: x:=PARAMCOUNT;

PARAMSTR (N);
AB Version 3.0 verfuegbare Funktion mit einem INTEGER-Parameter und Ergebnistyp STRING, die den N.Uebergabeparameter als Ergebnis hat. Das bedeutet, dass ein als .COM-File (bzw..CMD-File) uebersetztes Programm mit Parametern vom Betriebssystem (z.B. CP/M) aufgerufen werden kann,so dass die Parameter als STRINGs vom Programm verarbeitet werden koennen.
Beispiel: x:=PARAMSTR(n);

RANDOM;
Das Ergebnis dieser Funktion ist eine zufaellige Zahl vom Typ REAL groesser oder gleich Null und kleiner als Eins.
Beispiel: x:=RANDOM;dann hat einen zufaelligen Wert.

RANDOM (Zahl);
Es gibt eine zufaellige Zahl groesser oder gleich Null und kleiner als Zahl. Zahl und Ergebnis sind vom Typ INTEGER.
Beispiel: x:=RANDOM(200);dann hat x den ganzzahligen Wert zwischen 0 und 199.

SIZEOF (Argument);
Ergibt den Speicherplatz in Byte, den eine Variable oder ein Typ namens Argument im Speicher einnimmt. Das Ergebnis ist vom Typ INTEGER.
Beispiel: x:=SIZEOF(x);dann hat x den Wert 2, da es vom Typ INTEGER sein muss.

SWAP (Zahl);
Tauscht hoeherwertiges und niederwertiges Byte der Zahl vom Typ INTEGER um. Ergebnis vom Typ INTEGER.
Beispiel: x:=SWAP(8446);dann hat x den Wert -480

oder x:=SWAP($20FE);gleiches Ergebnis ($20FE=-480).

UPCASE (Zeichen);
Hat als Ergebnis den Grossbuchstaben des entsprechenden Zeichens, sofern dieser existiert, andernfalls ist das Argument das Ergebnis. Argument und Ergebnis sind vom Typ CHAR.
Beispiel: x:=UPCASE('a');dann hat x den Wert A.

Weitere Standardfunktionen siehe Kap. 3.4.8.1. und 8.2.

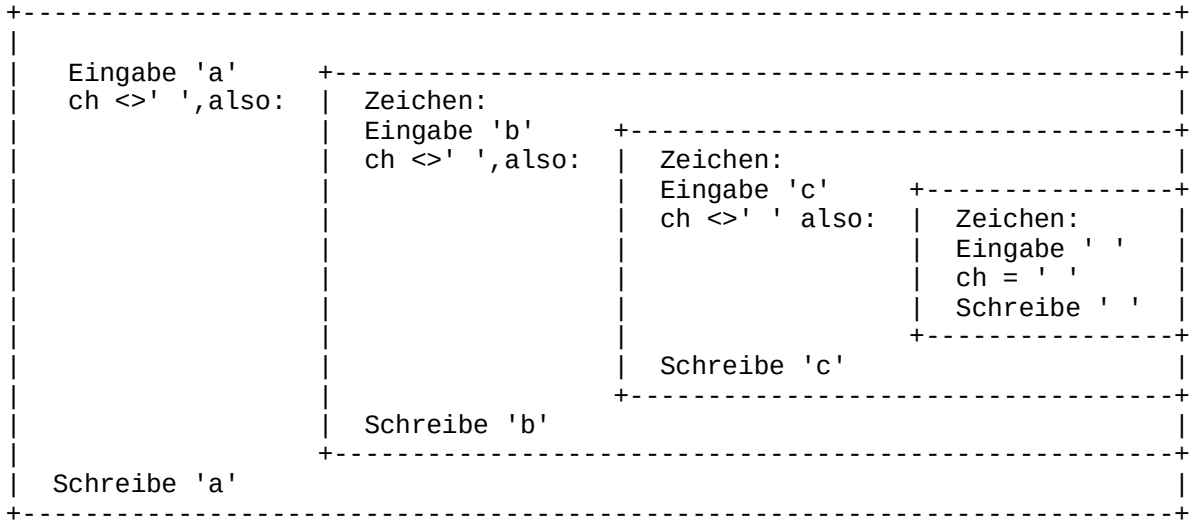
6.5 Rekursionen

Ein entscheidender Vorteil einer Programmiersprache, die lokale und globale Variablen kennt, ist die einfache Moeglichkeit, Rekursionen zu programmieren. Von einer Rekursion sprechen wir, wenn eine Prozedur oder Funktion sich selbst aufruft. Dies ist in anderen Programmiersprachen auch moeglich, jedoch koennen in Pascal lokale Variablen verwendet werden, die bei jedem Neuaufruf der entsprechenden Prozedur oder Funktion neuen Speicherstellen zugeordnet werden.
In Turbo Pascal ist allerdings zu beachten, dass eine Prozedur oder Funktion, die einen rekursiven Aufruf enthaelt, durch zwei Compiler-Anweisungen eingerahmt wird, die dafuer sorgen, dass rekursiver Code erzeugt werden kann. In der Regel erzeugt der Turbo Compiler naemlich nichtrekursiven Code. Mit (*$A-*) wird der Compiler angewiesen, dass er rekursiven Code erzeugen soll, und mit (*$S+*) wird diese Moeglichkeit wieder abgeschaltet.

Warum laesst man dann nicht das ganze Programm mit (*$A-*) uebersetzen ? Rekusiver Code braucht mehr Speicherplatz und ist in der Ausfuehrung etwas langsamer.
Hier nun ein einfuehrendes Beispiel fuer eine Rekursion:

```
PROGRAM Was_macht_das;  (* Idee: Pascal-Kurs der FU-Hagen *)

  (*$A-*)
  PROCEDURE Zeichen;
    VAR ch : CHAR;

    BEGIN
      READ (KBD,ch);WRITE (ch);
      IF ch <> ' ' THEN Zeichen;
      WRITE (ch)
    END; (* von Zeichen *)
  (*$A+*)

  BEGIN (* Hauptprogramm *)
    Zeichen
  END.
```

Was macht das Programm ?
Zur Uebung sollte jetzt das Buch beiseite gelegt werden.
Dass es sich hier um eine Rekursion handelt, erkenne wir an dem Aufruf der Prozedur Zeichen innerhalb der Prozedur Zeichen. Die Prozedur ruft sich also selbst auf.
Nehmen wir an, wir geben die Buchstaben "abc" und danach eine Leerstelle ein.
Das Hauptprogramm ruft die Prozedur Zeichen auf:



Die Eingabe eines Zeichens wird also solange fortgesetzt, bis das Leerzeichen eingegeben wird. Dann werden alle Zeichen rueckwaerts wieder ausgegeben.
In dieser Art ist die Rekusion natuerlich nur moeglich, wenn bei jedem Aufruf der Prozedur Zeichen eine neue Variable ch bereitgestellt wird. Dies ist dadurch gewaehrleistet, dass ch eine lokale Variable ist und somit nur innerhalb der Prozedur definiert ist. Obwohl die Variablen gleiche Namen haben, sind sie doch unterschiedlich.

Merke: Wenn eine Prozedur oder Funktion sich selbst aufruft (rekursiv aufruft), dann werden nach der Ausfuehrung der Prozedur/Funktion die restlichen Anweisungen noch abgearbeitet.

Was passiert, wenn wir die Zeile

```
IF ch <>' ' THEN Zeichen;
```

austauschen gegen

```
Zeichen;
```

d.h. den Prozeduraufruf ohne Bedingung ausfuehren ?

Nun die Prozedur ruft sich dann "unendlich" lange selbst auf. Tatsaechlich hoert das Programm allerdings irgendwann ziemlich unschoen auf, weil der Speicherbereich ueberschritten wurde.

Fehlerquelle: Es ist stets darauf zu achten, dass eine Rekursion eine Abbruchbedingung hat. Wenn also eine Prozedur/Funktion sich selbst aufruft, so sollte der Aufruf von einer Bedingung abhaengen, deren Wahrheitsgehalt sich irgendwann zu FALSE aendert.

Wir wollen uns nun noch ein kleines Beispiel fuer eine rekursive Funktion ansehen. Aus der Mathematik ist sicher die sogenannte Fakultaet bekannt.
Es ist z.B. 5!=1*2*3*4*5.
Oft wird das Bildungsgesetz fuer eine beliebige Fakultaet jedoch so dargestellt:

```
1! = 1 und n! = (n-1)! * n
```

Dies ist eine rekursive Beschreibung der Fakultaet, denn n! wird berechnet durch das Produkt aus n und der Fakultaet das Vorgaengers von n. Wir tun so als wenn wir (n-1)! berechnen koennten. Koennen wir auch, denn
(n-1)! = (n-2)! * (n-1) und so wieter.

Jedoch waere dies ein unendlicher Prozess, gaebe es nicht die Abbruchbedingung (oft auch Rekursionsanfang genannt), die lautet 1!=1.

Als Pascal-Funktion liest sich das so:

```
(*$SA-*)
FUNCTION reku_Fak (n : INTEGER) :INTEGER;
  BEGIN
    IF n=1 THEN  reku_Fak := 1
           ELSE reku_Fak := reku_Fak(n-1) * n
  END; (* von reku_Fak *)
(*$A+*)
```

So einfach ist das. Wir haben nur die Rekursionsvorschrift von n! in Pascal-Anweisungen umgeschrieben.

Hinweis: Da Fakultaeten schnell sehr gross werden, ist MAXINT rasch ueberschritten. Dies ist bei der Funktion nicht beruecksichtigt, um nicht vom Thema abzulenken.

Haetten wir ohne die Moeglichkeit der Rekursion die Funktion Fak nicht schreiben koennen ? Doch, sicherlich.
Der Vorteil der Rekursion liegt allerdings in der Einfachheit und Uebersichtlichkeit. Dass Rekursionen auch erhebliche Nachteile haben koennen, zeigt der naechste Paragraph.
Jede Rekursion laesst sich jedoch durch eine Iteration (Nacheinanderausfuehrung von Anweisungen) ersetzen. Am Beispiel der Fakultaet saehe das so aus:

```
FUNCTION iter_Fak (n : INTEGER) : INTEGER;
  Var i, Hilf : INTEGER;
  BEGIN
    Hilf := 1;
    IF n=1 THEN Hilf := 1
           ELSE FOR I:=1 to n DO Hilf := Hilf * i;
    iter_Fak := Hilf
   END; (* von iter_Fak *)
```

Beachten Sie: Ohne die Hilfsvariable Hilf kaemen wir nicht aus, da die Zuweisung iter_Fak := iter_Fak * i nicht erlaubt waere. Warum ?

Ein kleines Programm zum Testen der neuen Funktion:

```
PROGRAM Fakutest;
 VAR a:INTEGER;

(*$A-*)
FUNCTION reku_Fak (n : INTEGER) : INTEGER;
  BEGIN
    IF n=1 THEN reku_Fak := 1
           ELSE reku_Fak := reku_Fak(n-1) *n
  END; (* von reku_Fak *)
(*$A+*)

FUNCTION iter_Fak (n : INTEGER) : INTEGER+
  Var i, Hilf : INTEGER;
  BEGIN
    Hilf := 1;
```

```
      FOR i:=1 to n DO Hilf := Hilf * i;
      iter_Fak := Hilf
    END; (* von iter_Fak *)

  BEGIN
   WRITELN ('Zahl: ');
   READLN (a);
   WRITELN ('Fakultaet rekursiv: ', reku_Fak(a));
   WRITELN ('Fakultaet iterativ: ', iter_Fak(a))
  END.
```

```
+-------------------------------------------------------------------------+
| Rekursion:                                                              |
|                                                                         |
| Von einer Rekursion sprechen wir, wenn sich eine Prozedur oder Funktion |
| selbst aufruft. Dabei sind Konstanten und Variablen beim erneuten Auf-  |
| ruf der Prozedur/Funktion nicht mit den gleichnamigen der aufrufenden   |
| Prozedur/Funktion identisch.                                            |
| Eine Rekursion muss stets eine Abbruchbedingung haben, die auch erreicht |
| wird. Anderenfalls kaeme es zu einer unendlichen Rekursion.             |
+-------------------------------------------------------------------------+
```

Hinweis: Wenn eine Prozedur/Funktion sich explizit selbst aufruft, sprechen wir von einer direkten Rekursion.Eine indirekte Rekursion dagegen liegt vor, wenn eine Prozedur/Funktion eine andere aufruft, die dann ihrerseits einen Aufruf der ersten Prozedur/Funktion enthaelt.

Hinweis fuer CP/M 80: In rekursiven Programmteilen darf eine zu einem Unterprogramm lokale Variable nicht als Variablenparameter (VAR) in rekursive Aufrufe uebergeben werden.

Probleme der Rechnerzeit und des Speicherplatzes mit Rekursionen

```
fib(0) = 0              fib(6) = 8
fib(1) = 1              fib(7) =13
fib(2) = 1              fib(8) =21
fib(3) = 2              fib(9) =34
fib(4) = 3              ...
fib(5) = 5
```

Die Zahlen dieser merkwuerdigen Zahlenreihe nennt man Fibonacci-Zahlen. Sie werden gebildet, indem jeweils die letzte und vorletzte Fibonacci-Zahl zusammenaddiert werden.
Als mathematisches Bildungsgesetz sieht das folgendermassen aus:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)
```

Na, also. Wieder eine Rekursion, wie wir hellen Koepfe sofort erkannt haben. Da wir ja nun schon einige Uebung im Programmieren haben, koennen wir sicher sofort die dazugehoerige Pascal-Funktion schreiben:

```
(*SA-*)
FUNCTION fib (n : INTEGER) : INTEGER;
  BEGIN
    IF n<2 THEN fib := n
```

```
            ELSE fib := fib(n-1) + fib(n-2)
    END;  (* von fib *)
  (*SA+*)
```

Testen wir unsere Funktion in einem Programm:

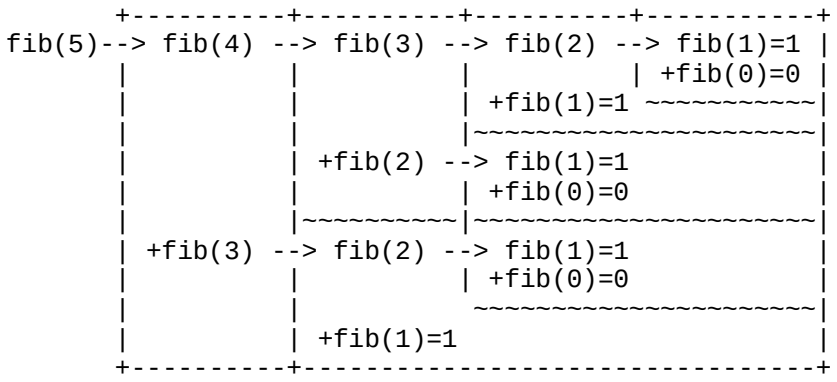
```
  PROGRAM Rekufib;
    VAR a : INTEGER;

    (*SA-*)
    FUNCTION fib (n : INTEGER) : INTEGER;
      BEGIN
        IF n<2 THEN fib := n
               ELSE fib := fib(n-1) + fib(n-2)
      END; (* von fib *)
    (*SA+*)

    BEGIN (* Hauptprogramm *)
      WRITE ('Eingabe einer zahl: ');
      READLN (a);
      WRITELN ('fib(', a, ')=', fib(a))
    END.
```

Alle Zahlen unserer oben aufgefuehrten Fibonacci-Reihe berechnet das Programm auch einwandfrei (testen sie das!).
Geben wir nun einmal eine etwas groessere Zahl ein:22.
Es dauert beim APPLE etwa 25 Sekunden, bis wir das Ergebnis fib(22)=17711 erhalten.
Zum Vergleich wollen wir die Funktion fib iterativ schreiben und an Stelle der rekursiven Funktion in unser Programm aufnehmen:

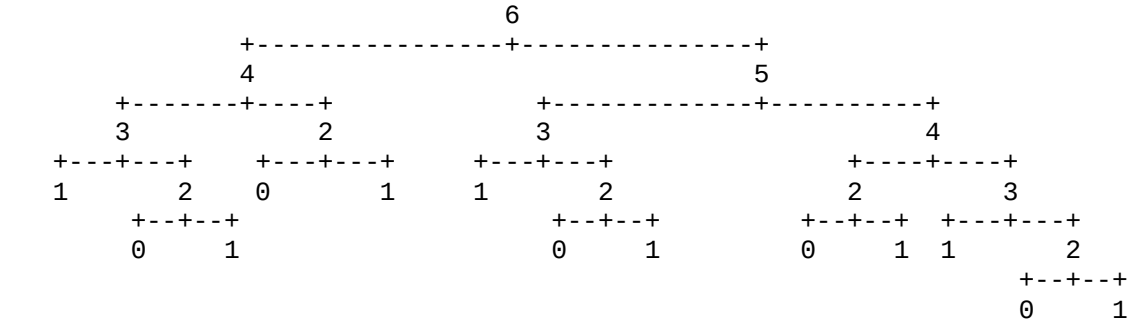
```
  PROGRAM Iterfib;
    VAR a : INTEGER;

    FUNCTION fib (n : INTEGER) : INTEGER;
      VAR i, Hilf, Letzt, Vorletzt : INTEGER;
      BEGIN
        Letzt := 1;
        Vorletzt := 0;
        IF n<2 THEN Hilf := n
        ELSE FOR i:=2 TO n DO BEGIN
               Hilf := Letzt + Vorletzt;
               Vorletzt := Letzt;
               Letzt := Hilf
             END; (* von FOR *)
        fib := Hilf
      END; (* von fib *)

    BEGIN (* Hauptprogramm *)
      WRITE ('Eingabe einer Zahl: ');
      READL (a);
      WRITELN ('fib(', a, ')=', fib(a))
    END.
```

Bei dieser Version des Programms stoppen wir eine Zeit unter einer Sekunde fuer fib(22).
Woher dieser gewaltige Unterschied?

Machen wir uns einmal deutlich, wie fib(5) rekursiv berechnet wird:



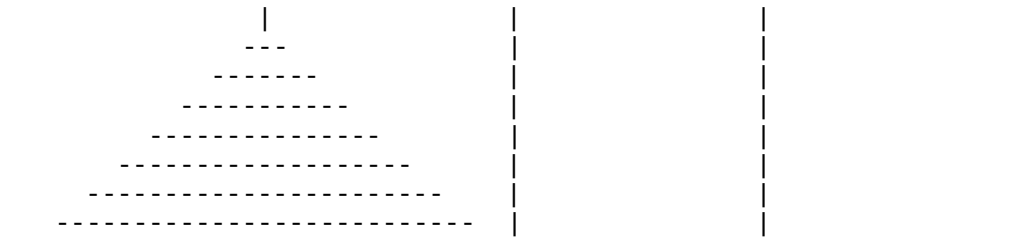
Addieren wir alle Endwerte (1) bei den Rekursionsabbruechen fib(1) und fib(0), so erhalten wir als Ergebnis 5. Wir erkennen, dass fuer n=5 die Funktion 15mal aufgerufen wird.
Eine andere Darstellung der Aufrufe der rekursiven Funktion ist ein Baum (hier fuer fib(6)=8):



Hier wird die Funktion schon 25mal aufgerufen.
Wir sehen also, dass die Zahl der Funktionsaufrufe explosionsartig ansteigt, da sich die Funktion zweimal selbst aufruft. Daher die enorme Rechenzeit bei der Rekursion. Fuer die Iteration wird nur einmal die Funktion aufgerufen und eine Schleife von 2 bis n abgearbeitet.
Ein anderes Problem bei Rekursionen spielt ebenfalls eine grosse Rolle: der Stapelspeicher (Stack).
Wenn eine Funktion/Prozedur aufgerufen wird, muss der Rechner sich die Ruecksprungadresse merken, denn nach dem Abarbeiten der Funktion/Prozedur soll es ja an der gleichen Stelle weitergehen.

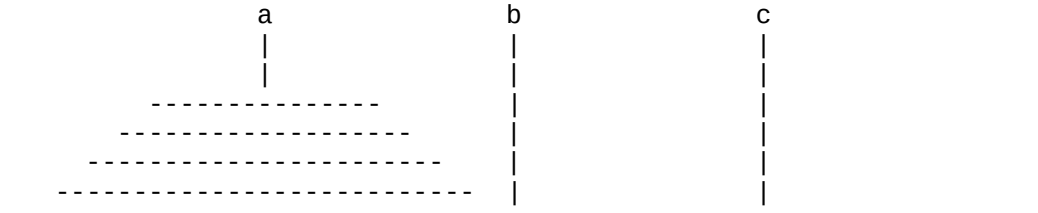
Ausserdem muessen jeweils die lokalen Variablen der Funktion/Prozedur abgelegt weden.
All dies geschieht im Stack (oder Stapelspeicher oder Kellerspeicher). Es handelt sich hier um einen LIFO-Stack (Last-In-First-Out). Diesen Speicherbereich koennen wir uns vorstellen wie einen Parkgroschenspender:

Der letzte Groschen, der in den Spender eingelegt wird, wird auch als erster wieder heraus genommen. Fuer Funktionen/Prozeduren heisst das, die zuletzt aufgerufene Funktion/Prozedur kehrt als erste wieder zu ihrer Ausgangsposition zurueck.
Bei Rekursion kann es nun (Abhaengig von der Groesse des Stacks unseres Rechners) leicht zu einer Ueberlastung des Stacks kommen, wenn die Funktion/Prozedur sich zu haeufig aufruft (zu grosse Rekursionstiefe).
In unserem vorigen Beispiel der Fakultaet spielten diese Probleme kaum eine Rolle, da die Funktion sich selbst nur einmal aufruft, d.h. fuer n! wird die Funktion n-mal abgearbeitet.
Hier noch ein Beispiel fuer die Anwendung einer Rekursion, die sich nicht so einfach durch einen iterativen Algorithmus ersetzen laesst:



Einigen ist sicherlich das Spiel " Tuerme von Hanoi " bekannt. Eine Holzausfuerung dieses Spieles ist auf dem Bild dargestellt.
Ziel das Spiels: Bringe alle Scheiben von einem Turm auf einen anderen.
Spielregeln: Bewege jeweils nur eine Scheibe von einem Turm zu einem anderen. Lege stets die kleinere auf die groessere Scheibe, nie umgekehrt.
Das Spiel laesst sich im Prinzip mit beliebig vielen Scheiben spielen. Auf der Abbildung sind es sieben.
An dieser Stelle sollten wir den Computer einmal abschalten und das Buch beiseite legen, um ein wenig zu spielen. Mit einigen unterschiedlich grossen Muenzen laesst sich " Tuerme von Hanoi " auch spielen, wenn keine andere Ausfuehrung vorhanden ist.

Aufgabe: Spielen Sie " Tuerme von Hanoi " mit 4 Scheiben. Versuchen Sie so wenig Zuege wie moeglich zu machen. Schreiben sie die Zuege auf.
Wir nehmen die Tuerme a, b und c.



Nun wollen wir ein Programm schreiben, das uns die Zuege fuer dieses Spiel mit vorgegebener Scheibenanzahl ausgibt.
Beim realen Spiel wird man sicherlich erst einmal mit wenigen Scheiben anfangen. So leicht wollen wir es uns nicht machen. Nehmen wir an unser Spiel habe 70 Scheiben. Diese 70 Scheiben sollen von Turm a nach Turm b gebracht werden. Dies kann nicht auf einmal geschehen, da nur jeweils eine Scheibe bewegt werden darf.
Der Loesungsalgorithmus, der hier verfolgt werden soll, ist folgender: Wir bringen erst einmal alle Scheiben mit Ausnahme der untersten auf den Hilfsturm. Dann bewegen wir die unterste Scheibe auf den Zielturm.

Schliesslich bringen wir die Scheiben vom Hilfsturm auf den Zielturm.
In unserem Beispiel heisst das: Bringe 69 Scheiben auf Turm c. Bewege eine Scheibe auf Turm b. Bringe 69 Scheiben auf Turm b. Hinter dem " Bringen von 69 Scheiben " verbirgt sich sicherlich mehr als nur ein Befehl, es laesst sich durch drei aehnliche Anweisungen ersetzen: " Bringe 68 Scheiben ... ".

Wir planen eine Prozedur: Bringe (70,a,b,c).

Diese Prozedur soll folgendes ausfuehren: Bringe 70 Scheiben von Turm a nach Turm b unter Zuhilfenahme von Turm c als Zwischenlager fuer die 69 Scheiben.

Folgende Anweisung fuehrt die Prozedur Bringe aus:

```
Bringe (70,a,b,c): Bringe (69,a,c,b)
                   Bewege eine Scheibe von a nach b
                   Bringe (69,c,b,a)
```

In Worten: Bringe 70 Scheiben von a nach b ueber c heisst: Bringe 69 Scheiben von a auf den Hilfsturm c unter Zuhilfenahme von b. Bewege eine Scheibe (die unterste) von a nach b. Bringe die 69 Scheiben vom Hilfsturm c nach b unter Zuhilfenahme von a.
Jetzt hat es jeder gemerkt: Wir benutzen eine Rekursion. Die Prozedur Bringe ruft sich selbst zweimal auf.
Schauen wir uns noch an, was dann Bringe (69,a,c,b) macht:

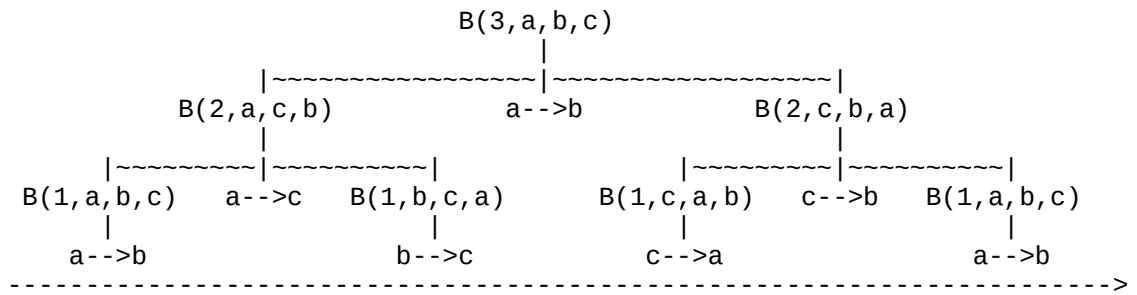
```
Bringe (69,a,c,b): Bringe (68,a,b,c)
                   Bewege eine Scheibe von a nach c
                   Bringe (68,b,c,a)
```

Aufgabe: Was macht Bringe (69,c,b,a) ?

Allgemein koennten wir fuer n Scheiben schreiben:

```
Bringe (n,a,b,c): Bringe (n-1,a,c,b)
                  Bewege eine Scheibe von a nach b
                  Bringe (n-1,c,b,a)
```

Da wir uns jetzt mit Rekursion auskennen, wissen wir, dass jede Rekursion eine Abbruchbedingung braucht.
In unserem Fall ist es natuerlich nur sinnvoll, weitere Prozeduren mit n-1 Scheiben aufzurufen, wenn die Anzahl der Scheiben groesser als Null ist.
Bevor wir uns nun an ein Pascal-Programm heranmachen, sollten wir an einem einfachen Beispiel die Prozedur Bringe testen.
Wir bringen 3 Scheiben von a nach b ueber c. Hierbei heisst B(3,a,b,c) soviel wie Bringe(3,a,b,c) und a-->b soviel wie "bewege eine Scheibe von a nach b".



Die einzigen Operationen, die ueberhaupt ausgefuehrt werden koennen, sind die Bewegungen einer Scheibe (z.b. a-->b). Lesen wir diese Bewegung in der Reihenfolge ihrer Ausfuehrung (Hilfe: Senkrechte Ebene von links nach rechts), so erhalten wir:

```
a-->b
a-->c
b-->c
a-->b
c-->a
c-->b
a-->b
```

Aufgabe: Schreiben Sie ein solches Diagramm fuer 4 Scheiben.

Nun aber die Pascal-Loesung:

```
(*$A-*)
PROCEDURE Bringe (n : INTEGER ; a, b, c : CHAR);

  BEGIN
    IF n > 0 THEN BEGIN
                    Bringe (n-1, a, c, b);
                    WRITELN (a, ' -> ', b);
                    Bringe (n-1, c, b, a)
                  END
  END; (* von Begin *)
(*$A+*)
```

Das ist schon die ganze Prozedur (wie oben geplant). Die Variablen fuer die Tuerme sind vom Typ CHAR. Weisen wir ihnen im Programm der Einfachheit halber die Buchstaben a, b und c zu.

```
PROGRAM Hanoi;

  VAR Anzahl : INTEGER;

  (*$A-*)
  PROCEDURE Bringe (n : INTEGER; a, b, c : CHAR);

    BEGIN
      IF n > 0 THEN BEGIN
                      Bringe (n-1, a, c, b);
                      WRITELN (a, ' -> ', b);
                      Bringe (n-1, c, b, a)
                    END
    END; (* von Bringe *)
  (*$A+*)

  BEGIN (* Hauptprogramm *)
    WRITELN ('Tuerme von Hanoi');
    WRITE ('Eingabe der Scheibenanzahl: ');
    READLN (Anzahl);
    Bringe (Anzahl, 'a', 'b', 'c')
  END.
```

Aufgabe: Testen Sie das Programm mit 3 und 4 Scheiben, und vergleichen Sie mit der oben "zu Fuss" erstellten Loesung. Testen Sie nun mit mehr Scheiben, und spielen Sie nach.

Hinweis: Da es zu sehr vielen Zuegen kommen kann, sollte das Programm eventuell fuer einen Drucker umgeschrieben werden.

```
PROGRAM Hanoi;

  VAR Anzahl : INTEGER;

  (*$A-*)
  PROCEDURE Bringe (n : INTEGER; a, b, c : CHAR);

    BEGIN
      IF n > 0 THEN BEGIN
                      Bringe (n-1, a, c, b);
                      WRITELN (LST, a, ' -> ', b);
                      Bringe (n-1, c, b, a)
                    END
    END; (* von Bringe *)
  (*$A+*)

  BEGIN (* Hauptprogramm *)
    WRITELN ('Tuerme von Hanoi');
    WRITE ('Eingabe der Scheibenanzahl: ');
    READLN (Anzahl);
    Bringe (Anzahl, 'a', 'b', 'c')
  END.
```

7.1 Felder

Eindimensionales Feld

Wir wollen eine Namenliste aller Schueler unserer Klasse erstellen. Nehmen wir an, es waeren maximal 40 Schueler in der Klasse zulaessig (die Paedagogen moegen es mir verzeihen!). In unserer Klasse sind aber nur 21 Schueler. Wir koennten als Variablen fuer die Schueler die Namen Schueler1, Schueler2, ..., Schueler 40 erfinden. Wie aber sollen wir nun alle Schuelernamen eingeben und spaeter wieder ausgeben? Diese Methode zwingt uns zu 40 Ein- und Ausgabeanweisungen.
Natuerlich geht es in Pascal (wie in allen Programmiersprachen) einfacher. Wir koennen indizierte Variablen verwenden. Dabei heissen in unserem Beispiel alle Variablen Schueler, gefolgt von einer Nummer 1 bis 40. Im Deklarationsteil des Programms muessen wir diesen Datentyp als Feld (Array) erklaeren.

```
VAR Schueler: ARRAY[1..40] OF STRING[80];
```

Die Variable Schueler kann also die Nummern 1 bis 40 haben, wobei jede dieser Variablen vom Typ STRING ist.
In den Anweisungen werden die Variablen als Schueler [1], Schueler [2], ..., Schueler [40] verwendet.
Die Zahl in der Klammer wird Index genannt. Als Index kann auch eine Variable vom Datentyp INTEGER verwendet werden.

Beispiel:

```
I := 5;
READLN (Schueler[I]);
```

hat den gleichen Effekt wie:

```
READLN (Schueler[5]);
```

Hinweis: In Programmiersprachen, die mit einem Compiler uebersetzt werden, muss die Deklaration eines Feldes genau festgelegt werden, welche Nummern als Indizes auftreten koennen, damit der Compiler genauso viele Speicherplaetze fuer die indizierten Variablen bereitstellen kann.
Das Programm fuer eine Schuelerliste:

```
PROGRAM Liste;

   CONST n=40;
   VAR i, e : INTEGER;
      Schueler : ARRAY [1..N] OF STRING[80];

   BEGIN
      WRITE ('Schuelerzahl: ');
      READLN (e);
      WRITELN ('Eingabe:');
      FOR i := 1 TO e DO BEGIN
         WRITE ('Schueler Nr. ',i :2,': ');
```

```
         READLN (Schueler [i])
       END; (* von FOR i *)
       WRITELN ('Ausgabe:');
       FOR i:= 1 TO e DO WRITELN (Schueler [i])
    END.
```

Beispiele fuer richtige Felddeklarationen:

```
    CONST Anfang = 7;
          Ende = 15;

    VAR A:ARRAY [1..30] OF INTEGER;
        B:ARRAY [2..Anfang] OF REAL;
        C:ARRAY [20..25] OF BOOLEAN;
```

Eine falsche Deklaration ist:

```
    VAR N : INTEGER;
        E : ARRAY[1..N] OF REAL;
```

weil N keine Konstante ist.

Mehrdimensionales Feld

Bis jetzt hatten die dargestellten Felder nur eine Dimension, d.h. jedes Element des Feldes hatte einen Index (eine Nummer). Jedoch kommt es haeufig vor, dass Elemente von Feldern zwei oder mehr Nummern erhalten sollen.Dies ist oft bei Tabellen der Fall.

```
T[ 1, 1] , T[ 1, 2] , T[ 1, 3] ,..., T[ 1,14] , T[ 1,15]
T[ 2, 1] , T[ 2, 2] , T[ 2, 3] ,..., T[ 2,14] , T[ 2,15]
T[ 3, 1] , T[ 3, 2] , T[ 3, 3] ,..., T[ 3,14] , T[ 3,15]
...

