Handbuch TURBO-Pascal

0. Vorwort

Dieses Handbuch ist eine leicht gekuerzte Uebersetzung des TURBO-Pascal Reference Manual 1983 von Borland International Inc.
Das Programmiersystem TURBO-Pascal wurde 1983 entwickelt von BORLAND INTERNATIONAL Scotts Valley California USA. Es ist auf allen CP/M-kompatiblen Mikrorechnern implementierbar.
Pascal ist eine allgemein verwendbare Programmier-Hochsprache, die von Professor Niklaus Wirth von der Technischen Universitaet in Zuerich, Schweiz entwickelt und nach dem beruehmten franzoesischen Philosophen und Mathematiker Blaise Pascal benannt wurde.
Professor Wirth's Definition der Sprache Pascal wurde 1971 veroeffentlicht mit dem Ziel die Computerprogrammierung zu systematisieren, speziell die Strukturierte Programmierung einzufuehren.
Pascal wurde bald auf allen Rechenanlagen implementiert und ist heute eine der fuehrendsten Programmiersprachen der Welt.
TURBO-Pascal wurde entwickelt um die unterschiedlichen Beduerfnisse der verschiedenen Nutzer zu befriedigen. ES besticht durch seine leicht handhabbaren interaktiven Elemente, die den Lernprozess aktiv foerdern und ein extrem effektives Arbeitsmittel bereitstellen, indem sie Entwicklung, Uebersetzung und Abarbeitung der Programme als Einheit in dem System TURBO-Pascal behandeln.
TURBO-Pascal haelt sich eng an die Definition von Standard-Pascal, wie es von K.Jensen und N.Wirth in dem PASCAL User Manual and Report (1974) definiert wurde. Es gibt nur wenige Abweichungen, die eingefuehrt wurden, um die Effizienz zu erhoehen. Es sind dies:

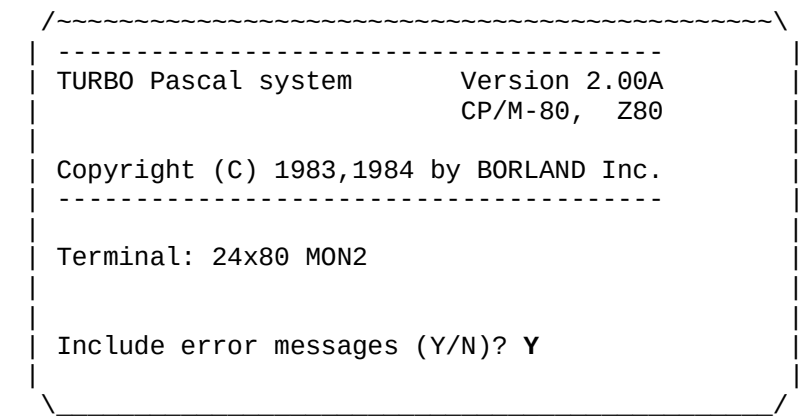
- GET und PUT sind nicht implementiert, dafuer wurden READ und WRITE erweitert.
- GOTO darf nicht den aktuellen Block velassen.
- Die Prozedure PAGE ist nicht implementiert.
- PACKED ist nicht speziell implementiert, da der Compiler automatisch packt, wenn is moeglich ist.
- Prozeduren und Funktionen duerfen nicht als Parameter verwendet werden.

1. Programmiersystem TURBO-Pascal

Der Aufruf des Systems erfolgt durch den Eingabe des Namens TURBO:

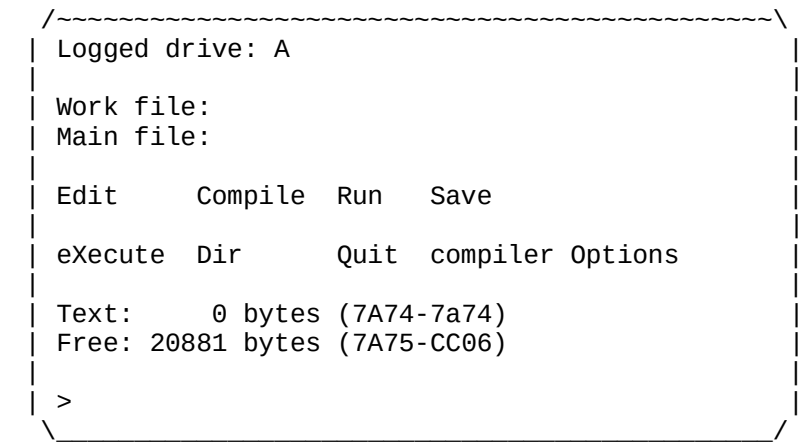
```
A>TURBO
```

Danach erscheint das Titelbild



Die Frage ist mit Y oder N zu beantworten. Bei Y wird die Komponente TURBO.MSG in den TPA geladen. Sie enthaelt die Fehlermitteilungen. Durch das Laden wird der TPA-Bereich fuer Pascalprogramme um 1462 Bytes verkleinert. Dafuer werden aber bei auftretenden Fehlern ausser der Fehlernummer auch die Fehlermitteilung ausgegeben.
Beispiel fuer Compilerfehlermeldung:
Error 5. Press <ESC> bei N.
Error 5: ')' expected. Press <ESC> bei Y.

Nach Beantwortung der Frage erscheint das TURBO Grundmenue:



Als letztes Zeichen erscheint das TURBO-Promptzeichen >. Durch Eingabe eines Kommandobuchstabens wird die entsprechende Komponente des TURBO-Systems aufgerufen. Nach Abarbeitung dieser Komponente erscheint wieder das TURBO-Promptzeichen und eine andere Funktion des TURBO-Systems kann aufgerufen werden. Dabei arbeitet das Terminal meist im Rollmodus. Wird jedoch nach dem Promptzeichen eine Taste gedrueckt, die keinem Kommandobuchstaben entspricht, dann erfolgt ein Loeschen des Teminals und das TURBO-Grundmenue erscheint. Dieses Wiederaufrufen des Grundmenues sollte

man grundsaetzlich durchfuehren, falls aktualisierte Werte im Menue gewuenscht werden.

Es gibt folgende Kommandobuchstaben:

L	Laufwerkzuweisung als aktuelles.
W	Workfile aufbauen oder laden.
M	Mainfile aufbauen oder laden.
E	Editieren des Workfile
C	Compilieren
R	Run, Abarbeiten des kompilierten Programmes.
S	Save, Retten des Workfile auf Diskette
X	Execute, Laden und Starten beliebiger CP/M-Programme.
D	Directory des aktuellen Laufwerkes anzeigen.
Q	Quit, Rueckkehr zum CP/M-Betriebssystem (CCP).
O	Setzen der Compiler-Options M,C,H und F.

Die Kommandos werden sofort nach Eingabe ausgefuehrt. Es ist also kein <CR> zu druecken.

1.1 **Laufwerkzuweisung**

Nach Eingabe von L erscheint Aufforderung zur Laufwerkzuweisung:

```
 /~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\
| >L                                             |
| New drive:B:<CR>                               |
|                                                |
 \______________________________________________/
```

Damit wird als aktuelles Laufwerk B zugewiesen. Wird nur <CR> gedrueckt, bleibt die alte Zuweisung erhalten.
Bei jeder Ausfuehrung des Kommandos-L wird automatisch die Funktion von ^C ausgefuehrt. Da ^C notwendig nach Diskettenwechsel ist, kann dies durch Auffruf des Kommandos-L ohne Angabe eines neuen Laufwerkes (nur <CR>) erfolgen.
Die Neuzuweisung eines Laufwerkes wird im Menue nicht sofort angezeigt. Dazu ist nach erscheinen des Promptzeichens > druecken z.B. <CR> notwendig.

1.2 **Workfilezuweisung**

Nach Eingabe von W erscheint Aufforderung zur Eingabe des Namens fuer das Workfile nach den bekannten Regeln fuer CP/M:

```
 /~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\
| >W                                         |
|                                            |
| Work file name:F:PAWEL<CR>                 |
|                                            |
| Loading F:PAWEL.PAS                        |
|                                            |
 \__________________________________________/
```

Damit wird das File PAWEL.PAS vom Laufwerk F in den Arbeitsbe-

reich geladen und kann danach mit E oder C weiterbearbeitet werden.

Existiert das File noch nicht, erscheint die Ausschrift:

```
 /~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\
| New File                                              |
| >                                                     |
 \_____________________________________________________/
```

In diesem Falle kann danach mit E das File aufgebaut werden.

Falls noch im Arbeitsbereich ein noch nicht gerettetes bearbeitetes File steht, wird dies angezeigt durch:

```
 /~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\
| Workfile X:FILENAME.TYP not saved. Save (Y/N)?: Y           |
|                                                             |
 \___________________________________________________________/
```

Bei Antwort Y wird das alte Workfile auf dem Laufwerk X gerettet, und das neue Workfile zugewiesen.
Bei Antwort N wird das im Arbeitsbereich stehende alte Workfile durch das neue ueberschrieben.

1.3 **Mainfilezuweisung**

Nach Eingabe von M erscheint Aufforderung zur Eingabe des Namens fuer das Mainfile nach den bekannten Regeln fuer CP/M:

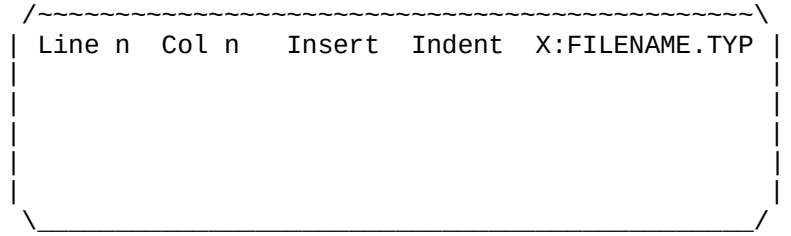
```
 /~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\
| >M                                                          |
| Main file name:E:HPPROG                                     |
|                                                             |
| >                                                           |
 \___________________________________________________________/
```

Damit wird das auf Laufwerk E befindliche Programm HPPROG.PAS als Mainprogramm zugewiesen.
Die Benutzung von einem Mainfile wird notwendig, wenn ein Pascal-Progamm zu gross wird. Das Programm wird in ein Wurzelprogramm und mehrere Teilprogramme zerlegt und zwar so, dass die Teilprogramme in das Wurzelprogramm durch Includeanweisungen eingefuegt werden. Der Compiler beginnt dann bei der Uebersetzung mit dem Mainprogramm und ladet dann nacheinander die durch Includeanweisungen bennanten Teilprogramme in den Workfilebereich, d.h. die Teilprogramme werden nacheinander Workfiles.
Treten bei der Compilierung Fehler auf, kann das fehlerhafte File wie ueblich korrigiert werden. Das Retten des geaenderten Files erfolgt automatisch und bei einem erneuten Compilerlauf wird auch das Mainfile wieder geladen.

1.4 **Editor-Aufruf**

Nach Eingabe von E erscheint der Text des Workfiles auf dem Terminal und der Editor wird gestartet, d.h. der auf dem Terminal stehende Text kann wie mit WORDSTAR (TP) bearbeitet werden.

Existiert noch kein Workfilenamen, so erfolgt die Aufforderung zur Eingabe dieses Namens, danach wird das File geladen, oder wenn es nicht gefunden wurde, die Ausschrift "New File" ausgegeben und der Editor gestartet. Es erscheint das Bild:



Die erste Zeile bleibt als Statuszeile stets auf dem Terminal stehen. Die Informationen haben folgende Bedeutung:

Line n	n=Zeilennummer des Cursor
Col n	n=Spaltennumer des Cursor
Insert	Anzeige Insert ON. Bei OFF steht dort Overwrite.
Indent	Zeigt automatisches Einruecken an
X:FILENAME.TYP	Filename des gerade editierten Textes.

In den der Statuszeile folgenden Zeilen kann der Programmtext geschrieben werden. Jede Zeile ist durch <CR> abzuschliessen. In den untenstehenden Kommandos kann das zweite Controll-Zeichen durch den ensprechenden normalen Buchstaben ersetzt werden, d.h. statt ^Q^L kann man auch ^QL druecken.

Fuer das Editieren koennen folgende Kommandos verwendet werden, die weitgehend mit denen von WORDSTAR uebereinstimmen:

Cursor Kommandos

Zeichen links	^S	Wirkt nur bis Zeilenanfang.
Zeichen rechts	^D	Wirkt nur bis Spalte 125.
Wort links	^A	Zum Wortanfang. Worte werden begrenzt durch: <space> {}<>,.()[]^'*+-/$ Wirkt ueber Zeilengrenzen.
Wort rechts	^F	Analog oben.
Zeile hoch	^E	Rollmodus bei Erreichen oberer Zeile.
Zeile tief	^X	Rollmodus bei vorletzter Zeile
Rollen hoch	^Z	Rollen um eine Zeile zurueck.
Roellen tief	^W	Rollen um eine Zeilen vorwaerts.
Blaettern hoch	^R	Rollen um ein Bild zurueck.
Blaettern tief	^C	Rollen um ein Bild vorwaerts.
Zeilenanfang	^Q^S	Nach erster Spalte.
Zeilenende	^Q^D	Nach Spalte hinter dem letzten Zeichen.
Bildanfang	^Q^E	Nach oberster Bildzeile.
Bildande	^Q^X	Nach unterster Bildzeile.
Fileanfang	^Q^R	Nach erster Textzeile des Files.
Fileende	^Q^C	Nach letzter Textzeile des Files.

Blockbeginn	^Q^B	Zum Blockbeginn.
Blockende	^Q^K	Zum Blockende.
Letzte Position	^Q^P	Rueckkehr zur letzten Cursorposition, nach FIND/REPLACE/SAVE-Kommando o.a.

Insert und Delete Kommandos

Insert-Modus ON/OFF	^V	Einfuegen oder ueberschreiben.
Loeschen links	DEL	Wirkt ueber Zeilengrenzen.
Loeschen Zeichen	^G	Wirkt ueber Zeilengrenzen.
Loeschen Wort rechts	^T	Wirkt ueber Zeilengrenzen.
Zeile einfuegen	^N	Wenn Cursor nicht am Zeilenanfang, wird Zeile an dieser Stelle getrennt.
Zeile loeschen	^Y	Zeile loeschen, in der Cursor steht.
Zeilenrest loeschen	^Q^Y	Loescht Zeile ab Cursor bis Zeilenende.

Block Kommandos

Blockbeginn	^K^B	Die gesetzte Marke ist nicht sichtbar.
Blockende	^K^K	Die gesetzte Marke ist nicht sichtbar.
Markenloeschen	^K^H	Blockmarken werden geloescht, unabhaengig von der Cursorstellung.
Wort markieren	^K^T	Markieren des Wortes an Cursorposition oder links von ihm. Wort wird wie Block behandelt.
Block kopieren	^K^C	Kopieren an die Cursorposition. Marken wandern mit.
Block verschieben	^K^V	Verschieben an die Cursorposition. Marken wandern mit.
Block loeschen	^K^Y	Achtung! Ein geloeschter Block ist nicht zurueckholbar.
Block lesen	^K^R	Block von einem angeforderten File lesen.
Block schreiben	^K^W	Block in ein angefordertes File schreiben. Marken und Block verbleiben an alter Stelle.

Bei nichtvorhandenem Block bewirken die Blockkommandos keine Fehlermeldung. Es passiert absolut nichts.

Spezielle Kommandos

Editieren Ende	^K^D	Rueckkehr zum TURBO-Grundmenue. Es ist zu beachten, dass Workfile noch nicht gesichert ist.
Tab	^I	Die Tabpositionen werden durch die Wortanfaenge der vorhergehenden Zeile bestimmt. Achtung! Bei Insert werden Tab eingefuegt!
Indent ON/OFF	^Q^I	Bei <CR> springt Cursor in naechster Zeile nicht zur Spalte 1, sondern unter das erste Wort. Bei On wird Indent in Statuszeile angezeigt. Bei OFF ist diese Stelle leer.
Restore Zeile	^Q^L	Werden Veraenderungen in einer Zeile durchgefuehrt, so koennen diese alle durch dieses Kommando wieder rueckgaengig gemacht werden, solange dabei der Cursor die Zeile nicht verlassen hat.
Suchen (Find)	^Q^F	Die Suchkette kann aus 30 Zeichen bestehen. Kann CTRL-Zeichen enthalten und wird durch

		<CR> beendet. Zeilenende kann durch ^M^J erzeugt werden. In Suchkette kann ^A als Wildcard (Maskenzeichen) verwendet werden. Optionen: B Rueckwaerts suchen G Globales suchen (top to down) n Suchen des n.Auftretens U Ignorieren Gross/Kleinbuchstaben W Nur Worte suchen Optionen ohne Zwischenraum schreiben und mit <CR> beeenden.
Suchen und Ersetzen	^Q^A	Suchkette und Optionen werden wie bei FIND angegeben. Die Zeichenkette zum Ersetzen kann 30 Zeichen lang sein. Sie kann CTRL-Zeichen enthalten (^A ohne Bedeutung). Abschluss durch <CR>. Optionen: B Rueckwaerts suchen G Globales suchen (top to down) n n-maliges Ersetzen N Ersetzen ohne Fragen:(Replace (Y/N)?) U Ignorieren Gross/Kleinbuchstaben W Nur Worte suchen Optionen ohne Zwischenraum schreiben und mit <CR> beeenden.
Find wiederholen	^L	Wiederholen des letzten ^Q^F oder ^Q^A Kommandos.
Eingabe CTRL-Zeichen	^P	Im Programmtext koennen CTRL-Zeichen durch Voranstellen von ^P eingegeben werden. Beispiel: ^G durch ^P^G.
Abort-Kommando	^U	Jedes Editor-Kommando kann durch ^U sofort abgebrochen werden.

1.5 **Compiler-Aufruf**

Nach Eingabe von C erfolgt die Compilierung des
- Mainfiles, wenn es existiert.
- Workfile im anderen Fall.

Existiert noch kein Workfile, erfolgt die Aufforderung zur Eingabe des Workfilenamens.
Wird das Workfile nicht auf dem angegebenen Datentraeger gefunden, erscheint die Mitteilung "New File" und es wird versucht, das leere File zu uebersetzen. Das Ergebnis ist die Meldung "Error 91: Unexpected end of source. Press <ESC>". Nach dem Druecken der Taste <ESC> wird der Editor aufgerufen und das neue File kann erstellt werden. Sollte dies nicht geschehen, kann der Editorlauf durch ^KD beendet werden. Damit wird das TURBO-Grundmenue erreicht. Das leere File wird nicht gerettet.

Die Durchfuehrung der Compilierung und die Form des Objektcodes ist abhaengig von verschiedenen Compiler-Optionen. Es sind fuer diese Standardwerte festgelegt, die eine zeitguenstige Uebersetzung, minimalen Objektcode und schnelle Abarbeitung des uebersetzten Programmes ergeben. Fuer bestimmte Faelle koennen diese Standardwerte veraendert werden. (Siehe Kommando O).

Bei Auftreten eines Fehlers im Quelltext wird die Compilierung abgebrochen in der Form:

```
 /~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\
| >C                                                   |
|                                                      |
| Compiling                                            |
|   10 lines                                           |
|                                                      |
| Error 5: ')' expected. Press <ESC>                   |
 \____________________________________________________/
```

wobei der Fehlertext nur dann ausgeschrieben wird, wenn TURBO.MSG geladen wurde (Eingabe Y bei Include error messages (Y/N)? nach dem Laden von TURBO). Nach Druecken der Taste <ESC> zeigt der Editor den fehlerhaften Quelltext und die Fehlerstelle auf dem Terminal (Cursorstelle). Es kann sofort editiert und danach erneut compiliert werden.
Bei fehlerfreier Uebersetzung erscheint folgendes Bild:

```
 /~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\
| Compiling                                            |
|   12 lines                                           |
|                                                      |
| Code:   230 bytes (7B4D-7C33)                        |
| Free: 21450 bytes (7C34-CFFE)                        |
| Data:     7 bytes (CFFF-D006)                        |
|                                                      |
|                                                      |
| >                                                    |
 \____________________________________________________/
```

Es werden die Anzahl der uebersetzten Quelltextzeilen, die Anzahl der Bytes und die Adressen fuer den Programmcode und den Variablenbereich, sowie den noch verbleibenden freien Speicherbereich angegeben.

Je nach festgelegter Option (Kommando O) wird das erzeugte Objektcodefile gespeichert

- -im TPA (M) (Standard)
- -als COM-File (C)
- -als CHN-File (H).

1.6 **Programmstart**

Nach Eingabe von R wird das uebersetzte Programm aktiviert und gestartet:

```
 /~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\
| >R                                                   |
| ......                                               |
| ......                                               |
| ......                                               |
| >                                                    |
 \____________________________________________________/
```

Je nach Compiler-Option bei der Uebersetzung wird das Programm
- -aus dem TPA (M)
- -als COM-File von Diskette (C)

aktiviert und gestartet.

Wurde kein uebersetztes Programm gefunden, so uebersetzt automatisch der Compiler das Workfile und startet nach erfolgreicher Uebersetzung sofort auch das Programm.
Wurde noch kein Workfile angegeben, so wird zur Eingabe des Workfilenamens aufgefordert. Existiert auch das Workfile selbst noch nicht auf dem Datentraeger, so wird analog wie bei Kommando C der Editor gestartet.

1.7 **Save-Kommando**

Nach Eingabe von S wird das Workfile auf das entsprechende Laufwerk ausgegeben und gerettet:

```
 /~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\
| >S                                                          |
|                                                             |
| Saving F:PAWEL.PAS                                          |
| >                                                           |
 \___________________________________________________________/
```

Falls bereits auf der Diskette ein Programm mit gleichem Filenamen existiert, wird dies in ein BAK-File umgewandelt und die neue Version unter dem urspruenglichen Namen eingetragen.