T[19, 1] , T[19, 2] , T[19, 3] ,..., T[19,14] , T[19,15]
T[20, 1] , T[20, 2] , T[20, 3] ,..., T[20,14] , T[20,15]
```

Wollen wir diese Tabelle mit 20 Reihen und 15 Spalten (d.h. in jeder Reihe 15 Werte) anlegen, so liesse sich ein Feld wie folgt definieren:
(Die Daten sollen vom Typ INTEGER sein)

```
TYPE Reihe = ARRAY[1..15] OF INTEGER;
     Tabelle = ARRAY[1..20] OF Reihe;

VAR T : Tabelle;
```

Nun koennen wir ein Tabellenelement z.B. ansprechen mit:

```
T [10][5] := 130;
```

Dies bedeutet, dass der Wert in der 10. Reihe und der 5. Spalte 130 betragen soll.
Bei dem so deklarierten Typ Tabelle handelt es sich also um ein eindimensionales Feld von eindimensionalen Feldern. Umstaendlich, nicht wahr!
Einfacher und uebersichtlicher laesst sich dieser Typ als zweidimensionales Feld deklarieren:

```
   TYPE Tabelle = ARRAY[1..20,1..15] OF INTEGER;

   VAR T : Tabelle;
```

Mit der Zuweisung

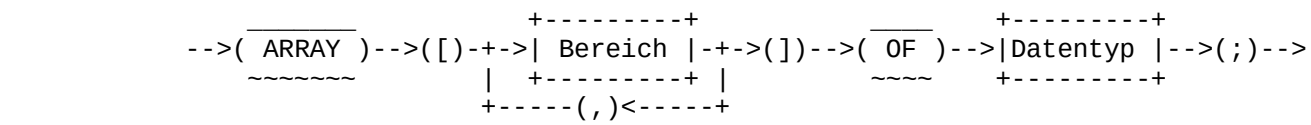
```
   T[10,5] := 130;
```

Solche zweidimensionalen Felder kennen wir auch aus der Mathematik: das Koordinatensystem, Matrizen, zweidimensionale Vektoren etc. Sicher sind auch noch mehr Dimensionen moeglich. Beispielsweise Koordinaten im Raum:

```
   VAR Koordinate : ARRAY [1..100,1..100,1..100] OF INTEGER;
```

Aber Vorsicht! Der Speicherplatzbedarf der Variablen Koordinate ist schon recht beachtlich: 100 mal 100 mal 100 mal 2 Bytes (fuer INTEGER) macht 2 Millionen Bytes - weit mehr als der Speicherplatz gaengiger Tischrechner. Aus Gruenden des Speicherplatzbedarfes werden sicher selten mehr als zwei Dimensionen gebraucht.

Mehrdimensionale Felder werden wie folgt deklariert:



Beispiele fuer korrekte Felddeklarationen:

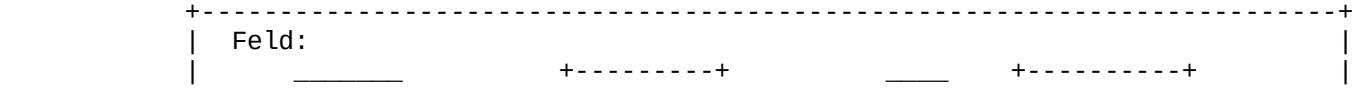
```
   TYPE Inber = 6..20;
        Letter = 'a'..'z';
        Farbe = (rot, gruen, gelb, blau, weiss, schwarz);

        Feld1 = ARRAY [Inber] OF REAL;
        Feld2 = ARRAY [Letter,Letter] OF Letter;
        Feld3 = ARRAY [20..30,5..8 ] OF Farbe;
        Feld4 = ARRAY [1..6, Farbe] OF BOOLEAN;
        Feld5 = ARRAY [TRUE..FALSE,1..2] OF CHAR;
```

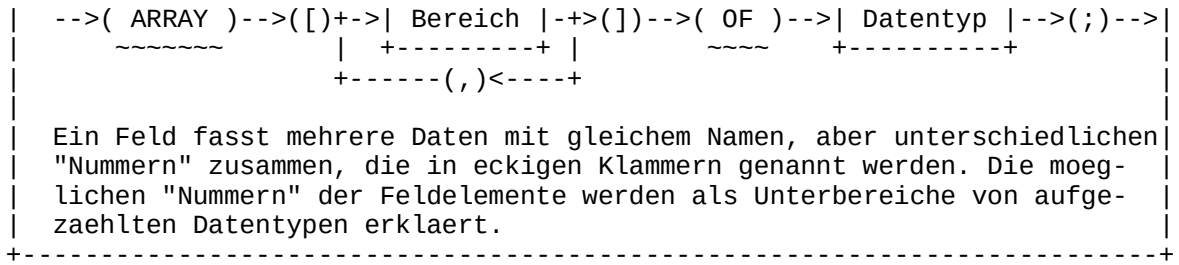
Hinweis: Der Datentyp STRING, der in Standard-Pascal nicht vorhanden ist, ist deklariert als:

```
   TYPE STRING = ARRAY [2..255] OF CHAR;
```

Ist s vom Typ STRING, so kann allerdings nicht direkt auf s[0] zugegriffen werden. In diesem Element befindet sich ein Kode fuer die Laenge des Strings. Geben wir aber dem Compiler die Anweisung, auf die Ueberpruefung des gueltigen Bereiches (range check) zu verzichten, so laesst sich die Laenge wie folgt ermitteln (Laenge: INTEGER):

```
   (* $R- *) (* Range check aus *)
   Laenge := ORD(s[0]);
   (* $R+ *) (* Range check an *)
```





Beipielprogramme zu Feldern

1. Zufallstest mit Histogramm

Wir wollen die Zuverlaessigkeit unseres Zufallszahlengenerators testen. Dazu erzeugen wir Zufallszahlen zwischen 0 und 9 und stellen deren Haeufigkeit in einem sogenannten Histogramm (Balkendiagramm) dar. Mit einem 80-Zeichen-Bildschirm koennten wir bis zu 75 Kreuze in der Breite auf dem Bildschirm ausgeben, wobei jedes Kreuz fuer das vorkommen der entsprechenden Zufallszahl steht. Einige Zeichen muessen wir fuer die Ziffern freilassen.

```
PROGRAM Randomtest;

   CONST Breite = 75; (* bei anderen Bildschirmen aendern *)
   VAR Maximum, i, j, n : INTEGER;
       Z : ARRAY [0..9] OF INTEGER;

   BEGIN
      CLRSCR;
      RANDOMIZE; (* neue Zufallszahlen *)
      WRITELN; WRITELN;
      Maximum := 0;
      FOR i := 0 TO 9 DO Z[i] := 0; (* alle Zahlen auf 0 setzen *)
      REPEAT
         n := RANDOM (10); (* Zahl zwischen 0 und 9 einschl. *)
         Z[n] := Z[n] + 1; (* Anzahl erhoehen *)
         IF Z[n] > Maximum THEN Maximum := z[n] (* Maximum der Anzahlen
                           auf neuesten Stand bringen *)
      UNTIL Maximum >= Breite; (* mehr Kreuze passen nicht auf Bild *)
      FOR i := 0 TO 9 DO BEGIN
         WRITE (i:1, ':');
         FOR j := 1 TO Z[i] DO WRITE('X');(* Anzahl der Kreuze fuer i *)
         WRITELN; WRITELN
      END (* von FOR i *)
   END.
```

Das Programm koennte folgendes Ergebnis haben:

```
0:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
2:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
3:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
4:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
5:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
6:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
7:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
   8:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   9:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

2. Lottozahlen

Das folgende Programm hilft uns beim Ausfuellen eines Lottoscheines (natuerlich ohne Gewaehr).
Dazu werden sechs Zahlen L[1] bis L[6] als Zufallszahlen erzeugt, wobei keine Mehrfachnennungen vorkommen duerfen.
Anschliessend wird der Lottoschein "ausgefuellt":

```
     10  20  30  40
  1  11  21   X  41
  2  12  22  32  42
  X  13  23  33  43
  4  14  24  34  44
  5   X  25  35  45
  6  16  26   X  46
  7  17  27  37   X
  8  18  28  38  48
  9  19   X  39  49
```

Das Programm dazu:

```
   PROGRAM Lottoblock;

      VAR i, j, z : INTEGER;
          L : ARRAY [1..6] OF INTEGER;
          Kreuz, Neuezahl, Ende : BOOLEAN;

      BEGIN
         RANDOMIZE; (* neue Zufallszahlen *)
            FOR i := 1 TO 6 DO BEGIN
               REPEAT
                  z := RANDOM(48) + 1; (* Zahl zwischen 1 und 49 *)
                  Neuezahl := TRUE;
                  FOR j := 1 TO i DO IF z = L[j] THEN Neuezahl := FALSE
                     (* ist wahr, wenn z neue Zufallszahl ist *)
               UNTIL Neuezahl;
               L[i] := z
            END; (* von FOR i *)
            CLRSCR;
            WRITELN ('Die Lottozahlen:');
            WRITELN;
            WRITE ('  ');
            Ende := FALSE;
            i := 10;
            REPEAT
               Kreuz := FALSE;
               FOR i := 1 TO 6 DO IF L[j] = i THEN Kreuz := TRUE;
               IF Kreuz THEN WRITE ('  X')
                        ELSE WRITE (i:3);
               IF i >= 49 THEN Ende := TRUE;
               IF i >= 40 THEN BEGIN  WRITELN;
                                      i := i-39
                               END
                          ELSE i := i + 10
            UNTIL ende
```

```
END.
```

3. Groesster und kleinster Messwert

In dem folgenden Beispiel soll eine Reihe von Messwerten (Datentyp REAL) eingegeben und der groesste und der kleinste Messwert bestimmt werden. Ausserdem berechnen wir als Nebenprodukt den Mittelwert der Messwerte und die groesste Abweichung vom Mittelwert.

Das Programm:

```
PROGRAM Messwerte;

   CONST N = 50;
   VAR Max, Min, Mittelwert, Abweichung, Dif :REAL;
       i, Anz : INTEGER;
       Wert : ARRAY [1..N] OF REAL;

   BEGIN
      WRITELN ('Messwerterfassung und Analyse: ');
      WRITE ('Wie viele Messwerte (max. 50) : ');
      READLN (Anz);
      FOR i := 1 TO Anz DO BEGIN
         WRITE (i:2, '.Wert: ');
         READLN (Wert[i])
      END; (* von FOR i *)
      Max := Wert[1];
      Min := Max; (* bestimmten Wert zuweisen *)
      Mittelwert := 0;
      FOR i := 1 TO Anz DO BEGIN
         IF Wert[i] > Max THEN Max := Wert[i];
         IF Wert[i] < Min THEN Min := Wert[i];
         Mittelwert := Mittelwert + Wert[i] (* vorlaeufig *)
      END; (* von FOR i *)
      Mittelwert := Mittelwert / Anz;
      Abweichung := 0;
      FOR i := 1 TO Anz DO BEGIN
         Dif := ABS (Mittelwert - Wert[i]);
         IF Dif > Abweichung THEN Abweichung := Dif
      END (* von FOR i *)
      CLRSCR;
      WRITELN; WRITELN;
      WRITELN ('Das Maximum ist: ', Max:7:2);
      WRITELN ('Das Minimum ist: ', Min:7:2);
      WRITELN ('Der Mittelwert : ', Mittelwert:7:2);
      WRITELN ('Mit der groessten Abweichung von:');
      WRITELN ('+/-', Abweichung:7:2)
   END.
```

4. Stundenplan

Ein Serviceprogramm fuer Ihren ganz persoenlichen Stundenplan. Mit diesem Programm kann der Benutzer einen Stundenplan eingeben, den ganzen Plan auf den Bildschirm ausgeben, einzelne Stunden abfragen und einzelne Stunden aendern.

```
PROGRAM Stundenplan;

   CONST N = 6;
   TYPE Wort    = STRING[10];
        Tage    = (Mo, Di, Mi, Don, Fr, Sa); (* Do ist reserviert *)
        Stunden = 1..N;
        Feld    = ARRAY [Tage,Stunden] OF Wort;

   VAR Plan     : Feld;
       Tag      : Tage;
       Std      : Stunden;
       ch       : CHAR;

   PROCEDURE Drucke (t : Tage);
      BEGIN
         CASE t OF
            Mo : WRITE ('Montag':12);
            Di : WRITE ('Dienstag':12);
            Mi : WRITE ('Mittwoch':12);
           Don : WRITE ('Donnerstag':12);
            Fr : WRITE ('Freitag':12);
            Sa : WRITE ('Samstag':12)
         END (* von CASE *)
      END; (* von Drucke *)

   PROCEDURE Weiter;
      BEGIN
         GOTOXY(10,20);
         WRITELN (CHR(7), 'Weiter mit <Return>');
         READLN
      END; (* von Weiter *)

   PROCEDURE Wann (VAR Tag : Tage; VAR Std : Stunden);
      VAR Antwort : Wort;
      BEGIN
         WRITELN ('Welcher Tag?');
         READLN (Antwort);
         Tag := Mo; (* fuer falsche Eingabe *)
         CASE Antwort[i] OF
            'M' : IF Antwort[2] = '0' THEN Tag := Mo
                                      ELSE Tag := Mi;
            'D' : IF Antwort[2] = 'I' THEN Tag := Di
                                      ELSE Tag := Don;
            'F' : Tag := Fr;
            'S' : Tag := Sa
         END; (* von CASE *)
         WRITELN ('Welche Stunde ?');
         REPEAT
            READLN (Std)
         UNTIL (Std > 0) AND (Std <= N)
      END; (* von Wann *)

   PROCEDURE Ganzer_Plan;
      BEGIN
         CLRSCR;
         WRITE ('   '); FOR Tag := Mo TO Do Drucke(tag);
         WRITELN;
         FOR Std := 1 TO N DO BEGIN
```

```
         WRITE (Std:3);
         FOR Tag := Mo TO Sa DO WRITE (Plan [Tag,Std] : 12);
         WRITELN
      END; (* von FOR Std *)
      Weiter
   END; (* von Ganzer_Plan *)

PROCEDURE Eine_Stunde;
   BEGIN
      CLRSCR;
      Wann (Tag,Std);
      WRITE ('Am '); Drucke(Tag);
      WRITELN (' in der ', Std:3,'. Stunde');
      WRITELN ('haben Sie ',Plan [Tag,Std] : 12);
      Weiter
   END; (* von Eine_Stunde *)

PROCEDURE Aenderung;
   BEGIN
      CLRCLS;
      WRITELN ('Aenderung:');
      Wann (Tag,Std);
      WRITELN ('Alter Wert: ', Plan [Tag,Std] : 12);
      WRITE ('Neuer Wert: ');
      READLN (Plan[Tag,Std]);
      Weiter
   END; (* von Aenderung *)

BEGIN (* Hauptprogramm *)
   CLRSCR; WRITELN ('Eingage des Plans: ');
   FOR Tag := Mo TO Sa DO BEGIN
      Drucke (Tag); WRITELN(':');
      FOR Std := 1 TO N DO BEGIN
         WRITE (Std:3,'. Stunde: ');
         READLN (Plan [Tag,Std])
      END; (* von FOR Std *)
      WRITELN
   END; (* von FOR Tag *)

   REPEAT
      CLRSCR;
      WRITELN (' G(anzen Plan ausgeben  ');
      WRITELN (' E(eine Stunde ausgeben ');
      WRITELN (' A(endern einer Stunde  ');
      WRITELN (' S(chluss               ');
      WRITELN;
      WRITELN (' -> Waehlen Sie: ');
      REPEAT
         READ (KBD,ch)
      UNTIL (ch = 'G') OR (ch = 'E') OR (ch = 'A') OR (ch = 'S');
      CASE ch OF
         'G' : Ganzer_Plan;
         'E' : Eine_Stunde;
         'A' : Aenderung
      END (* von CASE *)
   UNTIL ch = 'S'
END.
```

7.2 Sortieren von Feldern

Daten werden haeufig in eindimensionalen Feldern abgespeichert. Ein sinnvoller Zugriff auf die Daten erfolgt am besten mit sortierten Daten.

55 94 67 12 6 18 42

Diese acht Zahlen wollen wir sortieren.
Mit unserem Erfahrungsschatz ueber Zahlen faellt uns das sehr leicht. Wir wuerden moeglicherweise so vorgehen, dass wir uns erst die kleinste Zahl suchen, sie aufschreiben, dann die kleinste der verbliebenen Zahlen usw.
Natuerlich ueberblicken wir sofort, welches die kleinste Zahl ist. Dies ist aber nur deswegen moeglich, weil es so wenige Zahlen sind. Haetten wir 10 000 Zahlen zu ordnen, so muessten wir uns schon eine andere Methode suchen. Wir geben unseren Zahlen Namen mit Nummern:

```
a1 a2 a3 a4 a5 a6 a7 a8
55 94 67 44 12  6 18 42
```

Nun behaupten wir, die Zahl mit der kleinsten Nummer, a1, sei die kleinste Zahl. Natuerlich sieht jeder, dass das hier nicht zutrifft. Daher muessen wir die Zahl a1 mit allen folgenden (a2, a3, a4, a5, a6, a7, a8) vergleichen und den Platz tauschen, wenn wir eine kleinere Zahl als a1 gefunden haben:

```
a1 vergleichen mit a2 - OK
a1 vergleichen mit a3 - OK
a1 vergleichen mit a4 - vertauschen a1 <-----> a4

a1 a2 a3 a4 a5 a6 a7 a8
44 94 67 55 12  6 18 42

a1 vergleichen mit a5 - vertauschen a1 <-----> a5

a1 a2 a3 a4 a5 a6 a7 a8
12 94 67 55 44  6 18 42

a1 vergleichen mit a6 - vertauschen a1 <-----> a6

a1 a2 a3 a4 a5 a6 a7 a8
 6 94 67 55 44 12 18 42

a1 vergleichen mit a1 - OK
a1 vergleichen mit a8 - OK
```

Damit haben wir die kleinste Zahl = 6 gefunden.

Bei den Vertauschungen, die noetig waren, faellt uebrigens gleich der entscheidende Mangel dieses Algorithmus auf. Die zweitkleinste Zahl 12, die schon einmal an vorderster Stelle stand ist durch die Vertauschungen wieder weit nach hinten gerutscht. Bei anderen Zahlen ist es aehnlich.

Nun muessen wir die zweitkleinste Zahl a2 finden:

```
a2 a3 a4 a5 a6 a7 a8
94 67 55 44 12 18 42

a2 vergleichen mit a3 - vertauschen a2 <-----> a3
```

```
a2 a3 a4 a5 a6 a7 a8
67 94 55 44 12 18 42

a2 vergleichen mit a4 - vertauschen a2 <-----> a4

a2 a3 a4 a5 a6 a7 a8
55 94 67 44 12 18 42

a2 vergleichen mit a5 - vertauschen a2 <-----> a5

a2 a3 a4 a5 a6 a7 a8
44 94 67 55 12 18 42

a2 vergleichen mit a6 - vertauschen a2 <-----> a6

a2 a3 a4 a5 a6 a7 a8
12 94 67 55 44 18 42

a2 vergleichen mit a7 - OK
a2 vergleichen mit a8 - OK
```

Damit ist a2 die zweitkleinste Zahl.
Nun wird a3 als drittkleinste Zahl gesucht. Dazu vergleichen wir a3 mit allen folgenden a4 bis a8 und vertauschen, wenn noetig. Wir finden a3 = 18 als dritte Zahl.

Es bleibt:

```
a4 a5 a6 a7 a8
94 67 55 44 42
```

Die Prozedur des Vergleichens und Vertauschens wird fortgefuehrt, bis wir a8 vergleichen muessten. Dies ist selbsverstaendlich nicht mehr noetig, da a8 nur noch die groesste Zahl sein kann.
Aufgabe: Schreiben Sie die Vergleiche und noetigen Vertauschungen fuer a3 bis a8 ebenfalls auf.
Was haben wir nun getan ?
Wir haben nacheinander die Zahlen a1 bis a7 mit allen jeweils folgenden Zahlen verglichen. Wenn dabei eine kleinere Zahl gefunden wurde, haben wir die Zahlen vertauscht.
Dazu planen wir eine Prozedur Namens Sort in Pascal. Die Variablen im Hauptprogramm sollen sein a[1], a[2], ... ,a[N], wobei N eine Konstante mit dem Wert 8 ist.

Das Programm mit der Prozedure Sort:

```
   PROGRAM Tauschsort;

      CONST N = 8;
      VAR i : INTEGER;
          a : ARRAY [1..N] OF INTEGER;

      PROCEDURE Sort;
         VAR i : INTEGER;
             Hilf : INTEGER;

           BEGIN
```

```
            FOR i := 1 TO N-1 DO
               FOR j := i TO N DO
                  IF a[i] > a[j] THEN BEGIN
                    Hilf := a[i];
                    a[i] := a[j];
                    a[j] := Hilf (* Speichertausch *)
                  END (* von IF *)
         END; (* von Sort *)

   BEGIN (* Hauptprogramm *)
     WRITELN ('Geben Sie ', N:3,' Zahlen ein: ');
     FOR i:= 1 TO N DO READLN (a[i]);
     Sort;
     WRITELN ('Sortiert: ');
     FOR i:= 1 TO N DO WRITELN (a[i]:3)
   END.
```

In der Sortierprozedur werden also die Zahlen a[1] bis a[7] mit allen folgenden Zahlen a[i+1] bis a[8] verglichen und noetigenfalls vertauscht.

Damit wir die Vorgaenge bei den Vergleichen und Vertauschungen besser verstehen koennen, hier ein etwas erweitertes Programm, das die Zahlen bei jedem Schleifendurchlauf auf einem Drucker ausdruckt. Mit dem Programm koennte auch eine Teilfolge sortiert werden, da der Anfangsindex l (fuer links) und der Endindex r (fuer rechts) gesetzt wird.Im Ausdruck wird jeweils der Wert fuer l, r, i und j ausgegeben. In der Zahlenfolge werden die Zahlen unterstrichen,die vertauscht worden sind. Zaehlt man nach ,so ergibt sich bei unserer Beispielzahlenfolge eine Anzahl von 21 Vertauschungen. Das bedeutet, dass 21mal Daten transportiert werden muessen.

Hinweis zum Programm: Bei dem verwendeten Drucker wird das Unterstreichen mit der Kontrollzeichensequenz "ESC, '-', CHR(0/1)" gesteuert. Fuer andere Drucker muessen die Programmzeilen geaendert werden.

Programm mit Druckerausgabe:

```
PROGRAM Tauschsort;
   CONST N = 8;
   VAR i : INTEGER;
       a : ARRAY [1..N] OF INTEGER;

   PROCEDURE Ausgabe (l, r, i, j : INTEGER);
      VAR k : INTEGER;
      BEGIN
         WRITE (LST,CHR(27), '-', CHR(1)); (* Unterstreichen an *)
         WRITE (LST, '   l:  r:  i  j:       ');
         FOR k := 1 TO N DO WRITE (LST, k:4);
         WRITE (LST, CHR(27), '-', CHR(0)); (* Unterstreichen aus *)
         WRITELN (LST);
         WRITE (LST, l:4, r:4, i:4, j:4);
         WRITE (LST,'    a[i]:  ');
         FOR k := 1 TO N DO BEGIN
            IF (k = i) OR (k = j) THEN BEGIN
               WRITE (LST, CHR(27), '-', CHR(1)); (* Unterstreichen an *)
               WRITE (LST, a[k]:4);
               WRITE (LST, CHR(27), '-', CHR(0)) (*Unterstreichen aus *)
            END (* von IF *)
            ELSE WRITE (LST, a[k]:4)
```

```
         END; (* von k *)
         WRITELN (LST);
         WRITELN (LST)
      END; (* von Ausgabe *)

   PROCEDURE Sort (l, r :INTEGER);
      VAR i, j : INTEGER;
          w : INTEGER;

      BEGIN
         FOR i := 1 TO r-1 DO
            FOR j := i TO r DO BEGIN
               IF a[i] > a[j] THEN BEGIN
                  w := a[i]; a[i] := a[j]; a[j] := w;
                  Ausgabe(l, r, i, j)
               END (* von IF *)
            END (* von FOR j *)
      END; (* von Sort *)

   BEGIN (*Hauptprogramm *)
      WRITELN ('Geben Sie ', N:3, ' Zahlen ein.');
      FOR i := 1 TO N DO READLN (a[i]);
      WRITELN (LST, 'Anfang:');
      WRITE (LST, '                          ');
      FOR i := 1 TO N DO WRITE (LST, a[i]:4);
      WRITELN (LST);
      Sort (1,N);
      WRITELN (LST, 'Ende:');
      WRITE (LST, '                             ');
      FOR i := 1 TO N DO WRITE (LST, a[i]:4);
      WRITELN (LST);
   END.
```

Aufgabe: Testen Sie das Programm, und versuchen Sie anhand des Ausdruckes den Sortiervorganges zu verstehen!
Wenn wir eine andere Anzahl von Daten sortieren wollen, so aendern wir einfach die Konstante N.
Sollen Daten von anderen Typen sortiert werden, so wird der Datentyp des ARRAYs und der Typ der Hilfvariablen Hilf geaendert. Wenn wir Daten vom Typ CHAR oder STRING sortieren, dann wird die Reihenfolge des entsprechenden Zeichensatzes, hier ASCII (siehe Kap. 3.3). verwendet.
Unser Algorithmus in der Prozedur Sort kann leicht dadurch entscheidend verbessert werden, dass die innere Schleife, welche die jeweils nachfolgenden Elemente untersucht, nicht vorwaerts , sondern rueckwaerts durchlaufen wird.
Das Programm mit dieser Aenderung:

```
PROGRAM Tauschsort2;
   CONST n = 8;
   VAR i : INTEGER;
       a : ARRAY [1..N] OF INTEGER;

   PROCEDURE Sort;
      VAR i, j : INTEGER;
          Hilf : INTEGER;

      BEGIN
         FOR i := 1 TO N-1 DO
```

```
            FOR j := N DOWNTO i DO     (* hier geaendert *)
               IF a[i] > a[j] THEN BEGIN
                  Hilf := a[i];
                  a[i] := a[j];
                  a[j] := Hilf        (* Speichertausch *)
               END (* von IF *)
         END; (* von Sort *)

      BEGIN (* Hauptprogramm *)
         WRITELN ('Geben Sie ', N:3, 'Zahlen ein: ');
         FOR i := 1 TO N DO READLN (a[i]);
         Sort;
         WRITELN ('Sortiert: ');
         FOR i:=1 TO N DO WRITELN (A[i]:3)
      END.
```

Der hier benutzte Sortieralgorithmus wird oft auch BUBBLESORT genannt. Beim Sortiern steigen die kleineren Zahlen wie Blasen (Bubbles) zum Anfang des Datenfeldes auf.
Diese Eigenschaft des Sortieralgorithmus koennen wir uns auch hier wieder durch ein Demonstrationsprogramm auf einem Drucker sichtbar machen:

```
   PROGRAM Tauschsort2;

      CONST N = 8;
      VAR i : INTEGER;
          a : ARRAY [1..N] OF INTEGER;

      PROCEDURE Ausgabe (l, r, i, j : INTEGER);
         VAR k : INTEGER;
         BEGIN
            WRITE (LST, CHR(27), '-', CHR(1)); (* Unterstreichen an *)
            WRITE (LST, '   l:  r:  i:  j:           ');
            FOR k := 1 TO N DO WRITE (LST, k:4);
            WRITE (LST, CHR(27),'-', CHR(0)); (* Untersteichen aus *)
            WRITELN (LST);
            WRITE (LST,l:4, r:4, i:4, j:4);
            WRITE (LST,'    a[i]:  ');
            FOR K :=1 TO N DO BEGIN
               IF (K = i) OR (k = j) THEN BEGIN
                  WRITE (LST, CHR(27),'-',CHR(1)); (* Unterstreichen an *)
                  WRITE (LST,a[k]:4);
                  WRITE (LST,CHR(27),'-',CHR(0)) (* Unterstreichen aus *)
               END (* von IF *)
               ELSE WRITE (LST,a[k]:4)
            END;  (* von FOR *)
            WRITELN (LST);
            WRITELN (LST)
         END; (* von Ausgabe *)

      PROCEDURE Sort (l, r : INTEGER);
         VAR i,J : INTEGER;
             w : INTEGER;

         BEGIN
            FOR i := 1 TO r-1 DO
               FOR j := r DOWNTO i DO BEGIN
                  IF a[i] > a[j] THEN BEGIN
```

```
                w := a[i]; a[i] := a[j]; a[j] := w;
                Ausgabe (l,r,i,j)
             END (* von IF *)
          END (* von FOR j *)
       END; (* von Sort *)

       BEGIN (* Hauptprogramm *)
          WRITELN ('Geben Sie ',N:3, 'Zahlen ein.');
          FOR i := 1 TO N DO READLN (a[i]);
          WRITELN (LST,'Anfang:');
          WRITE (LST,'                          ');
          FOR i := 1 TO N DO WRITE (LST,a[i]:4);
          WRITELN (LST);
          SORT (1,N);
          WRITELN (LST, 'Ende:');
          WRITE (LST,                           ');
          FOR i := 1 TO N DO WRITE (LST,a[i]:4);
          WRITELN (LST);
       END.
```

Aufgabe: Testen Sie auch dieses Programm mit der Beispielzahlenfolge aus dem vorigem Beispiel.
Im Gegensatz zu 21 Vertauschungen in der alten Version braucht unser Programm nur noch 11 Vertauschungen vorzunehmen.

Als dritter Sortieralgorithmus soll nun noch ein rekursiver Algorithmus behandelt werden, der vor allem bei grossen Datenmengen besonders schnell ist: Quicksort.
Die (rekursive) Prozedur Sort(l,r) sortiert einen Teil des Datensatzes angefangen bei einem "linken" Element (l) bis zu einem "rechten" Element (r). Beim ersten Aufruf der Prozedur (im Hauptprogramm) sind l und r natuerlich die erste und die letzte Nummer.
Nun wird ein Datensatz unterteilt. Dazu wird ein beliebiges, hier das mittlere, Element gesucht (x := a[(l+r) DIV 2]).
Als naechstes wird der Datensatz, von links (mit dem Index i) und von rechts (mit dem Index j) kommend, durchsucht, bis sich die Indizes i und j treffen. Dabei wird x mit einem "linken" Element getauscht, wenn dieses groesser als x ist, und x wird mit einem "rechten" Element getauscht, wenn dieses kleiner ist als x.
Nach dieser Schleife sind im linken Teil nur Elemente, die kleiner, und im rechten Teil nur Elemente, die groesser sind als x.
Die Prozedur ruft sich selbst zweimal auf, damit nach dem gleichen Schema die beiden Teile des Datensatzes weiter sortiert werden, solange der entsprechende Teil aus mehr als einem Element besteht.

Als Programm stellt sich der Algorithmus folgendermassen dar:

```
   PROGRAM Quicksort;
   (* SA- *)
      CONST N = 8;
      VAR i : INTEGER;
          a : ARRAY[1..N] OF INTEGER;

      PROCEDURE Sort (l, r : INTEGER);
         VAR i, j : INTEGER;
             x, Hilf : INTEGER;

         BEGIN
```

```
         i := l; j := r;
         x := a[(l+r) DIV 2]; (* mittleres Element *)
         REPEAT
            WHILE a[i] < x DO i := i + 1;
            WHILE x < a[j] DO j := j - 1;  (* Annaehern *)
            IF i <= j THEN BEGIN
               Hilf := a[i]; a[i] := a[j]; a[j] ;= Hilf; (* Tausch *)
               i := i+1; j := j-1;
            END (* von IF *)
         UNTIL i > j;
         IF l < j THEN Sort (l,j);
         IF i < r THEN Sort (i,r)
      END; (* von Sort *)
      (* SA+ )

   BEGIN (* Hauptprogramm *)
      WRITELN ('Geben Sie ', N:3, ' Zahlen ein  ');
      FOR i := 1 TO N DO READLN(a[i]);
      Sort(1,N);
      WRITELN ('Sortieren: ');
      FOR i:= 1 TO N DO WRITELN (a[i]:3);
   END.
```

Auch bei diesem Algorithmus wollen wir zum besseren Verstaendnis schrittweise ausdrucken lassen, welche Vertauschungen gerade vorgenommen werden. Ausserdem ist es hier noetig, zu verfolgen, welche Sortierprozedur aufgerufen worden ist.
Dazu wieder ein Programm, dass ein Protokoll des Sortierverfahrens ausdruckt:

```
PROGRAM Quicksort;

   COPNST N = 8;
   VAR i : INTEGER;
       a : ARRAY[1..N] OF INTEGER;
       u : BOOLEAN;

   PROCEDURE Ausgabe (l, r, i, j, x :INTEGER);
      VAR k : INTEGER;
      BEGIN
         WRITE (LST, CHR(27), '-', CHR(1)); (: Unterstreichen an *)
         WRITE (LST, '   l:  r:  i:  j:  x:        ');
         FOR k := 1 TO  N DO WRITE (LST, k:4);
         WRITE (LST, CHR(27), '-', CHR(0)); (: Unterstreichen aus *)
         WRITELN (LST);
         WRITE (LST, l:4, r:4, i:4, j:4,x:4);
         WRITE (LST, '    a[i]:  ');
         FOR k := 1 TO N DO BEGIN
            IF u THEN IF (k=i) OR (k=j) THEN WRITE (LST, CHR(27), '-',
                                                    CHR(1));
            WRITE (LST, a[k]:4);
            WRITE (LST, CHR(27), '-', CHR(0))
         END; (* von FOR k *)
         WRITELN (LST);
      END; (* von Ausgabe *)

   (* SA- *)
   PROCEDURE Sort (l, r: INTEGER);
      VAR i, j: INTEGER;
```

```
        x, w: INTEGER;

    BEGIN
       WRITELN (LST, 'Sort(',l:2,', ',r:2,')');
       i := l; j := r;
       x := a[(l + r) DIV 2];
       REPEAT
          u := FALSE;
          WHILE a[i] < x DO BEGIN Ausgabe(l, r, i, j, x); i:= i+1 END;
          WHILE x < a[j] DO BEGIN Ausgabe(l, r, i, j, x); j:= j-1 END;
          IF i <= j THEN BEGIN
             w := a[i]; a[i] := a[j]; a[j] := w;
             u := TRUE;
             Ausgabe (l, r, i, j, x);
             i := i + 1; j := j - 1
          END (* von IF *)
       UNTIL i > j;
       IF l < j THEN Sort (l, j);
       IF i < r THEN Sort (i, r)
    END; (* von Sort *)
    (* SA+ *)

BEGIN (* Hauptprogramm *)
   WRITELN ('Geben Sie ', N:3, ' Zahlen ein.');
   FOR i := 1 TO N DO READLN (a[i]);
   WRITELN (LST, 'Anfang:');
   WRITE (LST, '                              ');
   FOR i := 1 TO N DO WRITE (LST, a[i]:4);
   WRITELN (LST);
   Sort(1, N);
   WRITELN (LST, 'Ende:');
   WRITE (LST, '                              ');
   FOR i := 1 TO N DO WRITE (LST, a[i]:4);
   WRITELN (LST);
END.
```

Aufgabe: Testen Sie auch dieses Programm mit der Beispielzahlenfolge aus den vorherigen Beispielen.
Der Quicksort-Algorithmus nimmt bei unserer Testzahlenfolge (gleiche Zahlenfolge fuer alle drei Programme) nur 9 Vertauschungen vor. Es scheint also, als sei Quicksort der schnellste Algorithmus. So einfach koennen wir uns diese Bewertung jedoch nicht machen. Zur Untersuchung der Qualitaet eines Sortieralgorithmus muessen wir mit unterschiedlich grossen und unterschiedlich geordneten Datenmengen arbeiten, sowie unterschiedliche Datentypen untersuchen. Ausserdem gibt es noch einige andere Sortieralgorithmen, die hier aus Platzgruenden nicht aufgefuehrt werden koennen. Eine genauere Analyse der bekannten Sortieralgorithmen gibt N. Wirth in seinem Buch "Algorithmen und Datenstrukturen" .
Fuer unseren "taeglichen" Bedarf an Sortieralgorithmen koennen wir festhalten, dass sich Quicksort oft als der schnellste Algorithmus bei groesseren Datenmengen herausstellt. Bubblesort dagegen ist wegen seiner Einfachheit zu empfehlen, wenn es nicht so sehr auf die Rechenzeit ankommt. Immerhin ist Bubblesort einer der "schnellern" Algorithmen. (Zu Quicksort siehe auch Kap.10.1).
Zum Schluss noch eine kleine Funktion, die ein sortiertes Feld von ganzen Zahlen nach einem bestimmten Element durchsucht, um dann den logischen Wert TRUE zu erhalten, wenn das Element gefunden ist, und FALSE, wenn es nicht vorhanden ist. Die vorgestellte Art des Suchens nennt sich binaers Suchen,

weil von der Mitte des Datensatzes ausgehend abhaengig vom Wert der Suchzahl nur nach links oder nach rechts weiter gesucht wird. Beim Durchsuchen der entstandenen Haelfte des Datensatzes wird wieder entsprechend vorgegangen, d.h. der Datensatz wieder halbiert.

```
FUNKTION binsuch (suchzahl : INTEGER): BOOLEAN;
   VAR links, rechts, position: INTEGER;
       gefunden : BOOLEAN;
   BEGIN
      links := 1; rechts := max;
      REPEAT
         positionen := (links + rechts) DIV 2;
         IF suchzahl > element[position] THEN links := position - 1
                                         ELSE rechts := position + 1;
         gefunden := suchzahl = element[position]
      UNTIL (links > rechts) OR gefunden;
      binsuch := gefunden
   END; (* von binsuch *)
```

Aufgabe: Machen Sie sich den Suchvorgang auf einem Blatt Papier anhand einer Zahlenfolge klar.

7.3 Verbunde

Es gibt Dinge, die gehoeren einfach zusammen:

Name, Vorname, Strasse, Hausnummer und Ort
Zaehler und Nenner
Name, Alter, Geschlecht, Gewicht und Schuhgroesse
Nummer, Name, Regalnummer, Preis, Anzahl und Art

Oft treffen wir auf Daten, die sinnvoll unter einem Oberbegriff zusammengefasst werden sollen: Adressen, Brueche, Personaldaten, Artikel. Eine solche Zusammenfassung ist in Pascal moeglich durch den strukturierten Datentyp Verbund (RECORD).

```
TYPE   Geschlecht =   (maennlich, weiblich);
       Art        =   (Buecher, Bekleidung, Lebensmittel, Elektro);

       Adresse       RECORD
                       Name, Vorname, Strasse, Ort : STRING[20];
                       Hausnummer : INTEGER
                     END;

       Bruch      =  RECORD
                       Zaehler, Nenner : INTEGER
                     END;

       Persondat  =  RECORD
                       Name    : STRING[20];
                       Alter   : INTEGER;
                       Gewicht : REAL;
                       Sex     : Geschlecht;
                       Schuhgr : 17..46
                     END;

       Artikel    =  RECORD
```

```
                      Nummer  : INTEGER;
                      Name    : STRING[20];
                      Regalnr : INTEGER;
                      Preis   : REAL;
                      Anzahl  : INTEGER;
                      Woher   : Art
                    END;
```

Mit diesen Typdeklarationen sollen folgende Variablen verwendet werden:

```
VAR Pers    : ARRAY[1..30] OF Adresse;
    Produkt : Artikel;
    a, b    : Bruch;
    x, y, z : Persondat;
```

Dann sind korrekte Operationen:

```
   Produkt.Woher := Elektro;
   Pers[15].Name := 'Meier';
   Pers[16] := Pers[15];
   a.Zaehler := 7;
   b.Nenner := a.Zaehler;
   x.Name := Pers[15].Name;
   y := x; z := x;
```

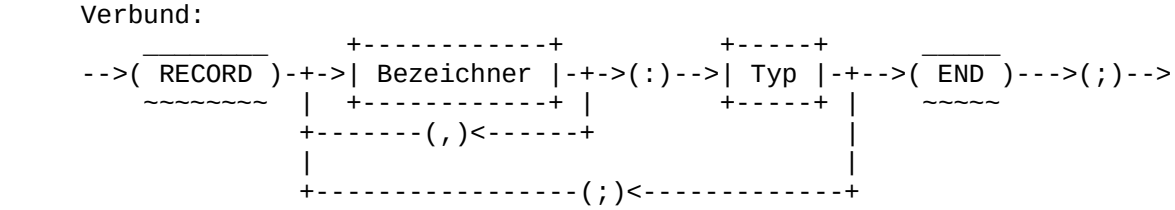
Mit dem Variablennamen x ist der ganze Verbund mit allen seinen Teilen gemeint. Dagegen wird mit x.Name nur auf einen bestimmten Teil zugegriffen.

Falsche Operationen sind dagegen:

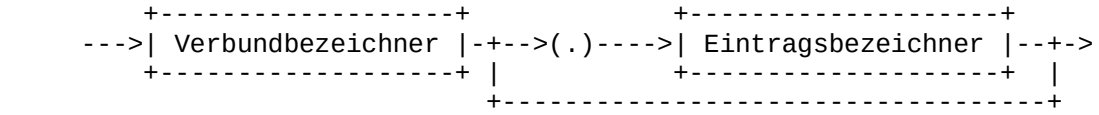
```
   Woher := Elektro;
   Produkt.Name := 'Buecher';
   Pers.Name[15] := 'Meier';
   Nenner := Zaehler;
```

Achtung: Bei der Zuweisung eines Verbundes an einen anderen werden alle Teile des Verbundes zugewiesen. Daher ist diese Zuweisung nur moeglich, wenn beide Variablen vom gleichen Typ sind!
Bei der Zuweisung von Teilen von Verbunden muessen beide Teile vom gleichen Typ sein.

Verbund:



Zugriff auf Verbundvariablen:



Wie in den Beispielen zu sehen war, koennen auch Felder von Verbunden auftreten. Weiterhin sind Felder als Teile von Verbunden moeglich:

```
TYPE Beispiel = RECORD
                 a : ARRAY[1..5] OF INTEGER;
                 b : BOOLEAN
                END;

VAR Wert : ARRAY[1..10] OF Beispiel;
```

Nun sind folgende Zuweisungen moeglich:

```
   Wert[8].b := TRUE;
   Wert[3].a[5] := 120;
   Wert[5].a[3] := 40;
```

Aufgabe: Bilden Sie weitere gueltige Zuweisungen!

Auch Schachtelungen von Verbunddeklarationen koennen von Nutzen sein, werden aber oft recht komplex.

```
TYPE Wohnung = RECORD
                  Strasse   : STRING[20];
                  Nr,PLZ    : INTEGER;
                  Ort       : STRING[20]
               END;

     Person  = RECORD
                  Name,Vorname : STRING[20];
                  Sitz         : Wohnung
               END;
```

Statt dieser Typdeklaration koennen wir auch schreiben:

```
TYPE Person = RECORD
               Name,Vorname : STRING[20];
               Sitz         : RECORD
                               Strasse  : STRING[20];
                               Nr,PLZ   : INTEGER;
                               Ort      : STRING[20]
                              END;
               END;
```

Im Ergebnis scheinen beide Deklarationen gleich zu sein. Doch ist mit der zweiten Version keine Variable vom Typ Wohnung moeglich !

Nun deklarieren wir die Variablen:

```
   VAR a, b : Person;
```

Gueltige Zuweisungen sind dann:

```
   a.Name := 'Meier';
   a.Sitz.Strasse := 'Hauptstrasse';
   b.Name := a.Name;
   b.Sitz.Ort := a.Name;
```

Falsche Zuweisungen sind dagegen:

```
   a.Sitz := 'Unna';
```

```
   a.Sitz.PLZ := '4750';
```

Aufgabe: Bilden Sie weitere gueltige Zuweisungen!

Nun noch eine Hilfe fuer die Arbeit mit Verbunden: die WITH-Anweisung. Wenn wir oefter im Programm Daten zu behandeln haben, die Teile von Verbunden darstellen, so ist dies oft sehr aufwendig und laestig, da die Teile allein nicht definiert sind.

Mit Variablen vom Typ

```
   VAR Buch : RECORD
               Titel,Autor : STRING[20];
               Seitenzahl  : INTEGER;
               Preis       : REAL;
              END;
```

ist eine Zuweisung

```
   Autor := 'Rollke';
```

falsch ! Dies liegt nicht an dem Autor, sondern daran, dass eine Variable Autor nicht deklariert ist. Vielmehr muss es heissen

```
   Buch.Autor := 'Rollke';
```

Wollen wir eine Inventurliste unserer Buecher aufstellen, so muessten wir fuer jedes Buch die Eingabe wie folgt strukturieren:

```
   READLN (Buch.Autor);
   READLN (Buch.Titel);
   READLN (Buch.Seitenzahl);
   READLN (Buch.Preis);
```

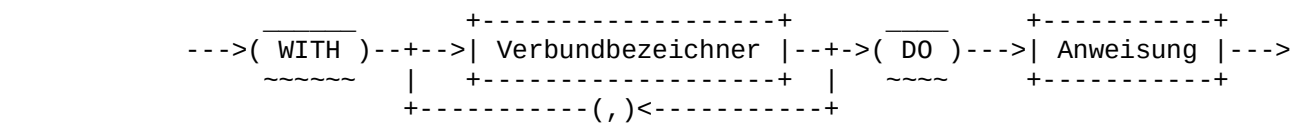
Hier bietet Pascal eine Hilfe an:

```
   WITH Buch DO BEGIN
      READLN (Autor);
      READLN (Titel);
      READLN (Seitenzahl);
      READLN (Preis)
   END; (* von WITH *)
```

Auch die Zuweisung

```
   WITH Buch DO Autor := 'Rollke';
```

ist nun richtig.



Zur Uebung des RECORD-Typs ein kleines Beispielprogramm, das nach Belieben erweitert werden kann (und soll). Das folgende Programm laesst den Benutzer die Daten des Periodensystems der Elemente eingeben:

Elementname, Abkuerzung, Atomgewicht, gibt es radioaktive Isotope ?
Die Eingaben erfolgen in der Reihenfolge der Ordnungszahlen. Anschliessend werden die Elemente jeweils nach Namen und Atomgewichten geordnet und ausgegeben.
Erweiterungsvorschlaege fuer das Programm:
- nach anderen Kriterien ordnen;
- Menuesteuerung einbauen;
- Liste auf Drucker ausgeben;
- alle (nicht-)radioaktiven Elemente ausgeben;
- pro Element noch Isotope speichern (mit ARRAY);
- spaeter (wenn Dateien bekannt sind) zur Speicherung der Daten aendern.

```
   PROGRAM Periodensystem;

      TYPE Elementtyp = RECORD
                          Name     : STRING[20];
                          Abk      : STRING[2];
                          Ordnzahl : INTEGER;
                          Radioakt : BOOLEAN;
                          Atommasse: REAL
                        END; (* von RECORD *)

      VAR Element : ARRAY[1..118] OF Elementtyp;
                n : INTEGER;
               ch : CHAR;

      PROCEDURE Sort (Mouds, Ende : Integer);
         VAR i, j : INTEGER;


         PROCEDURE Tausche (VAR a, b :Elementtyp);
            VAR Hilf : Elementtyp;
            BEGIN
               Hilf := a;
               a := b:
               b := Hilf
            END; (* von Tausche *)

         BEGIN
            FOR i := 1 TO Ende-1 DO
               FOR j := Ende DOWNTO i DO
                  CASE Modus OF
                     1 : IF Element[i].Name > Element[j].Name THEN
                            Tausche (Element[i], Element[j];
                     2 : IF Element[i].Atommasse > Element[j].Atommasse THEN
                            Tausche (Element[i], Element[j])
                  END (* von CASE *)
         END; (* von Sort *)

      PROCEDURE Ausgabe (Ende : INTEGER);
         VAR i : INTEGER;
         BEGIN
            FOR i := 1 TO Ende DO WITH Element[i] DO BEGIN
               WRITE(Ordnzahl:3,Name:20,Abk:4,' Atommasse=',Atommasse:8:4);
               IF Radioakt Then WRITE (' hat ') ELSE WRITE (' ohne ');
               WRITELN (' radioaktive Isotope')
            END (* von WITH *)
         END; (* von Ausgabe *)
```

```
BEGIN (* Hauptprogramm *)
   WRITELN ('Eingabe der Daten '); WRITELN;
   n := 0;
   REPEAT
      n := n + 1;
      WITH Element[n] DO BEGIN
         Ordnzahl := n;
         WRITELN (n:3, '. Element:');
         WRITE ('Name:       '); READLN (Name);
         WRITE ('Abkuerzung: '); READLN (Abk);
         WRITE ('Atommasse:  '); READLN (Atommasse);
         WRITE ('Radioaktiv (J/N) ? '); READ (KBD,ch);
         Radioakt := (ch = 'J') OR (ch = 'j');
         WRITELN; WRITELN
      END; (* von WITH *)
      WRITE ('Weiter (J/N)? '); READ (KBD,ch); WRITELN
   UNTIL (ch <> 'J') AND (ch <> 'j');
   CLRSCR;
   WRITELN ('Liste nach Namen sortiert: '); WRITELN;
   Sort (1,n);
   Ausgabe (n);
   WRITELN; WRITELN ('RETURN-Taste druecken...');
   READLN;
   WRITELN ('Liste nach Atommassen sortiert: '); WRITELN;
   Sort (2,n);
   Ausgabe (n)
END.
```

Aufgabe: Testen und erweitern Sie das Programm.

Varianten

Zum guten Schluss, als Kroenung sozusagen, wollen wir uns mit einer besonderen Form des Verbundes beschaeftigen, mit der Variante. Oft wird diese Form auch als varianter RECORD bezeichnet.
Es handelt sich hier um die Moeglichkeit, in der Typdeklaration Objekten mit gleichem Namen abhaengig von bestimmten Eigenschaften verschiedene Strukturen zuzuordnen.
In unserem Beispiel (Periodensystem) koennten wir z.B. daran interessiert sein, dass nur bei den Elementen, die radioaktive Isotope besitzen, auch noch die Strahlungsarten mitgespeichert werden.

```
TYPE Strahlung = (alpha, beta, gamma);

VAR Element = RECORD
               Name  : STRING[20];
               Masse : REAL;
               CASE Radioak : BOOLEAN OF
                  (TRUE:Art:Strahlung)
               END;
```

Nun gibt es zwei verschiedene Moeglichkeiten, die Variable ELEMENT mit Daten zu versorgen:

```
   +-------------------+    +----------------------+
   |  Name:    Helium  |    |   Name:    Uran226   |
```

```
   |  Masse:   4.0026  |    |   Masse:   238.029   |
   |  Radioak: False   |    |   Radioak: True      |
   |    ...    ...     |    |   Art:     alpha     |
   +-------------------+    +----------------------+
```

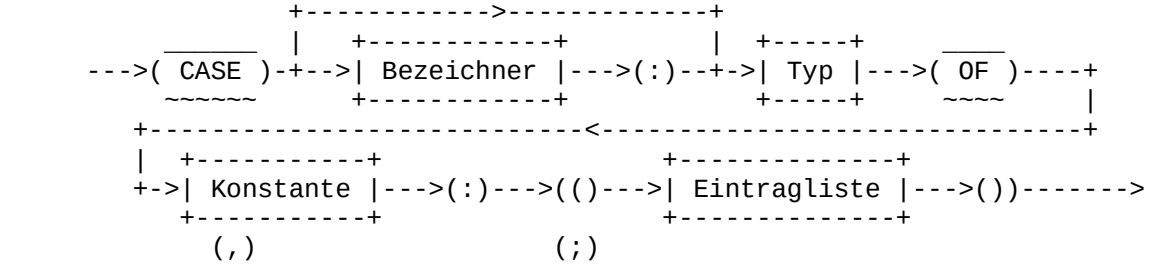
Ein weiteres Beispiel:

```
TYPE Schueler = RECORD
                  Name,Vorname      : STRING[20];
                  Einschulung       : RECORD
                                        Tag : 1..31;
                                        Mon : 1..12;
                                        Jahr: INTEGER
                                      END;
                  CASE Klassenstufe : INTEGER OF
                  5,6,7,8,9,10      : (Klasse : STRING[20];
                                       Klassenlehrer : STRING[20])
                  11,12,13          : (Tutor : STRING[20];
                                       Kurse : ARRAY[1..8] OF STRING[20])
                  END;
```

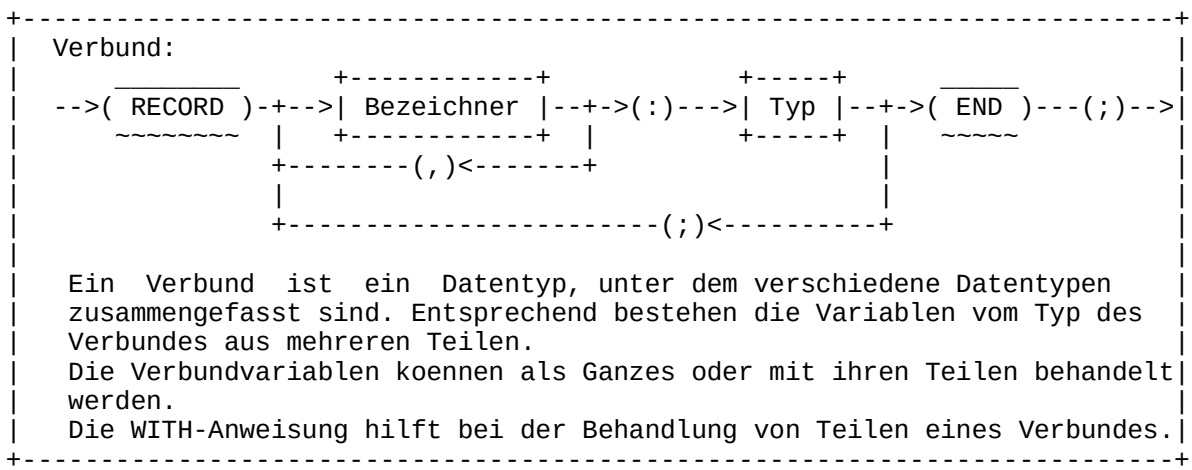
Pro Verbund darf ein variabler Teil (mit CASE) enthalten sein, der am Schluss des Verbundes stehen muss.

Achtung: Klammern (um die Eintragliste) in der CASE-Anweisung beachten !

Variabler Teil:

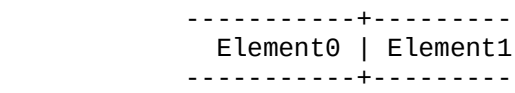


Das Syntaxdiagramm der Variante zeigt, dass das Variantenkennzeichen auch entfallen kann und nur der Typ angegeben werden kann (sog. "free unions").



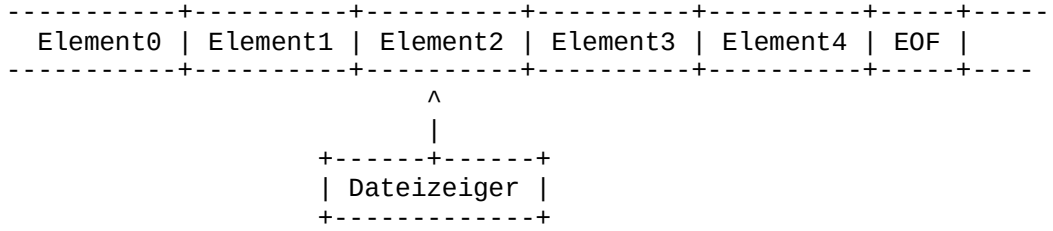
8.1 Dateien

Zur Datenverarbeitung gehoert nicht nur das Verarbeiten, sondern auch das Speichern von Daten.
Kurzfristig werden die Daten im Rechner gespeichert (im RAM, Random Access Memory). Hier sind die Daten jedoch fluechtig, d.h. wenn wir den Rechner ausschalten, sind die Daten geloescht.
Nun ist es sicher sinnvoll, die zu verarbeitenden oder verarbeiteten Daten auf externen Speichermedien zu speichern, die nicht fluechtig sind.
Ein Speicher ist schon der Drucker. Dieser kann aber nur Daten aufnehmen und nicht wieder zurueck in den Rechner geben.
Als schnelles Speichermedium, das zugleich eine grosse Speicherkapazitaet besitzt, verwenden wir die Disketten. Auf ihnen koennen wir nicht nur Programme, sondern auch Daten speichern. Dieser Datenspeicherung liegt in Pascal ein spezielles Konzept zugrunde, das, wenn es richtig verstanden ist, eine recht komfortable Datenspeicherung zulaesst. Zunaechst werden wir uns mit diesem Konzept beschaeftigen und dann die Moeglichkeiten der Dateibefehle kennenlernen.
Es gibt generell zwei Arten von Dateien (files im Englischen): Dateien mit wahlfreiem Zugriff (random access files) und Dateien mit sequentillem Zugriff (sequential access files).
"Random access files": Diese Dateien zeichnen sich dadurch aus, dass man auf jedes Element der Datei direkt zugreifen kann. Bespielsweise ist der Rechnerspeicher so organisiert. Jede Speicherzelle (Byte) ist mit einer Nummer versehen (Adresse), durch die sie direkt beschrieben oder gelesen werden kann. In einigen Betriebssystemen (z.B. DOS3.3) kennt man diese Art von Dateien ebenfalls.
"Sequential access files": Alle Daten sind hintereinander angeordnet, und es kann nur in dieser Reihenfolge auf sie zugegriffen werden. Diese Art der Dateien ist noetig, wenn Magnetbaender als Speichermedien verwendet werden.
In Pascal sind nur sequentielle Dateien moglich. Der Zugriff auf die Daten auf einer Diskette kann jedoch von seiten des Beriebssystems des verwendeten Rechners durch wahlfreien Zugriff geschehen. So sind sequentielle Dateien in Verbindung mit Diskettenlaufwerken nicht langsamer als andere.
Den Aufbau einer Datei in Pascal koennen wir uns so vorstellen: Alle Datenelemente werden in der Reihenfolge ihrer Eingabe in die Datei auf einem Band gespeichert (Band nur symbolisch). Dadurch hat jedes Element automatisch eine Nummer, naemlich seine Platznummer auf dem Band.



Die Nummern, auf die eine spaeter zu besprechende Prozedur zugreift, beginnen mit 0.
Das Ende der Datei wird durch ein Element namens EOF (End-Of-File, Ende der Datei) gekennzeichnet.
Eine leere Datei besteht wenigstens aus dem Element EOF.
Die Datenelemente der Datei haben alle den gleichen Typ. Dieser Typ der Datenelemente wird im Deklarationsteil in der Dateideklaration vereinbart.

Das Element EOF ist unabhaengig von dieser Typdeklaration.
Soll nun auf ein Element der Datei zugegriffen werden, so schafft man sich ein "Fenster" in die Datei. Mit der Typdeklaration der Datei wird automatisch eine Variable mit deklariert, der sogenannte Dateizeiger. Diese Variable zeigt auf ein Datenelement der Datei und erlaubt den Zugriff auf dieses Element.
Mit der Prozedur SEEK wird der Dateizeiger auf ein bestimmtes Element positioniert.



Dieser Dateizeiger zeigt also auf ein bestimmtes Element und stellt sozusagen eine Verbindung mit dem Element her.
Durch die READ-Anweisung wird ein Datenelement aus der Datei gelesen. Durch die WRITE-Anweisung wird ein Datenelement in die Datei geschrieben. Ausserdem kann abgefragt werden, ob der Dateizeiger auf EOF, d.h. auf das Ende der Datei zeigt.
Die Daten, die in einer Datei gespeichert werden, koennen von allen Typen sein. So sind Dateien vom Typ INTEGER, REAL, CHAR, BOOLEAN sowie der Aufzaehlungstypen oder der zusammengesetzten Typen moeglich. Als spezielle Dateien sind Binaerdateien zu nennen, die Daten vom Typ BOOLEAN beinhalten. Dazu gehoeren auch Codefiles von Programmen oder Bildschirmgrafiken.
Eine weitere Form spezieller Dateien sind die sogenannten Textfiles. Darunter fallen z.B. die Progammtexte, aber auch selbst erstellte Texte. Ein sehr starkes Werkzeug liegt darin, dass der Benutzer vom Programm aus auch auf Programmtexte zugreifen kann.
Spezielle Textdateien sind die Standarddateien zur Ein- und Ausgabe, wie z.B. fuer die Tastatur und den Bildschirm.
Wir wollen zur Uebung der neuen Datenstruktur Datei zwei kleine Programme schreiben:

1. Ein Programm zum Erstellen einer neuen Datei von zehn ganzen Zahlen.

2. Ein Programm zum Lesen dieser (dann schon bestehenden) Datei von zehn ganzen Zahlen.

Zuerst brauchen wir eine Variable fuer die Datei, nennen wir sie "f".

```
VAR f: FILE OF INTEGER;
```

Mit dieser Variablendeklaration haben wir f als Name fuer eine Datei von ganzen Zahlen deklariert.

Weiterhin sei eine Variable i vom Typ INTEGER zu deklarieren.

WRITE (f,i); schreibt das Datenelement i in die Datei an die aktuelle Stelle des Dateizeigers und rueckt den Dateizeiger eine Stelle weiter.

READ (f,i); liest ein Datenelement aus der Datei an der Stelle des Dateizeigers; weist es der Variablen i zu und rueckt den

Dateizeiger eine Stelle weiter.

Bevor wir aber ueberhaupt auf Elemente der Datei zugreifen koennen, muss die Datei eroeffnet werden, d.h. der Rechner muss seine Datenleitungen zu der Stelle verknuepfen, wo die Datei steht, bzw. wo die neue Datei stehen soll.

ASSIGN (f, filename); verknuepft unsere (logische) Datei f mit einer tatsaechlichen (physischen) Datei mit dem angegebenen Namen filename auf der Diskette. Der filename ist ein STRING (Konstante oder Variable), der den Dateinamen enthaelt.
Beispiel: ASSIGN (f,'B:TEST.DAT');

REWRITE (f); setzt den Dateizeiger einer neuen Datei auf das 0. Element und eroeffnet damit die Datei. Existiert die Datei schon, so wird sie ueberschrieben.

RESET (f); setzt den Dateizeiger einer bereits existierenden Datei auf das 0. Element und eroeffnet damit die Datei. Existiert die Datei noch nicht, so tritt eine Fehlermeldung auf.

Achtung: in unserem Beispiel ist "f" der Name der Datei im Programm und "filename" der Name der Datei auf der Diskette!

Wenn wir mit unseren Dateioperationen fertig sind (spaetestens jedoch zum Programmende),muss die Datei noch geschlossen werden mit

```
CLOSE (f);
```

Damit sind alle Aenderungen in der Datei permanent gespeichert, und die Datei wird in das Inhaltsverzeichnis der Diskette uebernommen.
Nun kommen wir zu unserem Programm:

```
PROGRAM Schreibe;
   VAR i : INTEGER;
       f : FILE OF INTEGER;

   BEGIN
      ASSIGN (f,'A:Daten.DAT');
      REWRITE (f);         (* neue Datei eroeffnen *)
      FOR i:=1 TO 10 DO WRITE (f,i);
      CLOSE (f)            (* Datei permanent speichern *)
   END.
```

Aufgabe: Tippen Sie das Programm ein, und fuehren Sie es aus!

Sehen wir uns mit dem DIR-Befehl das Inhaltsverzeichnis der Diskette an, so sehen wir,dass die neue Datei namens Daten.DAT gespeichert ist.
In unserer Datei sollten die Zahlen 1 bis 10 gespeichert sein.

Achtung: Die Zahlen sind in den Elementen 0 bis 9 gespeichert. Element 10 enthaelt EOF (End Of File).

So, nun lesen wir die Datei, um uns davon zu ueberzeugen, dass die Daten tatsaechlich gespeichert sind:

```
PROGRAM Lese;
```

```
VAR f : FILE OF INTEGER;
    i : INTEGER;

BEGIN
   ASSIGN (f,'A:Daten.DAT');
   RESET (f);                  (* alte Datei eroeffnen *)
   WHILE NOT EOF(f) DO BEGIN  (* solange nicht Dateiende unter
                               Zeiger *)
     READ (f,i);              (* Datenelement lesen *)
   WRITELN (i)                (* ausgeben *)
   END;   (* while *)
   CLOSE (f)
END.
```

Aufgabe: Tippen Sie das Programm ein, und fuehren Sie es aus!

Die Funktion EOF (f);

ergibt den Wert TRUE, wenn der Datenanzeiger auf des Ende der Datei f zeigt (keine Ausgabe dieses "Wertes" mit WRITE(LN) moeglich!) sonst FALSE. Die WHILE-Schleife verhindert, dass versucht wird, nach dem Ende der Datei etwas auszugeben.
Eine Besonderheit der RESET-Anweisung besteht darin, dass sie auch in einer schon eroeffneten Datei verwendet werden kann. Mit

RESET (f);

wird der Datenanzeiger der (schon eroeffneten) Datei f auf das Element 0 zurueckgesetzt.

Etwas mehr Komfort

Bei unserem kleinen Testprogramm faellt unangenehm auf, dass der Name der Datei nicht variabel gehalten ist. Die koennen wir leicht aendern, indem wir eine Variable "filename" vom Typ STRING definieren, die wir den Benutzer eingeben lassen.
Weiterhin muessten wir die Eroeffnung der Datei mit RESET oder REWRITE davon abhaengig machen, ob die Datei schon besteht oder neu angelegt werden soll. Wird naemlich eine schon bestehende Datei mit REWRITE eroeffnet, so ist der alte Dateninhalt verloren. Hier hilft uns die Moeglichkeit, die Fehlermeldung abzuschalten und softwaremaessig Fehler festzustellen.

Die Compileranweisung

(*$I-*) oder {$I-} Achtung: Kein Kommentar!

schaltet die Fehlermeldung des Turbo Pascal-Systems ab und

(*$I+*) oder {I+}

schaltet sie wieder ein.

Die Fehlermeldung sollte aber nur abgeschaltet werden, wenn der Programmierer sicher ist, dass nur ein ganz bestimmter (gewollter) Fehler und sonst keiner auftritt. Der Fehler, den wir erwarten, ist, dass sich die Datei nicht mit RESET eroeffnen laesst, weil sie noch nicht existiert.
Das System bricht bei Auftritt eines Fehlers nicht das Programm ab, gibt

aber einer vordefinierten Funktion

```
IORESULT
```

einen Wert ungleich Null mit.

Achtung: Nachdem diese Funktion abgefragt wurde, wird der Wert wieder auf Null gsetzt.
Der Wert von IORESULT beschreibt den aufgetretenen Fehler (siehe Anhang).

Die Programmsequenz koennte sein:

```
WRITE ('Name der Datei: ');
READLN (filename);
ASSIGN (f,filename);
(*$I-*)                      (* Fehler aus *)
RESET (f);                   (* Versuch: Alte Datei *)
(*$I+*)                      (* Fehler an *)
IF IORESULT<>0 THEN BEGIN    (* wenn Fehler... *)
  WRITELN ('Neue Datei eroeffnen...');
  REWRITE (f)
END; (* von if *)
```

Eine weitere Anweisung im Zusammenhang der Datei ist

```
SEEK (f,Nummer);
```

Hierbei ist f der Variablenname der Datei und "Nummer" einer ganzen Zahl groesse oder gleich Null.
Mit SEEK wird der Datenzeiger auf ein bestimmtes Element mit der Nummer "Nummer" gesetzt. Dabei wird aber keine WRITE- oder READ-Anweisung ausgefuehrt.
Mit dieser neuen Anweisung sind wir in der Lage, eine sequentielle Datei im "random-access-Verfahren" (d.h. mit wahlfreiem Zugriff) zu behandeln.

Eine Anwendung

Wir wollen nun unser Wissen ueber Dateien und Verbunde zusammenfassen in einem Programm zum "klassischen" Problem einer Adressdatei. Aus Platzgruenden wird hier keine allzu komfortable Version des Programms dargestellt. Es sei dem Leser ueberlassen, das Programm zu erweitern.
Bedienen wir uns der Menue-Technik, um den Programmablauf zu steuern, und geben wir dem Benutzer folgende Moeglichkeiten:

```
+-----------------------------------------------------------------+
|  V (eraendern von Daten                                         |
|  L (oeschen von Daten                                           |
|  A (lphabetisch Ordnen                                          |
|  D (rucken der Datei                                            |
|  S (uchen nach Kriterien                                        |
|  Z (um Schluss                                                  |
+-----------------------------------------------------------------+
```

Menue bedeutet, dass der Benutzer jedezeit ein Programmteil durch einfaches Tippen des ersten Buchstabens der Zeile auswaehlen kann.
Zu Beginn muss der Benutzer nach dem Dateinamen gefragt werden. Sollte die Datei nicht existieren, wird gefragt, ob eine neue Datei erstellt werden

soll. Andernfalls wird das Programm beendet.

Wir erstellen zunaechst das "Rohprogramm" mit der Deklaration der globalen Variable, den noch leeren Prozeduren und dem Hauptprogramm.
Die Funktion Lieszeichen ist schon erstellt. Sie ermoeglicht eine recht komfortabel Eingabe von Zeichen. Die Funktion EOLN(KEYBOARD) darin erhaelt den Wert TRUE, wenn die Return-Taste gedrueckt wird.
Ausserdem ist die Prozedur Start fertig, die den Datenanmen abfragt, die Datei eroeffnet oder eine neue Datei erstellt (erst als Leerdatei, um sicher zu gehen, dass der Diskettenplatz ausreicht) mit der Prozedur Neu und die Prozedur Menue aufruft.
Die Prozedur Menue, ebenfalls schon fertig, ist die "zentale Schaltstelle" des Programms. Von hier aus werden die anderen Prozedure aufgerufen und das Programm kehrt nach dem Durchlaufen der Prozeduren hierher zurueck. Wenn in der Prozedur Menue das Programmende gewaehlt wird, so wird zuerst die Prozedur Start beendet, die die Datei schliesst, bevor sie zum Hauptprogramm zurueckkehrt.
Zur vorzeitigen Beendigung der Prozedur Start wird ein Label namens Exit1 definiert und an das Ende der Prozedur gesetzt. Sollte der Benutzer einen falschen Dateinamen eingegeben haben, aber nicht gewillt sein, eine neue Datei zu eroeffnen (weil er z.B. den korrekten Dateinamen nur vergaessen hat), so kann er hier das Programm verlassen, um sich das Inhaltsverzeichnis der Diskette erneut anzusehen.
Die Mengen J.N und JN werden verwendet, um die haeufige Abfrage von Ja und Nein zu erleichtern.

```
PROGRAM Adressdatei;
 CONST Maxn = 40;

 TYPE Setofchar  = SET OF CHAR;
      Kurzstring = STRING [20];
      Adresse    = RECORD
                     Test : BOOLEAN;
                     Name, Vorname, Strasse : Kurzstring;
                     PLZ : INTEGER;
                     Ort, Tel :Kurzstring
                   END; (* Adresse *)
      Filetyp    = FILE OF Adresse;

 VAR Person      : ARRAY [1..Maxn] OF Adresse;
     Leer        : Kurzstring;
     Datei       : Filetyp;
     Dateiname   : STRING [15];
     ch          : CHAR;
     Anzahl, Nr, i, k : INTEGER
     J, JN, N    : SET OF CHAR;

FUNCTION Lieszeichen (m : Setofchar) : CHAR;
VAR ch : CHAR;
    OK : BOOLEAN;
BEGIN
  REPEAT
    READ (KBD, ch);                      (* Lies Zeichen ohne Echo *)
    IF EOLN (KBD) THEN ch:=CHR(13);      (* <CR>-Taste *)
      OK := ch IN m;
    IF NOT OK THEN WRITE (CHR(7))            (* Bell *)
              ELSE IF ch IN [''..CHR(126)]   (* druckbare Zeichen *)
                      THEN WRITE (ch)
```

```
  UNTIL OK;
 Lieszeichen := ch
END;   (* Lieszeichen *)

PROCEDURE Lesen;
 BEGIN
  (* folgt noch *)
 END;

PROCEDURE Schreiben;
 BEGIN
  (* folgt noch *)
 END;

PROCEDURE Veraendern;
 BEGIN
  (* folgt noch *)
 END;

PROCEDURE Loeschen;
 BEGIN
  (* folgt noch *)
 END;

PROCEDURE Alphasort;
 BEGIN
  (* folgt noch *)
 END;

PROCEDURE Drucken;
 BEGIN
  (* Uebung *)
 END;

PROCEDURE Suchen;
 BEGIN
  (* Uebung *)
 END;

PROCEDURE Menue;
VAR Frage : CHAR;
BEGIN
 REPEAT
  clrscr;
  WRITELN; WRITELN;
  WRITELN ('Die Datei hat', Anzahl:3);
  WRITELN ('tatsaechliche Elemente.');
  WRITELN; WRITELN ('Waehlen Sie:');
  WRITELN; WRITELN;
  WRITELN (' V (eraendern von Daten   ');
  WRITELN (' L (oeschen von Daten     ');
  WRITELN (' A (lphabetisch Ordnen    ');
  WRITELN (' D (rucken der Datei      ');
  WRITELN (' S (uchen nach Kriterien  ');
  WRITELN;
  WRITELN (' Z (um Schluss            ');
  Frage:=Lieszeichen(['V','v','L','l','A','a','D','d','S','s','Z','z']);
```

```
  CASE Frage OF
   'V','v' : Veraendern;
   'L','l' : Loeschen;
   'A','a' : Alphasort;
   'D','d' : Drucken;
   'S','s' : Suchen
  END (* case *)
 UNTIL Frage IN ['Z','z']
END;  (* von Menue *)

 PROCEDURE Start;
 LABEL Exit1;
 VAR Satz : Adresse;

 PROCEDURE Neu;
  BEGIN   (* von Neu *)
   Leer:='               ';     (* 20mal Leer *)
   ASSIGN(Datei,Dateiname);
   REWRITE (Datei);
   FOR i:=1 TO Maxn+1 DO BEGIN
    WITH Person [i] DO BEGIN    (* leere Datei erzeugen *)
     Test := FALSE; Name := Leer; Vorname := Leer;
     Ort := Leer; PLZ := 0; Strasse := Leer; Tel := Leer
    END; (* With *)
    WRITE(Datei,Person [i]);
   END; (* for *)
   CLOSE (Datei)
  END; (* von Neu *)

 BEGIN   (* von Start *)
  Anzahl := 0;  (* noch keine Datei *)
  WRITELN; WRITELN;
  WRITE ('Eingabe des Dateinamens: '); READLN (Dateiname);
  ASSIGN(Datei, Dateiname);
  (*$I-*) RESET(Datei); (*$I+*)
  IF IORESULT <> 0 THEN BEGIN
   WRITELN ('Datei existiert nicht! ');
   WRITE ('Wollen Sie eine neue Datei (J/N)?');
   ch := Lieszeichen (JN);
   IF ch IN J THEN Neu ELSE GOTO Exit1
  END; (* von if *)
  RESET(Datei);
  FOR i:=1 TO Maxn DO BEGIN
   SEEK(Datei,i);
   READ(Datei,Satz);
   IF Satz.Test THEN Anzahl := Anzahl +1
  END; (* von for *)
  Menue;
  CLOSE (Datei);
  Exit1:
 END; (* von Start *)

BEGIN (* Hauptprogramm *)
J  := ['J','j'];
N  := ['N','n'];
JN := ['J','j','N','n'];
CLRSCR;
Start;
```

```
     CLRSCR;
     WRITELN ('Das war es!')
     END.
```

Die Prozeduren Drucken und Suchen seien dem Leser als Uebung empfohlen. Die restlichen Prozeduren werden hier noch entwickelt. Das Rohprogramm ist allerdings schon in seiner bisherigen Form lauffaehig. Daher kann es schon getestet werden.
Diese Art der Programmierung nennt man "top-down-Methode". Dabei wird erst das grobe Geruest des Programms erstellt und diese nach und nach mit Modulen (Prozeduren und Funktionen) ausgefuellt.
Mit den noch zu entwickelnden Prozeduren Lesen, Schreiben, Veraendern, Sortieren und Loeschen hat das Programm seine wichtigsten Funktionen und kann schon recht komfortabel angewendet werden.
Als erstes folgt nun die Prozedur Lesen. Sie hat zwei Variable, von denen eine zurueckgegeben wird. Die Prozedur hat die Aufgabe, einen Datensatz mit der Nummer N zu lesen und auf dem Bildschirm darzustellen. Sie enthaelt selbst zwei Prozeduren Maske und Fuellen, die eine Bildschirmmaske generieren und dies mit dem Dateninhalt fuellen.

```
    PROCEDURE Lesen (N : INTEGER; VAR Satz : Adresse);
    PROCEDURE Maske;
     BEGIN (* von Maske *)
      GOTOXY(1,2 ); WRITE ('Nummer:     ');
      GOTOXY(1,8 ); WRITE ('Name:       ');
      GOTOXY(1,10); WRITE ('Vorname:    ');
      GOTOXY(1,12); WRITE ('Strasse:    ');
      GOTOXY(1,14); WRITE ('Plz:        ');
      GOTOXY(1,16); WRITE ('Wohnort:    ');
      GOTOXY(1,18); WRITE ('Telefon:    ')
     END; (* von Maske *)

    PROCEDURE Fuellen;
     BEGIN (* von Fuellen *)
      WITH Satz DO BEGIN
       GOTOXY(13,2 ); WRITE (N : 3);
       GOTOXY(13,8 ); WRITE (Name);
       GOTOXY(13,10); WRITE (Vorname);
       GOTOXY(13,12); WRITE (Strasse);
       GOTOXY(13,14); WRITE (PLZ);
       GOTOXY(13,16); WRITE (Ort);
       GOTOXY(13,18); WRITE (Tel)
      END (* von with *)
     END; (* von Fuellen *)

     BEGIN (* VON LESEN *)
      CLRSCR;
      SEEK (DATEI, N);
      READ(Datei,Satz);
      Maske;
      Fuellen
     END; (* von Lesen *)
```

Als naechstes die Prozedur Schreiben. Auch sie hat zwei Parameter. Sie macht nichts anderes, als einen Datensatz mit der Nummer N und dem Inhalt Satz auf die Diskette zu schreiben.

```
     PROCEDURE Schreiben (N : INTEGER; Satz : Adresse);
```

```
BEGIN
 SEEK (Datei,N);
 WRITE(Datei,Satz);
END; (* von Schreiben *)
```

Die Prozedur Veraendern dient dazu, Datensaetze zu aendern. Dies bedeutet zweierlei. Es koennen schon vorhandene Daten umgeaendert werden oder mit weiterfortlaufender Nummer neue Datensaetze eingegeben werden. Ein entsprechendes Untermenue erlaubt die Auswahl.
Diese Pozedur enthaelt die Pozedur Aendern, die eine Aenderung der Daten selbst uebernimmt. Dazu wird der Datensatz zuerst gelesen. Dann wird mit der Prozedur Tasten, die ihrerseits Teil von Aendern ist, in die Bildschirmmaske gegangen und geaendert. Die Prozedur Tasten erlaubt auch einfaches Druecken der Return-Taste. Dann werden die alten Daten einfach uebernommen.
Am Ende der Aenderung kann sich der Benutzer noch entscheiden, ob er tatsaechlich die geaenderten Daten speichern will. Tut er dies nicht, so wird der Datensatz nicht gueltig, und bei der Neueingabe wird die aktuelle Anzahl der Daten wieder zurueckgesetzt.

```
PROCEDURE Veraendern;
 VAR Frage : CHAR;

 PROCEDURE Aendern (VAR N : INTEGER);
  VAR Satz : Adresse;
      Hilf : Kurzstring;

  PROCEDURE Tasten (x,y : INTEGER; VAR Wort : Kurzstring);
   VAR Wort2 : Kurzstring;
   BEGIN (* von Tasten *)
     GOTOXY (x,y); READLN (Wort2);
     IF Wort2 <> '' THEN Wort : Wort2;
      (* nur Aenderung, wenn nicht <CR> *)
     GOTOXY (x,y); WRITE (Wort)
   END; (* von Tasten *)

  BEGIN (* von Aendern *)
   Lesen (N, Satz);
   WITH Satz DO BEGIN
    Tasten (13,8,Name); Tasten (13,10,Vorname); Tasten (13,12,Strasse);
    STR (PLZ, Hilf);  (* PLZ voruebergehend als String *)
    Tasten (13,14,Hilf);
    Hilf := CONCAT (Hilf,'0000');   (* falls leer *)
     PLZ := 0; k:=1000;
    FOR i:=1 TO 4 DO BEGIN (* Umwandeln in Integer *)
     PLZ := PLZ + k * (ORD (Hilf[i]) - 48); k := k DIV 10
    END; (* von for *)
     Tasten (13,16,Ort); Tasten (13,16,Tel)
    END; (* von with *)
    GOTOXY(1,21);
    WRITE ('Wollen Sie abspeichern (J/N) ?');
    ch := Lieszeichen (JN);
    IF ch IN J THEN BEGIN
     Satz.Test := TRUE;
     Schreibe (N, Satz);
    END (* von if *)
    ELSE N:=N-1
   END; (* von Aendern *)
```

```
   BEGIN (* von Veraendern *)
    REPEAT
     CLRSCR;
     WRITELN; WRITELN;
     WRITELN ('  N (euneingabe');
     WRITELN ('  U (maendern  ');
     WRITELN ('  M (enue      ');
     WRITELN; WRITELN ('Bitte waehlen Sie');
     Frage := Lieszeichen (['N','n','U','u','M','m']);
     IF Frage IN ['N','n'] THEN BEGIN (* 1. Element oder naechstes *)
      Nr := Anzahl+1;
      Aendern (Nr);
      Anzahl := Nr
     END; (* von if *)
     IF Frage IN ['U','u'] THEN BEGIN
      WRITELN;
      WRITE ('Welche Nr. aendern? ');
      READLN (Nr);
      Aendern (Nr)
     END (* von if *)
    UNTIL Frage IN ['M','m']
   END; (* von Veraendern *)
```

Weiterhin haben wir noch die Prozedur Loeschen, die erlaubt, einen Datensatz zu loeschen. Dazu wird dem Benutzer angeboten, die Nummer des Datensatzes oder den Namen der zu loeschenden Person anzugeben.Zur Sicherheit wird der Datensatz noch einmal angezeigt und gefragt, ob wirklich geloescht werden soll. Soll noch nicht geloescht werden, so kann man an das Ende der Prozedur springen und so die Prozedur vorzeitig verlassen.

```
PROCEDURE Loeschen;
LABEL Exit;
VAR Nnam, Vnam : Kurzstring;
    Ende : BOOLEAN
    Satz : Adresse;

PROCEDURE Weg (N : INTEGER);
 Var Satz : Adresse;
 BEGIN (* von Weg *)
  Lesen (N, Satz);
   WRITELN;
  WRITELN ('Ist das die Person (J/N) ?');
   ch := Lieszeichen (JN);
   IF ch IN J THEN BEGIN
    WRITELN ('Loeschen...');
   FOR k:=N TO Anzahl-1 DO BEGIN (* Umspeichern *)
    SEEK (Datei,k+1); READ(Datei,Satz);
    SEEK (Datei,k); WRITE(Datei,Satz)
   END; (* von for *)
    Anzahl := Anzahl - 1
  END (* von if *)
 END; (* von Weg *)

BEGIN (* von Loeschen *)
 CLRSCR;
 WRITELN (' Was wollen Sie loeschen?');
 WRITELN ('  N (ummer  ');
```

```
     WRITELN ('  P (erson  ');
     WRITELN ('  E (nde    ');
     ch := Lieszeichen (['N','n','P','p','E','e']);
     WRITELN;
     IF ch IN ['E','e'] THEN GOTO Exit;
     IF ch IN ['P','p'] THEN BEGIN
      WRITE ('Vorname: '); READLN (Vnam);
      WRITE ('Name   : '); READLN (Nnam);
      i:=1; Ende:=FALSE;

      REPEAT
       SEEK (Datei,i); READ(Datei,Satz);
       IF (Satz.Name = Nnam) AND (Satz.Vorname = Vnam) THEN BEGIN
        Weg (i);
        Ende := TRUE
       END; (* von if *)
       IF (i>Anzahl) AND (NOT Ende) THEN BEGIN
        WRITELN ('Person nicht vorhanden');
        Ende := TRUE
       END; (* von if *)
       i := i + 1
      UNTIL Ende
     END (* von if *)
      ELSE BEGIN
       WRITE ('Welche Nummer loeschen? ');
       READLN (i);
       Weg (i)
      END; (* von ELSE *)
      Exit:
     END; (* von Loeschen *)
```

Zum Schluss betrachten wir noch die Prozedur Alphasort zum alphabetischen Sortieren des Datensatzes nach Nachnamen. Es wird ein einfacher Bubblesort-Sortieralgorithmus gewaehlt.

```
    PROCEDURE Alphasort;
     VAR i,j : INTEGER;
         Buf : Adresse;
     BEGIN
      RESET(Datei);
      SEEK(Datei,1);
      FOR i:=1 TO Anzahl DO READ(Datei,Person[i]);
      FOR i:=1 TO Anzahl-1 DO
       FOR Aj:=Anzahl DOWNTO i DO
        IF Person[i].Name > Person[j].Name THEN BEGIN
         Buf:=Person[i]; Person[i]:=Person[j]; Person[j]:=Buf END;
      SEEK(Datei,1);
      FOR i:=1 TO Anzahl DO WRITE(Datei,Person[i]);
     END; (* von Alphasort *)
```

Fuer ein komfortables Arbeiten mit Dateien stellt Turbo Pascal noch folgende Standardprozeduren und -funktionen bereit:

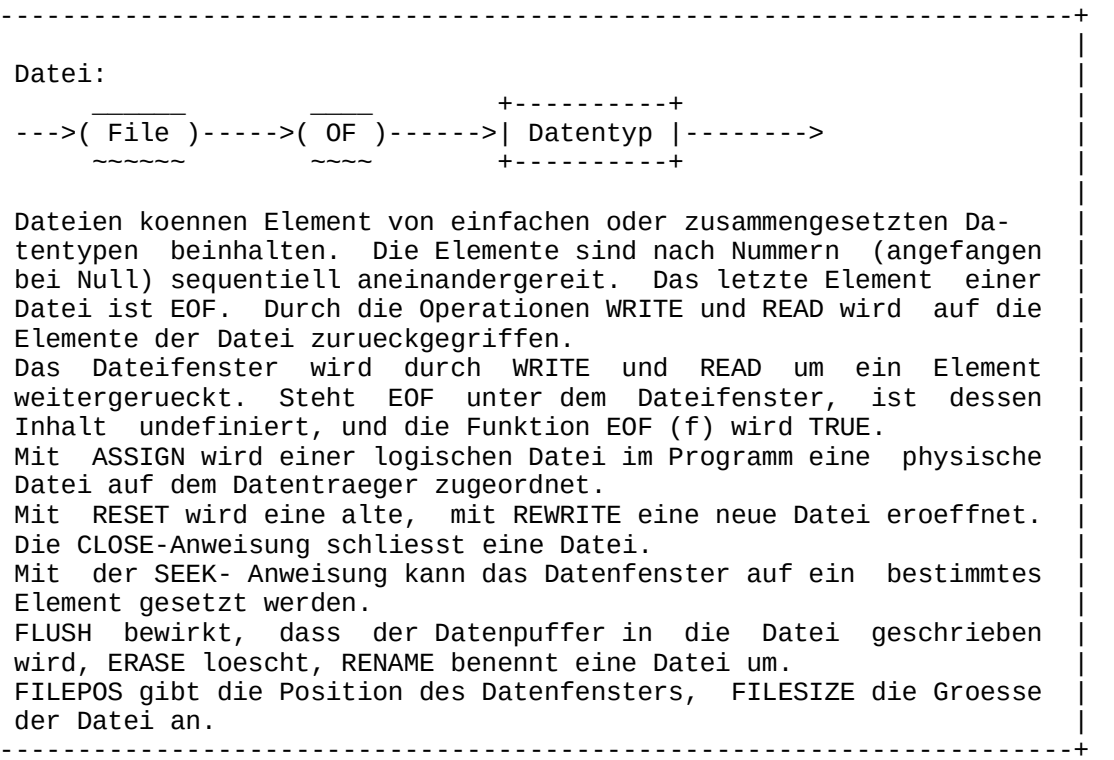
Prozeduren:

FLUSH (f); sorgt dafuer, dass der Datenpuffer in die Datei geschrieben wird. Sonst wird nicht bei jeder Dateioperation das entsprechende Datenelement gelesen/geschrieben. Erst werden

mehrere solcher Operationen in einem Puffer (Zwischenspeicher) verwaltet.

ERASE (f); loescht eine (vorher geschlossene) Datei.

RENAME (f,Name); benennt eine (vorher geschlossene) Datei f um. Der STRING Name wird der neue Name der Datei auf der Diskette. Diese Prozedur hebt folglich die Wirkung von ASSIGN auf.



Funktionen:

i:=FILEPOS(f); Das Ergebnis vom Typ INTEGER dieser Funktion enthaelt die aktuelle Position des Datenzeigers.

i:=FILESIZE(f); ergibt die Anzahl der Datensaetze der Datei f. Ergebnis vom Typ INTEGER.

8.2 Textdateien

Eine besondere Form der Dateien sind die Textdateien. Der Datentyp einer Textdatei wird mit dem reservierten Wort

```
TEXT
```

erklaert und steht stellvertretend fuer

```
FILE OF CHAR
```

In der Variablendeklaration koennen wir z.B. eine Variable erklaeren:

```
VAR t : TEXT;
```

Neben den Operationen READ und WRITE gibt es bei Textdateien noch die Operationen READLN und WRITELN.
Zuerst wird der Datei t wie jeder anderen Datei mit ASSIGN ein Dateiname zugewiesen. Dann wird eine neue Datei mit REWRITE und eine bestehende Datei mit RESET eroeffnet. Alsdann gibt es die Anweisungen:

```
WRITE(t,<text>);   und
WRITELN(t,<text>);
```

um Text ohne oder mit Zeilenvorschub in die Datei zu schreiben, und

```
READ(t,<zeichen>);
READLN(t,<text>);
```

um ein Zeichen oder eine Textzeile aus der Datei zu lesen.
Weiterhin gibt es die Funktion

```
EOF(t)     (End Of File)
```

die TRUE ergibt, wenn das Ende der Datei erreicht ist, und

```
EOLN(t)    (End Of LiNe)
```

die TRUE ergibt, wenn das Ende der Zeile erreicht ist (d.h. wenn das naechste zu lesende Zeichen ein Zeilenvorschub,<CR>, ist).
Die Prozeduren/Funktionen SEEK, FLUSH, FILEPOS und FILESIZE duerfen nicht auf Textdateien angewendet werden!
Wir wollen nun ein Druckersteuerungsprogramm schreiben, das eine Textdatei liest, nach vereinbarten Sonderzeichen absucht und den Text dann auf dem Drucker ausgibt. Dazu vereinbaren wir:
Ist das erste Zeichen der Zeile ein Punkt, so folgt genau ein Druckersteuerzeichen. Die Steuerzeichen sollen sein:

```
F: Seitenvorschub (formfeed)
D: Deutscher Zeichensatz
A: ASCII-Zeichensatz
e: Schriftart Elite
p: Schriftart Pika
d: Doppeldruck
n: Normaldruck
I: Schriftart Italic an
i: Schriftart Italic aus
```

Mit diesem Steuerprogramm sind wir z.B. in der Lage, deutsche Umlaute und eckige Klammern in ein und demselben Text zu verarbeiten. Wir wissen ja, dass der ASCII-Zeichensatz keine Umlaute kennt. Drucker, die auf den deutschen Zeichensatz umschaltbar sind, ersetzen dann einige Sonderzeichen (z.B. Die eckigen Klammern) durch Umlaute, da die Anzahl der Zeichen begrenzt ist. Ohne dieses Steuerprogramm koennten wir also nicht deutschen Text mit Pascal-Programmen mischen, die mit eckigen oder geschweiften Klammern versehen sind.

```
PROGRAM drucktext;
```

```
CONST esc = $1B;
VAR   name:STRING[20];
         s:STRING[80];
        ch:CHAR;
         t:TEXT;
        Ok:BOOLEAN;
 BEGIN
  WRITE(LST,chr(2));
  CLRSCR;
  REPEAT
   WRITE('Zu druckender Text: ');
   READLN (name);
   ASSIGN (t,name);
   (*$I-*) RESET(t); (*$I+*)
   Ok:=IORESULT=0;
   IF NOT Ok THEN WRITELN (^G,'Datei nicht vorhanden!');
  UNTIL Ok;
  WHILE NOT EOF(t) DO BEGIN
   (* lies zeilenweise den Text *)
   READLN(t,s);
   IF POS('.',s)=1 THEN CASE s[2] OF  (* suche Steuerzeichen *)
                         'F':WRITE(LST,CHR(12));        (*FF*)
                         'D':WRITE(LST,esc,'R',CHR(2)); (*Deutsch*)
                         'A':WRITE(LST,esc,'R',CHR(0)); (*Ascii*)
                         'e':WRITE(LST,esc,'M');        (*Elite*)
                         'p':WRITE(LST,esc,'P');        (*Pika*)
                         'd':WRITE(LST,esc,'G');        (*Doppel*)
                         'n':WRITE(LST,esc,'H');        (*normal*)
                         'I':WRITE(LST,esc,'4');        (*Italic*)
                         'i':WRITE(LST,esc,'5');        (*Italic aus*)
                        END
                  ELSE WRITELN(LST,s)
  END;
  CLOSE(t)
 END.
```

Die Befehlscodierungen fuer den Drucker beziehen sich auf einen EPSON-FX 80-Drucker. Fuer andere Drucker koennen diese Zeichen natuerlich anders lauten (siehe Handbuch fuer den Drucker).
Die Abfrage nur des ersten Zeichens in der Zeile hat den Vorteil, dass das Programm enorm schnell ist.
Ebenfalls denkbar mit dieser Methode ist ein Programm, das bei Auftauchen eines bestimmten Zeichens (z.B. &) im Text das darauffolgende Zeichen als Sonderzeichen interpretiert. Hierbei muesste allerdings jedes Zeichen einer Zeile abgefragt werden.

Standarddateien

Eine besondere Form der Textdateien sind die sogenannten Standarddateien. Diese brauchen und duerfen vom Benutzer nicht mit ASSIGN, RESET, REWRITE und CLOSE behandelt werden. Sie sind bestimmten logischen Geraeten zugeordnet.

INPUT : Eingabedatei, durch die standardmaessig (d.h. wenn nichts anderes genannt ist) die Eingabe geschieht. Sie ist dem Geraet CON: (Console) zugeordnet. Mit der Compiler-Option (*$B-*) kann sie TRM: (Terminal) zugeordnet werden.

OUTPUT : Ausgabedatei, ueber die standardmaessig (d.h. wenn nichts anderes genannt ist) die Ausgabe erfolgt. Sie ist dem Geraet CON: (Console) zugeordnet. Mit der Compiler-Option (*$B-*) kann sie TRM: (Terminal) zugeordnet werden.

CON : Ein- und Ausgabedatei, die dem Geraet CON: (Console) zugeordnet ist. Das ist normalerweise der Bildschirm und die Tastatur des Rechners. Eingaben ueber die Tastatur bewirken ein zusaetzliches Echo des eingegebenen Zeichens auf dem Bildschirm. Ein Eingabepuffer verwaltet eine Eingabezeile.

TRM : Ein- und Ausgabedatei, die dem Geraet TRM: (Terminal) zugeordnet ist. Wie CON, aber ohne Puffer

KBD : Eingabedatei, die dem Geraet KBD: (Keyboard) zugeordnet ist. Das ist die Tastatur. Eingaben erscheinen nicht auf dem Bildschirm.

LST : Ausgabedatei, die dem Geraet LST: (Lister) zugeordnet ist. Das ist der angeschlossene Drucker.

AUX : Ein- und Ausgabedatei, die dem Geraet AUX: (Auxiliary device) zugeordnet ist. Oft handelt es sich um ein Modem.

USR : Ein- und Ausgabedatei, die dem Geraet USR: (User defined device) zugeordnet ist. Hier kann der Benutzer ein Geraet zur Ein-/Ausgabe selbst definieren. (Siehe Handbuch zu Turbo-Pascal.)

In READ(LN)- und WRITE(LN)-Befehlen ist die Datei, ueber die die Ein- und Ausgabe erfolgen soll, zuerst zu nennen. Wird keine Datei genannt, so erfolgt die Ausgabe ueber OUTPUT und die Eingabe ueber INPUT.
Das bedeutet:

```
WRITELN ('Test'); ist gleich WRITELN (OUTPUT,'Test');
READLN (wort); ist gleich READLN (INPUT,wort);
```

Mit ASSIGN koennen den logischen Geraeten auch andere Variablen fuer Textdateien zugewiesen werden.

```
+------------------------------------------------------------------------+
|  Textdateien:                                                          |
|                                                                        |
|      TEXT                                                              |
|                                                                        |
|  Eine Textdatei ist eine Datei, deren Elemente vom Typ CHAR sind.      |
|  Textdateien werden nicht mit SEEK, FLUSH, FILEPOS und FILESIZE        |
|  behandelt.                                                            |
|  Die Anweisungen ASSIGN, REWRITE und RESET werden mit Textdateien      |
|  benutzt wie mit anderen Dateien. Die Ein- und Ausgaben geschehen mit  |
|  READ, READLN, WRITE und WRITELN.                                      |
|  Die Funktion EOF gibt das Dateiende, die Funktion EOLN das Zeilenende |
|  an. Vordefinierte Textdateien sind: INPUT, OUTPUT, CON, TRM, KBD, LST,|
|  AUX und USR. Vordefinierte Dateinamen sind: CON:, TRM:, KBD:, LST:,   |
|  AUX: und USR:.                                                        |
+------------------------------------------------------------------------+
```

8.3 Typenlose Dateien

Als Anwendung der sogenannten typenlosen Dateien wollen wir ein Kopierprogramm fuer Dateien konzipieren.
Mit den bisher bekannten Dateitypen wuerde es sehr schwer fallen, ein solches Programm zu erstellen. Insbesondere muessten wir wissen, von welchem Datentyp die einzelnen Elemente der Datei sind. Dann wuerden wir so viele Datenelemente lesen, wie wir in den Rechnerspeicher bekommen, und sie sodann in eine neue Datei schreiben.
Bei der Benutzung typenloser Dateien entfaellt die genaue Kenntnis der Typen der einzelnen Datenelemente. Eine solche Datei wird mit dem Wort FILE deklariert:

```
VAR f: FILE:
```

Die Elemente einer solchen Datei sind vom Typ BYTE, d.h. von der Groesse der kleinsten zusammenhaengenden Speichereinheit. Auf der Diskette werden dann 128 Bytes zu einem Block zusammengefasst. Solche 128-Byte-Bloecke lassen sich dann mit zwei Prozeduren lesen und schreiben:

BLOCKREAD (f, puffer, Anzblocks); liest eine Anzahl Bloecke (durch Anzblocks vom Typ INTEGER erklaert) aus der offenen Datei f in eine Variable puffer, die sinnvollerweise gross genug ist (z.B. als ARRAY), um den Dateninhalt der zu lesenden Bloecke aufzunehmen.

BLOCKWRITE (f, puffer, Anzblocks); schreibt eine Anzahl Bloecke aus der Variablen puffer in die offene Datei f.

Weiterhin gibt es auch mit typenlosen Dateien die Standardprozeduren und -funktionen ASSIGN, RESET, REWRITE, ERASE, RENAME, FILEPOS, FILESIZE und EOF.
Allerdings gibt FILESIZE die Groesse der Datei in Bloecken (pro 128 Bytes) wieder. Entsprechend gibt FILEPOS die Position des Dateizeigers in Bloecken wieder.
Nun kommen wir zu unserem Programm. Wir definieren uns einen ARRAY mit 16 KByte Speicherplatz als Puffer zum Einlesen und Schreiben der Daten. Dazu wird eine Konstante Speicher=$4000 definiert. Der hexadezimale Wert $4000 ist gleich 16384, das entspricht genau 16 KByte (1 KByte = 1024 Byte). Hat Ihr Rechner mehr Speicherplatz zur Verfuegung, so kann diese Konstante einfach geaendert werden. Weiterhin geben wir mit einer Konstanten an, wie gross der Block ist.
Nach der Frage nach den jeweiligen Datennamen und der Eroeffnung der Datei wird festgestellt, wie gross die Anzahl der mit einer Speicherfuellung zu kopierenden Bloecke (Anzbloecke) und die Anzahl der von der Datei zu kopiereden Bloecke (Noch_zu_kopieren) ist. Dann werden immer so viele Bloecke kopiert, wie in den Pufferspeicher passen.
Zum Schluss werden die Dateien geschlossen, was insbesondere fuer die neue Datei wichtig ist, damit sie ins Inhaltsverzeichnis der Diskette uebernommen wird.

```
PROGRAM Kopiere_Files;

CONST Speicher = $4000;  {16 KByte}
      Block    = 128;    {Blockgroesse}

VAR   Puffer          : ARRAY [1..Speicher] OF BYTE;
      Original, Kopie : STRING[20];
      Originalfile,
```

```
          Kopiefile       : FILE;
          Anzbloecke, Liesbloecke,
          Noch_zu_kopieren: INTEGER

    BEGIN
     REPEAT
      CLRSCR;
      WRITELN ('Kopierprogramm: ');
      WRITE ('Name der zu kopierenden Datei: ');
      READLN (Original);
      ASSIGN (Originalfile, Original);
      {$I-} RESET (Originalfile); {$I+}   {existiert file?}
     UNTIL IORESULT = 0;
     WRITE ('Name der Kopie: ');
     READLN (Kopie);
     ASSIGN (Kopiefile,Kopie);
     REWRITE (Kopiefile);
     Anzbloecke := Speicher DIV Block;  {Zahl der zu kopierenden Bloecke}
     Noch_zu_kopieren := FILESIZE (Originalfile);
     WHILE Noch_zu_kopieren > 0 DO BEGIN  {wenn noch etwas da}
      IF  Anzbloecke <= Noch_zu_kopieren THEN Liesbloecke := Anzbloecke
                                 ELSE Liesbloecke := Noch_zu_kopieren;
      BLOCKREAD (Originalfile, Puffer, Liesbloecke);
      BLOCKWRITE (Kopiefile, Puffer, Liesbloecke);
      Noch_zu_kopieren := Noch_zu_kopieren - Liesblocke
     END;
     CLOSE (Originalfile);
     CLOSE (Kopiefile)
    END.
```

Hinweis: Wir haetten auch den Datentyp CHAR fuer die Elemente des ARRAYs waehlen koennen, ohne etwas zu veraendern, da CHAR auch ein Byte Speicherplatz einnimmt. Wuerden wir den Dateneyp INTEGER z.B. als Elementtyp des ARRAYs verwenden, so muesste der Bereich des Feldes halbiert werde, da INTEGER zwei Bytes Speicherplatz braucht.

+--+
| Typenlose Datei: |
| |
| VAR f : FILE; |
| |
| Typenlose Dateien gestatten den direkten Zugriff auf Diskettendateien |
| ohne Ruecksicht auf das Datenformat. Die Elemente bei dieser |
| Zugriffsmethode sind als vom Typ Byte anzusehen. |
| Die Standardprozeduren und -funktionen ASSIGN, RESET, REWRITE, ERASE, |
| RENAME, FILESIZE, FILEPOS und EOF sind zu benutzen. Die Funktionen |
| FILESIZE und FILEPOS beziehen sich auf Bloecke pro 128 Byte. Mit |
| BLOCKREAD und BLOCKWRITE koennen solche Bloecke aus der Datei gelesen |
| oder in die Datei geschrieben werden. |
+--+

8.4 Zeiger

Es gibt schon merkwuerdige Typen im Pascal...

```
TYPE Zeiger = ^Objekt;
```

Eine Variable dieses Typs ist nicht selbst vom Typ Objekt (dann muesste es ja TYPE Zeiger = Objekt heissen), sondern eine Variable, die auf das Objekt zeigt.
Der Datentyp Zeiger ist von einfachem Typ (wie INTEGER, CHAR, REAL, BOOLEAN und Aufzaehlungen).

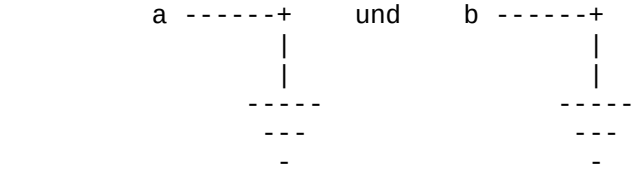
Deklarieren wir uns zwei Zeigervariablen:

```
TYPE Zeiger = ^CHAR;
VAR a,b : Zeiger;     (VAR a,b : ^CHAR; waere auch moeglich gewesen!)
```

Achtung: Die Variablen a und b sind **nicht** vom Typ CHAR.
Unsere beiden Zeiger zeigen nun so in die Weltgeschichte hinein. Dies ist ein ziemlich unbefriedigender, weil undefinierter Zustand. Daher gibt es eine Konstante namens

```
NIL       (engl. Nichts),
```

die jedem Zeiger beliebigen Typs zugeordnet werden kann. Wir nennen diese Konstantye NIL auch Erdung (aus der Physik bekannt).
Mit
a : NIL;
b : NIL; haben die Zeiger a und b definierte Zustaende. Sie sind geerdet.



Der Sinn dieser Erdung liegt darin, dass wir spaeter z.B. das Ende einer Liste durch eine solche Erdung kennzeichnen koennen oder entscheiden koennen, ob ein Zeiger auf ein Element zeigt oder nicht.
So, nun wollen wir unsere Zeiger mal auf Objekte vom Typ CHAR zeigen lassen. Dazu benoetigen wir die Prozedur

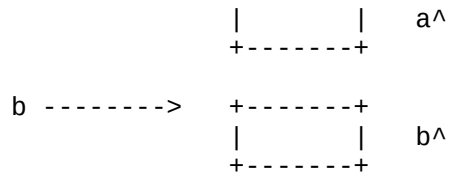
```
NEW (<zeiger>);
```

Mit dieser Prozedur wird eine vorerst leere Speicherstelle (Variable) erzeugt, auf die der Zeiger zeigt. Der Datentyp dieser Variablen ist durch den Zeigerdatentyp festgelegt.

```
NEW (a);
NEW (b);  erzeugt Variablen mit den Namen

a^ und
b^        vom Typ CHAR (erinnern wir uns: Der Zeigertyp ist ^CHAR).
```

Nun sieht das Bild folgendermassen aus:

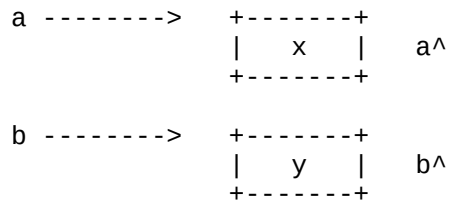
```
a -------->   +-------+
```



Die Kaestchen (Variablen) mit Namen a^ und b^ sind noch leer. Nun weisen wir ihnen Speicherinhalte zu:

```
a^:='x';
b^:='y';
```

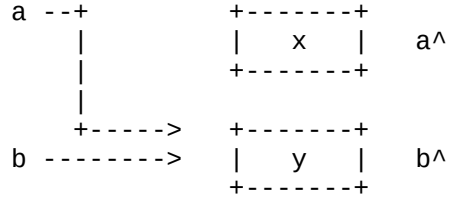
Im Bild sieht das so aus:



Die Zeiger a und b zeigen nun auf die Variablen a^ und b^ mit den Inhalten x und y.
Folgende Zuweisungsarten zwischen diesen Zeigern muessen wir jetzt genau unterscheiden:

```
a := b;
```

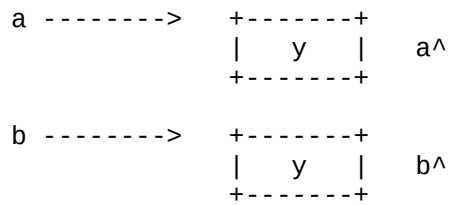
Im Bild:



Mit dieser Zuweisung wurde der Zeiger a so "verbogen", dass er auf das selbe Objekt zeigt wie der Zeiger b.
Beachten Sie: Auf das Objekt a^ zeigt nun kein Zeiger mehr. Dieses Objekt ist verloren! Es gibt keine Moeglichkeit mehr, auf dieses Objekt zuzugreifen. Werden Zeigervariablen einander zugewiesen, so muessen die Objekte, auf die die Zeiger zeigen, vom selben Typ sein.
Haetten wir die andere Zuweisungsart gewaehlt, naemlich

```
a^ := b^;
```

so ergaebe sich folgendes Bild:



Hier haben wir nur den Inhalt der Speicherstelle geaendert, auf die a zeigt. Im Speicher des Rechners gibt es nun den Buchstaben y zweimal.
In Turbo Pascal ist die DISPOSE-Anweisung das Gegestueck zur NEW-Anweisung.

Mit

```
DISPOSE (a);
```

wird die Speicherstelle , die der Zeiger a einnimmt wieder freigesetzt. In Version 1.0 ist DISPOSE nicht verfuegbar.

Statt dessen wird hier das Befehlspaar

```
MARK     und
RELEASE  verwendet in Form

VAR a : ^INTEGER;
  ...
MARK (a);
  ...
RELEASE (a);
  ...
```

Die MARK-Anweisung markiert zum Zeitpunkt ihrer Ausfuehrung den Beginn des Speichers fuer Zeigervariablen. Mit RELEASE werden alle Zeigervariablen seit dem letzten MARK geloescht. Dabei ist a ein Zeiger auf eine INTEGER-Variable, a zeigt naemlich auf den Anfang des Variablenstacks.
Die Anweisung DISPOSE und das Anweisungspaar MARK und RELEASE sind in einem Programm nicht zu mischen. Man hat sich fuer eine Form zu entscheiden.
Allerdings kann in Version 2.0 auch MARK und RELEASE verwendet werden, was z.B. sinnvoll sein kann, wenn man Umsteiger auf Turbo Pascal ist und alte Programme mit wenigen Veraenderungen weiterbenutzen will.
Folgende zusaetzliche Prozeduren und Funktionen sind mit Zeigervariablen definiert:

`GETMEM (p,i);` Prozedur, um einer Zeigervariablen p einen Speicherplatz von genau i Byte fuer den Heap bereitzustellen. Im Gegensatz zu NEW, das dem Zeiger so viel Platz bereitstellt, wie er benoetigt, wird mit GETMEM der Platz vorher begrenzt.

`FREEMEM (p,i);` ist die Umkehrprozedur zu GETMEM. Mit ihr wird der Zeigervariablen p der Platz von i Byte zurueckgegeben, i muss den gleichen Wert haben, der durch GETMEM definiert wurde.

`i:=MAXAVAIL;` ist eine Funktion mit einem Ergebnis vom Typ INTEGER, die die Groesse des groessten zusammenhaengenden Speicherplatzes angibt, der fuer Zeigervariablen zu verwenden ist. Unter CP/M wird die Groesse in Byte unter MS-DOS in Paragraphen angegeben.

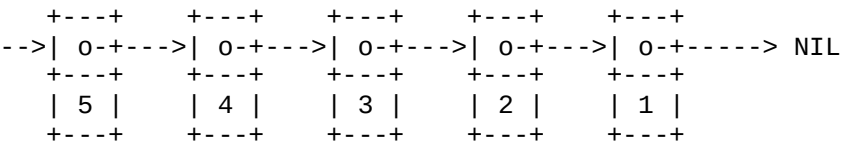
8.5 Listen

Wir haben schon eine Datenstruktur kennengelernt, die eine Liste darstellt: das eindimensionale Feld.
Die Elemente eines solchen Feldes sind alle aneinandergereiht. Die Reihenfolge der Elemente wird durch die Nummer der Elemente bestimmt. Der Nachteil dieses Feldes liegt daran, dass die Anzahl der Elemente von vornherein bestimmt werden muss. Sicher ist das eine Speicherplatzverschwendung, wenn das Feld im Verlauf des Programmes nicht ausgefuellt wird, und andererseits aergerlich, wenn das Programm mehr Daten erzeugt, als das Feld aufnehmen kann.

Abhilfe schafft eine Liste, die gebildet wird durch Datenelemente, die aus zwei Komponenten bestehen:

1. einem Zeiger auf ein anderes Element
2. den Dateninhalt

Eine solche Liste koennte so aussehen:



Der Zeiger eines jeden Elements der Liste zeigt auf das benachbarte Element. Der Anfang der Liste stellt ein Element namens Wurzel dar, das Ende wird durch die Erdung NIL angegeben.
Die Dateninhalte der Listenelemente sind Zahlen von 1 bis 5. Die umgekehrte Reihenfolge kommt durch die besondere Art der Listenerstellung zustande.
Schaffen wir uns nun zunaechst eine Datenstruktur fuer die Listenelemente. Der Datentyp muss sicher ein Verbund sein, der sowohl den Zeiger auf das naechste Element als auch auf den Dateninhalt des Elements umfasst. Der Dateninhalt soll vom Typ INTEGER sein.

```
TYPE Element = RECORD
                 Naechster : Zeiger;
                 Inhalt    : INTEGER
               END;
```

Das Problem besteht jetzt im Datentyp Zeiger. Denn mit

```
TYPE Zeiger = ^Element;
```

haben wir es mit einer rekursiven Typdeklaration zu tun. Was deklarieren wir denn nun zuerst?
Das Problem wird in Pascal dadurch geloest, dass vorgeschrieben ist, den Zeiger zuerst zu deklarieren. Unsere Typdeklaration sieht wie folgt aus:

```
TYPE Zeiger = ^Element

     Element = RECORD
                 Naechster : Zeiger;
                 Inhalt    : INTEGER
               END;
```

Nehmen wir noch zwei Variablen dazu:

```
VAR Wurzel,z : Zeiger;
```

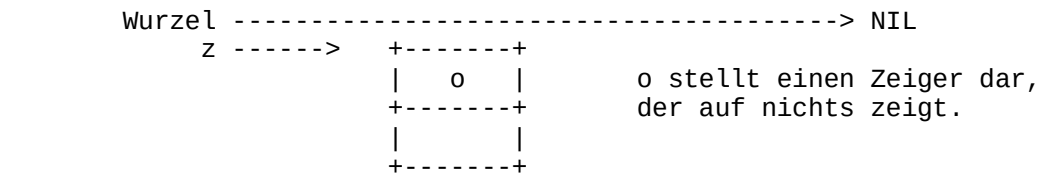
Um die Liste zu erzeugen, erden wir zunaechst den Zeiger Wurzel:

```
Wurzel := NIL;
```

Nun erzeugen wir ein leeres Element:

```
NEW(z);
```

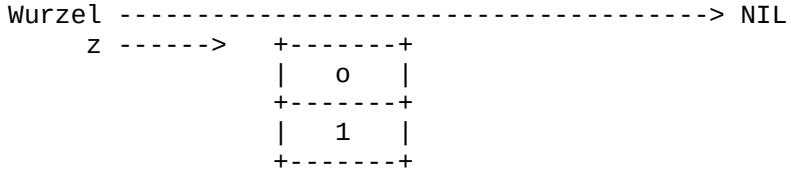
Die Situation stellt sich jetzt wie folgt dar:



Lesen wir einen Dateninhalt ein:

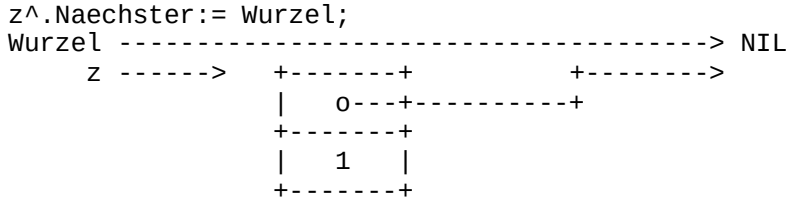
```
z^.Inhalt := 1;
```

Mit dieser Programmzeile weisen wir dem Inhalt-Teil des Elements, auf das z zeigt, die Zahl 1 zu.



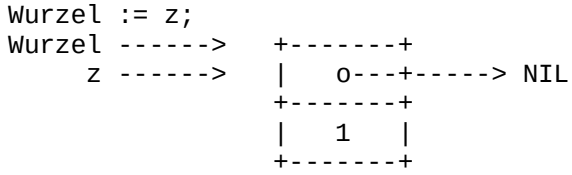
Nun soll der Zeiger-Teil des Elements auf Erdung zeigen:

```
z^.Naechster:= Wurzel;
```



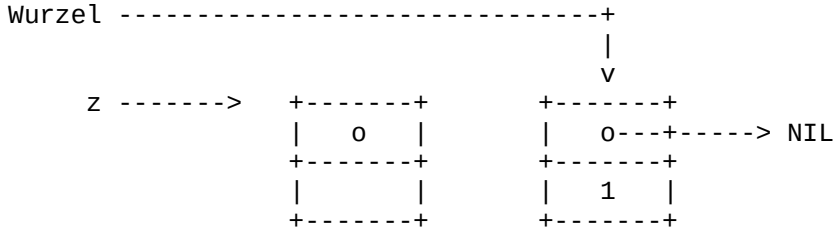
Der letzte Schritt besteht darin, das wir die Wurzel auf das neueste Element (d.h. bisher unser einziges Element zeigen lassen:

```
Wurzel := z;
```



Was ist schon eine Liste mit einem einzigen Element? Also schaffen wir ein zweites...

```
NEW (z);   erstellt uns ein neues Leerelement.
```

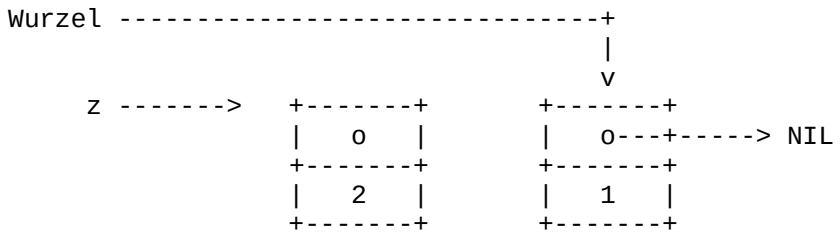


An dieser Stelle verstehen wir auch, warum der Zeiger namens Wurzel noetig

ist. Haetten wir ihn nicht auf das zuletzt eingefuegte Element zeigen lassen, so waere der Zugriff auf dieses Element verloren, nachdem wir die NEW(z)-Anweisung ausgefuehrt haben.
Durch

```
z^.Inhalt := 2;
```

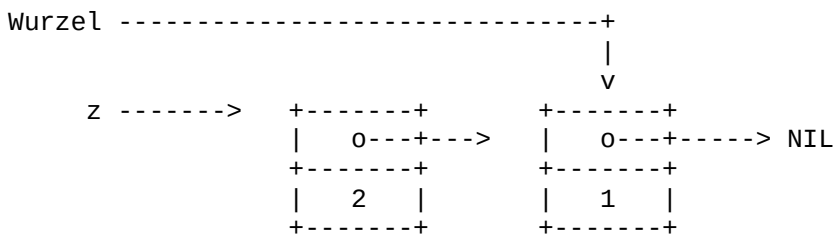
weisen wir dem neuen Element einen Dateninhalt zu.



Mit der Anweisung

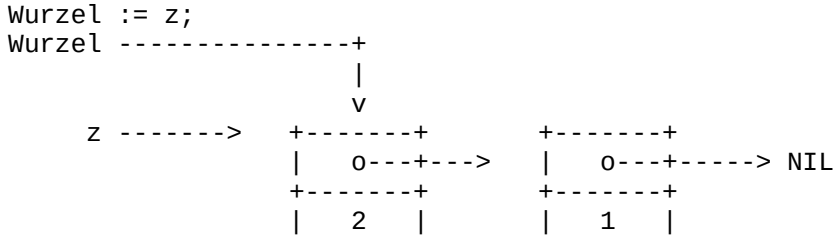
```
z^.Naechster := Wurzel;
```

schaffen wir die Veknuepfung zum letzten Element.



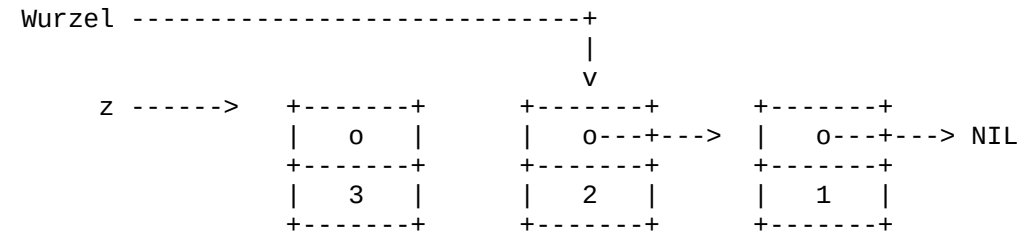
Als letzte Operation ruecken wir den Zeiger Wurzel wieder weiter auf das neueste Element:

```
Wurzel := z;
```



Nocheinmal schreiben wir diese vier Schritte zur Erzeugung und Anbindung eines dritten Elements, jedoch in etwas kuerzerer Form:

```
NEW (z);
z^.Inhalt := 3;
```



```
    z^.Naechster := Wurzel;

  Wurzel ------------------------------+
                                       |
                                       v
       z ------>   +-------+      +-------+      +-------+
                   |   o---+--->  |   o---+--->  |   o---+---> NIL
                   +-------+      +-------+      +-------+
                   |   3   |      |   2   |      |   1   |
                   +-------+      +-------+      +-------+

    Wurzel := z;

  Wurzel --------------+
                       |
                       v
       z ------>   +-------+      +-------+      +-------+
                   |   o---+--->  |   o---+--->  |   o---+---> NIL
                   +-------+      +-------+      +-------+
                   |   3   |      |   2   |      |   1   |
                   +-------+      +-------+      +-------+
```

Wir haben nun schon eine Liste mit drei Elementen erzeugt. Auffallend sind dabei zwei Tatsachen:

1. Die Liste wird rueckwaerts aufgebaut.
2. Wir haben immer die gleichen vier Anweisungen gebraucht.

Damit sind wir in der Lage, ein Programm zu schreiben, das eine Liste von Zahlen erstellt. Als Abbruchbedingungen definieren wir: Wenn die Zahl Null eingegeben wird, so stellt sie das Ende der Liste dar.

```
     PROGRAM Liste1;

       TYPE Zeiger = ^Element;

            Element = RECORD
                        Naechster : Zeiger;
                        Inhalt    : INTEGER
                      END;

       VAR Wurzel, z : Zeiger;

       BEGIN
        Wurzel := NIL;                (* Erdung der Liste *)
        REPEAT
         NEW (z);                     (* neues Element schaffen *)
         READLN (z^.Inhalt);          (* Dateninhalt einlesen *)
         z^.Naechster := Wurzel;      (* mit naechstem E1. verbinden *)
         Wurzel := z                  (* Wurzel aktualisieren *)
        UNTIL z^.Inhalt = 0
       END.
```

Eine Liste zu erstellen ist ziemlich unbefriedigend, wenn wir sie nicht auch wieder lesen koennen. Dazu erweitern wir das Programm durch eine Schleife, in der ein Zeiger, der Zeiger z, von der Wurzel an die Liste durchlaeuft und die Dateninhalte der Elemente ausgibt, solange er nicht auf NIL zeigt.

Achtung: Niemals den Zeiger Wurzel veraendern, da sonst der Anfang der Liste verloren waere.

```
    PROGRAM Liste

      TYPE Zeiger = ^Element;

          Element = RECORD
                      Naechster : Zeiger;
                      Inhalt    : INTEGER
                    END;

      VAR Wurzel, z : Zeiger;

      BEGIN
       WRITELN ('Erstellen der Liste.');
       WRITELN ('Geben Sie Zahlen ein. Ende mit Null.');
       Wurzel := NIL;            (* Erdung der Liste *)
       REPEAT
        NEW (z);                 (* Neues Element schaffen *)
        READLN (z^.Inhalt);      (* Dateninhalt einlesen *)
        z^.Naechster := Wurzel;  (* Mit naechstem El. verbinden *)
        Wurzel := z              (* Wurzel aktualisieren *)
        UNTIL z^.Inhalt = 0;
        (*--------------------------------*)
        WRITELN ('Ausgabe der Liste: ');
        z := Wurzel;             (* Laufzeiger auf Wurzel setzen *)
        WHILE z<>NIL DO BEGIN
         WRITELN (z^.Inhalt);    (* Lese Element unter Laufzeiger *)
         z := z^.Naechster       (* Laufzeiger weiterruecken *)
        END
       END.
```

Aufgabe: Testen Sie das Programm. In welcher Reihenfolge erscheinen die eingegebenen Zahlen wieder in der Ausgabe?

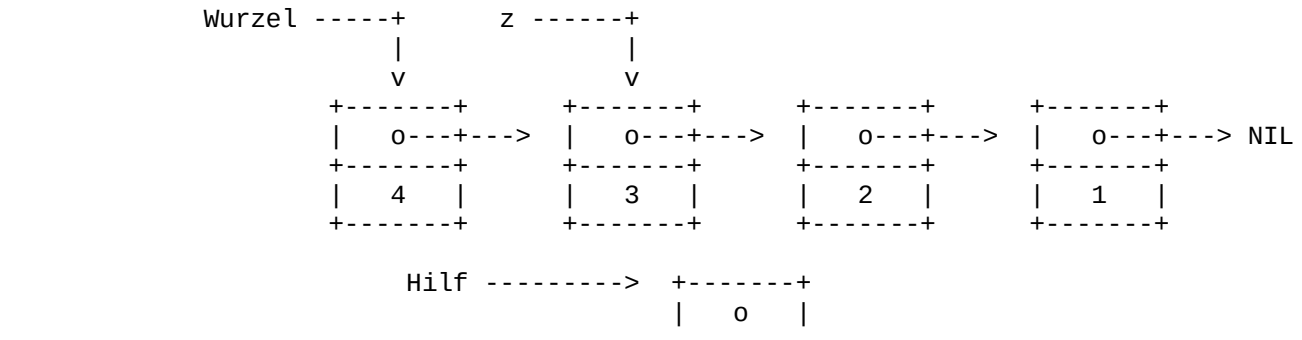
Beim Arbeiten mit einer Liste sind zwei Operationen wichtig:

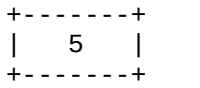
1. Einfuegen
2. Loeschen

Was bedeutet es, ein neues Element nach einem bestehenden Element einzufuegen? Dazu deklarieren wir noch eine weitere Zeigervariable:

```
    VAR Wurzel,z,Hilf : Zeiger;
```

Die Situation stellt sich folgendermassen dar:





Es existiert eine Liste aus vier Elementen. Der Laufzeiger z zeigt auf das Element, nach dem das neue Element eingefuegt werden soll.
Mit

```
NEW (Hilf);
Hilf^.Inhalt := 5;
```

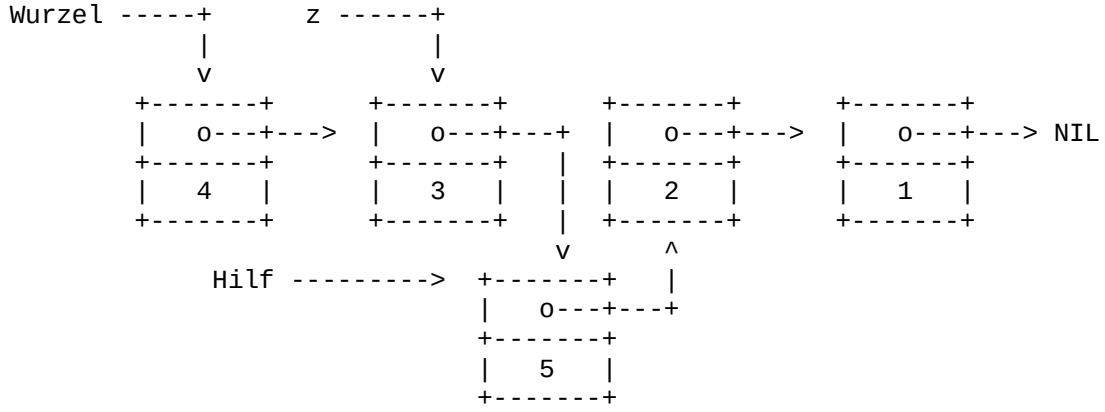
wurde ein neues Element erzeugt und mit Dateninhalt gefuellt. Einfuegen bedeutet nun, dass der Zeiger des neuen Elements auf das Element nach z zeigen soll:

```
Hilf^.Naechster := z^.Naechster;
```

und das Element, auf das z zeigt, selbst auf das neue Element zeigen soll:

```
z^.Naechster := Hilf;
```

Die neue Situation:



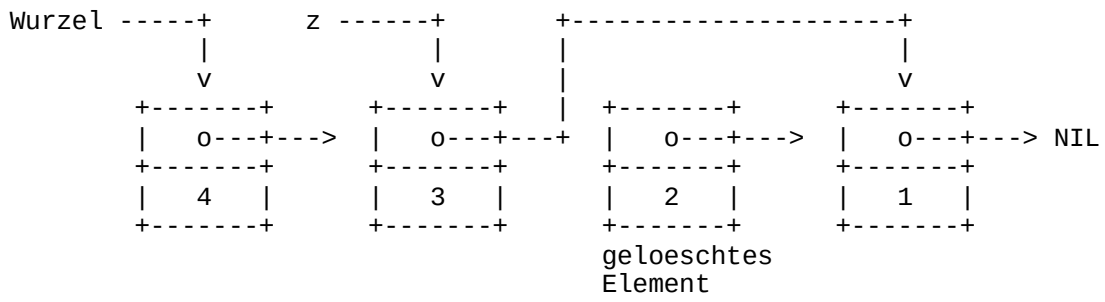
Frage: Warum wurde die Reihenfolge der Zuweisungen so gewaehlt?

Machen wir eine Prozedur aus diesen Erkenntnissen:

```
PROCEDURE Einfuegen (Hinterdiesem, Neues : Zeiger);
  BEGIN
   Neues^.Naechster := Hinterdiesem^.Naechster;
   Hinterdiesem^.Naechster := Neues
  END;
```

Die Prozedur fuegt ein Element Neues hinter dem Element Hinterdiesem ein. Dabei muss natuerlich das Hauptprogramm ein neues Element schaffen, ihm einen Inhalt zuweisen, den Platz zum Einfuegen finden und dann diese Prozedur aufrufen.
Achtung: Pruefen wir die moeglichen Probleme dieser Prozedur. Ist das Element, hinter dem eingefuegt werden soll, das letzte Element, so geht dies auch. Soll aber das Element als erstes Element der Liste eingefuegt werden, so funktioniert diese Methode nicht.
Aufgabe: Schreiben Sie eine Prozedur, die ein Element **vor** einem anderen Element einfuegt!
Nun kommen wir zum Problem des Loeschens. Das Loeschen stellt sich ebenfalls als recht einfach dar. Das Prinzip besteht darin, dass der Zeiger .Naechster des Elements vor dem zu loeschenden Element auf das Element nach diesem

"verbogen" wird.



Das Verbiegen des Zeigers erreichen wir mit

```
z^.Naechster := z^.Naechster^.Naechster;
```

Dabei ist das zu loeschende Element das, welches den Dateninhalt

```
z^.Naechster^.Inhalt          hat.
```

Bei unserer Loeschprozedur muessen wir allerdings auch den Fall betrachten, dass das zu loeschende Element das erste Element der Liste ist. Dies erreichen wir mit der folgenden Konstruktion, bei der der Variablenparameter Neuewurzel die alte Wurzel aktualisiert, wenn das erste Element geloescht wird.

```
PROCEDURE Loesche (Zuloeschen : Zeiger; VAR Neuewurzel : Zeiger);
 VAR Hilf : Zeiger;
 BEGIN
  IF Zuloeschen = Neuewurzel THEN Neuewurzel := Zuloeschen^.Naechster
    ELSE BEGIN
     Hilf := Neuewurzel;
     WHILE Hilf^.Naechster <> Zuloeschen DO
       Hilf := Hilf^.Naechster;
     Hilf^.Naechster := Hilf^.Naechster^.Naechster
    END  (* Else *)
 END;  (* Loesche *)
```

Aufgabe: Testen Sie auch diese Prozedur in einem Programm.
Mit der natuerlich etwas abgewandelten Einfuegeprozedur laesst sich ein Programm zum Erstellen einer geordneten Liste von Zeichen erstellen.

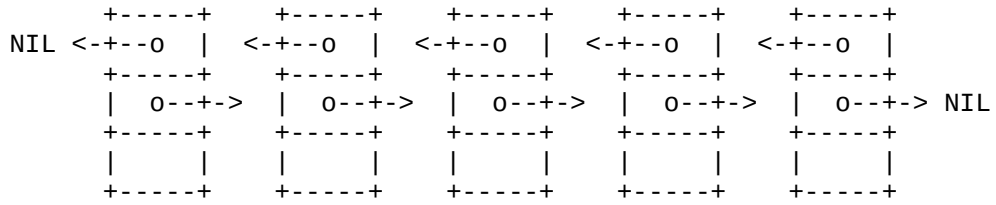
```
PROGRAM Ordnerliste;

  TYPE Zeiger=^Objekt;

       Objekt=RECORD
                Naechster: Zeiger;
                Daten    : CHAR
              END;  (* Record *)

  VAR Anfang, Hilf, Z : Zeiger;
      Zeichen         : CHAR;
      Eingefuegt      : BOOLEAN;

  BEGIN (* Hauptprog *)
    WRITE('Eingabe eines Zeichens:');
```

```
        READ (KBD,Zeichen); WRITELN (Zeichen);
        NEW (Anfang);
        Anfang^.Daten := Zeichen;
        Anfang^.Naechster := NIL;

        REPEAT
          WRITE('Eingabe eines Zeichens:');
          READ (KBD,Zeichen); WRITELN (Zeichen);
          NEW (Z);
          Z^.Daten := Zeichen;
          IF Anfang^.Daten > Zeichen THEN BEGIN
                                            Z^.Naechster:=Anfang;
                                            Anfang:=Z
                                          END (*if*)
             ELSE BEGIN
                    Eingefuegt:=FALSE;
                    Hilf:=Anfang;
                    WHILE (Hilf^.Naechster<>NIL) AND
                    (Not Eingefuegt) DO BEGIN
                     IF Hilf^.Naechster^.Daten>Zeichen THEN BEGIN
                        Z^.Naechster:=Hilf^.Naechster;
                        Hilf^.Naechster:=Z;
                        Eingefuegt:=TRUE
                     END; (*if*)
                     Hilf:=Hilf^.Naechster;
                    END; (*while*)
                    IF NOT Eingefuegt THEN BEGIN
                       Hilf^.Naechster:=Z;
                       Z^.Naechster:=Nil
                    END; (*if*)
             END; (*else*)
        UNTIL Zeichen = '/';

        (*Lesen:*)
        Hilf := Anfang;
        WHILE Hilf<>Nil DO BEGIN
          WRITE(Hilf^.Daten);
          Hilf := Hilf^.Naechster;
        END; (*while*)
        WRITELN;
      END.
```

Aufgabe: Analysieren Sie das Programm, und testen Sie es.
Ausser den hier besprochenden einfach verketteten Listen sind auch mehrfach verkettete Listen denkbar. So koennnten wir mit der Datenstruktur

```
    TYPE Zeiger =^Objekt;

         Objekt = RECORD
                    Links, Rechts : Zeiger;
                    Inhalt : CHAR
                  END;
```

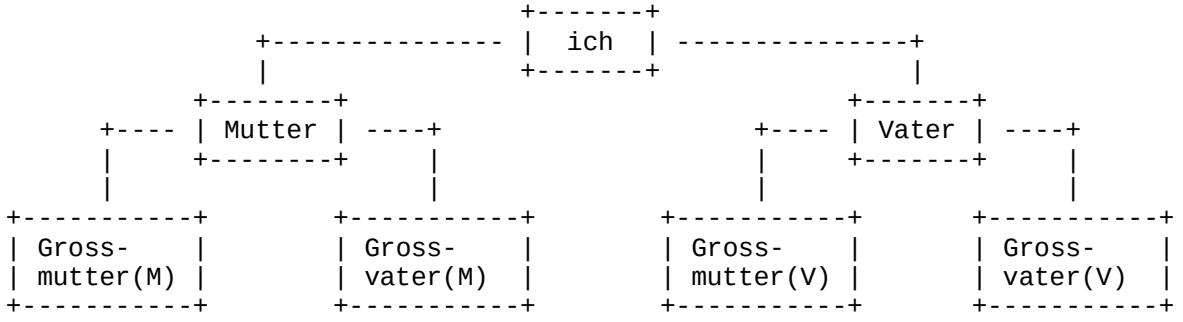
eine zweifach verkettete Liste darstellen. Jedes Element hat dann neben dem Dateninhalt zwei Zeiger, von denen einer auf den linken und einer auf den rechten Nachbarn des Elements zeigt. Eine solche Liste hat natuerlich einen linken und einen rechten Anfang und zwei Enden mit NIL.



Durch einfaches Verbinden des Anfangs einer Liste mit ihrem Ende erhalten wir einen Ring (eine zyklische Struktur). Verbinden wir die Anfaenge und Enden der zweichfach verketteten Liste miteinander, so erhalten wir einen doppelt verketteten Ring, der sich sehr leicht in beide Richtungen lesen laesst.

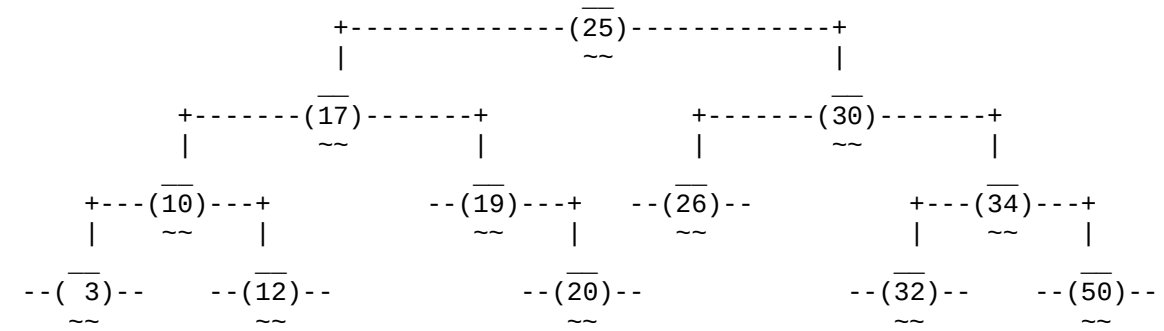
8.6 Baeume

Schreiben wir einmal unseren Stammbaum in der Art, dass wir bei uns selbst anfangen.



Und so weiter . . .

Eine solche Struktur der Datendarstellung von Daten nennen wir Baum.
Wir koennen auch andere Daten in Form eines Baumes darstellen.



Beide Baeume haben einiges gemeinsam:
- Die Wurzel des Baumes ist oben.
- Von jedem Knotenpunkt gehen hoechsten zwei Verzweigungen ab.
- Von Endknoten gehen keine Verzweigungen aus.

Baeume, die an jeden Knoten hoechsten zwei Verzweigungen haben, nennen wir binaere Baeume. Wir wollen uns im folgenden nur mit binaeren Baeumen beschaeftigen.

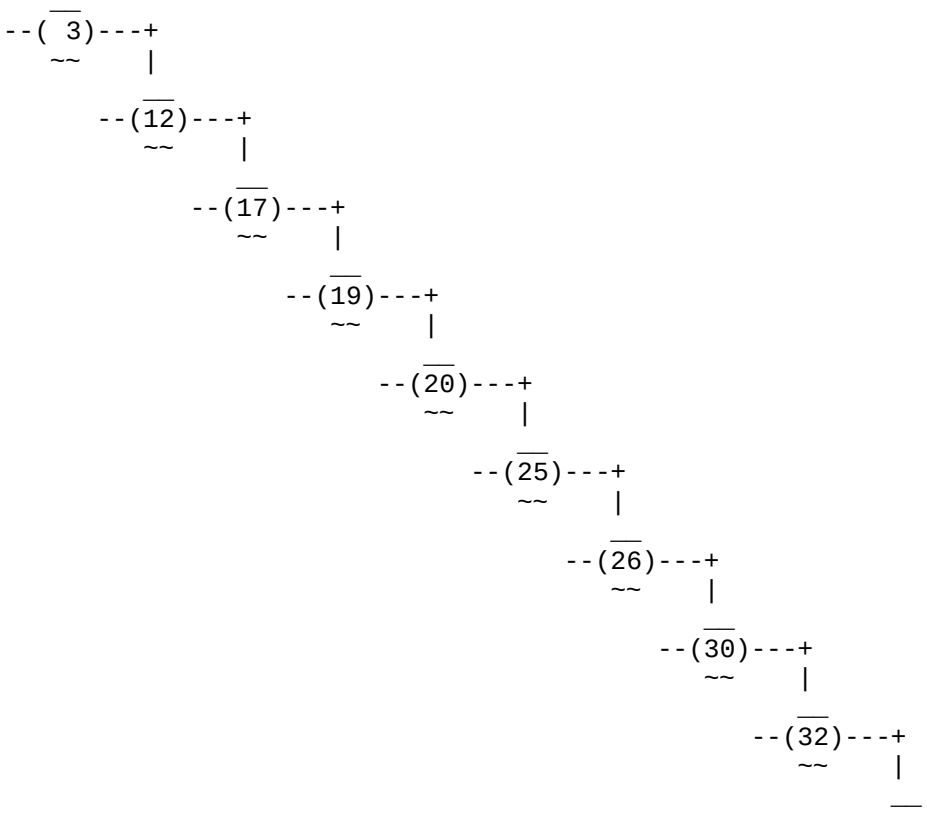
- Knoten: Von hier gehen Verzweigungen zu weiteren Elementen aus.
- Wurzel: Der Knoten, zu dem selbst kein anderes Element zeigt, d.h. der

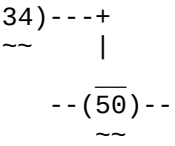
"obere" Knoten.
- Endknoten: Ein Knoten, der auf kein weiteres Element zeigt. Die Zeiger des Endknotens sollten also geerdet sein.
- Teilbaum: Ein Teilbaum ist ein Baum, dessen Wurzel ein Knoten eines anderen Baumes ist.

Der Zahlenbaum im obigen Beispiel hat ausserdem die Eigenschaft, dass er geordnet ist. Von jedem Knoten aus finden wir links stets kleinre, rechts stets groessere Elemente.
Welchen Sinn haben nun diese Strukturen in der Informatik?
Schreiben wir uns die Zahlen unseres Zahlenbaumes einmal in aufsteigender Reihenfolge (in einer Liste) auf:

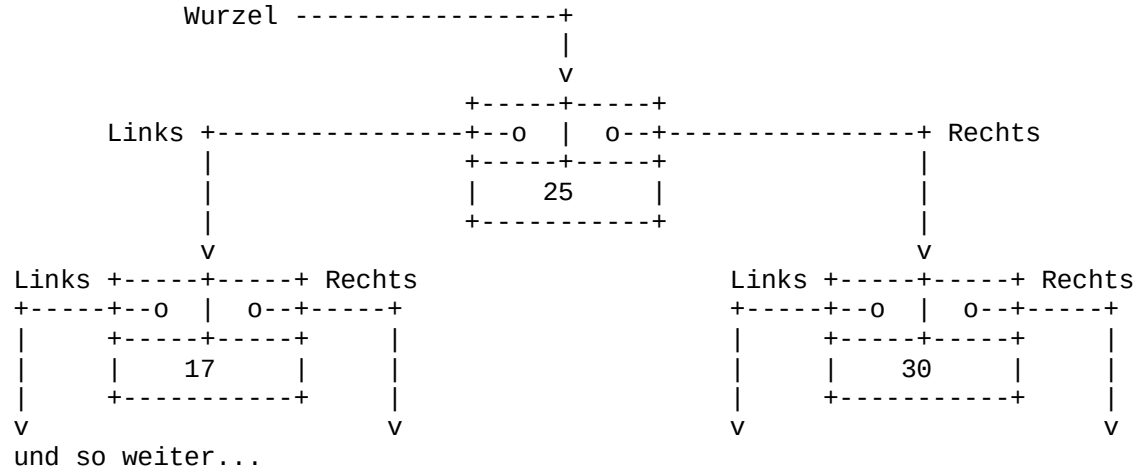
3 10 12 17 19 20 25 26 30 32 34 50

Wenn wir in einer Liste Daten suchen, so koennen wir Glueck haben, dass das Gesuchte am Anfang steht, oder Pech, dass es recht weit hinten zu finden ist. Im schlechtesten Fall muessten wir alle Elemente durchsuchen, wenn wir z.B. die Zahl 50 suchen.
Ganz anders in unserem Baum. Wir schauen im obersten Knoten (der Wurzel) nach und stellen fest, dass 50 groesser als dieser ist. Folglich brauchen wir nur noch nach rechts zu gehen. Dort finden wir die Zahl 30, 50 ist ebenfalls groesser als dieser Knoten. Genauso ist es beim naechsten Knoten.
Insgesamt muessen (einschliesslich der 50) 4 Vergleiche vorgenommen werden. Bei der Liste haetten wir 12 Vergleiche zu machen.
Allgemein kann man sagen, dass in einer Liste mit N Elementen im schlechstesten N Vergleiche vorzunehmen sind, in einem geordneten Baum jedoch nur so viele Vergleiche, wie die Verzweigungstiefe des Baumes ist.
Hier ist allerdings gleich anzumerken, dass es durchaus geordnete Baeume gibt, die sehr unguenstig gestaltet sind.





Einen solchen Baum nennen wir entartet. Von jedem Knoten geht nur jeweils eine Verzweigung aus. Es handelt sich genau genommen um eine lineare Liste. Natuerlich haben wir bezueglich des Suchens bei diesem Baum keinen Vorteil. Besonders effektiv sind die ausgewogenen Baeume (AVL-Baeume, benannt nach Adelson, Velskii und Landis). Hier heisst es:
Ein Baum ist genau dann ausgeglichen, wenn sich fuer jeden Knoten die Hoehen der von ihm ausgehenden Teilbaeume um hoechstens 1 unterscheiden.
Wir wollen hier das Thema Baeume nicht in aller Ausfuehrlichkeit behandeln - dafuer reicht der Platz nicht aus. Vielmehr wollen wir uns eine Datenstruktur schaffen, die die Konstruktion eines Baumes zulaesst, dann ein Programm entwickeln, das Daten in einem geordneten Baum abspeichert, und zu guter Letzt einen ausgewogenen Baum erzeugen.
Weitergehende Informationen zu Baeumen sind u.a. zu finden in Wirth, Algorithmen und Datenstrukturen.
Wir koennen uns die Datenstruktur wieder in einem Bild verdeutlichen:



Die Datenstruktur zu einem binaeren Baum sieht sinnvollerweise folgendermassen aus:

```
TYPE Zeiger =^Knoten;
     Knoten = RECORD
                Links, Rechts : Zeiger;
                Inhalt        : INTEGER (*Dateninhalt*)
              END;
```

Was bedeutet es nun, einen Baum aufzubauen ?
Diese Taetigkeit wird einzig und allein durch Einfuegen neuer Elemente bestimmt. Nehmen wir an, wir haben einen Laufzeiger namens Lauf. Dieser soll den Baum beim Einfuegen neuer Elemente durchlaufen. Folgende Situation koennte sich dann ergeben, wenn Lauf auf einen Knoten (am Anfang auf die Wurzel) des Baumes zeigt:
1. Lauf zeigt auf NIL. In diesem Fall ist der Knoten noch unbesetzt, und es

kann das neue Element hier eingefuegt werden. Dabei ist auch dafuer zu sorgen, dass die Zeiger des neuen Elements auf NIL gesetzt werden.

2. Lauf zeigt nicht auf NIL. In diesem Fall ist der Knoten schon besetzt. Nun ist zu unterscheiden, in welche Richtung der Baum weiter zu durchsuchen ist, um das neue Element der Groesse nach einzufuegen.
   a)Das neue Element ist groesser als der Knoten. In diesem Fall muessen wir an diesem Knoten nach rechts gehen und in unsere Vorschrift wieder bei Punkt 1. anfangen.
   b)Das neue Element ist kleiner als der Knoten. In diesem Fall muessen wir an diesem Knoten nach links gehen und in unserer Vorschrift wieder bei Punkt 1. anfangen.

Aufgabe: Erzeugen Sie nach dieser Methode einen Baum auf einem Blatt Papier.

Aus der Formulierung geht hervor, dass es sich um einen rekursiven Algorithmus handelt.Schreiben wir die Prozedur Einfuege in Pascal. Die Prozedur braucht zwei Parameter: die einzufuegende Zahl als Wert und den Laufzeiger als Variable (Warum?).

```
(*$A-*)
PROCEDURE Einfuege (Zahl : INTEGER; VAR Lauf : Zeiger);
  BEGINN
   IF Lauf =  NIL THEN BEGIN  (* hier einfuegen *)
     NEW (Lauf);
     WITH Lauf^ DO BEGIN
      Inhalt := Zahl;
      Links  := NIL;
      Rechts := NIL
     END (* with *)
   END (* if *)
   ELSE
    IF Zahl < Lauf^.Inhalt THEN Einfuege (Zahl, Lauf^.Links)
     ELSE
      IF Zahl > Lauf^.Inhalt THEN Einfuege (Zahl, Lauf^.Rechts)
END; (* Einfuege *)
(*$A+*)
```

Aufgabe: Zum Verstaendnis der Prozedur Einfuege ist folgende Ueberlegung notwendig. Zeichnen Sie sich einen beliebigen (nicht zu grossen) Baum auf. Lassen Sie einen Zeiger namens Wurzel auf die Wurzel des Baumes zeigen. Waehlen Sie eine beliebige Zahl N, die eingefuegt werden soll.
Nun simulieren Sie den Prozeduraufruf: Einfuege (N, Wurzel);. Bedenken Sie dabei, dass bei jedem Selbstaufruf der Prozedur (Rekursion) ein neuer Zeiger namens Lauf gebildet wird.
Als naechstes muessen wir noch eine Prozedur Druckebaum erstellen, die uns die Struktur des Baumes auf dem Bildschirm zeigt. Eine einfache Methode besteht darin, den Baum um 90 Grad gedreht darzustellen, zum Beispiel so:

```
+-------------------------------------------------------------------------+
|                                |--> 55                                  |
|                           |--> 40                                       |
|                                |--> 32                                  |
|                      |--> 30                                            |
|                                |--> 10                                  |
|                           |-->  5                                       |
|                                |-->  1                                  |
+-------------------------------------------------------------------------+
```

Auch diese Prozedur formulieren wir rekursiv. Wir starten mit einem Laufzeiger bei der Wurzel und durchsuchen den Baum.
Wenn es noch einen weiteren Knoten gibt, d.h. wenn der Laufzeiger nicht auf NIL zeigt, muessen wir den Baum erst rechts weiter durchsuchen, dann den Knoten drucken und schliesslich links weiter durchsuchen. So erreichen wir, dass die Zahlen in ihrer Sortierung (mit der groessten Zahl angefangen) untereinander gedruckt werden. Um die Knoten auch noch eingerueckt zu drukken, muessen wir einen Wert immer dann um eins erhoehen, wenn wir noch einen weiteren Knoten suchen. Dieser Wert wird der Prozedur bei jedem Aufruf mitgegeben.

So erhalten wir als vollstaendiges Programm:

```
    PROGRAM Sorttree;

       Type Zeiger = ^Knoten;

            Knoten = RECORD
                      Links, Rechts : Zeiger;
                      Inhalt        : INTEGER;
                     END;
       VAR N : INTEGER;
           Wurzel : Zeiger;

       (*$A=*)
       PROCEDURE Druckebaum (Lauf : Zeiger; Stelle : INTEGER);
         VAR i : INTEGER;
         BEGIN
          IF Lauf <>-NIL THEN   (* noch weitere Knoten vorhanden *)
          WITH Lauf^ DO BEGIN
            Druckebaum (Rechts, Stelle + 1);
            FOR i := 1 TO Stelle DO WRITE (' ':7);
            WRITE (' I--->');WRITELN (Inhalt : 3);
            Duckebaum (Links, Stelle + 1)
          END (* with *)
         END; (* Druckebaum *)

      PROCEDURE Einfuege (Zahl : INTEGER; VAR Lauf : Zeiger);
         BEGIN
          IF Lauf = NIL THEN BEGIN (* hier einfuegen *)
            NEW (Lauf);
            WITH Lauf^ DO BEGIN
             Inhalt := Zahl;
             Links  := NIL;
             Rechts := NIL;
            END (* with *)
         END (* if *)
         ELSE
          IF Zahl < Lauf^.Inhalt THEN Einfuege (Zahl, Lauf^.Links)
           ELSE
            IF Zahl > Lauf^.Inhalt THEN Einfuege (Zahl, Lauf^.Rechts)
       END; (* Einfuege *)
       (*$A+*)

      BEGIN (* Hauptprogramm *)
        Wurzel := NIL;
        WRITELN ('Unausgewogener Sortierbaum:');
```

```
        WRITELN ('Geben Sie Zahlen ein, ');
        WRITELN ('Ende mit 0 !');
        WRITE ('Zahl: ');
        READLN (N9;
        WHILE N <> 0 DO BEGIN
         Einfuege (N, Wurzel);
         WRITE ('Zahl: ');
         READLN (N);
        END; (* while *)
        Druckebaum (Wurzel, 0)
      END.
```

Aufgabe: Testen Sie das Programm, und simulieren Sie auf einem Blatt Papier die Prozedur Druckebaum.

Bei Eingabe der Zahlenreihe

25 10 30 8 4 70 3 28 9 8 2 0

(in dieser Reihenfolge) erhalten wir folgende Ausgabe:

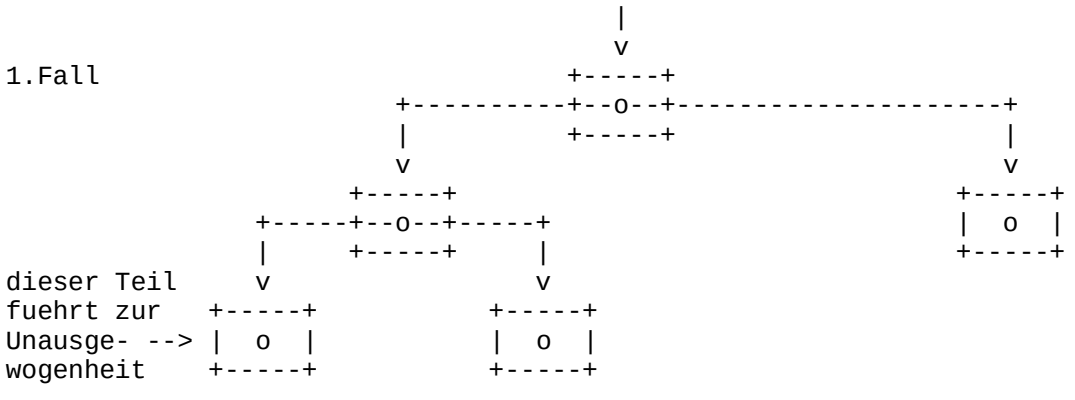
```
+-------------------------------------------------------------------------+
|                  |--> 70                                                |
|             |--> 30                                                     |
|                  |--> 28                                                |
|        |--> 25                                                          |
|             |--> 10                                                     |
|                       |-->  9                                           |
|                  |-->  8                                                |
|                       |-->  4                                           |
|                            |-->  3                                      |
|                                 |-->  2                                 |
+-------------------------------------------------------------------------+
```

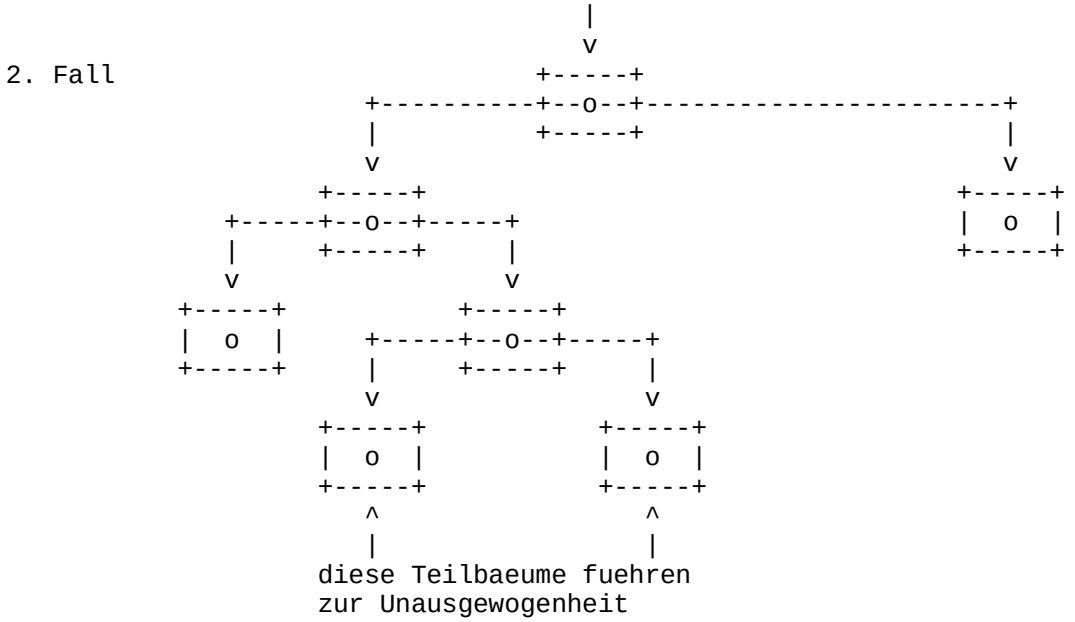
Aufgabe: Geben Sie die Zahlen in anderer Reihenfolge ein (insbesondere in sortierter Reihenfolge).
Es ergeben sich je nach Reihenfolge der Eingabe andere Baeume, die mehr oder weniger gut ausgewogen sind. Das Programm achtet also nicht auf die Ausgewogenheit des Baums, sondern lediglich auf die Sortierung der Elemente. Wollen wir nun einen ausgeglichenen, sortierten Baum erstellen, muessen wir die Prozedur Einfuege dahingehend veraendern, dass nicht nur je nach Groesse des neuen Elements nach rechts oder links erweitert wird, sondern der Baum umgeschichtet wird, wenn die Differenz der rechten und linken Teilbaeume groesser als eine Ebene wird.
Dazu geben wir dem Zeiger (nach Wirth) einen zusaetzlichen Dateninhalt bal mit, der die Werte -1, 0 oder +1 haben kann.

```
    TYPE Zeiger = ^Knoten;
         Knoten = RECORD
                   Links,Rechts : Zeiger;
                   bal          : -1 .. +1;
                   Inhalt       : INTEGER
                  END;

    VAR Lauf : Zeiger;
```

Lauf^.bal = +1 : Der rechte Teilbaum ist eine Ebene groesser.
Lauf^.bal = 0 : Beide Teilbaeume sind gleich gross.
Lauf^.bal = -1 : Der linke Teilbaum ist eine Ebene groesser.

Entsteht die Situation, dass wir ein Element im linken Teilbaum einfuegen muessen, so wird im ersten Fall die Unausgeglichenheit aufgehoben (Lauf^.bal = 0), im zweiten Fall wird der Baum links einer Ebene groesser (Lauf^.bal = -1), und im letzten Fall muss der Baum ausgeglichen werden, in dem ein Element vom linken Teilbaum in den rechten Teilbaum uebertragen wird. (Beim Einfuegen eines Elemnts in den rechten Teilbaum gehen wir analog vor.)
Da die Prozedur Einfuege rekursiv geschrieben wird, koennen wir z.B. beim Einfuegen eines Elements in einen linken Teilbaum, bei dem ein Ausgleichen notwendig wird, nur zwei Situation entstehen:





Diese beiden Faelle sind in der IF ... THEN ... ELSE - Anweisung des Ausgleichens unterschieden.
Wie wir wissen, muss jede Rekursion einen Rekursionsabbruch besitzen. Dies gewaehrleisten wir durch eine Uebergabevariable namens Dif, die vom Typ BOOLEAN ist und angibt, ob die Hoehe des entsprechenden Teilbaums zugenommen hat. Zu Beginn der Prozedur (wenn noch kein neues Element eingefuegt wurde) hat diese Variable den Wert FALSE.
Hier nun das Programm, das einen ausgewogenen, sortierten Baum von Zahlen erstellt und ausgibt.

```
PROGRAM Baltree; (* Nach Wirth *)

  TYPE Zeiger = ^Knoten;
       Knoten = RECORD
                  Links, Rechts : Zeiger;
                  bal           : -1 .. +1;
                  Inhalt        : INTEGER
                END;

  VAR N : INTEGER;
      b : BOLLEAN;
      Wurzel : Zeiger;

  (*$A-*)
  PROCEDURE Druckebaum (Lauf : Zeiger; Stelle : INTEGER);
    VAR i : INTEGER;
    BEGIN
     IF Lauf <> NIL THEN   (* noch weitere Knoten vorhanden *)
     WITH Lauf^ DO BEGIN
       Druckebaum (Rechts, Stelle +1);
       FOR i := 1 TO Stelle DO WRITE (' ':7);
       WRITE (' I--->'); WRITELN (Inhalt : 3);
       Druckebaum (Links, Stelle + 1)
     END (* with *)
    END; (* Druckebaum *)

  PROCEDURE Einfuege (Zahl : INTEGER;VAR Lauf : Zeiger;VAR dif:BOOLEAN);
    VAR p1, p2 : Zeiger;      (* dif = FALSE *)
    BEGIN
      IF Lauf=NIL THEN BEGIN (* Zahl nicht im Baum, einfuegen *)
        NEW(Lauf);
        dif:=TRUE;
        WITH Lauf^ DO BEGIN
          Inhalt:=Zahl;
          Links:=NIL;
          Rechts:=NIL;
          bal:=0;
        END (* WITH *)
      END (* IF *) ELSE
      IF Zahl<Lauf^.Inhalt THEN BEGIN
        Einfuege(Zahl,Lauf^,Links,dif);
        IF dif THEN (* Linker Ast groesser *)
          CASE Lauf^.bal OF
            1: BEGIN Lauf^.bal:=0;dif:=FALSE END;
            0: Lauf^.bal:=-1;
           -1: BEGIN (* Ausgleichen *)
                 p1:=Lauf^.Links;
                 IF p1^.bal=-1 THEN BEGIN
                   Lauf^.Links:=p1^.Rechts;
                   p1^.Rechts:=Lauf;
                   Lauf^.bal:=0;
                   Lauf:=p1;
                 END (* IF *)
                 ELSE BEGIN
                   p2:=p1^.Rechts;
                   p1^.Rechts:=p2^.Links;
                   p2^.Links:=p1;
                   Lauf^.Links:=p2^.Rechts;
```

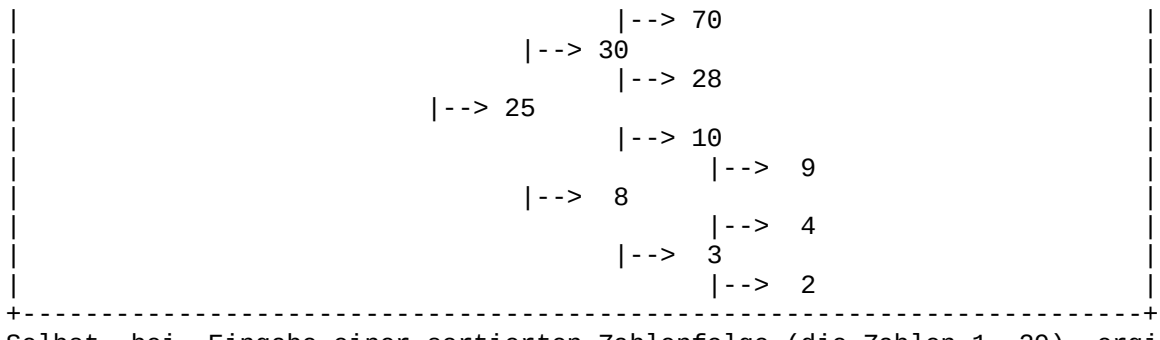
```
                  p2^.Rechts:=Lauf;
                  IF p2^.bal=-1 THEN Lauf^.bal:=+1
                                ELSE Lauf^.bal:=0;
                  IF p2^.bal=+1 THEN Lauf^.bal:=-1
                                ELSE Lauf^.bal:=0;
                  Lauf:=p2;
                END; (* ELSE *)
                Lauf^.bal:=0;
                dif:=FALSE
              END (* CASE -1 *)
            END (* CASE *)
          END (* IF *)
          ELSE
    IF Zahl>Lauf^.Inhalt THEN BEGIN
      Einfuege(Zahl,Lauf^.Rechts,dif);
      IF dif THEN (* rechter Ast groesser *)
         CASE Lauf^.bal OF
           -1: BEGIN Lauf^.bal:=0;dif:=FALSE END;
           0: Lauf^.bal:=+1;
           1: BEGIN (* Ausgleichen *)
                 p1:=Lauf^.Rechts;
                 IF p1^.bal=+1 THEN BEGIN
                   Lauf^.Rechts:=p1^.Links;
                   p1^.Links:=Lauf;
                   Lauf^.bal:=0;
                   Lauf:=p1
                 END (* IF *)
                 ELSE BEGIN
                   p2:=p1^.Links;
                   p1^.Links:=p2^.Rechts;
                   p2^.Rechts:=p1;
                   Lauf^.Rechts:=p2^.Links;
                   p2^.Links:=Lauf;
                   IF p2^.bal=+1 THEN Lauf^.bal:=-1
                                 ELSE Lauf^.bal:=0;
                   IF p2^.bal=-1 THEN Lauf^.bal:=+1
                                 ELSE Lauf^.bal:=0;
                   Lauf:=p2;
                 END; (* ELSE *)
                 Lauf^.bal:=0;
                 dif:=FALSE
               END (* CASE +1 *)
             END (* CASE *)
           END (* IF *)
         ELSE dif:=FALSE
END; (* Einfuege *)
(*$A+*)

BEGIN (* Hauptprogramm *)
  Wurzel := NIL;
  b := FALSE;
  WRITELN ('Ausgewogener Sortierbaum:');
  WRITELN ('Geben Sie die Zahlen ein, ');
  WRITELN ('Ende mit 0!');
  WRITE ('Zahl: ');
  READLN (N);
  WHILE N <> 0 DO BEGIN
   Einfuege (N, Wurzel, b);
```

```
         WRITE ('Zahl: ');
         READLN (N)
        END; (* while *)
        Druckebaum (Wurzel, 0)
      END.
```

Bei Eingabe der Zahlenreihe

```
                  25  10  30  8  4  70  3  28  9  8  2  0
```

(in einer beliebigen Reihenfolge) erhalten wir die Ausgabe:

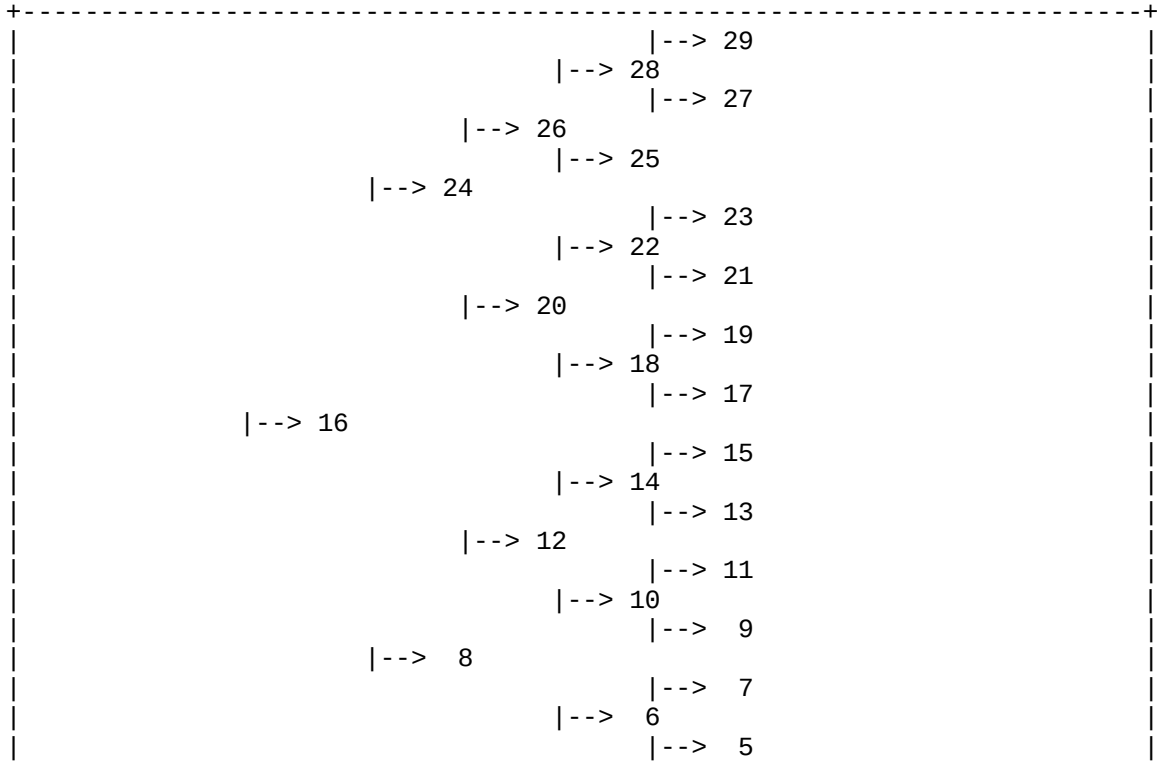


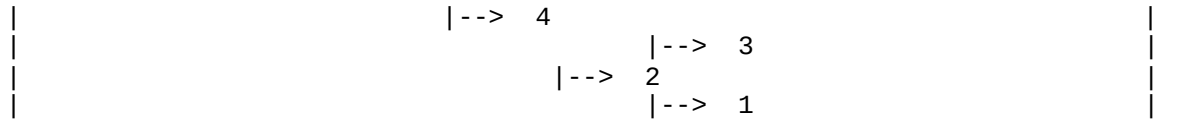
Selbst bei Eingabe einer sortierten Zahlenfolge (die Zahlen 1..29) ergibt sich ein ausgewogener Baum, wie Sie aus der Abbildung auf der gegenueberliegend Seite ersehen koennen.

Fazit:

Ein Baum ist immer dann als Datenstruktur sinnvoll zu verwenden, wenn eine grosse Datenmenge zu verarbeiten ist. Insbesondere zum schnellen Suchen von Elementen eignen sich AVL-Baeume hervorragend, da die Zahl der Vergleiche erheblich verringert wird.

Baeume sind sofort bei der Eingabe der Elemente sortiert.





Besonders bei der Verwaltung sogenannter Datenbanken werden Baeume verwendet. Dabei werden die Daten auf der Diskette nicht sortiert. Statt dessen wird ein sogenannter Suchbaum mit den Nummern der Elemente der Datei und den Suchworten aufgebaut. Mit SEEK laesst sich denn schnell auf ein Element zugreifen.

Ist der Datensatz groesser als das Fassungsvermoegen einer Diskette, wird der gesamte Datensatz in sogenannte Teilbaeume zerlegt.

Ausblick:

Es koennen, besonders zur weiteren Reduzierung der Vergleiche beim Suchen, auch Baeume mit hoeherem Grad als Zwei aufgebaut werden. Dabei gibt es dann in jedem Knoten mehr als zwei Zeiger auf nachfolgende Knoten.

Hier sind z.B. die sogenannten B-Baeume zu nennen (siehe z.B. Wirth: Algorithmen und Datenstrukturen).

Eine Anwendung fuer B-Baeume bietet das Turbo Toolbox-System Turbo Access (siehe Kap. 10.2).

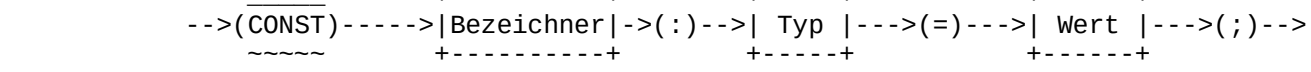
9.1 Typkonstanten

Eine Besonderheit von Turbo Pascal, die ueber den Pascal-Standard hinausgeht, sind die sogenannten Typkonstanten.
Erinnern wir uns daran, was Konstanten sind. Dies sind Bezeichner, denen im Deklarationsteil nach dem reservierten Wort CONST ein Dateninhalt zugewiesen wird, der ab dieser Stelle ein fuer allemal festgelegt ist. Der Datentyp der Konstanten ergibt sich aus der Darstellung des Wertes der Konstanten. Fuer die Deklaration von Konstanten sind nur einfache Typen erlaubt. Das bringt den grossen Nachteil, dass z.B.eine Menge oder ein Feld, das vordefiniert sein soll, nicht als Konstante deklariert werden kann.
In Standard-Pascal bleibt hier nur die Loesung, eine Variable zu verwenden und diese im Programm mit den gewuenschten Werten zu belegen.In Turbo Pascal koennen wir Typkonstanten verwenden.
Typkonstanten sind eigentlich keine Konstanten in dem Sinne, dass sie im Programm nicht mehr geaendert werden duerfen. Vielmehr sind es Variablen, die im Konstantendeklarationsteil mit einem Wert vorbelegt werden, aber im Programm durchaus als Variablen verwendet werden duerfen.

Beispiel:

```
CONST  Seitenzahl  :INTEGER = 200;
       Faktor      :REAL = 3.7;
       Piep        :CHAR = ^G;
       Wahr        :BOOLEAN = TRUE;
```

Wir sehen, dass Typkonstanten offenbar nach dem folgendem Syntaxdiagramm gebildet werden:



Im Programm laesst sich nun z.B. die Typkonstante Seitenzahl verwenden wie eine Variable dieses Namens, allerdings mit dem Unterschied, dass eine Variable keinen oder eine zufaelligen Inhalt hat, wenn das Programm startet und noch kein Wert zugewiesen wurde. Eine Typkonstante dagegen hat einen festen Anfangswert. Auch eine Zuweisung im Programm ist moeglich:

```
Seitenzahl := 210;
```

Achtung: Die Bezeichner der Typkonstanten duerfen natuerlich nicht noch einmal vergeben werden, insbesondere duerfen sie nicht noch einmal fuer eine Variablendeklaration verwendet werden. Ausserdem duerfen Typkonstanten nicht in Felddeklarationen verwendet werden, da sie keine Konstanten im eigentlichen Sinn sind.

So ist

```
CONST Max:INTEGER = 20;
VAR  Feld:ARRAY[1...Max] OF REAL;
```

nicht korrekt. Hier haette es heissen muessen:

```
CONST Max = 20;
```

Fuer Typenkonstanten sind nicht nur einfache Datentypen zulaessig, sondern auch zusammengesetzte.
Dies ist ein sehr leistungsfaehiges Werkzeug, da es haeufig vorkommt, dass Mengen, Felder oder Verbunde schon mit Anfangswerten belegt werden sollen.

eispiele:

```
TYPE Set_of_Char = SET OF CHAR;
   Family = ARRAY[1..4] OF STRING[10];
   Quadrat = ARRAY[1..2,1..3];

CONST Ende : STRING[4] = 'Ende';
   JaNein : Set_of_Char = ['J','j','N','n'];
   Familie : Family = ('Vater','Mutter','Sohn','Tochter');
   Magisch : Quadrat = ((4,9,2),(3,5,7),(8,1,6));
```

Mit

```
VAR i,j:INTEGER
```

ergibt die Programmsequenz

```
FOR i:=1 TO 3 DO BEGIN
  FOR j:=1 TO 3 DO WRITE (Magisch[i,j]:3);
  WRITELN
END;
```

dann die Ausgabe:

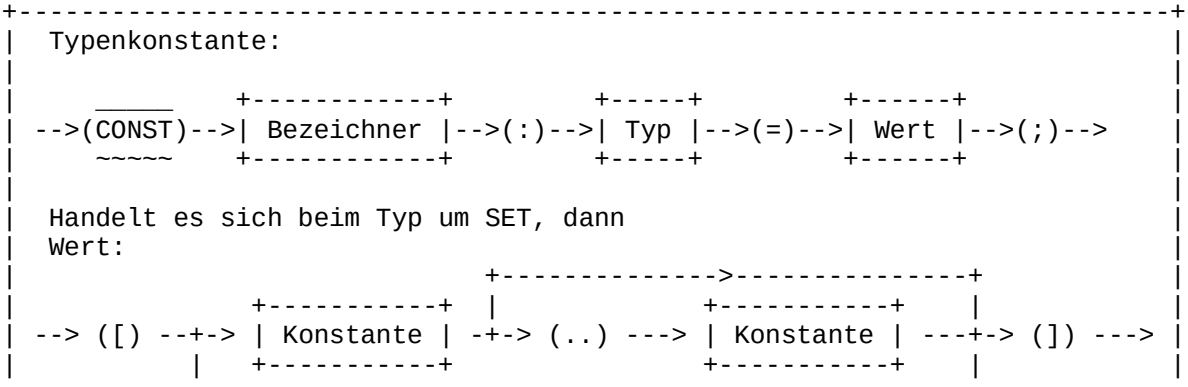
```
4  9  2
3  5  7
8  1  6
```

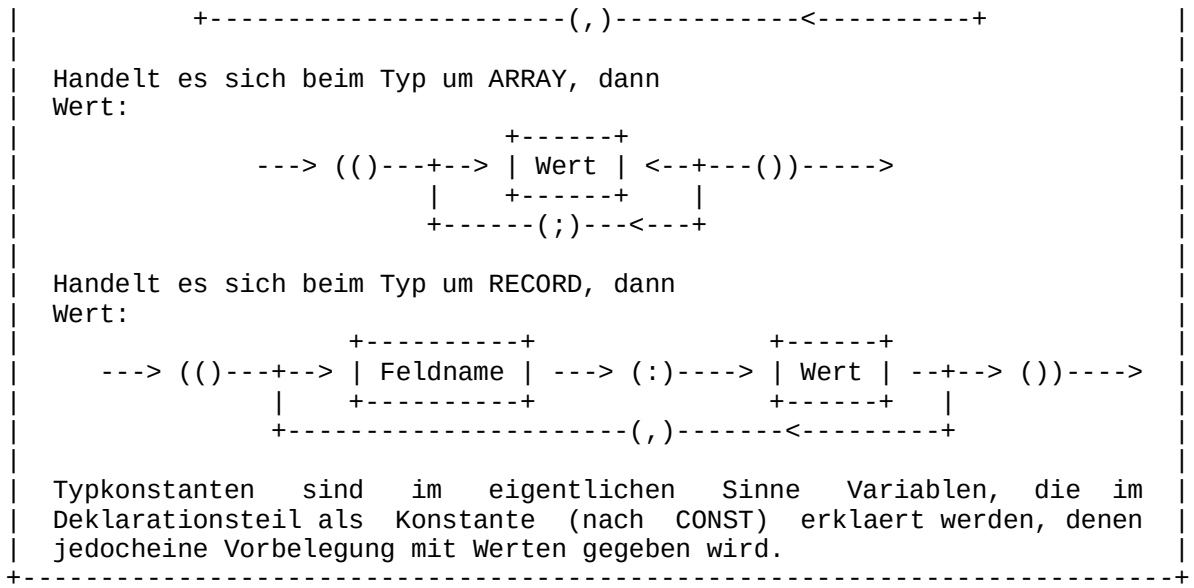
also ein magisches Quadrat.

Bei mehrdimensionalen Feldern ist darauf zu achten, dass sich die innere Klammer in der Feldbelegung auf den zuletzt deklarierten Index bezieht.
Auch Verbunde koennen so vorbelegt werden:

```
TYPE Kurzadresse = RECORD
                     Nummer:INTEGER
                     Name,Ort:STRING[20]
                   END;

CONST Leername:Kurzadresse = (Nummer:0;Name:'';Ort:'');
```

Die Zuweisung der Elemente des Verbundes geschieht in einer Klammer nach dem Gleichheitszeichen in der Reihenfolge der Deklaration der Elemente. Die Zuweisung der Werte geschieht mit einem Doppelpunkt als Trennsymbol. Die verschiedenen Zuweisungen werden durch Semikolon getrennt.





9.2 Zugriff auf Speicherstellen - Externe Prozeduren - Inline

In diesem Abschnitt wird es etwas spezieller, da wir uns hier mit der Verknuepfung von Pascal und maschinenabhaengigen Speicherstellen und Programmteilen beschaeftigen.
Hier sollen nur einige Hinweise auf die noetigen Turbo Pscal-Befehle gegeben werden. Fuer eine weitergehende Beschaeftigung mit der Maschinensprache wird auf die einschlaegige Literatur verwiesen. Auch die speziellen Speicherstellen eines Rechners haengen sehr vom verwendeten Modell ab, so das dazu auf die Handbuecher des Rechners vewiesen werden muss. Ausfuehrliche Erlaeuterungen mit Beispielen, differenziert nach 8- und 16-Bit-Rechnern, gibt das Handbuch zu Turbo Pascal.

Zugriff auf Speicherstellen

Der Zugriff auf Speicherstellen im Rechner wird von Standard-Pascal nicht unterstuetzt, da die Sprache rechnerunabhaengig sein sollte. In der Praxis jedoch haben wir oefter Anwendungen, bei denen wir auf solche Speicherstellen zugreifen muessen.

Hinweis: Der Zugriff auf rechnerspezifische Speicherstellen verhindert, dass das entsprechende Programm ohne Aenderungen auch auf jeden anderen Rechner lauffaehig ist. Daher sollte davon nur spaerlich Gebrauch gemacht werden.
Turbo Pascal besitzt ein vordefiniertes Feld namens MEM, das nichts anders darstellt als den Speicher des Rechners. Die Indizes des Feldes sind die Nummern der Speicherplaetze.

Auch hier muessen wir zwischen 8- und 16-Bit-System unterscheiden.

8-Bit-Systeme:
Das Feld MEM[i] mit dem Index i (INTEGER) und einem Dateninhalt vom Typ BYTE stellt die Zugriffsmoeglichkeit auf einen Speicherplatz dar. Mit i wird die Speicherstelle angegeben.

Beispiele:

```
MEM[$E000]:= 128;
```

```
MEM[28500]:= 63;
WRITELN (MEM[300]);
MEM[ist]:= MEM[ist+30]; mit ist:INTEGER
```

16-Bit-Systeme:
Das Feld MEM[s:i] mit den Indizes s und i (beide INTEGER) und einem Dateninhalt vom Typ BYTE erlaubt ein Lesen des angegebenen Speicherplatzes, wobei s die Segmentnummer und i die Speicherplatznummer angibt.
Zum Schreiben in den Speicher wird das Feld MEMW[s:i] verwendet.
MEMW schreibt einen INTEGER-Wert in die angegebene Speicherstelle, mit dem niederwertigen Byte zuerst.

Beispiele:

```
Wert := MEM[$0000:$0081];
WRITELN( MEM[$0000:$0010];
MEMW[$0000:$0081]:= Wert;
```

Externe Prozeduren

Auch externe Prozeduren und Funktionen lassen sich in ein Turbo Pascal-Programm einfuegen. Eine externe Prozedur ist ein Maschienenprogramm, das von Turbo aus aufgerufen werden soll.
Im Pascal-Programm hat eine externe Prozedur nur einen Prozedurkopf, der das reservierte Wort EXTERNAL enthaelt, aber keinen Prozedurrumpf. Dieser wird von dem entsprechenden Maschinenprogramm dargestellt.

Beispiele fuer 8-Bit-Systeme:

```
PROCEDURE Copydisk;EXTERNAL $E000:

PROCEDURE Grafinit (Seite:INTEGER);EXTERNAL $C000;

PROCEDURE Gross (VAR Wort:Kurzstring);EXTERNAL
$C0FF;(*Mit TYPE Kurzstring=STRING[80]*)

FUNCTION Druckeran : BOOLEAN;EXTERNAL $C100;
FUNCTION Addiere (a,b : INTEGER):INTEGER;EXTERNAL $C200;
```

Beispiele fuer 16-Bit-Systeme

```
PROCEDURE Copydisk;EXTERNAL'Copy';

PROCEDURE Grafinit (Seite:INTEGER);EXTERNAL'Grafik';

PROCEDURE Gross (VAR Wort:Kurzstring);EXTERNAL'Stuff';
(*Mit TYPE Kurzstring = STRING [80]*)

FUNCTION Druckeran : BOOLEAN;EXTERNAL'Druck';

FUNCTION Addiere (a,b : INTEGER):INTEGER;EXTERNAL'Math';
```

Bei der Verwendung eines 8-Bit-Rechners wird die Anfangsspeicherstelle der externen Prozedur angegeben, bei 16-Bit-Systemen wird der Name eines Maschinenprogramms auf der Diskette angegeben, das speicherplatzunabhaengig sein muss (relocatable).

Achtung: als erstes muss die externe Prozedur die Ruecksprungadresse (1 Wort

= 2 Bytes) vom Stack nehmen, um sie spaeter zum Prozedurende wieder auf den Stack zu legen. Bei 16-Bit-Systemen sind die Register BP, CS, DS und SS entsprechend zu retten.
Etwas gewoehnungsbeduerftig ist die Variablenuebergabe bei externen Prozeduren. Wir wollen uns nur hier mit einfachen Datentypen beschaeftigen. Dem fortgeschrittenen Benutzer sei das Handbuch empfohlen.
Werteparameter werden auf den Stack gelegt und koennen (nach dem die Ruecksprungadresse gerettet ist !) von der Prozedur von diesem herunter genommen werden.
Variablenparameter werden ebenfalls ueber den Stack uebergeben. Jedoch wird hier die absolute Speicherstelle auf den Stack gelegt, an der sich die entsprechende Variable befindet.
Die Parameteruebergabe hat fuer einfache Datentypen folgende Formate:
BYTE, CHAR, BOOLEAN: 1 Wort auf dem Stack. Das hoeherwertige Byte ist Null, weil diese Datentypen nur ein Byte Speicher brauchen.
INTEGER: 1 Wort auf dem Stack.
STRING: 1 Wort auf dem Stack. Darin ist die Anfangsadresse des Strings angegeben. In dieser Adresse steht die Laenge des Strings, die nachfolgenden Adressen enthalten die Zeichen des Strings.
Bei Funktionen ist folgendes zu beachten:
Eine Funktion hat stets ein Ergebnis. Dieses Ergebnis wird, wenn es von skalarem Typ ist (BYTE, CHAR, BOOLEAN, INTEGER), ueber das HL-Register bei 8-Bit-Rechnern oder ueber das AX-Register bei 16-Bit-Systemen zurueckgegeben. Ist das Ergebnis ein Byte lang, so steht im hoeherwertigen Byte eine Null.

Inline

Mit Hilfe des INLINE-Befehls lassen sich Maschinenprogrammanweisungen in den Programmfluss einbinden.

```
INLINE (<Codierungen>);
```

Die Codierung der Maschinenbefehle und Daten werden durch Schraegstriche voneinander getrennt.
Als Beispiel folgen zwei sehr aehnliche Prozeduren, die ein Wort in Grossbuchstaben bzw. Kleinbuchstaben wandeln. Dazu wird die Anfangsadresse des Wortes (STRING) in das HL-Register geladen. In dieser Adresse steht die Laenge des Wortes. Diese wird in das B-Register geladen (als Zaehlregister). In einer Schleife, in der das B-Register jeweils um eins vermindert wird, wird vom Wert des jeweiligen Zeichens $20 subtrahiert, sofern es sich um Kleinbuchstaben handelt (mit Codierungen zwischen $61 und $7B).
Analog funktioniert die Wandlung in Kleinbuchstaben.
In einem Programm sieht das so aus:

```
PROGRAM Inlinetest;

TYPE Wort = STRING [20];
VAR w : Wort;

PROCEDURE Wandel_klein (VAR word : Wort);
(* Wandelt ein Wort word in Kleinbuchstaben *)
BEGIN
 INLINE ($2A/word/     (*       LD  HL,(word)  ;Anfangsadresse  *)
         $46/          (*       LD  3,(HL)     ;Zaehlregister   *)
         $04/          (*       INC B          ;erhoehe B       *)
         $05/          (*weiter DEC B          ;vermindere B    *)
         $CA/*+20/     (*       JP  Z,ende     ;B=C dann ende   *)
```

```
          $23/          (*       INC HL          ;naechst. Zeichen*)
          $7E/          (*       LD  A,(HL)      ;in Accu         *)
          $FE/$41/      (*       CP  'A'         ;vergl. mit 'A'  *)
          $DA/*-9/      (*       JP  C,weiter    ;ist kleiner     *)
          $FE/$5B/      (*       CP  'Z'         ;vergl. mit 'Z'  *)
          $D2/*-14/     (*       JP  NC,weiter   ;ist groesser    *)
          $C6/$20/      (*       ADD 20H         ;addiere 20H     *)
          $77/          (*       LD  (HL),A      ;aendere Stelle  *)
          $C3/*-20 );   (*       JP  weiter      ;naechst. Zeichen*)
                        (* ende                                   *)
END;

PROCEDURE Wandel_gross (VAR word : Wort);
(* Wandelt ein Wort word in Grossbuchstaben *)
BEGIN
 INLINE ($2A/word/      (*       LD  HL,(word)   ;Anfangsadresse  *)
          $46/          (*       LD  B,(HL)      ;Zaehlregister   *)
          $04/          (*       INC B           ;erhoehe B       *)
          $05/          (*weiter DEC B           ;vermindere B    *)
          $CA/*+20/     (*       JP  Z,ende      ;B=0 dann ende   *)
          $23/          (*       INC HL          ;naechst. Zeichen*)
          $7E/          (*       LD  A,(HL)      ;in Accu         *)
          $FE/$61/      (*       CP  'a'         ;vergl. mit 'a'  *)
          $DA/*-9/      (*       JP  C,weiter    ;ist kleiner     *)
          $FE/$7B/      (*       CP  'z'         ;vergl. mit 'z'  *)
          $D2/*-14/     (*       JP  NC,weiter   ;ist groesser    *)
          $D6/$20/      (*       SUB 20H         ;subtrahiere 20H *)
          $77/          (*       LD  (HL),A      ;aendere Stelle  *)
          $C3/*-20 );   (*       JP weiter       ;naechst. Zeichen*)
                        (* ende                                   *)
END;

BEGIN (* Hauptprogramm *)
 WRITE ('Wort: ');
 READLN (w);
 Wandel_klein (w);
 WRITELN ('Klein: ',w);
 Wandel_gross (w);
 WRITELN ('Gross: ',w)
END.
```

9.3 Chain und Execute

Eine beliebte Programmiertechnik fuer groessere Programme besteht darin, von einem "Hauptprogramm" aus menuegesteuert andere Programme aufzurufen und abzuarbeiten.
Dazu bietet Turbo Pascal zwei Anweisungen: CHAIN und EXECUTE.

CHAIN

Mit der CHAIN-Anweisung wird ein anderes Turbo Pascal-Programm aufgerufen, das mit der cHn-Option des Compilers uebersetzt wurde. Im aufrufenden Programm wird eine Dateivariable vom Typ FILE erklaert, dann der Dateivariablen mit ASSIGN der Programmname des aufzurufenden Programms zugeordnet und zuletzt mit

```
CHAIN (Dateiname);
```

das Programm aufgerufen.

Auch eine Variablenuebergabe ist moeglich. Dazu muss das aufgerufene Programm einen zum aufrufenden Programm fast identischen Deklarationsteil haben. Und zwar muessen die gemeinsamen Variablen in beiden Programmen als erste und voellig identisch deklariert werden. Damit wird gewaehrleistet, dass das aufgerufene Programm arbeitet.

Ein Programmpaar dient als Beispiel:

```
PROGRAM Aufrufendes;
VAR Satz : STRING [80];
    next : FILE;

BEGIN
 WRITELN ('Geben Sie einen Satz ein: ');
 READLN (Satz);
 ASSIGN (next,'ZAEHLEN.CHN');
 CHAIN (next)
END.
```

Dieses Programm wird mit der Compileroption C als HAUPT.COM uebersetzt.

```
PROGRAM Aufgerufenes;
VAR Satz : STRING [80];   (* hier selbe Variable *)
    Gross, Klien, i : INTEGER;

BEGIN
  Gross := 0;
  Klein := 0;
  FOR i:=1 TO LENGHT (Satz) DO BEGIN
    IF Satz [i] IN ['A'..'Z'] THEN Gross:=Gross+1;
    IF Satz [i] IN ['a'..'z'] THEN Klein:=Klein+1;
  END;
  WRITELN ('Der Satz hat ',Gross:3,' Grossbuchstaben');
  WRITELN ('         und ',Klein:3,' Kleinbuchstaben')
END.
```

Dieses Programm wird mit der Compileroption H als ZAEHLEN.CHN uebersetzt. Nun kann das Programm HAUPT.COM aufgerufen und abgearbeitet werden. Beide Programme "teilen" sich die Variable Satz. Sie stellt sozusagen eine globale Variable dar.
Hinweis: Ein CHN-Programm braucht wesentlich weniger Speicherplatz als ein COM-Programm,weil wesentliche Teile des Betriebssystems nicht mit uebersetzt wurden. Daher ist es selbst auch nicht lauffaehig, sondern muss von einem anderen Turbo-Programm aufgerufen werden.

EXECUTE

Mit der EXECUTE-Anweisung koennen beliebige Programme, die als COM-File auf der Diskette zugreifbar sind, aufgerufen werden. Hier koennte man sich vorstellen, dass kompilierte BASIC-Programme oder andere Fremdprogramme benutzt werden.
EXECUTE wird genauso verwendet wie CHAIN. Zuerst muss mit ASSIGN einem Dateinamen eine physische Datei auf der Diskette zugewiesen werden. Dann wird mit

```
EXECUTE (Dateiname);
```

das Programm aufgerufen.
Als Beispiel wollen wir ein Programm das Tubo-System aufrufen lassen.

```
PROGRAM Executetest;
VAR Datei : FILE;

BEGIN
  WRITELN ('Nach Return wird Turbo gestartet.');
  WRITELN ('Druecken Sie RETURN...');
  READLN;
  ASSIGN (Datei,'TURBO.COM');
  EXECUTE (Datei)
END.
```

Aehnlich koennten Dienstleistungsprogramme gestartet werden oder nach dem obigen Konzept eine " Hauptprogramms " mehrere Programme menuegesteuert abgearbeitet werden.
Ein " Hauptprogramm " ruft ein anderes Programm nach Benutzerwahl auf, das an seinem Ende wiederum das " Hauptprogramm " aufruft.

9.4 Include

Ein sehr nuetzliches Programmierwerkzeug stellt die Compileranweisung

```
{$I...} oder (*$I...*)
```

dar. Mit ihr kann der Programmierer bestehende Programmtexte aus anderen Textdateien in den gerade zu bearbeitenden Text einfuegen. Das kann aus zwei verschiedenen Gruenden notwendig sein:

1. Der Programmtext wird zu lang und kann daher nicht als ganzer im Rechner verarbeitet werden.

2. Es existieren schon Programmteile, wie z.B. Typdeklarationen fuer eine Grafik, fertige Prozeduren aus einer Art Prozedurbibliothek usw., die in einem anderen Progammtext wiederverwendet werden sollen, ohne sie direkt in den Text aufzunehmen.

Um einen Text in einen Programmtext zur Zeit des Uebersetzens einfuegen zu lassen, muss folgede Programmzeile benutzt werden:

```
{$I<Textdatei>} oder (*$I<Textdatei>*)
```

Dann wird die so spezifizierte Textdatei vom Compiler waehrend des Uebersetzens eingefuegt.
Achtung: Die Compileranweisungen sehen zwar aus wie Kommentar, stellen aber keinen dar.

Beispiele:

```
PROGRAM Test;

(*$I 'B:DEKLARATIO.PAS'*)
(*$I 'A:PROC1.PAS*)
(*$I 'B:PROC2.PAS*)

BEGIN
  Eingabe; (* steht in PROC1 *)
  Rechne; (* steht in PROC2 *)
```

```
    Ausgabe (* steht in PROC1 *)
  END.
```

Nach dem Beispiel muss es die Textdatei PROC1.PAS auf der Diskette in Laufwerk A und die Textdateien DEKLARATIO.PAS und PROC2.PAS auf der Diskette in Laufwerk B geben. In DEKLARATIO.PAS stehen sehr wahrscheinlich Deklarationen, waehrend in PROC1.PAS die Prozeduren der Eingabe und Ausgabe und in PROC2.PAS der Text der Prozedur Rechne stehen.

9.5 Overlays

Dem fortgeschrittenen Programmierer kann es schon einmal passieren, dass sein Programm zu gross wird. Handelt es sich nur um ein zu grosses Textfile, so schafft die Include-Anweisung des Compilers Abhilfe (siehe Kap.9.4).
Passt aber das Codefile, d.h. das uebersetzte Programm, nicht mehr in den Speicher, so ist auf eine andere Technik zurueckzugreifen: Overlays.
Hierbei handelt es sich um eine recht raffinierte Art von Prozeduren, die sich gemeinsam ein und den selben Speicherplatz teilen. Dazu muessen sie allerdings auf der Diskette " parken " und werden nur fuer den Fall ihres Aufrufs von der Diskette in den Rechner geholt. Das kostet natuerlich mehr Zeit, als speicherresidente Prozeduren zu verwenden. Daher sind einige Regeln zu beachten, die nach der Erklaerung dieser Struktur folgen.
Im Prozedur- oder Funktionskopf steht das reservierte Wort OVERLAY vor dem Wort PROCEDURE oder FUNCTION. Dann handelt es sich um eine entsprechende Overlay-Prozedur oder -Funktion.

```
OVERLAY PROCEDURE <Titel><Parameterliste>;
OVERLAY FUNCTION <Titel><Parameterliste>:<Ergebnistyp>;
```

Trifft der Compiler waehrend des Uebersetzens auf einen entsprechenden Prozedur-/Funktionkopf, so wird der uebersetzte Maschinencode nicht mehr in das Programm, sondern in eine Overlay-Datei uebersetzt, die den Namen des Programm-Codefiles hat, jedoch mit Suffix .000 bis .099. Als Einschraenkung muss erwaehnt werden, dass Programme, die Overlays enthalten, nur mit den Compileroptionen C und H uebersetzt werden duerfen.

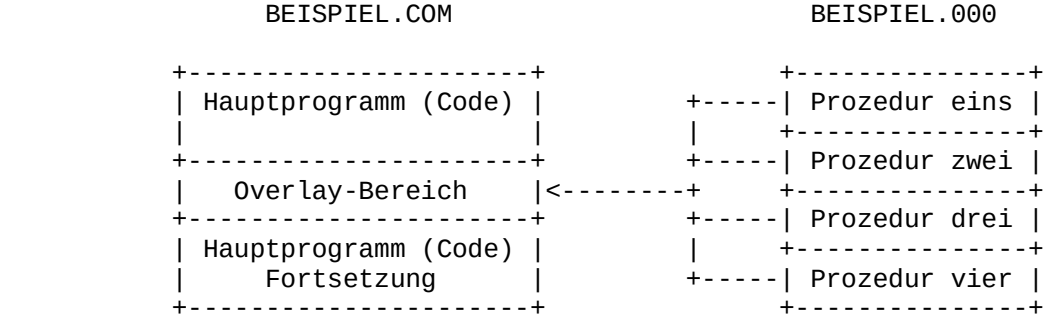
Beispiel:

```
PROGRAM Overtest1;
...
OVERLAY PROCEDURE eins;
 BEGIN
 ...
 END;

OVERLAY PROCEDURE zwei;
 BEGIN
 ...
 END;

OVERLAY PROCEDURE drei;
 BEGIN
 ...
 END;

OVERLAY PROCEDURE vier;
 BEGIN
```

```
 ...
 END;

BEGIN (* Hauptprogramm *)
...
END.
```

In diesem Beispielprogramm teilen sich die vier Prozeduren ein und den selben Speicherplatz im Programmcode, und zwar sinnvollerweise so, dass Platz fuer die groesste der vier Prozeduren freigehalten wird. Nehmen wir an, das Programm wird als BEISPIEL.COM uebersetzt; dann befinden sich alle vier Prozeduren in dem File BEISPIEL.000.

Waehrend des Programmlaufs wird jeweils die Prozedur von der Diskette in den Speicher geladen, die gerade gebraucht wird, weil sie aufgerufen wurde. Die Prozedur bleibt im Speicher verfuegbar, bis eine andere Overlay-Prozedur dieser Overlay-Datei aufgerufen wird.
In einer Grafik stellt sich das wie folgt dar:

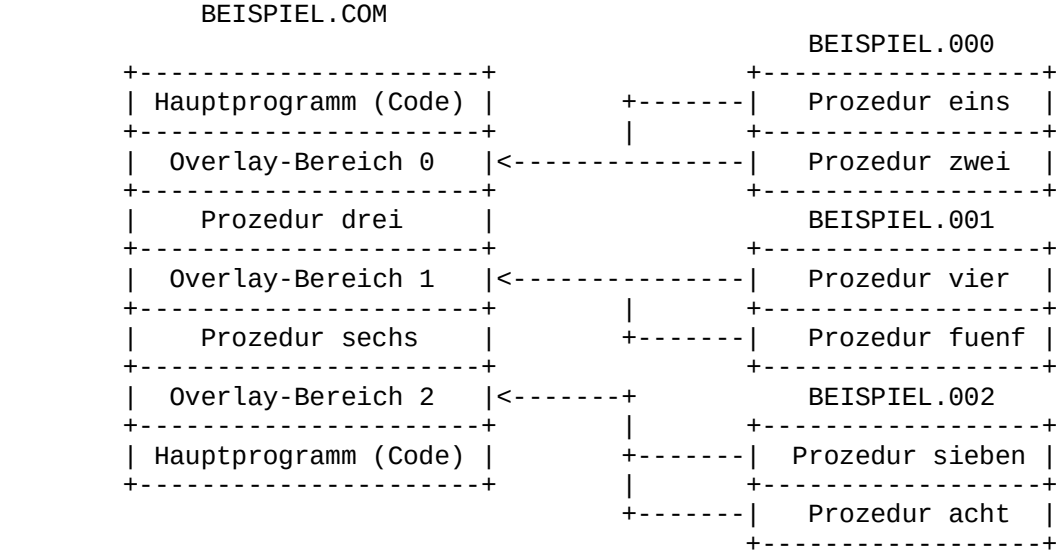


Overlay-Prozeduren und Funktionen, die nacheinander im Text auftreten, werden in dasselbe Overlay-File geschrieben. Liegen zwischen mehreren Overlay-Bereichen normale Prozeduren, so werden Overlay-Files mit unterschiedlichen Nummern angelegt (Maximum 100).
Beispiel:

```
PROGRAM Overtest2;
...
OVERLAY PROCEDURE eins;
 BEGIN
 ...
 END;

OVERLAY PROCEDURE zwei;
 BEGIN
 ...
 END;

PROCEDURE drei;
 BEGIN
 ...
 END;

OVERLAY PROCEDURE vier;
 BEGIN
 ...
 END;
```

```
OVERLAY PROCEDURE fuenf;
 BEGIN
 ...
 END;

PROCEDURE sechs;
 BEGIN
 ...
 END;

OVERLAY PROCEDURE sieben;
 BEGIN
 ...
 END;

OVERLAY PROCEDURE acht;
 BEGIN
 ...
 END;

BEGIN (* Hauptprogramm *)
 ...
 END.
```

In einer Grafik stellt sich das wie folgt dar:



In diesem Beispiel werden also drei Dateien mit den Namen BEISPIEL.000 , BEISPIEL.001 und BEISPIEL.002 angelegt, die den uebersetzten Code der jeweiligen Prozeduren enthalten. Im Programm wird fuehr jeden Overlay-Bereich soviel Platz freigehalten, dass die jeweils groesste Prozedur hineinpasst.
Eine Erweiterung des Overlay-Konzepts stellen die geschachtelten Overlays dar. Jede Overlay-Prozedur oder Funktion kann selbst wieder wie ein Hauptprogramm gestaltet werden, d.h. Overlays enthalten. Das bedeutet, dass in dem Programmcode der Overlay-Prozedur wieder Platz gelassen wird fuer einen weiteren Overlay-Bereich.
Von welcher Diskette liest das Programm die Overlays ? - Normalerweise

werden sie von dem Laufwerk gelesen, das gerade in Betrieb ist (aktuelles Laufwerk). Wird ein anderes Laufwerk gewuenscht, so kann dies mit Hilfe einer entsprechenden Compiler-Anweisung geschehen.
Mit

{$OA} oder (*$OA*)

wird z.B. das Laufwerk A: spezifiziert als das Laufwerk, von dem ab dieser Stelle die Overlays gelesen werden.
Mit

{$O@} oder (*$O@*)

wird das gerade aktuelle Laufwerk gewaehlt.
In der Version 3.0 wird statt der Compiler-Anweisung eine Standardprozedur namens OVRDRIVE (bzw.OVRPATH) benutzt. Die Compiler-Anweisung (*$O...*) wird in dieser Version nicht mehr unterstuetzt !

Bei 8-Bit-Systemen wird mit

OVRDRIVE (Laufwerknummer);

ein Laufwerk spezifiziert, von dem Overlays zu lesen sind. Die Nummer 0 bedeutet das aktuelle Laufwerk, 1 bedeutet A:, 2 bedeutet B: usw. Waehrend des Programms kann die Laufwerksnummer mehrfach geaendert werden.

Bei 16-Bit-Systemen wird mit

OVRPATH (Pfadname);

ein Subdirectory-Pfad als STRING-Parameter spezifiziert.

Wichtige Hinweise zu Overlays:

- Im Programm-Code wird soviel Platz freigehalten, wie die groesste Overlay-Prozedur des entsprechenden Overlay-Bereichs benoetigt.
- Zusammenhaengende Overlay-Bereiche werden in derselben Overlay-Datei abgespeichert. Sind Overlay-Bereiche durch normale Prozeduren voneinander getrennt, so werden verschiedene Overlay-Dateien angelegt (numeriert von 000 bis 099). Dadurch vergroessert sich natuerlich der Platzbedarf, die Ladezeiten fuer Overlay-Prozeduren verringern sich jedoch, da mehr Overlay-Prozeduren gleichzeitig praesent sind.
- Overlay-Prozeduren duerfen nicht FORWARD deklariert werden.
- Overlay-Prozeduren duerfen nicht rekursiv sein.
- Prozeduren, die oft aufgerufen werden, sollten keine Overlay-Prozeduren sein (insbesondere Prozeduren, die in Schleifen aufgerufen werden), da das Einladen von der Diskette erheblich mehr Zeit benoetigt als Zugriffe im Rechnerspeicher.
- Programme, die Overlays enthalten, duerfen nicht mit der M-Option des Compilers uebersetzt werden.

9.6 TLIST

Es gibt grundsaetzlich zwei verschiedene Moeglichkeiten, einen Programmtext und andere Texte auf den Drucker zu bringen.
1. Vom CP/M-System aus wird mit CTRL-P der Drucker eingeschaltet und mit
   TYPE<Name der Textdatei mit Suffix>
   die Textdatei gelistet. Anschliessend wird mit CTRL-P der Drucker wieder ausgeschaltet.
2. Man verwendet ein Ausdruckprogramm.

Auf der Turbo Pascal-Diskette befindet sich, sozusagen als Zugabe, ein entsprechendes Programm namens TLIST.
Wird TLIST aufgerufen, so meldet es sich mit

```
   Enter name of file to list (?) for help <Return> to skip:
```

Hier geben wir ein:
   Name der zu listenden Textdatei oder
   ?,   um Hilfen zu bekommen, oder
   <Return>, um das Programm zu verlassen.
Haben wir einen Dateinamen eingegeben (wie BEISPIEL.PAS), so werden wir nach Optionen gefragt. Hier koennen wir L,M oder LM eingeben
   L : Der Text wird mit Zeilennummern ausgegeben.
   M : Reservierte Woerter werden markiert (d.h. unterstrichen).
   LM: Beides.
   Keine Eingabe : Weder noch.
Dann wird der Text gedruckt.

Druckparameter im Text

In den Text der ausgedruckt werden soll, koennen Parameter eingefuegt werden, die das Ausdruckformat steuern. Die Anweisungen werden als Kommentare in den Text eingebaut, die in der ersten Spalte der Zeile anfangen muessen. Hier bedeutet:

{.PLn} oder (*.PLn*) :	Die Seitenlaenge soll n Zeilen betragen.
{.CPn} oder (*.CPn*) :	Seitenvorschub, wenn auf der aktuellen Seite kein Platz fuer n Zeilen mehr vorhanden ist.
{.PA} oder (*.PA*) :	Seitenvorschub
{.POn} oder {*.POn*) :	Einstellen des linken Randes auf n Zeichen.
{.HEtext} oder (*.HEtext*) :	Text, der in die Kopfzeile jeder Seite gedruckt werden soll.
{.FOtext} oder (*.FOtext*) :	Dito fuer Fusszeile.
{.L-} oder (*.L0*) :	Textlisting abschalten.
{.L+} oder (*.L+*) :	Textlisting einschalten.

Include-Files werden ebenfalls mit dem Programmtext gelistet, wenn die Include-Anweisung in der ersten Spalte der Zeile steht.
Mit / kann der Text der Kopf- und Fusszeile mehrzeilig sein.
Steht im Text der Kopf- oder Fusszeile ein #-Zeichen, so stellt dies die Stelle dar, an der die Seitennummer gedruckt wird.

9.7 Fehlerbehandlung ab Version 3.0

Ab der Versionsnummer 3.0 des Turbo Pascal-Systems eroeffnet sich dem Programmierer eine voellig neue Fehlerbehandlungs- und -erkennungsmoeglichkeit. Bislang wurde das Programm einfach abgebrochen und eine entsprechende Fehlererkennung des Systems an den Benutzer ausgegeben. Die Aussagen zum aufgetretenen Fehler sind englisch und mit Codierungen versehen, so dass ein unkundiger Benutzer damit nichts anfangen kann. Ausserdem sind eventuell geoeffnete Dateien beim auftreten eines Fehlers verloren, da sie nicht geschlossen werden. Mit der der neuen Moeglichkeit kann der Programmierer dafuer sorgen, dass dem Benutzer eine Fehlermeldung in Klartext ausgegeben wird und dass alle Dateien geschlossen werden, bevor das Programm abbricht.
Dazu wird im Programm eine Prozedur zur Fehlerbehandlung geschrieben, deren Adresse einem Fehler-Pointer des Systems namens ERRORPTR uebergeben wird. Die Prozedur muss zwei Werteparameter vom Typ INTEGER besitzen, denen vom System (in dieser Reihenfolge) die Fehlernummer und die Fehleradresse uebergeben wird. Die Fehlernummer teilt sich auf in die Codierung fuer den Feh-

lertyp (hoeherwertiges Byte) und die Fehlernummer selbst (niederwertiges Byte). Beim Fehlertyp kann es sich handeln um:

        0 : Benutzer-Unterbrechung mit CTRL-C
        1 : Ein-/Ausgabefehler
        2 : Laufzeitfehler

Im folgenden sehen wir ein kleines Demonstrationsprogramm, das auf eine Fehlermeldung im Klartext verzichtet, da dazu eine aufwendige Fallunterscheidung noetig waere. Hier soll nur das Prinzip der Fehlererkennung deutlich gemacht werden.
In dem Programm werden zwei vom Benutzer einzugebende ganze Zahlen durcheinander geteilt. Fehler koennen dadurch simuliert werden, dass beim Programmlauf die zweite Zahl gleich Null ist (Laufzeitfehler:Teilen durch Null), der Benutzer das Programm durch CTRL-C unterbricht oder ein Buchstabe anstelle einer Zahl eingegeben wird (Ein-Ausgabefehler).

```
PROGRAM Errortest;

VAR i,j, : INTEGER;

PROCEDURE Error (ErrNo, ErrSdr : INTEGER);
BEGIN
 WRITELN ('Fehler!');
 WRITELN ('Typ: ',HI(ErrNo));
 WRITELN ('Nummer: ',LO(ErrNo));
 WRITELN ('Adresse: ',ErrAdr);
 HALT
END;

BEGIN
 ERRORPTR := ADDR(Error);
 WRITE ('Zahl: '); READLN (i);
 WRITE ('geteilt durch: '); READLN (j)
 WRITELN ('gleich ',i/j:10:4)
END.
```

Die Namen der Prozedur sowie der beiden Werteparameter sind gleichgueltig.

Bei 16-Bit-Systemen muss es statt ERRORPTR := ADDR(Error); dann analog heissen

```
        ERRORPTR := OFS (Error);
```

10.1 Sortieren mit Turbo Sort

Zusaetzlich zum Turbo Pascal bietet die Herstellerfirma ein Programmpaket namens Turbo Toolbox an, das zwei voneinander unabhaengige Programmhilfen umfasst: Turbo Sort und Turbo Access.
Bei Turbo Sort handelt es sich um Hilfsprogramme zum komfortablen Sortieren, waehrend Turbo Access Routinen zur Dateiverwaltung mit sogenannten B-Baeumen beinhaltet.
Turbo Sort umfasst einige Prozeduren, die das Sortieren eines Datensatzes nach dem Quicksort-Algorithmus erlaubt. Der Benutzer ist weitgehend frei in der Wahl der zu sortierenden Daten. Auch braucht er sich nicht um die Verwaltung des Speichers waehrend des Sortierens zu kuemmern. Maximal koennen 32767 Datensaetze sortiert werden.
Passt der ganze Datensatz in den Speicher, so wird im Speicher sortiert, anderenfalls dient die Diskette als virtueller Speicher.
Turbo Sort ist eine Textdatei, die die zu verwendenden Prozeduren und Funktionen sowie einen Deklarationsteil enthaelt. Im eigenen Programm wird es vor dem Aufruf einer Turbo Sort-Prozedur mit

```
{$IA : SORT.BOX}
```

in den Programmtext eingefuegt. Dazu muss die Textdatei SORT.BOX waehrend des Uebersetzens auf der Diskette im Laufwerk A sein. Anderenfalls ist ein anderes Laufwerk anzugeben.
Der Sortiervorgang wird durch den Aufruf einer Funktion begonnen:

```
x := TurboSort(ItemSize);
```

wobei x eine Variable vom Typ INTEGER ist und ItemSize ein ganzzahliger Wert, der die Groesse der Datensaetze angibt. Um ItemSize zu berechnen, benutzen wir einfach die Standardfunktion SIZEOF. Nehmen wir an, unsere Daten seien vom Typ

```
TYPE Datentyp = RECORD
                  Name,Vorname : STRING[20];
                  Alter        : INTEGER
                END;
```

und

```
VAR Feld : ARRAY[1..1000] OF Datentyp;
       x : INTEGER;
```

Dann laesst sich Turbo Sort aufrufen mit

```
x := TurboSort(SIZEOF(Datentyp));
```

Allerdings funktioniert das Sortieren so noch nicht. Zuerst muessen die Daten an die Sortierprozedur uebergeben werden. Das geschieht mit der Prozedur Inp, die im Include-Text FORWARD deklariert ist, so dass eine Parameterliste entfaellt.

```
PROCEDURE Inp;
```

```
BEGIN
   { Hier die Eingabe der Daten. Z.B.:}
   FOR x := 1 TO 1000 DO BEGIN
      WITH Feld[x] DO BEGIN
         WRITE ('Name: ');
         READLN (Name);
         WRITE ('Vorname: '); READLN (Vorname);
         WRITE ('Alter: '); READLN (Alter)
      END;
      {Ende der Eingabe eines Datenelements}
      SortRelease (Feld[x];
   END;
END;
```

Mit dem Aufruf der Prozedur SortRelease wird ein Datenelement des deklarierten Datentyps an den Sortierteil uebergeben.
Natuerlich haetten wir auch die Datenelemente aus einer Datei lesen koennen. Weiterhin ist anzugeben, nach welchen Kriterien sortiert werden soll. Nehmen wir an, es sei nach dem Namen zu sortieren, so erklaeren wir durch die FORWARD deklarierte Funktion Less:

```
FUNKTION Less;
VAR erst  : Datentyp ABSOLUTE X;
    zweit : Datentyp ABSOLUTE Y;
BEGIN
   Less := erst.Name < zweit.Name
END;
```

Die Absolute Adresse X und Y sind von Turbo Sort vordefiniert. Haetten wir eine Sortierung nach dem Alter gewuenscht, so muesste es heissen:

```
Less := erst.Alter < zweit.Alter
```

Als letztes ist eine Ausgabeprozedure zu schreiben, die festlegt, wie die Ausgabe geschehen soll.
Nehmen wir an, die sortierten Daten sollen wieder in das bestehende Feld eingeordnet werden. Dann koennte es heissen:

```
PROCEDURE OutP;
BEGINN
   x := 1;
   REPEAT
      SortReturn (Feld[x]);
      x := x+1
   UNTIL SortEOS
END;
```

Auch die Prozedur OutP ist FORWARD deklariert und braucht von uns nur noch ohne Parameterliste geschrieben werden.
Durch den Aufruf der Prozedure SortReturn werden die sortierten Datenelemente in ihrer Reihenfolge an eine vom Benutzer zu spezifizierende Variable uebergeben. Eine logische Funktion SortEOS erhaelt das Ergebnis TRUE, wenn alle Daten ausgegeben wurden.
Nun folgt ein komplettes Programmbeispiel, das ein Feld von Verbunden sortieren soll, dessen Elemente je eine zufaellige Zahl und ein zufaelliges Zeichen enthalten. Der Benutzer kann angeben , ob er nach der Zahl (Nummer) oder nach dem Zeichen sortiert haben moechte. Die unsortierten Daten werden

in der 10. Spalte, die sortierten in der 20. Spalte des Bildschirmes unter-
einander ausgegeben.

```
PROGRAM Sortbox_Demo;

CONST Max = 15;

TYPE Datentyp = RECORD
                 Zahl : INTEGER;
                 Zeichen : CHAR;
                END;

VAR a : ARRAY [1..Max] OF Datentyp;
    ch : CHAR;
    istZahl : BOOLEAN;

{$I B:SORT.BOX}

PROCEDURE Inp;
VAR i : INTEGER;
BEGIN
 FOR i := 1 TO Max DO BEGIN
  a[i].Zahl := RANDOM(100);
  a[i].Zeichen := CHR(32+RANDOM(96));
  GOTOXY(10,i);WRITE (a[i].Zahl:5,a[i].Zeichen:3);
  SortRelease(a[i])
 END
END;

PROCEDURE OutP;
VAR i : INTEGER;
BEGIN
 i := 1;
 REPEAT
  SortReturn(a[i]);
  GOTOXY(20,i);
  WRITELN(a[i].Zahl:5,a[i].Zeichen:3);
  i := i+1;
 UNTIL SortEOS;
 GOTOXY(30,1);
 WRITE('Return druecken...');
 READLN
END;

FUNKTION Less;
VAR erst : Datentyp ABSOLUTE X;
    zweit : Datentyp ABSOLUTE Y;
BEGIN
 IF istZahl THEN Less := erst.Zahl < zweit.Zahl
            ELSE Less := erst.zeichen < zweit.Zeichen
 END;

 BEGIN (* Hauptprogramm *)
  CLRSCR;
  WRITE('Sortieren nach N(ummer oder Z(eichen?');
  REPEAT
   READ(KBD,ch)
  UNTIL ch IN ['n','N','z','Z'];
```

```
  istZahl := ch IN ['n','N'];
  CLRSCR;
  WRITELN(TurboSort(SIZEOF(Datentyp)))
 END.
```

Als Alternative koennte das Programm dahingehend geaendert werden, dass nach beiden Kriterien (Nummer und Zeichen) geordnet wird. Dazu wird die Sortierprozedure zweimal mit geaenderter Sortieranweisung in der Funktion Less aufgerufen.

```
PROGRAM Sortbox_Demo;

CONST Max = 20;

TYPE Datentyp = RECORD
                 Zahl : INTEGER;
                 Zeichen : CHAR;
                END;

VAR a : ARRAY[1..Max] OF Datentyp;
    ch : CHAR;
    Dummy,y : INTEGER;
    istZahl : BOOLEAN;

{$I b:SORT>BOX}

PROCEDURE Erzeuge;
VAR i : INTEGER;
BEGIN
 FOR i := 1 TO Max DO BEGIN
  a[i].Zahl := RANDOM(100);
  a[i].Zeichen := CHR (32 + RANDOM(96));
  GOTOXY(10,i); WRITE (a[i].Zahl:5,a[i].Zeichen:3);
 END;
END;

PROCEDURE Inp;
VAR i : INTEGER;
BEGIN
 FOR i := 1 TO Max DO
  SortRELEASE(a[i])
END;

PROCEDURE OutP;
VAR i : INTEGER;
BEGIN
 i := 1;
 REPEAT
  SortReturn (a[i]);
  GOTOXY(y,i);
  WRITELN (a[i].Zahl:5, a[i].Zeichen:3);
  i := i + 1;
 UNTIL SortEOS;
 IF y = 30 THEN BEGIN
  GOTOXY(40,1);
  WRITE ('Return druecken...');
  READLN
```

```
 END
END;

FUNKTION Less;
VAR erst : Datentyp ABSOLUTE X;
    zweit : Datentyp ABSOLUTE Y;
BEGIN
 IF istZahl THEN Less := erst.Zahl < zweit.Zahl
            ELSE Less := erst.Zeichen < zweit.Zeichen
END;

BEGIN {Hauptprogramm}
 CLRSCR;
 Erzeuge;
 y := 20; istZahl := TRUE;
 Dummy := TurboSort(SIZEOF(Datentyp));
 y := 30; istZahl := FALSE;
 Dummy := TurboSort(SIZEOF(Datentyp))
END.
```

Das angebene Beispiel ist sicher keine sehr nuetzliche Anwendung des Turbo Sort-Paketes. Es soll nur die Funktion dieser Programmierhilfe erlaeutern. In der Praxis werden die Datensaetze sicher etwas anders aussehen. Jedoch bleibt das Prinzip gleich.
Zur Uebung koennten sie das Adressprogramm aus Kap. 8.1 mit Hilfe von Turbo Sort ueberarbeiten.

Hinweis: Die Funktion TurboSort hat ein Ergebnis, das Aufschluss ueber eventuelle Fehler beim Sortieren gibt:

```
 0 : Sortieren erfolgreich.
 3 : Nicht genug Speicher vorhanden.
 8 : Unzulaessige Laenge des Datensatzes (muss groesser 1 ).
 9 : Mehr als 32767 Datensaetze.
10 : Fehler beim Sortieren (Diskette defekt oder voll).
11 : Fehler beim Lesen von der Diskette.
13 : Fehler beim Erzeugen einer Datei.
```

10.2 B-Dateien mit Turbo Access

Der zweite grosse Bereich, den die Turbo Toolbox abdeckt, ist eine effektive Dateiverwaltung. Dazu verwendet man das Turbo Access-System. Es beinhaltet mehrere Textdateien, die als Include-Files in das eigene Programm eingebaut werden und eine Reihe sehr effektiver Prozeduren und Funktionen zur Verfuegung stellen.

```
ACCESS.BOX  enthaelt alle Prozeduren zur Verwaltung der eigentlichen Datei
            sowie grundlegende Deklarationen.
GETKEY.BOX  enthaelt Suchroutinen fuer eine Indexdatei.
ADDKEY.BOX  enthaelt Routinen zum Einfuegen von Schluesseln in die Index-
            datei.
DELKEY.BOX  enthaelt Routinen zum Loeschen von Schluesseln in der Index-
            datei.
```

Das Softwarepaket unterstuetzt nicht nur die Verwaltung und Pflege einer Datendatei, sondern zudem noch den Aufbau und die Verwaltung von Indexdateien. Das bedeutet, dass der Benutzer in mehreren Indexdateien Schluesselwoer-

ter verwalten kann, die zum Suchen in der Datei verwendet werden. So kann z.B. mit einem Datensatz Adresse einer Datendatei erstellt und verwaltet werden, und zum Suchen nach bestimmten Kriterien werden Indexdateien eingerichtet, in denen fuer jeden Dateneintrag z.B. Name, Postleitzahl, Ort und Kundennummer verwaltet werden. So kann dem Benutzer des endgueltigen Programmes gestattet werden, nach diesen Kriterien im Datensatz zu suchen, ohne dass der ganze Datensatz von Anfang bis Ende durchsucht werden muss. Vielmehr wird nur in der Indexdatei gesucht, die die Form eines B-Baumes hat, und nach dem erfolgreichen Suchen die Nummer des gesuchten Datensatzes ausgegeben, der nun aus der Datei gelesen werden kann.
Eine kurze Darstellung soll verdeutlichen, welche Vorteile B-Baeume in diesem Zusammenhang haben. Im Gegensatz zu Binaerbaeumen haben die Elemente eines solchen Baumes mehr als zwei Verzweigungen. Die Knoten eines solchen Baumes heissen Seiten. Sie beinhalten Elemente (namens Schluessel) und Verknuepfungen zu weiteren Elementen.

Ein Baum der n. Ordnung hat dann folgende Eigenschaften:

1. Jede Seite enthaelt hoechstens 2*n Elemente (Schluessel).

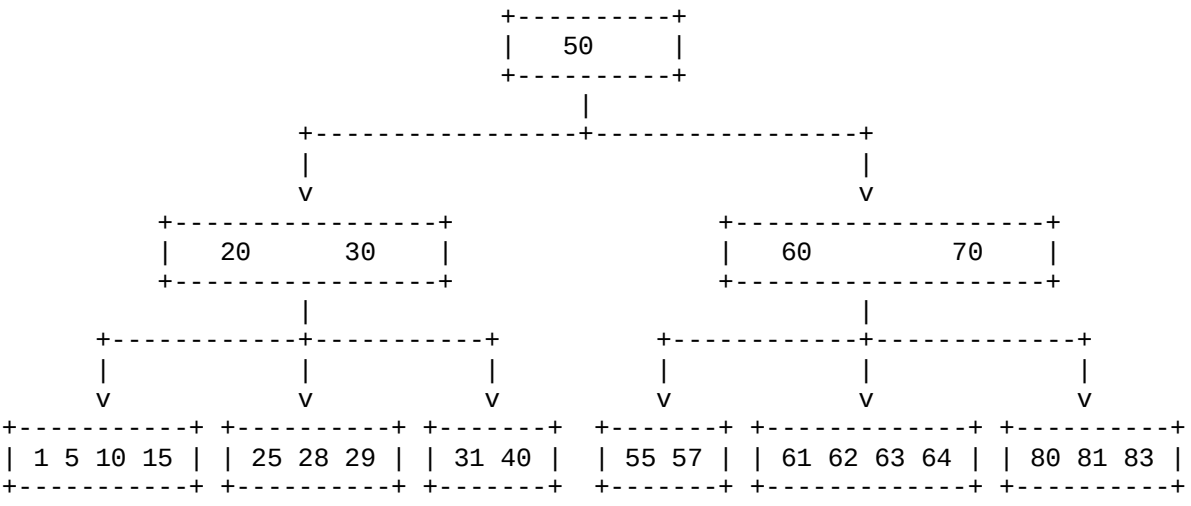
2. Jede Seite, ausser der Wurzelseite, enthaelt mindestens n Elemente.

3. Jede Seite ist entweder eine Blattseite, d.h. hat keinen Nachfolger,
   oder sie hat m+1 Nachfolger, wobei m die Zahl ihrer Schluessel ist.

4. Alle Blattseiten liegen auf der gleichen Stufe.
   (Nach Wirth: Algorithmen und Datenstrukturen)

Auch in einem B-Baum sind die Elemente durch die Einfuegevorschrift bedingt geordnet. Weiterhin besteht der Vorteil, dass in einem B-Baum durch die o.g. Vorschriften die Ebenen des Baumes stets ausgeglichen sind. Wird eine Seite aufgefuellt, so muss sie geteilt werden, wenn sie mehr als 2*n Elemente enthaelt. Andererseits muessen zwei Seiten wieder zu einer zusammengefuegt werden, wenn die Anzahl der Elemente durch Loeschen n unterschreitet. Durch die groessere Anzahl der Verzweigungen vermindert sich der durchschnittliche Suchweg im Baum. Allerdings vergroessert sich bei zunehmender Ordnung n der Verwaltungsaufwand.

Ein Beispeil fuer einen B-Baum 2. Ordnung, dessen Schluessel ganze Zahlen sind, ist:



Auf die Entwicklung und Erlaeuterung der Einfuege- und Loeschoperationen soll hier verzichtet werden, da sie fuer die Benutzung von Turbo Access unerheblich sind.
Vielmehr sollte sich der Benutzer nur im klaren darueber sein, was in solcher Datei verwaltet wird. Die Datendatei ist eine sequentielle Datei von Record. Die Schluesseldateien (Indexdateien) dagegen werden als B-Baeume verwaltet, deren Seiten als Eintraege 1. ein Schluesselwort, 2. einen Datenquerverweis (d.h. die Datenrecord-Nummer) und 3. einen Schluesselquerverweis (d.h. die Verknuepfung zur naechsten Seite) beinhaltet. Wird eine Suchoperation nach einem Schluessel in einer solchen Datei ausgefuehrt, so geben die entsprechenden Prozeduren die Nummer des gesuchten Records der Datendatei zurueck. Die inhaltliche Verwaltung der Datei und der Schluessedatei obliegt natuerlich weiterhin dem Programmierer. Das Turbo Access-System stellt nur die Prozeduren zur Verwaltung zur Verfuegung.

Das Turbo Access-System wird folgendermassen benutzt:

```
PROGRAM...+

   < eigene Deklarationen >

   < Deklaration der Konstanten fuer Turbo Access >

   (* $I ACCESS.BOX *)
   (* $I GETKEY.BOX *)
   (* $I ADDKEY.BOX *)
   (* $I DELKEY.BOX *)

   < wobei die ...KEY.BOX - Teile weggelassen werden koennen, wenn keine
   Indexdatei verwendet wird >

   < eigene Unterprogramme >

   < Hauptprogramm >
```

Folgende Konstanten muessen fuer Turbo Access deklariert werden:

MaxDataRecSize	Die maximale Groesse der zu verwaltenden Records der Datei in Bytes.
MaxKeyLen	Die maximale Laenge der verwendeten Schluessel in Bytes (1..255).
PageSize	Maximale Anzahl der Schluesseleintraege pro Seite (normalerweise zwischen 16 und 32).
Order	Ordnung des B-Baumes. Hier muss die Haelfte von PageSize deklariert werden.
PageStackSize	Groesse des Seitenpuffers. Anzahl der Seiten, die gleichzeitig im Speicher sein koennen. Ein grosser Wert bewirkt, dass das Suchen sehr schnell ist, der Speicherplatzbedarf jedoch zunimmt.
MaxHeigth	Die maximale Hoehe des B-Baumes. Berechnet sich aus MaxHeight = Log(E)/Log(PageSize * 0.5), wobei E die Anzahl der Schluessel einer Indexdatei ist.

Sind diese Konstanten deklariert und die entsprechenden Include-Anweisungen in das Programm eingefuegt, so koennen die im folgenden beschriebenen Prozeduren und Funktionen verwendet werden. Dazu sind die Datentypen

```
IndexFile und
DataFile
```

zur Deklaration der entsprechenden Dateien zu verwenden, und es existiert eine Boolesche Variable

```
OK : BOOLEAN
```

die von den meisten Prozeduren und Funktionen veraendert wird, um anzuzeigen, ob die Operation erfolgreich ausgefuehrt wurde.

Prozeduren zur Datenverwaltung

```
PROCEDURE MakeFile (VAR DatF : DataFile;
                        FileN : STRING[14];
                        RecLen : INTEGER);
```

erstellt eine Datendatei DatF mit Namen FileN und Records der Laenge RecLen in Bytes. Ist OK FALSE, so findet sich nicht mehr genug Platz auf der Diskette.

```
PROCEDURE OpenFile (VAR DatF : DataFile;
                        FileN : STRING[14];
                        RecLen : INTEGER);
```

eroeffnet eine bestehende Datendatei (wie Make File). Ist OK FALSE, so laesst sich die Datei nicht eroeffnen.

```
PROCEDURE CloseFile (VAR DatF : DataFile);
```

schliesst eine geoeffnete Datendatei DatF.

```
PROCEDURE AddRec (VAR DatF : DatFile;
                  VAR DataRef : INTEGER;
                  VAR Buffer);
```

fuegt einen Datensatz, der der untypisierten Variablen Buffer uebergeben wird, in die Datei DatF an der Stelle DataRef (Record-nummer) ein. Die neue Recordnummer steht durch den Variablenpara-meter DataRef zur Verfuegung.

```
PROCEDURE DeleteRec (VAR DatF : DatFile;
                         DataRef : INTEGER);
```

loescht einen Datensatz in der Datei DatF mit der Recordnummer DataRef.

```
PROCEDURE GetRec (VAR DatF : DatFile;
                      DataRef : INTEGER;
                  VAR Buffer);
```

liest einen Datensatz aus der Datei DatF mit der angegebenen Recordnummer DataRef in die Puffervariable Buffer.

```
PROCEDURE PutRec (VAR DatF : DatFile;
                      DataRef : INTEGER;
                  VAR Buffer);
```

schreibt einen Datensatz, der mit der Variablen Buffer uebergeben wird, in die Datei DatF an die Stelle der Recordnummer DataRef.

```
FUNKTION FileLen (VAR DatF : DataFile) : INTEGER;
```

gibt die Anzahl der Datensaetze der Datei wieder (mit den geloeschten Datensaetzen).

```
FUNKTION UsedRecs (VAR DatF : DataFile) : INTEGER;
```

gibt die Anzahl der benutzten Datensaetze der Datei wieder.

Prozeduren zur Verwaltung von Schluesseln

```
PROCEDURE InitIndex;
```

wird zum Programmbeginn aufgerufen.

```
PROCEDURE MakeIndex (VAR IndexF : IndexFile;
                         FileN : STRING[14];
                         KeyLen,Status : INTEGER);
```

erstellt eine Indexdatei IndexF mit dem Namen FileN, die Schluesselworter der Laenge KeyLen enthaelt. Schluesselwoerter sind vom Typ STRING. Der Statusparameter kann den Wert 0 oder 1 haben, wobei 0 bedeutet, dass keine doppelten Schluesselwoerter erlaubt sind, und 1, dass doppelte Schluesselwoerter benutzt werden koennen. OK ist FALSE, wenn nicht genuegend Platz auf der Diskette zur Verfuegung steht.

```
PROCEDURE OpenIndex (VAR IndexF : IndexFile;
                         FileN : STRING[14];
                         KeyLen,Status : INTEGER);
```

eroeffnet eine bestehende Indexdatei. Sonst wie MakeIndex. OK ist FALSE, wenn die Datei nicht existiert.

```
PROCEDURE CloseIndex (VAR IndexF : IndexFile);
```

schliesst eine Indexdatei.

```
PROCEDURE AddKey (VAR IndexF : IndexFile;
                  VAR DataRef : INTEGER;
                  VAR Key);
```

fuegt eine Schluessel Key in die Indexdatei IndexF, der mit dem Datenelement der Nummer DataRef verbunden ist. OK ist FALSE, wenn der Schluessel schon existiert (nur bei Status 0).

```
PROCEDURE DelKey  (VAR IdexF : IndexFile;
                   VAR DataRef : INTEGER;
                   VAR Key);
```

loescht einen Schluessel. Sonst wie AddKey. OK ist FALSE, wenn der Schluessel nicht existiert.

```
PROCEDURE FindKey (VAR IdexF : IndexFile;
                   VAR DataRef : INTEGER;
                   VAR Key);
```

sucht genau den angegebenen Schluessl Key in der Indexdatei IndexF und gibt nach erfolgreicher Suche die Recordnummer des entsprechenden Datenelements ueber DataRef zurueck. OK ist FALSE, wenn der Schluessel nicht gefunden wurde.

```
PROCEDURE SearchKey (VAR IndexF : IndexFile;
                     VAR DataRef : INTEGER;
                     VAR Key);
```

wie FindKey. Allerdings braucht der angegebene Schluessel nur mit dem Anfang des gesuchten Schluessels uebereinstimmen.

```
PROCEDURE NextKey (VAR IndexF : IndexFile;
                   VAR DataRef : INTEGER;
                   VAR Key);
```

sucht den naechsten Schluessel.

```
PROCEDURE PrevKey (VAR IndexF : IndexFile;
                   VAR DataRef : INTEGER;
                   VAR Key);
```

sucht den vorhergehenden Schluessel.

```
PROCEDURE ClearKey (VAR IndexF : IndexFile);
```

setzt den Indexzeiger (abhaengig von einem folgenden NextKey oder PrevKey) auf den ersten oder letzten Eintrag der Indexdatei.

Hinweis: Die Veraenderung von Datenelementen bewirkt keine automatische Aenderung von Indexeintraegen und umgekehrt. Daher muss der Programmierer selbst dafuer sorgen, dass z.B. bei der Aenderung eines Datenelements die zugehoerigen Schluesseleintraege zuvor geloescht und dann mit den neuen Inhalt zugefuegt werden.

Im folgenden ein Programmbeispiel zur Verdeutlichung der o.g. Prozeduren und Funktionen. Es soll eine Mitgliederdatei erzeugt und verwaltet werden. Teile des Programms sind an das Adressprogramm aus Kap. 8.1 angelehnt. Erweiterungen und Verfeinerungen sollten vom Leser als Uebung vorgenommen werden. Die Datenelemente bestehen aus einer Mitgliedernummer (als STRING, der Einfachheit halber, weil Schluessel auch STRINGs sind) und einer Adresse. Zwei Suchkriterien sollen zugelassen sein: Mitgliedsnummer (genaue Uebereinstimmung und keine doppelten) und Name (Anfangsuebereinstimmung und doppelte zugelassen).
Die Elemente der Datei koennen veraendert (mit Neueingabe), geloescht und nach den o.g. Kriterien gesucht werden.
Das Programm sieht nun so aus:

```
PROGRAM Mitgliederdatei;
CONST Reclaenge = 80;
```

```
      J : SET OF CHAR = ['J','j'];
      N : SET OF CHAR = ['N','n'];
      JN :SET OF CHAR = ['J','j','N','n'];

      MaxDataReSize = Reclaenge;
      MaxKeyLen     = 22;
      PageSize      = 16;
      Order         = 8;
      PageStackSize = 5;
      MaxHeight     = 5;

{$I B:ACCESS.BOX}
{$I B:GETKEY.BOX}
{$I B:ADDKEY.BOX}
{$I B:DELKEY.BOX}

  TYPE Setofchar = SET OF CHAR;
       Kurzstring = STRING[20];
       Adresse    = RECORD
                      Nummer : Kurzstring;
                      Name, Vorname, Ort : Kurzstring;
                      END; { Adresse }
       Filetyp    =  FILE OF Adresse;

  VAR Person       : Adresse;
      DatF         : DataFile;
      NumF, NamF   : IndexFile;
      Num          : Kurzstring;
      Nam          : Kurzstring;
      Dateiname    : STRING[15];
      ch           : CHAR;
      Anzahl, Nr   : INTEGER;

  FUNKTION Lieszeichen (m : Setofchar) : CHAR;
  VAR ch : CHAR;
      OK : BOOLEAN;
  BEGIN
   REPEAT
    READ (KBD,ch);                        { Lies Zeichen ohne Echo }
    IF BOLN (KBD) THEN ch := CHR(13);     { <Return>-Taste }
     OK := ch IN m;
    IF NOT OK THEN WRITE (CHR(7))             { Bell }
              ELSE IF ch IN [' '..CHR(126)]  { druckbare Zeichen }
                      THEN WRITE (ch)
   UNTIL OK;
   Lieszeichen := ch
  END;  { von Lieszeichen }

  PROCEDURE Lesen (Var Satz : Adresse);
  PROCEDURE Maske;
   BEGIN   { von Maske }
    GOTOXY(1,2); WRITE ('Nummer:   ');
    GOTOXY(1,4); WRITE ('Name:     ');
    GOTOXY(1,6); WRITE ('Vorname:  ');
    GOTOXY (1,8); WRITE ('Wohnort: ');
   END;   { von Maske }

  PROCEDURE Fuellen;
```

```
 BEGIN   { von Fuellen }
  WITH Satz DO BEGIN
   GOTOXY(13,2); WRITE (Nummer);
   GOTOXY(13,4); WRITE (Name);
   GOTOXY(13,6); WRITE (Vorname);
   GOTOXY(13,8); WRITE (Ort);
  END {von WITH }
 END; { von Fuellen }

BEGIN   { von Lesen }
 CLRSCR;
 GetREC (DatF,Nr,Satz);
 Maske;
 Fuellen
END;   { von Lesen }

PROCEDURE Schreibe (Satz : Adresse);
BEGIN
 PutRec (DatF, Nr, Satz);
 DeleteKey (NumF,Nr,Num);
 AddKey (NumF,Nr.Nummer);
 DeleteKey (NamF,Nr,Nam);
 AddKey (NamF,Nr,Satz.Name);
END;   { von Schreibe }

PROCEDURE Suchen;
 BEGIN
  CLRSCR;
  WRITELN ('Sucheneines Datensatzes nach');
  WRITELN;
  WRITELN ('M(itgliedsnummer');
  WRITELN ('N(ame');
  ch := Lieszeichen(['m','M','n','N']);
  CASE ch OF
   'M','m' : BEGIN
              WRITE ('Gesuchte Nummer:');
              READLN (Num);
              FindKey (NumF,Nr,Num);
             END;
   'N','n' : BEGIN
              WRITE ('Gesuchter Name:');
              READLN (Nam);
              SearchKey (NamF,Nr,Nam);
             END;
END;
REPEAT
 IF OK THEN BEGIN
  Lesen (Person);
  Nam := Person.Name;
  Num := Person.Nummer
 END
 ELSE WRITELN (^G^G^G'Element existiert nicht !');
 GOTOXY(1,20);
 WRITELN ('W(eiter-N(aechster,V(orheriger Name');
 ch := Lieszeichen (['W','w','N','n','V','v']);
 CASE ch OF
  'N','n' : NextKey (NamF,Nr,Nam);
  'V','v' : PrevKey (NamF,Nr,Nam)
```

```
   END;
  UNTIL ch IN ['W','w']
 END;

PROCEDURE Veraendern;
VAR Frage : CHAR;

PROCEDURE Aendern;
 VAR Satz : Adresse;
     Hilf : Kurzstring;

PROCEDURE Tasten (x,y :INTEGER; VAR WORT : Kurzstring);
 VAR Wort2 : Kurzstring;
 BEGIN { von Tasten }
  GOTOXY (x,y); READLN (Wort2);
  IF Wort2 <> '' THEN Wort := Wort2;
  { nur Aenderung, wenn nicht <Return>}
  GOTOXY (x,y); WRITE (Wort)
 END; { von Tasten }

BEGIN  { von Aendern }
 Lesen (Satz);
 WITH Satz DO BEGIN
  Tasten (13,2,Nummer);
  Tasten (13,4,Name); Tasten (13,6,Vorname);
  Tasten (13,8,Ort);
 END;   { von WITH }
 Schreibe (Satz);
END;   { von Aendern }

BEGIN   { von Veraendern }
 REPEAT
       CLRSCR;
       WRITELN; WRITELN;
       WRITELN ('  N (eueingabe');
       WRITELN ('  U (Maendern ');
       WRITELN ('  M (enue     ');
       WRITELN; Wreiteln ('Bitte waehlen Sie');
       Frage := Lieszeichen (['N','n','U','u','M','m']);
       IF Frage IN ['N','n'] THEN BEGIN
        WITH Person DO BEGIN
         Nummer := ' ';
         Name   := ' ';
         Vorname := ' ';
         Ort := ' '
      END;
      AddRec (DatF,Nr,Person);
      AddKey (NumF,Nr,Person.Nummer);
      AddKey (NamF,Nr,Person.Name);
      Aendern
     END;   { von IF }
     IF Frage IN {'U','u'] THEN BEGIN
      WRITELN;
      Suchen;
      IF OK THEN Aendern
     END   { von IF }
    UNTIL Frage IN ['M','m']
   END;   { von Veraendern}
```

```
PROCEDURE Loeschen;
 BEGIN
  Suchen;
  IF OK THEN BEGIN
   WRITELN ('Soll das Element geloescht werden (J/N? ');
   REPEAT
    READ (KBD,ch)
   UNTIL ch IN JN;
   IF ch IN J THEN BEGIN
    DeleteRec (DatF,Nr);
    DeleteKey (NumF,Nr,Num);
    DeleteKey (NamF,Nr,Nam);
   END;
  END;
 END;

PROCEDURE Menue;
VAR Frage : CHAR;
BEGIN
 REPEAT
  CLRSCR;
  Anzahl := UsedRecs (DatF);
  WRITELN; WRITELN;
  WRITELN ('Die Datei hat ', Anzahl:3);
  WRITELN ('tatsaechliche Elemente.');
  WRITELN; WRITELN ('Waehlen Sie:');
  WRITELN; WRITELN;
  WRITELN ('  V (eraendern von Daten   ');
  WRITELN ('  L (Oeschen von Daten     ');
  WRITELN ('  S (uchen nach Kriterien  ');
  WRITELN;
  WRITELN ('  Z (um Schluss            ');
  Frage := Lieszeichen (['V','v','L','l','S','s','Z','z']);
  CASE Frage OF
   'V','v' : Veraendern;
   'L','l' : Loeschen;
   'S','s' : Suchen;
  END   { CASE }
 UNTIL Frage IN ['Z','z']
END;   { von Menue }

PROCEDURE START;
LABEL Exit1;
VAR Satz : Adresse;

PROCEDURE Neu;
 LABEL Exit2;
 BEGIN   { von Neu }
  MakeFile (DatF,Dateiname+'.DAT',SIZEOF(Person));
  IF NOT OK THEN BEGIN
   WRITELN ('Kein Platz auf der Diskette');
   GOTO Exit2;
  END
  ELSE BEGIN
   MakeIndex (NumF,Dateiname+'.NUM',SIZEOF(Num),0);
   MakeIndex (NamF,Dateiname+'.NAM'<SIZEOF(Nam),1);
  END;
```

```
 Exit2:
 END;   { von Neu }

 BEGIN   {von Start }
  WRITELN; WRITELN;
  WRITE ('Eingabe des Dateinamens: '); READLN (Dateinale);
  OpenFile (DatF,Dateiname+'.DAT',SIZEOF(Person));
  IF NOT OK THEN BEGIN
   WRITELN ('Datei existiert nicht !');
   WRITE ('Wollen Sie eine neue Datei (J/N)?');
   ch := Lieszeichen (JN);
   IF ch IN J THEN BEGIN
    Neu;
    IF NOT OK THEN GOTO Exit1
  END ELSE GOTO Exit1
 END   { von IF }
 ELSE BEGIN
  OpenIndex (NumF,Dateiname+'.NUM',SIZEOF(Num),0);
  OpenIndex (NamF,Dateiname+'.NAM',SIZEOF(Nam),1);
 END;

 Menue;
 CloseFile (DatF);
 CloseIndex (NumF);
 CloseIndex (NamF);
 Exit1:
END;   { von Start }

BEGIN   { Hauptprogramm }
 Nr := 0;
 CLRSCR;
 InitIndex;
 Start;
 CLRSCR;
 WRITELN 'Das war es !')
END.
```

Wird als Dateiname z.B. DATEI angegeben,so werden Dateien mit den Namen DATEI.DAT, DATEI.NUM und DATEI.NAM angelegt.
Als Uebung sollten Sie das Programm dahingehend erweitern und aendern, dass Namen als Schluesselwoerter nur in Grossbuchstaben gespeichert werden und vor dem Suchen das Suchwort entsprechend geaendert wird. Ausserdem sollte eine Ausgabeprozedur fuer die ganze Datei eingefuegt werden, die dafuer sorgt, dass die Datei in alphabetischer Reihenfolge der Namen ausgegeben wird. Eine Verbesserung liegt sicher auch darin, nicht nur nach Nachnamen, sondern nach diesen in Verbindung mit dem Vornamen zu suchen.

Standardfunktionen und -prozeduren in Turbo Pascal

Anhang F

Typenabkuerzungen: i : INTEGER, r : REAL, c : CHAR, s : STRING, b : BOOLEAN,
a : Aufzaehlungstyp, p : Zeiger, f : Datei, A : Array, x : beliebig,
/ : nichts

Bezeichner	Argumenttyp	bei Funktionen Ergebnistyp	Bezeichnung
ABS	r oder i	wie Argument	Betrag des Arguments
ARCTAN	i oder r	r	Arcustangens
ASSIGN	(f, s)	/	Weist Dateinamen zu
BLOCKREAD	(s, A, i, i)	i	Lesen eines Diskettenblockes
BLOCKWRITE	(s, A, i, i)	i	Schreiben eines Diskettenblockes
CHAIN	f	/	Ruft CHN-File auf
CHR	i	c	Zeichen aus ASCII mit Nummer i
CLOSE	f	/	Schliessen einer Datei
CLREOL	/	/	Loeschen bis Ende der Zeile
CLRSCR	/	/	Bildschirm loeschen
CONCAT	s	s	Verbindet Strings zu neuen Strings
COPY	(s, i, i)	s	Kopiert Teilstring aus String
COS	i oder r	r	Cosinus
CRTINIT	/	/	Terminal initialisieren
CRTEXIT	/	/	Terminal reset
DELAY	i	/	Warte Zeit i
DELETE	(s, i, i)	/	Loescht Teil aus String
DELLINE	/	/	Loesche Zeile
DISPOSE	p	/	loescht unbenutzte Zeiger
EOF	f	b	Ende einer Datei
EOLN	f	b	Ende einer Zeile
EXECUTE	f	/	Ruft COM-File auf
EXIT	/	/	Ausstieg aus Block
EXP	i oder r	r	e-Funktion
FILLCHAR	(x, i, c)	/	Fuellt Variable x mit Zeichen
FRAC	r oder i	r	Nachkommawert einer Zahl
GOTOXY	(i,i)	/	Setzt Cursor an bestimmte Stelle
HI	i	i	Hoeherwertiges Byte des Argumentes
INLINE	Codierung	/	Fuegt Maschienencode ein
INSERT	(s, s, i)	/	Fuegt String in andere String ein
INSLINE	/	/	Fuegt Zeile ein
INT	i oder r	r	Vorkommawert einer Zahl
IORESULT	/	/	Fehlercode
KEYPRESSED	/	b	Abfrage auf Tastendruck
LENGTH	s	i	Gibt Laenge des String an
LN	i oder r	r	Logarithmus Naturalis
LO	i	i	Niederwertiges Byte des Argumentes
LOG	i oder r	r	Dekadischer Logarithmus
LOWVIDEO	/	/	Low-Video Attribut
MARK	i	/	Markieren des Variablenstapels
MEMAVAIL	/	i	Freier Speicher
MOVE	(x, x, i)	/	Bewegt Anzahl von Byts

NEW	p	/	Erzeugen einer Zeigervariablen
NORMVIDEO	/	/	Norm-Video Attribut
ODD	i	b	Wahr, wenn i ungerade ist
ORD	c	i	Nummer des Zeichens c in ASCII
OVRDRIVE	i	/	Drive-Nr. fuer Overlay
OVRPATH	s	/	Pfadname fuer Overlay

Bezeichner	Argumenttyp	bei Funktionen Ergebnistyp	Bezeichnung
PARAMCOUNT	/	i	Anzahl der Parameter
PARAMSTR	i	s	n-ter Parameter
POS	(s, s)	i	Position eines Strings in anderem
PRED	c,i,b oder a	wie Argument	Vorgaenger
PWROFTEN	r oder i	r	Zehnerpotenz
RANDOM	/	r	Zufallszahl 0.0 bis 1.0
RANDOM	i	i	Zufallszahl von 0 bis 1
RANDOMIZE	/	/	Erzeugt neue Zufallszahlen
READ	i,r,c,s od. /	/	Lesen ohne Zeilenvorschub
READ	(f, x)	/	Lesen aus Datei f
READLN	i,r,c,s od. /	/	Lesen mit Zeilenvorschub
RELEASE	i	/	Ruecksetzen des Variablenstapels
RESET	(f)	/	Eroeffnen einer alten Datei
REWRITE	(f)	/	Eroeffnen einer neuen Datei
ROUND	r	i	Rundet zu ganzer Zahl
SEEK	(f, i)	/	Element einer Datei anwaehlen
SEEKEOF	f	b	Aehnlich EOF
SEEKEOLN	f	b	Aehnlich EOLN
SIN	i oder r	r	Sinus
SIZEOF	x	i	Speicherplatz der Variablen x
SQR	i oder r	wie Argument	Quadrat
SQRT	i oder r	r	Quadratwurzel
STR	long-i	s	Macht aus Zahl einen String
SUCC	c,i,b od. a	wie Argument	Nachfolger
SWAP	i	i	Tauscht LSB und MSB von i
TRUNC	long-i oder r	i	Ganzzahliger Teil des Arguments
UPCASE	c	c	Wandelt in Grossbuchstaben
WRITE	i,r,c,s od. /	/	Schreiben ohne Zeilenvorschub
WRITE	(f, x)	/	Schreiben in Datei f
WRITELN	i,r,c,s od. /	/	Schreiben mit Zeilenvorschub

Vordefinierte Typen und Konstanten

Anhang G

Vordefinierte Typen

```
INTEGER
BYTE        (Unterbereich 0..255)
REAL
CHAR
STRING      (mit Angabe der Maximalen Zeichenzahl)
BOOLEAN
```

Vordefinierte Konstanten

```
PI = 3.1415926536
FALSE
TRUE
MAXINT = 32767
NIL
```

Vordefinierte Felder

```
MEM
PORT
```

ASCII-Zeichensatz-Tabelle

Anhang H

Dezimale, oktale und hexadezimale Codewerte der ASCII-Zeichen

DEZ	OKT	HEX	Zei.	DEZ	OKT	HEX	Zei.	DEZ	OKT	HEX	Zei.	DEZ	OKT	HEX	Zei.
0	000	00	NUL	32	040	20	SP	64	100	40	@	96	140	60	`
1	001	01	SCH	33	041	21	!	65	101	41	A	97	141	61	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EOT	36	044	24	$	68	104	44	D	100	144	64	d
5	005	05	END	37	045	25	%	69	105	45	E	101	145	65	e
6	006	06	ACK	38	046	26	&	70	106	46	F	102	146	66	f
7	007	07	BEL	39	047	27	'	71	107	47	G	103	147	67	g
8	010	08	BS	40	050	28	(	72	110	48	H	104	150	68	h
9	011	09	HT	41	051	29	)	73	111	49	I	105	151	69	i
10	012	0A	LF	42	052	2A	`	74	112	4A	J	106	152	6A	j
11	013	0B	VT	43	053	2B	+	75	113	4B	K	107	153	6B	k
12	014	0C	FF	44	054	2C	,	76	114	4C	L	108	154	6C	l
13	015	0D	CR	45	055	2D	-	77	115	4D	M	109	155	6D	m
14	016	0E	SQ	46	056	2E	.	78	116	4E	N	110	156	6E	n
15	017	0F	SI	47	057	2F	/	79	117	4F	O	111	157	6F	o
16	020	10	DLE	48	060	30	0	80	120	50	P	112	160	70	p
17	021	11	DC1	49	061	31	1	81	121	51	Q	113	161	71	q
18	022	12	DC2	50	062	32	2	82	122	52	R	114	162	72	r
19	023	13	DC3	51	063	33	3	83	123	53	S	115	163	73	s
20	024	14	DC4	52	064	34	4	84	124	54	T	116	164	74	t
21	025	15	NAK	53	065	35	5	85	125	55	U	117	165	75	u
22	026	15	SYN	54	066	36	6	86	126	56	V	118	166	76	v
23	027	17	ETB	55	067	37	7	87	127	57	W	119	167	77	w
24	030	18	CAN	56	070	38	8	88	130	58	X	120	170	78	x
25	031	19	EM	57	071	39	9	89	131	59	Y	121	171	79	y
26	032	1A	SUB	58	072	3A	:	90	132	5A	Z	122	172	7A	z
27	033	1B	ESC	59	073	3B	;	91	133	5B	[	123	173	7B	{
28	034	1C	FS	60	074	3C	<	92	134	5C	\	124	174	7C	|
29	035	1D	GS	61	075	3D	=	93	135	5D	]	125	175	7D	}
30	036	1E	RS	62	076	3E	>	94	136	5E	^	126	176	7E	
31	037	1F	US	63	077	3F	?	95	137	5F	_	127	177	7F	DEL

Anmerkung: Der ASCII-Code verwendet nur 7 Bits eines Bytes. Das hoechstwertige Bit (Bit 7) ist in dieser Tabelle auf Null gesetzt. Es kann in anderen Faellen auch den Wert 1 haben. Dann ist zum dezimalen Codewert 128, zum oktalen 200 und zum hexadezimalen 80 zu addieren.