1.8 **Execute-Kommando**

Nach Eingabe von X kann aus TURBO ein beliebiges CP/M-Programm gestartet werden:

```
 /~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\
| >X                                                          |
|                                                             |
| Command:A:POWER                                             |
 \___________________________________________________________/
```

In diesem Falle wuerde das Programm POWER geladen und gestartet werden.
Bendet man das CP/M-Programm danach mit normalem Ausgang zum CP/M-System (bei POWER durch EXIT, bei TP durch X, bei DU durch ^C usw.), dann kehrt die Steuerung unmittelbar zum TURBO-System zurueck, d.h. es wird TURBO und die benoetigten Files geladen, einschlieslich des zuletzt bearbeiteten Workfiles und als letztes erscheint der TURBO-Prompt >. Ein nachfolgendes <CR> bringt das TURBO-Grundmenue zurueck, es kann aber auch mit jedem anderen TURBO-Kommando wie R,E,C usw. fortgefahren werden.
Falls zwischenzeitlich die Diskette mit dem TURBO-System aus dem betreffenden aktuellen Laufwerk entfernt wurde, wird der Nutzer aufgefordert die TURBO-Systemdiskette wieder in dieses einzulegen und mit <CR> zu starten:

```
 /~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\
```

```
| TURBO.COM not found.  Re-insert disk in drive E: and hit RETURN |
|                                                                 |
 \_____________________________________________________________/
```

Danach erfolgt das Laden und Starten von TURBO wie oben beschrieben.

1.9 **Directory-Aufruf**

Nach Eingabe eines D erfolgt die Aufforderung zur Eingabe einer Maske, die den Umfang der auszugebenden Programmnamen steuert.Die Maske wird in der CP/M ueblichen Weise mit * und ? gebildet und mit <CR> abgeschlossen. Wird nur <CR> eingegeben, dann wird die gesamte Directory des angegebenen Laufwerkes auf dem Terminal ausgegeben:

```
 /~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\
| >                                                                 |
| Dir mask: A:*.PAS                                                 |
| PAWEL    PAS : PROBE    PAS : PRO1   PAS : PRO2   PAS : PRO3   PAS |
| PRO4     PAS : PROGRAMM PAS                                       |
|                                                                   |
| Bytes Remaining On A: 25K                                         |
| >                                                                 |
 \_________________________________________________________________/_
```

Zum Abschluss wird der auf der Diskette noch freie Speicherbereich ausgegeben (z.B. 25k).

1.10 **Quit-Kommando**

Das Kommando Q dient zum Verlassen des TURBO-Systems und zur Rueckkehr zum CCP. Es erscheint das CP/M-Prompt X> mit dem aktuellen Laufwerk X. Falls noch ein editiertes geladenes Workfile existiert, das nicht gesichert wurde, wird gefragt, ob es gerettet werden soll.

1.11 **Compiler Optionen**

Mit diesem Kommando wird die Form des Ausgabefiles festgelegt, bzw. kann nach einem bei der Abarbeitung eines COM-Programmes aufgetretenen Fehler dessen Stelle gefunden werden.
Nach der Eingabe des Kommandos O erscheint:

```
 /~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\
| >O                                                          |
|                                                             |
|             Memory                                          |
|  compile -> Com-file                                        |
|             cHn-file                                        |
|                                                             |
| Start address: 1FC9 (min 1FC9)                              |
| End   address: CD40 (max D006)                              |
|                                                             |
```

```
|  Find run-time error  Quit                    |
|                                               |
|  >_                                           |
 \_____________________________________________/
```

Die Stelle des Pfeiles zeigt die aktuell wirkende Option. Sie kann durch Eingabe von M,C oder H veraendert werden. Mit F wird der Suchprozess zur Auffindung eines Fehlers im Quellprogramm gestartet und mit Q kann zum TURBO-Grundmenue zurueckgekehrt werden. Start- und Endadresse werden nur bei den Optionen C oder H ausgegeben. Die Optionen im Einzelnen:

M: (Standard) Der Code wird im TPA erzeugt und Programm kann durch Run gestartet werden (Kommando R).

C: Unter dem Namen des Work- bzw. Mainfile wird ein COM-File erzeugt, das den Code und die TURBO-Pascal-Bibliothek enthaelt. Gestartet wird das COM-File durch normalen CP/M-Aufruf oder durch das Kommando R. Diese Programme koennen groesser als bei der Option M sein, da ihre Anfangsadresse im TPA niedriger liegt und bei der Compilierung kein TPA fuer die Speicherung des Programmes benoetigt wird. Es wird die Startadresse und Endadresse des Programmbereiches fuer den Code im COM-File ausgegeben. Sie koenen mit S bzw. E geaendert werden.

H: Unter dem Namen des Work- bzw. Mainfiles wird ein CHN-File erzeugt. Es enthaelt den Programmcode, aber nicht die TURBO-Pascal-Bibliothek. Diese Programme koennen nur von einem anderen Pascalprogramm, das mit der COM-Option uebersetzt wurde, durch das Chain-Kommando aufgerufen werden. Damit besteht also die Moeglichkeit auch sehr grosse Pascal-Programmpakete zu erzeugen. Im Option-Menue wird die Start- und Endadresse des Programmbereiches duer den Code im CHN-File ausgegeben. Sie koennen mit S bzw. E geaendert werden.

F: Wurde ein Programm nicht mit der M-Option uebersetzt, kann die Stelle eines bei der Abarbeitung auftretenden Proggramm-fehlers nicht automatisch im Quelltext gefunden werden, da zu diesem Zeitpunkt das TURBO-Pascal-System nicht ohne weiteres zur Verfuegung steht. Der Abbruch des Programmes erfolgt mit der Fehlermeldung:

```
Run-time error nn, PC = 1B56
Progsamm aborted
```

Dabei ist nn die Fehlernummer und die Hexadezimalzahl gibt die Fehlerstelle im Code an. Um diese Stelle im Quellprogramm zu finden, muss das F-Kommandogegeben werden. Es wird die Eingabe der Fehleradresse gefordert:

```
Enter PC:1B56
```

Die Eingabe der Fehleradresse z.B. 1B56 fuehrt dann automatisch zum Suchprozess im Quellprogramm. Wurde die Fehlerstelle gefunden, wird dies mitgeteilt und zum Eintippen der Taste <ESC> aufgefordert. Danach erscheint das Quellprogramm und der Cursor steht hinter dem Sprachelement, das den Fehler verursachte.

S: Die Startadresse gibt die Adresse des ersten Byte vom Code

an. Das ist normalerweise die Adresse unmittelbar hinter der TURBO-Pascal-Bibliothek. Diese Adresse kann noch oben erhoeht werden, um Platz z.B. fuer absolute Variable, die von geketteten Programmen (CHN-Files) verwendet werden, zu schaffen. Nach Eingabe des Kommando S erscheint die Aufforderung zur Eingabe der Startadresse. Sie wird hexadezimal eingegeben und mit <CR> abgeschlossen. Wird nur <CR> eingegeben, so bleibt die alte Adresse erhalten.

E: Die Endadresse gibt die hoechste verfuegbare Adresse des Programmes an. Der Wert in Klammern ist die TPA-Grenze (BDOS-1). Die angegebene Adresse ist etwa 700 bis 1000 Bytes niedriger, um Platz fuer den CCP zubekommen, wenn die Programme abgearbeitet werden sollen. Falls die uebersetzen Programme auf einem anderen CP/M-Rechner abgearbeitet werden sollen, dessen TPA-Grenze einen anderen Wert hat, besteht mit dem Kommando E die Moeglichkeit, die Endadresse anzupassen. Nach Eingabe des Kommando E erfolgt die Aufforderung zur Eingabe der Endadresse hexadezimal. Sie wird mit <CR> abgeschlossen. Wird nur <CR> eingegeben, dann bleibt die alte Adresse erhalten.

2. Grundlegende Sprachelemente

2.1 Grundsymbole

Das Grundvokabular von TURBO-Pascal besteht aus Grundsymbolen, die zu folgenden Klassen zusammengefasst sind:

```
<Buchstaben>    ::= A|B|C|D|E|F|G|H|I|J|K|L|M|
                    N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
                    a|b|c|d|e|f|g|h|i|j|k|l|m|
                    n|o|p|q|r|s|t|u|v|w|x|y|z|_

<Ziffern>       ::= 0|1|2|3|4|5|6|7|8|9

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

Es gibt weiterhin einige Operatoren, die aus zwei Symbolen bestehen:

```
Zuweisungsoperator:     :=
Vergleichsoperatoren:   <>   <=   >=
Teilbereichsbegrenzer:  ..
Klammern:               (. .) gleichbedeutend mit [ ]
Kommentar:              (* *) gleichbedeutend mit { }
```

2.2 Reservierte Worte

Folgende Worte sind in TURBO-Pascal fest definiert und duerfen nur fuer die entsprechenden Zwecke verwendet werden. Mit Stern versehene Worte sind nicht in Standard-Pascal enthalten:

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

2.3 **Begrenzer**

Sprachelemente muessen durch wenigstens einen der folgenden Begrenzer getrennt werden:

<space>,
Zeilenende,
Kommentar

2.4 **Programmzeilen**

Die maximale Laenge einer Programmzeile betraegt 127 Zeichen. Alle weiteren Zeichen werden ignoriert.

2.5 **Standardbezeichner**

TURBO-Pascal verwendet eine Anzahl von Standardbezeichnern als Namen fuer Konstante, Typen, Variable, Prozeduren und Funktionen. Jeder dieser Bezeichner kann umdefiniert werden. Man sollte dies jedoch grundsaetzlich nicht tun, um von vornherein Missverstaendnisse und Probleme beim Austausch von Programmen zuvermeiden. Es wird deshalb die Umdefinition von Standardbezeichnern nicht erlaubt. Als Standardbezeichner werden verwendet:

Abs	ArcTan	Assign	Aux	AuxInPtr	AuxOutPtr
Bdos	BdosHL	BlockRead	BlockWrite	Bios	BiosHL
Boolean	Buflen	Byte	Chain	Char	Chr
Close	ClrEOL	ClrScr	Con	ConInPtr	ConOutPtr
Concat	ConstPtr	Copy	Cos	CrtExit	CrtInit
DelLine	Deley	Delete	Dispose	EOF	EOLN
Erase	Execute	Exp	False	FilePos	FileSize
FillChar	Flush	Frac	GetMem	GotoXY	HeapPtr
Hi	HighVideo	IOresult	Input	Insline	Insert
Int	Integer	Kbd	KeyPressed	Length	Ln
Lo	Lst	LstOutPtr	Mark	MaxInt	Mem
MemAvail	Move	New	NormVideo	Odd	Ord
Output	Overlay	Pi	Port	Pos	Pred
Ptr	Random	Randomize	Read	Readln	Real
RecurPtr	Release	Rename	Reset	Rewrite	Round
Seek	Sin	SizeOf	Sqr	Sqrt	StackPtr
Str	Succ	Swap	Text	Truc	True
Trunc	UpCase	Usr	UsrInPtr	UsrOutPrt	Val
Write	Writeln				

3.Standard-Scalartypen

Eine Datentyp-Definition legt die Menge der Werte fest, die Variable des entsprechenden Typs annehmen koennen und ordnet dem Typ einen Bezeichner zu.
Jede Variable im Programm muss einem und nur einem Datentyp zugeordent sein. Obwohl Datentypen in TURBO-Pascal gaenzlich veraendert werden koennen, werden sie alle aus einfachen (unstrukturierten) Typen aufgebaut.
Ein einfacher Typ kann entweder vom Programierer definiert werden (er heisst dann definierter Scalartyp) oder er ist ein Standard-Scalartyp: Integer, Real, Boolean, Char oder Byte.

3.1 **Integer**

Integer sind ganze Zahlen. In TURBO-Pascal ist ihr Wertebereich von -32768 bis 32767
Im Speicher benoetigen sie 2 Bytes. Ex ist zu beachten:

- -Ueberlauf von Integerzahlen wird nicht angezeigt.
- -Zwischenergebnisse muessen sich innerhalb des Wertebereiches halten, sonst ergeben sich falsche Werte. So ist beispielsweise 1000*100/50 nicht 2000 sondern -621 !

3.2 **Byte**

Im Speicher benoetigt dieser Typ genau 1 Byte. Er ist ein Teilbereich vom Typ Integer mit einem Wertebereich von:
0 bis 255.
Bytes sind deshalb mit Integervariablen kompatibel und koennen statt dessen definiert werden. In Ausdruecken koennen sie gemischt auftreten und Integervariablen koennen auch Bytes zugewiesen werden.

3.3 **Real**

Der absolute Wertebereich reicht fuer Realzahlen von
1E-38 bis 1E+38
(dabei bedeutet E: "mal 10 hoch") mit einer Mantisse von 11 signifikanten Zeichen. Der Typ benoetigt im Speicher 6 Bytes.
Bei Ueberlauf haelt das Programm an und meldet einen Ausfuehrungsfehler. Ein Unterlauf ergibt den Wert Null.
Obgleich Realzahlen ein Standardtyp sind, sollte man folgende Unterschiede zu den anderen Standardtypen beachten:

1) Sie duerfen in den Funktionen Pred und Suc nicht als Argumente auftreten.
2) Sie duerfen nicht als Indexzahlen in Bereichen verwendet werden.
3) Sie duerfen nicht den Grundtyp einer Menge definieren.
4) Sie duerfen nicht zur Steuerung von If und Case verwendet werden.
5) Von ihnen duerfen keine Teilbereiche definiert werden.

3.4 **Boolean**

Eine Boolesche Variable kann entweder den Wert wahr oder falsch haben: (True | False). Es ist festgelegt, dass False<True ist.

Der Type benoetigt im Speicher 1 Byte.

3.5 **Char**

Die Werte dieses Typs sind die in 2.1 definierte Zeichenmenge (ASCII-Zeichen). Die Menge ist entsprechend ihrem ASCII-Wert geordnet. Den normalen ASCII-Zeichen sind Zahlenwerte von 0-255 zugeordnet. Der Typ benoetigt im Speicher 1 Byte.

4. Nutzerdefinierte Sprachelemente

4.1 **Bezeichner**

Bezeichner werden zur Bezeichnung von Marken, Konstanten, Typen, Variablen, Prozeduren und Funktionen verwendet.

```
     <Bezeichner> ::= <Buchstabe>{<Buchstabe>|<Ziffer>}
```

Ein Bezeichner besteht also aus einem Buchstaben, dem Buchstaben und Ziffern folgen koennen. Die Laenge ist maximal 127 Zeichen und alle Zeichen sind signifikant.

Beispiel:

```
     TORBO
     square
     person_counted
     BirthDate
     3teWurzel          falsch! Ziffer am Anfang.
     zwei Worte         falsch! Leerzeichen unerlaubt.
```

Bei TURBO-Pascal gibt es zwischen grossen und kleinen Buchstaben keinen Unterschied. So sind

```
          SehrlangeDefinition = SEHRLANGEDEFINITION
```

identisch, aber der linke Bezeichner duerfte wohl klar leichter zu lesen sein.

4.2 **Zahlen**

Zahlen sind Konstante der Typen Integer und Real. Integerzahlen sind ganze Zahlen, die dezimal und hexadezimal dargestellt werden koennen. Hexadezimale Integerzahlen werden durch vorangestellte $-Zeichen erklaert.
Dezimalzahlen haben einen Bereich von -32768 .. +32767.
Hexadezimalzahlen haben einen Bereich von $0000 .. $FFFF.
Unmittelbar vor einer Dezimalzahl darf ein Vorzeichen stehen.

Beispiele:

```
     1
     12345
     -345
     $ABC
     $123
     $12c
     $12G           Falsch! G keine Hexadezimalzahl.
     $12.3          Falsch! Punkt keine Hexadezimalzahl.
```

Der Bereich der Realzahlen betraegt 1E-38 .. 1E+38 mit 11 signifikannten Ziffern. Die Exponentialdarstellung kann verwendet werden mit E als "mal 10 hoch". Eine Integerkonstante gilt ueberall dort, wo eine Realzahl gueltig ist. Trennzeichen duerfen nicht innerhalb von Zahlen stehen. Unmittelbar vor einer Zahl darf ein Vorzeichen stehen.

Beispiele:

```
     1.0
     1234.5678
     -0.024
     1E6
     2E-7
     -1.2345678901E+12
     1                  erlaubt, ist aber eine Integerkonstante.
```

4.3 **Zeichenkette**

Eine Zeichenkettenkonstante ist eine Folge von Zeichen, die in Hochkomma eingeschlossen sind:

```
          'Dies ist eine Zeichenketten '
```

In einer Zeichenkette kann auch ein Hochkomma enthalten sein, nur muss man es dann an der betreffenden Stelle zweimal angeben.
Zeichenketten, die nur aus einem Zeichen bestehen sind vom Typ Char. Eine Zeichenkette ist kompatibel mit einem Bereich von Char mit gleicher Laenge. Alle Zeichenkettenkonstanten sind mit allen Zeichenkettentypen kompatibel.

Beispiele:

```
     'TURBO'
     'Du wirst''s sehen!'
     ''''
     '.'
     ''                       leere Zeichenkette.
```

4.3.1 CXTRL-Steuerzeichen

TURBO-Pascal erlaubt die Verwendung von CTRL-Steuerzeichen als Zeichenketten. Dabei gibt es zwei Moeglichkeiten der Darstellung:

1) als #-Symbol gefolgt von einer dezimal oder hexadezimal Zahl. Damit wird ein Zeichen mit dem entsprechenden ASCII-Wert definiert.
2) als ^-Symbol gefolgt von einem ASCII-Zeichen. Damit wird das entsprechende CTRL-Zeichen definiert.

Beispiele:

```
     #10       entspricht     CTRL-J oder LINE FEED
     #$1B      entspricht     CTRL-[ oder ESCAPE
     ^G        entspricht     CTRL-G oder BELL
```

Folgen von Steuerzeichen koennen ohne Begrenzer aneinandergehaengt werden:

Beispiele:

```
     #13#10
     #27^U#20
     ^G^G^G^G
```

Steuerzeichen koennen auch mit anderen Zeichenketten gemischt auftreten:

Beispiele:
```
'Wach auf!'^G^G^G'Bitte, wach auf! '
```

4.4 **Kommentare**

Kommentare koennen ueberall im Programm stehen, wo Begrenzer stehen koennen. Sie werden durch geschweifte Klammern { } oder durch (* *) begrenzt.

Beispiele:
```
{Das ist ein Kommentar}
(* das ist das gleiche *)
```

Es koennen in Kommentaren nicht Kommentare mit den gleichen Begrenzern eingeschlossen werden. Aber es ist erlaubt in Kommentaren mit { } Kommentare mit (* *) einzuschliessen und umgekehrt. Damit kann man Quelltexte, die Kommentare enthalten in Kommentarklammern einschliessen und damit im Quelltext bei der Uebersetzung verschwinden lassen.

4.5 **Compiler-Direktiven**

Einige der Optionen des Compilers werden durch Direktiven gesteuert. Sie werden in den Quelltext als Kommentare mit einer speziellen Syntax eingefuegt. Sie koennen ueberall dort im Text stehen, wo Kommentare stehen koennen.
Eine Compiler-Direktive besteht aus einer oeffnenden Kommentar-Klammer der unmittelbar ein $-Zeichen und eine Compiler-Direktive oder eine Liste derartiger Direktiven folgt, die durch Komma untereinander getrennt sind. Die Syntax der Direktiv-Zeichen selbst haengt von der speziellen Direktive ab, die im Anhang E beschrieben sind.

Beispiele:
```
{$I-}
{$I E:INCLUDE.FIL}
{$R-,B+,V-}
```

5. Programmkopf und Programmblock

Ein Pascal-Programm besteht aus dem
- Programmkopf, dem der
- Programmblock folgt.

Der Programmblock selbst besteht aus dem
- Deklarationsteil und dem
- Anweisungsteil.

Im Deklarationsteil werden alle lokalen Objekte des Programmes definiert und im Anweisungteil stehen alle Aktionen, die mit diesen Objekten ausgefuehrt werden sollen.

5.1 **Programmkopf**

In TURBO-Pascal ist der Programmkopf voellig unverbindlich und hat fuer das Programm keinerlei Bedeutung. Wenn er angegeben wird, und dies ist sehr zu empfehlen, sollte er ausser dem reservierten Namen **Program** den Programmnamen und in der Parameterliste die speziell verwendeten Ein- und Ausgabegeraete enthalten. Die Liste Parameter besteht aus einer Folge von Bezeichnern, die durch Komma getrennt und in runde Klammern eingeschlossen sind.

Beipiele:

```
Program Circles;
Program Rechnung(Input,Output)
Program Druck(Input,Drucker)
```

5.2 **Deklarationsteil**

Der Deklarationsteil eines Blockes definiert alle Bezeichner, die im Anweisungsteil des Blockes und in dem in ihm moeglicherweise enthaltenen Bloecken verwendet werden. Er besteht aus fuenf Teilen:

1) Markenvereinbarungsteil
2) Konstantendefinitionsteil
3) Typdefinitionsteil
4) Variablenvereinbarungsteil
5) Prozedur- und Funktionsvereinbarungsteil

Waehrend Standard-Pascal festlegt, dass jeder dieser Teile nur Null oder einmal und nur in der oben angegeben Ordnung auftreten darf, erlaubt TURBO-Pascal, dass obige fuenf Teile mehrmals in beliebiger Ordnung im Deklarationsteil auftreten duerfen.

5.2.1 Markenvereinbarungsteil

Jede Anweisung ein einem Programm kann durch eine Marke gekennzeichnet werden. Dadurch wird es moeglich mittels eine GOTO-Anweisung direkt zu dieser Anweisung zu verzweigen.
Eine Marke besteht aus einem Markennamen, gefolgt von einem Doppelpunkt. Bevor eine Marke verwendet werden darf, muss sie im Markenvereinbarungsteil definiert sein. Die Definition beginn mit dem reservierten Wort **Label**, dem eine Liste von Markennamen, die alle durch Kommas voneinander getrennt sind, folgt und die durch ein Semikolon abgeschlossen wird.

Beispiel:

```
Label 10,error,999,Quit;
```

Waehrend Standard-Pascal nur maximal vierstellige ganze Zahlen als Markannamen zulaesst, erlaubt TURBO-Pascal sowohl die Verwendung von Ziffern als auch von Bezeichnern.

5.2.2 Konstantendefinitionsteil

Der Konstantendefinitionsteil enthaelt die Definitionen aller Synomyme fuer Konstanten, die in einem Block auftreten. Die Definition beginnt mit dem reservierten Wort **const**, dem eine Liste von Konstantenzuweisungen folgt, die durch Semikolon von-

einander getrennt sind. Jede Konstantenzuweisung besteht aus einem Bezeichner , dem ein Gleichheitszeichen und eine Konstante folgt.

Beispiel:

```
const
     Limit = 255;
     Max = 1024;
     PassWord = 'SESAM';
     CursHome = ^['V';
```

In TURBO-Pascal gibt es folgende fest definierten Konstante, die ohne vorherige Definition verwendet werden koennen:

Name	Typ	Wert
Pi	Real	-3.1415926536E+00
False	Boolean	Wahrheitswert: falsch
True	Boolean	Wahrheitswert: richtig
Maxint	Integer	32767

Im Konstantendefinitionsteil koennen auch typ-definierte Konstanten vereinbart werden (siehe 13).

5.2.3 Typdefinitionsteil

Ein Datentyp kann in Pascal entweder direkt im Variablendefinitionsteil oder durch einen Typbezeichner beschrieben werden. Es gibt bestimmte festgelegte Typbezeichner, aber der Nutzer kann durch Verwendung der Typdefinition diese Liste beliebig erweitern.
Die Typdefinition beginnt mit dem reservierten Wort **type**, dem eine oder mehrere Zuweisungen folgen, die durch Semikolon voneinander getrennt werden. Jede Zuweisung besteht aus einem Typ-Bezeichner, dem ein Gleichheitszeichen und ein Typ folgt.

Beispiele:

```
type
     Number = Integer;
     Tag = (Montag,Dienstag,Mittwoch,Donnerstag,
            Freitag,Sonnabend,Sonntag);
     List = array[1..10] of Real;
```

5.2.4 Variablenvereinbarungsteil

Jede in einem Programm auftretende Variable muss vorher definiert werden. Die Definition muss im Text vor der ersten Verwendung der Variablen auftreten, d.h. die Variable muss dem Compiler bekannt sein, bevor sie verwendet wird.
Eine Variablenvereinbarung beginnt mit dem reservierten Wort **Var**, dem ein oder mehrere Bezeichner folgen, die durch Kommas voneinander getrennt sind. Dieser Liste folgt ein Doppelpunkt und ein Typ. Damit werden von dem spezifizierten Typ neue Variable aufgebaut, denen als Namen die spezifizierten Bezeichner zugewiesen sind. Der Gueltigkeitsbereich der Bezeichner ist der Block, in

dem sie definiert wurden. Zu diesem Gueltigkeitsbereich gehoeren auch alle Bloecke, die in dem genannten Block enthalten sind. Man beachte jedoch, dass in jedem solchen Block, der in einem anderen enthalten ist, andere Variable definiert werden koennen, die die gleichen Bezeichner verwenden. Diese Variable heissen dann lokal zu diesem Block und zu anderen Bloecken, die in ihm enthalten sind. Variable, die ausserhalb eines Blockes definiert wurden, heissen globale Variable und sind, wenn lokale Variable mit gleichem Bezeichner auftreten in diesem Block dann nicht mehr zugaenglich.

Beispiel:

```
Var
     Result,Intermediate,SubTotal:Real;
     I,J,X,Y:Integer;
     Accepted,Valid:Boolean;
     Period:Tag
     Buffer:array[0..127] of Byte
```

5.2.5 Prozedur- und Funktionsvereinbarungsteil

Eine Prozedurvereinbarung dient der Definition einer Prozedur innerhalb einer anderen Prozedur oder eines Programmes (siehe 16.2). Eine Prozedur wird durch eine Prozeduranweisung aktiviert (siehe 7.1.2) und nach der Ausfuehrung dieser Prozeduranweisung wird das Programm mit der Anweisung fortgesetzt, die unmittelbar der Call-Anweisung der Prozedur folgt.
Eine Funktionsvereinbarung dient zur Defintion eines Programmteiles, das eine Variable berechnet und zurueckgibt (siehe 16.3). Eine Funktion wird aktiviert, wenn sein Bezeichner als Teil eines Ausdruckes auftritt (siehe 6.2).

5.3 **Anweisungsteil**

Der Anweisungsteil ist der letzte Teil eines Blockes. Er spezifiziert die Aktionen, die durch das Programm ausgefuehrt werden. Der Anweisungsteil besteht aus einer Verbundanweisung, der ein Punkt '.' folgt. Eine Verbundanweisung besteht aus dem reservierten Wort **begin**, dem eine Liste von einzelnen Anweisungen folgt. Diese sind in der Liste durch Semikolon voneinander getrennt und werden durch das reservierte Wort **end** abgeschlossen.

6. Ausdruecke

Ausdruecke sind Konstruktionen, die Regeln fuer das Rechnen mit den vorliegenden Werten von Variablen und die Erzeugung neuer Werte durch Anwendung von Operatoren zum Inhalt haben. Sie bestehen aus Operanden, d.h. Variablen, Konstanten und Funktionsbezeichnungen, die mittels der unten definierten Operatoren kombiniert werden. Dieser Abschnitt beschreibt, wie Ausdruecke aus Standard-Skalartypen Integer, Real, Boolean und Char gebildet werden. Ausdruecke, die definierte Skalartypen, Stringtypen und

Mengentypen enthalten, werden in 8.1, 9.2 und 12.2 respektive definiert.

6.1 **Operatoren**

Operatoren koennen entsprechend ihrer Rangfolge in fuenf Kategorien eingeteilt werden:

1) Minusvorzeichen (Minus mit einem Operanden).
2) Not Operator.
3) Multiplikationsoperatoren: *,/,div,mod,and,shl,shr.
4) Additionsoperatoren: +,-,or,xor.
5) Vergleichsoperatoren: =,<>,<,>,<=,>=,in.

Folgen von Operatoren gleichen Ranges werden von links nach rechts abgearbeitet.
Ausdruecke in Klammern werden zuerst abgearbeitet, unabhaengig von den davor oder dahinter stehenden Operatoren. Sind beide Operanden eines Multiplikations- oder Additionsoperators vom Typ Integer, dann ist auch das Ergebnis vom Integertyp. Wenn einer oder beide Operanden vom Typ Real sind, dann ist auch das Ergebnis vom Realtyp.

6.1.1 Minuszeichen

Das Minuszeichen bezeichnet die Negation des Operanden, der vom Typ Integer oder Real sein muss.

6.1.2 NOT-Operator

Der NOT-Operator negiert den logichen Wert eines Booleschen Operanden:

```
not True  = False
not False = True
```

TURBO-Pascal erlaubt auch die Anwendung des NOT-Operators auf Integer-Operanden. In diesem Falle erfolgt die Negation der einzelnen Bits.

Beispiele:

```
not 0     = -1
not -15   = 14
not $2345 = $DCBA
```

6.1.3 Multiplikationsoperanden

Operator	Operation	Operandentypen	Ergebnistyp
*	Multiplikation	Real,Real	Real
*	Multiplikation	Integer,Integer	Integer
*	Multiplikation	Integer,Real	Real
/	Division	Real,Real	Real
/	Division	Integer,Real	Real

/	Disision	Integer,Integer	Real
div	Ganzzahlige Division	Integer	Integer
mod	Modulo	Integer	Integer
and	Arithmetisches and	Integer	Integer
and	Logisches and	Boolean	Boolean
shl	Shift links	Integer	Integer
shr	Shift rechts	Integer	Integer

Beispiele:

```
     123*456        =    492 falsch, Ueberlauf der Integerzahl!
     123/4          =    30.75
     123 div 4      =    30
     12 mod 5       =    2
     true and false =    falsch
     12  and 22     =    4
     2   shl 7      =    256
     256 shr 7      =    2
```

6.1.4 Additionsoperatoren

Operator	Operation	Operandentypen	Ergebnistyp
+	Addition	Real,Real	Real
+	Addition	Real,Integer	Real
+	Addition	Integer,Integer	Integer
-	Subtraktion	Real,Real	Real
-	Subtraktion	Real,Integer	Real
-	Subtraktion	Integer,Integer	Integer
or	Arithmetisches or	Integer,Integer	Integer
or	Logisches or	Boolean,Boolean	Boolean
xor	Arithmetisches xor	Integer,Integer	Integer
xor	Logisches xor	Boolean,Boolean	Boolean

Beispiele:

```
     123 + 456      =    579
     456 - 123.0    =    333.0
     true or false  =    True
     12 or 22       =    30
     true xor false =    True
     12 xor 22      =    26
```

6.1.5 Vergleichsoperatoren

Vergleichsoperatoren koennen mit allen Standardtypen Real, Integer, Boolean, Char und Byte verbunden werden. Operanden vom Typ Integer, Real und Byte koennen gemischt auftreten. Der Ergebnistyp ist stets Boolean, d.h. True oder False.
Es bedeuten:

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

Beispiele:

```
     a =  b    True,wenn      a    gleich    b
     a <> b    True,wenn      a    ungleich  b
     a >  b    True,wenn      a    groesser  b
```

```
a <  b    True,wenn      a    kleiner   b
a >= b    True,wenn      a    groesser oder gleich  b
a <= b    True,wenn      a    kleiner  oder gleich  b
```

6.2 **Funktionsaufruf**

Ein Funktionsaufruf besteht aus einem Funktionsbezeichner, dem eine wahlfreie Parameterliste folgt. Diese Liste besteht aus einer oder mehreren Variablen oder Ausdruecken, die durch Komma voneinander getrennt und insgesamt in runde Klammern eingeschlossen sind.
Das Auftreten eines Funktionsaufrufes im Programm bewirkt die Aktivierung der Funktion, die durch sie bezeichnet wird. Wenn die Funktion keine Standardfunktion ist, muss sie vor der Aktivierung definiert sein.

Beispiele:

```
 Round(PlotPos)
 Writeln(Pi * (Sqr(R)))
 (Max(x,y) < 25) and (Z > Sqrt(x*y))
 Volumen(Radius,Hoehe)
```

7.Anweisungen

Der Anweisungsteil eines Programmes, einer Prozedur oder einer Funktion definiert die algorithmischen Aktionen als Folge ausfuehrbarer Anweisungen. Diese Folge besteht aus Anweisungen, die durch Semikolon voneinander getrennt sind. Sie beginnt mit dem reservierten Wort **begin** und endet mit dem reservierten Wort **end**. Anweisungen in Pascal sind entweder einfache oder strukturierte Anweisungen.

7.1 **Einfache Anweisungen**

Einfache Anweisungen sind Anweisungen, die keine weiteren Anweisungen enthalten. Es sind dies die

- Ergibtanweisung,
- Prozeduranweisung,
- Sprunganweisung und
- Leeranweisung.

7.1.1 Ergibtanweisung

Die Ergibtanweisung ist die fundamentalste aller Anweisungen. Sie wird verwendet, um einer bestimmten Variablen einen bestimmten Wert zuzuweisen. Eine Ergibtanweisung besteht aus einem Variablenbezeichner, dem Ergibtoperator := und einem Ausdruck. Eine Zuweisung ist moeglich zu Variablen von beliebigem Typ (ausser Files), wenn der Ausdruck vom gleiche Typ ist. Es gibt nur eine Ausnahme: Ist die Variable vom Realtyp, kann der Ausdruck vom Integertyp sein.

Beispiele:

```
Angle    :=   Angle * Pi;
AccessOk :=   False;
Entry    :=   Answer = Password;
SpherVol :=   4 * Pi * R * R;
UpCas    :=   (Ch > 'A') and (Num <= 'Z')
```

7.1.2 Prozeduranweisung

Eine Prozeduranweisung dient der Aktivierung einer Standardprozedur oder einer vorher vom Nutzer definierten Prozedur. Die Anweisung besteht aus dem Prozedurbezeichner, dem wahlweise eine Parameterliste folgt. Diese Liste besteht aus Variablen oder Ausdruecken, die durch Komma voneinander getrennt und in runden Klammern eingeschlossen sind. Wird die Prozedur bei der Programmausfuehrung erreicht, geht die Steuerung an die Prozedur ueber und die angegebenen Parameterwerte werden uebergeben. Wenn die Prozedur ausgefuehrt wurde, wird das Programm mit der Ausfuehrung der dieser Prozedur folgenden Anweisung fortgesetzt.

Beispiele:

```
 Find(Name,Adresse);
Sort(Adresse);
UpperCase(Text);
UpdateLastFile(LastRecord);
```

7.1.3 Sprunganweisung

Eine Sprunganweisung besteht aus dem reservierten Wort **goto**, dem ein Markenbezeichner folgt. Sie dient der Fortfuehrung des Programmes an der Stelle im Programmtext, an der die Marke steht.
Es gelten folgende Einschraenkungen:

1) Jede Marke muss durch eine Markenvereinbarung im Kopf des Blockes definiert und durch Kennzeichnung einer Anweisung festgelegt sein.
2) Der Gueltigkeitsbereich einer Marke ist der Block, in dem die Marke definiert ist. Deshalb ist es nicht moeglich in oder aus einer Prozedur oder Funktion zu springen.

7.1.4 Leeranweisung

Eine Leeranweisung ist eine Anweisung, die aus keinem Symbol besteht und die nichts macht. Sie kann entstehen, wenn die Pascal-Syntax eine Anweisung verlangt, aber keine vom Algorithmus her benoetigt wird.

Beispiele:

```
begin end.
while Antwort <> '' do;
repeat until KeyPressed; {Wartet bis Taste gedrueckt wurde}
```

7.2 **Strukturierte Anweisungen**

Strukturierte Anweisungen sind aus anderen Anweisungen zusammengesetzte Konstruktionen, die entweder nacheinander auszufuehren sind (Verbundanweisungen), bedingt auszufuehren sind (Bedingte Anweisungen) oder zu wiederholen sind (Zyklusanweisungen). Die Besprechung der WITH-Anweisung erfolgt in 11.2.

7.2.1 Verbundanweisung

Eine Verbundanweisung wird verwendet, wenn mehr als eine Anweisung an einer Stelle auszufuehren sind, an der die Pascal-Syntax die Spezifikation von nur einer Anweisung erlaubt. Sie besteht aus einer beliebigen Anzahl von Anweisungen, die durch Semikolon voneinander getrennt und in die reservierten Worte **begin** und **end** eingeschlossen werden. Die einzelnen Anweisungen in der Verbundanweisung werden nacheinander in der Reihenfolge ausgefuehrt, in der sie aufgeschrieben wurden.

Beispiel:

```
     if Small > Big than
          begin
               Tmp   :=   Small;
               Small :=   Big;
               Big   :=   Tmp;
          end;
```

7.2.2 Bedingte Anweisungen

Eine bedingte Anweisung waehlt eine einzelne ihrer Komponentenanweisungen zur Ausfuehrung aus.

7.2.2.1 **IF-Anweisung**

Die **IF**-Anweisung waehlt die nach dem reservierten Wort **then** stehende Anweisung nur dann zur Ausfuehrung aus, wenn eine bestimmte Bedingung (Boolescher Ausdruck) wahr ist. Ist sie falsch, dann wird entweder keine Anweisung oder die Anweisung ausgefuehrt, die dem wahlweise stehenden reservierten Wort **else** folgt. Da die bedingten IF-Anweisungen geschachtelt werden koennen, entstehen zweideutige Konstruktionen. Die Konstruktion :

```
     if  expr1  then
         if  expr2  then
         stmt1
         else
         stmt2
```

wird wie folgt ausgefuehrt:

```
     if  expr1  then
         begin
          if  expr2  then
          stmt1
          else
          stmt2
         end;
```

Damit ist klar: Die ELSE-Klausel gehoert stets zur letzten IF-Anweisung, die keine ELSE-Klausel hat.

Beispiele:

```
     if Interest > 25 then
          Usury := True
     else
          TakeLoan := OK;

     if (Entry < 0) or (Entry > 100) then
          begin
               Write('Bereich ist 1 bis 100, bitte eingeben:';
               Read(Entry);
          end;
```

7.2.2.2 **CASE-Anweisung**

Die CASE-Anweisung besteht aus einem Ausdruck (dem Selektor), der in die reservierten Worte **case** und **of** eingeschlossen ist, und einer Liste von Anweisungen, von denen jede durch eine Konstante vom Typ des Selektors markiert ist. Die Liste wird durch das reservierte Wort **end** abgeschlossen. Die CASE-Anweisung legt fest, dass die eine Anweisung auszufuehren ist, deren CASE-Marke gleich dem vorliegenden Wert des Selektors ist. Enthaelt keine der CASE-Marken den vorliegenden Wert, dann wird entweder keine Anweisung ausgefuehrt, oder wahlweise die Anweisung, die dem reservierten Wort **else** folgt. Die ELSE-Klausel ist eine Erweiterung gegenueber Standard-Pascal.
Eine CASE-Marke kann aus einer beliebigen Anzahl von Konstanten oder Teilbereichen bestehen, die durch Komma getrennt sind und durch einen Doppelpunkt abgeschlossen werden. Ein Teilbereich wird dargestellt durch zwei Konstanten, die durch den Teilbereichsbegrenzer '..' getrennt sind. Der Typ der Konstanten muss der gleiche wie der des Selektors sein. Die Anweisung, die der

CASE-Marke folgt, wird ausgefuehrt, wenn der Wert des Selektors gleich einer der Konstanten ist oder in einem der Teilbereiche liegt.
Gueltige Selektortypen sind alle einfachen Typen, d.h. alle Skalartypen ausser Real.

Beispiele:

```
     case Operator of
          '+': Result := Input + Result;
          '-': Result := Innput * Result;
          '/': Result := Input / Result;
          end;

     case Year of
          Min..1939: begin
                          Time := PreWordWar2;
                          write('The world at peace...');
                     end;
          1946..Max: begin
                          Time := PostWorldWar2;
                          write('Building a new world');
```

```
            end;
     else
          Time := WorldWar2;
          writeln('We are at war');
     end;
```

7.2.3 Zyklusanweisungen

Zyklusanweisungen legen fest, dass gewisse Anweisungen wiederholt ausgefuehrt werden sollen. Wenn die Anzahl der Wiederholungen vorher bekannt ist, d.h. bevor der Zyklus gestartet wird, ist die FOR-Anweisung die angemessene Konstruktion. Andernfalls sollte die WHILE-Anweisung oder die REPEAT-Anweisung verwendet werden.

7.2.3.1 **WHILE-Anweisung**

Die WHILE-Anweisung hat die Form:

```
     while expr do stmt;
```

Der Ausdruck, der die Wiederholungen steuert, muss vom Typ Boolean sein. Die Anweisung wird solange ausgefuehrt, wie der Ausdruck wahr ist. Ist der Ausdruck zu Beginn schon falsch, wird gar keine Anweisung ausgefuehrt.

7.2.3.2 **REPEAT-Anweisung**

Die Wiederholanweisung REPEAT hat die Form:

```
     repeat stmt1;stmt2;...;stmtN until expr;
```

Der die Wiederholung steuernde Ausdruck muss vom Type Boolean sein. Die Folge der Anweisungen zwischen repeat und until wird wiederholt (mindestens jedoch einmal) ausgefuehrt, bis der Ausdruck den Wert True erhaelt.

Beispiele:

```
     repeat
          write(^M,'Delete this item? (Y/N)');
          read(Answer)
     until UpCase(Answer) in ['Y','N'];
```

7.2.3.3 **FOR-Anweisung**

Die FOR-Anweisung gibt an, dass eine Anweisung wiederholt ausgefuehrt werden soll, waehrend einer Variablen (der Laufvariablen) eine Folge von Werten zugewiesen wird. Die Folge kann auf- oder absteigend sein bis zum Endwert. Die Laufvariable, der Anfangswert und der Endwert muessen alle vom gleichen Typ sein. Gueltig sind alle einfachen Typen, d.h. alle Skalartypen ausser Real.
Die Laufanweisung hat die Form:

```
     for var := Anfwert to Endwert do stmt;
```

oder

```
     for var := Anfwert downto Endwert do stmt;
```

Dabei darf die Laufvariable nicht durch die Anweisung geaendert werden.
Falls Anfangswert > Endwert bei einer TO-Klausel oder Angangswert < Endwert bei einer DOWNTO-Klausel ist, wird keine Anweisung ausgefuehrt.

Beispiele:

```
     for i := 2 to 100 do if A[i] > Max then Max := A[i];

     for i := 1 to NoOflines do
          begin
           readln(Line);
           if Length(Line)<Limit then ShortLines := ShortLines+1;
           else
              LongLines := LongLines+1;
          end;
```

Falls die Wiederholung beendet werden soll, bevor der Endwert erreicht ist, muss eine GOTO-Anweisung benutzt werden. In diesen Faellen ist es besser eine WHILE- oder REPEAT-Anweisung zu verwenden.
Wenn die FOR-Anweisung verlassen wird, hat die Laufvariable den Endwert, ausser wenn die Schleife nie durchlaufen wurde. In diesem Falle wird der Laufvariablen kein Wert zugewiesen.

8. Skalar- und Teilbereichstypen

Die fundamentalen Datentypen in Pascal sind die Skalartypen. Sie definieren jeweils eine endliche und lineargeordnete Menge von Werten. Obgleich der Standardtyp Real zu den Skalartypen gehoert, stimmt er nicht voellig mit dieser Definition ueberein. Aus diesem Grunde duerfen auch Realtypen nicht ueberall dort im Text verwendet werden, wo Skalartypen eingesetzt werden koennen.

8.1 Skalartypen

Abgesehen von den Standard-Skalartypen (Integer, Real, Boolean, Char und Byte) unterstuetzt TURBO-Pascal vom Nutzer definierte Skalartypen, sogenannte erklaerte oder definierte Skalartypen. Die Definition eines solchen Skalartypes gibt die einzelnen moeglichen Werte in geordneter Folge an. Die Werte dieses Types werden durch die Bezeichner dargestellt, die die Konstanten des neuen Types sein werden.

Beispiele:

```
     type
          Karte = (Karo,Herz,Pik,Kreuz);
```

```
          Tag   = (Mo,Di,Mi,Do,Fr,Sa,So);
          Monat = (Jan,Feb,Mrz,Apr,Mai,Jun,Jul,Aug,Sep,Nov,Dez);
          Operator = (Plus,Minus,Mult,Div);
```

Variable vom Typ Karte koennen also die vier Werte: Karo, Herz, Pik und Kreuz annehmen.
Bereits bekannt ist der Standard-Skalartyp Boolean. Mit diesem Definitionsverfahren kann man ihn definieren als:

```
     type
          Boolean = (False,True)
```

Die Vergleichsoperatoren =,<,>,<>,>=,<= kann man auf alle Skalartypen anwenden, solange beide Operanden vom gleichen Typ sind (Real und Integer duerfen gemischt auftreten). Die in der Definition festgelegte Ordnung wird fuer den Vergleich verwendet, d.h. links stehende Werte sind kleiner als rechts stehende. Fuer den Typ Karte gilt also:

```
          Karo < Herz < Pik < Kreuz
```

Bei folgenden Funktionen duerfen die Argumente Skalartypen sein:

```
     Succ(Pik)      Der Nachfolger von Pik ist Kreuz.
     Pred(Pik)      Der Vorgaenger von Pik ist Herz.
     Ord(Pik)       Der Ordnungswert von Pik ist 2.
```

Der Typ des Ergebnisses von Succ und Pred ist der gleiche, wie der Typ des Argumentes. Das Ergebnis von Ord ist vom Integertyp.

8.2 **Teilbereichstypen**

Ein Typ kann als Teilbereich eines bereits definierten Skalartypes definiert werden und zwar durch Angabe des kleinsten und des groessten Wertes des Teilbereiches. Die erste Konstante darf dabei nicht groesser als die zweite sein. Ein Teilbereich vom Typ Real ist nicht erlaubt.

Beispiele:

```
     type
          HemiSphere = (North,South,East,West);
          World      = (East,West);
          CompassRange = 0..360;
          Gross      = 'A'..'Z';
          Klein      = 'a'..'z';
          Grad       = (Celc,Fahr,Ream,Kelv);
          Wein       = (Rot,Weiss,Rose,Schaum);
```

Hierbei ist World ein Teilbereich des Skalartyps HemiSphere. Letzterer heisst der assoziative Skalartyp des ersteren. Der assoziative Skalartyp von CompassRange ist Integer und der von Gross und Klein ist Char.
Der bereits genannte Standard-Skalartyp Byte kann mit dieser Konstruktion definiert werden als:

```
     type
```

```
Byte = 0..255
```

Ein Teilbereichstyp behaelt alle Eigenschaften seines assoziativen Skalartyps und ist nur durch den Bereich seiner Werte beschraenkt.
Die Verwendung definierter Skalar- und Teilbereichstypen wird dringend empfohlen, da sie die Lesbarkeit von Programmen sehr verbessern. Ausserdem koenne leicht Pruefungen in den Programmtext eingefuegt werden (siehe 8.4), die waehrend der Ausfuehrung des Programmes die den Skalar- und Teilbereichsvariablen zugeordneten Werte ueberwachen. Ein weiterer Vorteil ist, dass diese Typen Speicherbereich sparen. TURBO-Pascal ordnet einer Variablen eines Skalar- oder Teilbereichstyps nur ein Byte zu, falls die Anzahl der Elemente des Typs kleiner als 256 ist. Analog benoetigen Integer Teilbereichsvariable, deren untere und obere Grenze innerhalb von 0 bis 255 liegen, ebenfalls nur einen Speicherplatz von einem Byte.

8.3 **Typumwandlung**

Die Funktion Ord kann man verwenden, um Skalartypen in Werte vom Integertyp umzuwandeln. Standard-Pascal enthaelt jedoch keine Moeglichkeit der Umwandlung in der anderen Richtung, d.h. der Umwandlung eines Integerwertes in einen anderen Skalarwert. TURBO-Pascal enthaelt jedoch die Moeglichkeit des "retyping": Man kann den Wert eines Skalartyps in den Wert eines anderen Skalartyps umwandeln, der die gleiche Ordnungszahl besitzt. Diese Umwandlung erreicht man, indem man den Typbezeichner des entsprechenden Typs als Funktionsbezeichner verwendet und in Klammern einen Parameter folgen laesst. Als Parameter setzt man den Wert des anderen Skalartyps ein. Als Wert wird der entsprechende Wert des ersten Skalartyps zurueckgegeben. Retyping kann man auf alle Skalartypen ausser Real anwenden.

Unter Verwendung obiger Definitionsbeispiele gilt dann folgendes:

```
Integer(Herz)  = 2
Monat(10)      = Nov
HemiSphere(2)  = East
Gross(14)      = 'N'
Grad(3)        = Kelv
Char(78)       = 'N'
Integer('7')   = 55
```

8.4 **Bereichspruefung**

Das Einfuegen von Bereichspruefungen fuer Skalar- und Teilbereichsvariable in den Programmtext wird durch die Compiler-Direktive {$R+} erzeugt. Der Standardwert fuer diese Direktive ist {$R-}, d.h. im Normalfall wird keine Bereichspruefung eingebaut. Wenn diese Direktive aktiv ist, wird bei jeder Zuweisung eines Wertes zu einer Skalar- oder Teilbereichsvariablen geprueft, ob dieser Wert auch in dem Bereich der betreffenden Variable liegt.

Es wird empfohlen, diese Pruefungen solange in neuen Programmen zu belassen, bis sie voellig ausgetestet sind.

Beispiele:

```
     program RangeCheck;
          type
               Digit = 0..9;
          var
               Digit1,Digit2,Digit3: Digit;
          begin
               Dig1 := 5;       {gueltig}
               Dig2 := Dig1 + 3;{gueltig}
               Dig3 := 47       {ungueltig, keine Fehlermeldung}
             {$R+} Dig3 := 55       {ungueltig, Fehlermeldung}
             {$R-} Dig3 := 167      {ungueltig, keine Fehlermeldung}
```

9. Zeichenkettentypen (Strings)

TURBO-Pascal gestattet den Gebrauch von Stringtypen, d.h. die Verarbeitung von Zeichenketten. Stringtypen sind strukturierte Typen, die in vielerlei Weise den Feldtypen (siehe 10: array) aehneln. Es gibt jedoch einen Hauptunterschied zwischen beiden. Die Anzahl der Zeichen in einer Zeichenkette (ihre Laenge) kann sehr stark dynamisch zwischen 0 und einer festgelegten oberen Grenze waehrend der Verarbeitung schwanken, im Gegensatz dazu ist die Anzahl der Elemente eines Feldes immer fest.

9.1 **Stringtyp-Definition**

Die Definition eines Stringtyp erfolgt durch Angabe der maximalen Anzahl der Zeichen, die er enthalten kann, durch seine maximale Laenge:

```
     type      name = string[n];
```

Die Definition besteht also aus dem reservierten Wort **string**, dem die Laenge n in eckigen Klammern folgt. Die Laenge n ist eine Integerkonstante im Bereich 1 bis 255. Es gibt keine Standard Laenge, sie muss immer definiert werden.

Beispiele:

```
     type
          FileName   = string[14];
          ScreenLine = string[80];
```

Eine Stringvariable der Laenge n belegt im Speicher n+1 Byte. Das zusaetzliche Byte enthaelt die aktuelle Laenge der Variablen. Die einzelnen Zeichen im String sind indiziert von 1 bis n.

9.2 **STRING-Ausdruecke**

Zeichenketten koennen durch Bildung von STRING-Ausdruecken manipuliert werden. STRING-Ausdruecke bestehen aus STRING-Konstanten, STRING-Variablen, Funktionsaufrufen und Operatoren.

Das Pluszeichen kann man als STRING-Operator verwenden, um zwei Zeichenketten miteinander zu verknuepfen. Die CONCAT-Funktion (siehe 9.5) vermag das gleiche, aber der + Operator wird vorangig verwendet. Das Ergebnis ist die Verkettung beider Operanden. Ist die Laenge des Ergebnis-String laenger als 255, entsteht ein Laufzeitfehler.

Beispiele:
```
     'TURBO-' + 'Pascal'       = 'TURBO-Pascal'
     '123'    + '456'          = '123456'
     'A' + 'B' + 'C' + 'D   '  = 'ABCD   '
```

Die Vergleichsoperatoren haben eine niedrigere Rangfolge, als der Kettungs-Operator +. Wenn sie auf STRING-Operanden angewendet werden, ist das Ergebnis ein Boolescher Wert (True oder False). Werden zwei Strings miteinander verglichen, erfolgt der Vergleich von links nach rechts. Sind die Strings von unterschiedlicher Laenge, aber gleich bis einschliesslich zum letzten Zeichen des kuerzeren Strings, dann ist der kuerzere kleiner. Strings sind nur dann gleich, wenn sie gleiche Laenge und gleichen Inhalt haben.

Beispiele:
```
     'A' < 'B'                                  =  richtig
     'A' > 'b'                                  =  falsch
     '2' < '12'                                 =  falsch
     'TURBO'   =   'TURBO'                      =  richtig
     'TURBO '  =   'TURBO'                      =  falsch
     'Pascal Compiler ' < 'Pascal compiler '  =  richtig
```

9.3 **STRING-Ergibtanweisung**

Der Ergibtoperator wird verwendet, um das Ergebnis eines STRING-Ausdruckes einer STRING-Variablen zuzuweisen.

Beispiele:
```
     Alter  :=  'fuenfzehnten';
     Zeile  :=  'Herzlichen Glueckwunsch zu Deinem ' + Alter +
                'Geburtstag';
```

Wird die maximale Laenge einer STRING-Variablen ueberschritten (durch Zuweisung zuvieler Zeichen), so werden die restlichen Zeichen abgeschnitten. Wurde beispielsweise die Variable Alter mit `type Alter = string[5];` definiert, dann ist ihr Wert nach obiger Zuweisung 'fuenf'.

9.4 **STRING-Prozeduren**

In TURBO-Pascal koennen die folgenden Standard-STRING-Prozeduren verwendet werden:

9.4.1 DELETE-Prozedur

Syntax: Delete(str,pos,num);

Die Parameter sind:

str	STRING Variable,
pos	Integer Ausdruck,
num	Integer Ausdruck.

Delete erzeugt aus str einen Teilstring durch Loeschen von num Zeichen beginnend ab Position pos. Wenn pos groesse als die Laenge von str ist, wird kein Zeichen geloescht. Wenn pos+num ausserhalb der Zeichenkette liegt, werden nur die Zeichen geloescht, die ab pos innerhalb liegen. Liegt pos nicht in 1..255, wird ein Laufzeitfehler erzeugt.

Beispiele:

```
type  str = string[10];
str :=  'ABCDEFG';
Delete(str,2,4);  ergibt 'AFG'.
Delete(str,2,10); ergibt 'A'.
```

9.4.2 INSERT-Prozedur

Syntax: Insert(obj,target,pos);

Die Parameter sind:

obj	STRING Ausdruck,
target	STRING Variable,
pos	Integer Ausdruck.

Insert fuegt den String obj in den String target ab Position pos ein. Ist pos groesser als die Laenge von target, wird obj an target angefuegt. Wenn das Ergebnis laenger als die maximale Laenge von target ist, werden die ueberstehenden Zeichen abgeschnitten und target erhaelt nur die links stehenden. Wenn pos ausserhalb von 1..255 liegt, entsteht ein Laufzeitfehler. Mit dem oben definierten String str='ABCDEFG' ergeben sich folgende

Beispiele:

```
Insert('XX',str,3);     ergibt  'ABXXEFG'
Insert('UVWXYZ',str,4); ergibt  'ABCUVWX'
```

9.4.3 STR-Prozedur

Syntax: Str(value,str-var);

Die Paramter sind:

value	WRITE-Prozedur-Parameter vom INTEGER/REAL-Typ
str-var	STRING-Variable.

Die STR-Prozedur konvertiert den numerischen Wert von value in eine Zeichenkette und speichert sie in str-var ab.

Beispiele:

```
Wenn      I = 1234
ergibt    Str(I:5,str1);    in str1 = ' 1234'
```

```
     Wenn      X = 2.5E4
     ergibt    Str(x:10:0,str2); in str2 = ' 25000'
```

9.4.4 VAL-Prozedur

```
Syntax:   Val(str,var,code);
```

Die Parameter sind:

str	STRING-Ausdruck,
var	INTEGER/REAL-Variable,
Code	INTEGER-Variable.

Der STRING-Ausdruck str muss den Regeln einer numerischen Konstanten (siehe 4.2) genuegen. Weder fuehrende noch nachfolgende Leerzeichen sind erlaubt. Val konvertiert die Konstante zu einem Wert vom gleichen Typ wie var und speichert diesen Wert in var ab. Wird kein Fehler festgestellt, ist der Wert der Variablen code=0. Andernfalls erhaelt code den Wert der Position des ersten fehlerhaften Zeichens in str und der Wert von var ist undefiniert.

Beispiele:

```
     Wenn      str1 = '234'   ergibt
     Val(str1,I,Result);      I = 234        Result = 0

     Wenn      str2 = ' 234'  ergibt
     Val(str2,I,Result);      I = undef.     Result = 1

     Wenn      str3 = '2.5E4' ergibt
     Val(str3,X,Result);      X = 25000      Result = 0
```

9.5 **STRING-Funktionen**

In TURBO-Pascal stehen die folgenden Sandard-STRING-Funktionen zur Verfuegung:

9.5.5 COPY-Funktion

```
Syntax    Copy(str,pos,num);
```

Die Parameter sind:

str	STRING-Ausdruck,
pos	INTEGER-Ausdruck,
num	INTEGER-Ausdruck.

Copy gibt einen String zurueck, der aus num Zeichen von str, beginnend ab Position pos, besteht. Wenn pos > Laenge(str) ist, besteht das Ergebnis aus der leeren Zeichenkette ''. Wenn pos+num ausserhalb von str liegt, werden nur die innerhalb von str liegenden Zeichen zurueckgegeben. Liegt pos nicht in 1..255, so entsteht ein Laufzeitfehler.

Beispiele:

```
     Wenn str1 := 'ABCDEFG' , dann ist
```

```
     str2 := Copy(str1,3,2);   gleich    'CD'
     str2 := Copy(str1,4,10);  gleich    'DEFG'
     str2 := Copy(str1,4,2);   gleich    'DE'
```

9.5.2 CONCAT-Funktion

Syntax: Concat(str1,str2{,strN});

Die Parameter sind:

str1	STRING-Ausdruck,
str2	STRING-Ausdruck,
....	.....
strN	STRING-Ausdruck.

Die Anzahl N der STRING-Ausduecke ist beliebig. Das Ergebnis der CONCAT-Funktion ist eine Zeichenkette, die aus den einzelnen Zeichenketten durch Kettung entsteht und zwar in der gleichen Reichenfolge, wie die STRING-Parameter in der Liste stehen. Wird die Laenge groesser als 255, entsteht ein Laufzeitfehler. Man kann, wie bereits in 9.3 erwaehnt, mit dem +-Operator das gleiche erhalten. CONCAT sichert nur die Kompatibilitaet mit anderen Compilern.

Beispiel:

```
     str1 := 'TURBO ';
     str2 := 'ist am schnellsten';
     str3 := Concat(str1,'-Pascal ',str2); ergibt
     str3 gleich 'TURBO-Pascal ist am schnellsten'
```

9.5.3 LENGTH-Funktion

Syntax: Length(str);

Der Parameter ist:

str	STRING-Ausdruck.

Diese Funktion gibt die Laenge des STRING-Ausdruckes str zurueck, d.h. die Anzahl der Zeichen von str. Der Typ des Ergebnisses ist Integer.

Beispiel:

```
     str := '123456789';
     Length(str) ist gleich 9.
```

9.5.4 POS-Funktion

Syntax: Pos(obj,target);

Die Parameter sind:

obj	STRING-Ausdruck,
target	STRING-Ausdruck.

Der Ergebnistyp ist Integer. Die POS-Funktion durchsucht den String target nach dem String obj. Das Ergebnis ist die Position

in target, an der das erste Zeichen von obj steht. Wenn obj nicht in target gefunden wurde, gibt POS den Wert 0 zurueck.

Beispiele:
```
str := 'ABCDEFG';
Pos('BC',str)  ergibt  2,
Pos('H',str)   ergibt  0.
```

9.6 **STRINGS und CHARACTER**

STRING-Typen und Standard-Skalar-Typen CHAR sind kompatibel. Deshalb kann man ueberall dort, wo STRING-Werte erlaubt sind, CHAR-Werte einsetzen und umgekehrt. Weiterhin kann man STRINGS und CHARACTER in Ausdruecken mischen. Wenn einem CHARACTER ein STRING-Wert zugewiesen wird, muss die Laenge des STRING genau 1 sein, sonst wird ein Laufzeitfehler angezeigt. Man kann die einzelnen Zeichen einer STRING-Variablen durch Indizierung erreichen. Dies geschieht durch Anfuegen eines Indexausdruckes vom INTEGER-Typ, eingeschlossen in eckigen Klammern, an den Bezeichner der STRING-Variablen.

Beispiele:
```
Buffers[5]
Line[Length(Line)-1]
Ord(Line[0])
```

Das erste Zeichen eines STRING (mit Index 0) enhaelt die STRING-Laenge. Dies ist genau der Wert von Ord(Line[0]). Wenn jedoch vom Programmierer der Laengenindikator selbst geaendert wird, muss er auch selbst sichern, dass dieser die maximale Laenge der STRING-Variablen nicht uebersteigt. Wenn die Compiler-Direktive R aktiv ist {$R+}, wird ein Code generiert, der sichert, dass der Wert des STRING-Index-Ausdruckes die maximale Laenge der STRING-Variablen nicht uebersteigt. Es ist jedoch auch moeglich, einen String ueber seine aktuelle dynamische Laenge hinaus zu indizieren. Die dann gelesenen Zeichen haben jedoch rein zufaellige Werte und haben keinerlei Bezug zu den wirklichen Werten der STRING-Variablen.

10. Feldtypen (ARRAY)

Ein Feld (ARRAY) ist ein strukturierter Typ, der aus einer festen Anzahl von Komponenten besteht. Die Komponenten sind alle vom gleichen Typ. Diesen bezeichnet man als Komponententyp oder Basistyp. Jede dieser Komponenten kann man exakt durch Indizierung des Feldes erreichen. Indizes sind INTEGER-Ausdruecke, die in eckigen Klammern geschrieben, an den Bezeichner des Feldes angehaengt werden. Ihr Typ heisst Indextyp.

10.1 **Feld-Definition**

Die Definition eines Feldes besteht aus dem reservierten Wort **array**, dem ein in eckigen Klammern eingeschlossener Indextyp

folgt. Nach diesem steht das reservierte Wort **of** und ein Komponententyp.

Beipiele:

```
     type
          Day      = (Mon,Die,Mit,Don,Fre,Sam,Son);
     Var
          WorkHour : array[1..8]  of Integer;
          Week     : array[1..52] of Day;

     type
          Players  = (Player1,Player2,Player3,Player4);
          Hand     = (One,Two,Pair,TwoPair,Three,Straight,Flush,
                      FullHouse,Four,StraightFlush,RSF);
          LegalBid = 1..200;
          Bid      = array[Players];
     Var
          Player   : array[Players] of Hand;
          Pot      : Bid;
```

Die Zuweisung zu einer Feld-Komponente erfolgt durch Angabe eines in eckigen Klammern eingeschlossenen Indexes hinter dem Variablenbezeichner.

Beispiele:

```
     Player[Player3] := FullHouse;
     Pot[Player3]    := 100;
     Player[Player4] := Flush;
     Pot[Player4]    := 50;
```

Es ist erlaubt, einer Variablen x den Wert einer anderen Variablen Y zuzuweisen, wenn beide den gleichen Typ haben. Analog ist es moeglich ganze Felder zu kopieren:

```
               Index1 := Index2.
```

Die Compiler-Direktive R steuert auch die Erzeugung eines Codes, der zur Laufzeit des Programmes die Index-Ausdruecke prueft, ob sie innerhalb des erlaubten Bereiches liegen. Standard ist dabei {$R-}, d.h. es muss im Programm {$R+} gesetzt werden, wenn die Index-Ausdruecke geprueft werden sollen.

10.2 **Mehrdimensionale Felder**

Der Komponententyp eines Feldes kann ein jeder Datentyp sein, d.h. der Komponententyp kann auch ein Feld sein. Eine solche Konstruktion heisst mehrdimensionales Feld.

Beispiele:

```
     type
          card = (Two,Three,Four,Five,Six,Seven,Eight,Nine,Ten,
                  Knight,Queen,King,Ace);
          Suit = (Hearts,Spade,Clubs,Diamonds);
          AllCards = array[Suit] of array[1..13] of Card;
     var
          Deck : AllCards;
```

Ein mehrdimensionales Feld kann auch durch mehrdimensionale Indices definiert werden:

```
type
     AlCards = array[Sui,1..13] of Card;
```

Aus diesem Grunde kann man auch folgende kuerzere Schreibweise fuer die Auswahl eines Feldes anwenden:

```
Deck[Hearts,10]   equivalent mit    Deck[Hearts][10]
```

Es ist natuerlich auch moeglich mehrdimensionale Felder in Termen vorher definierter Feldtypen zu definieren.

Beispiele:

```
type
     Pupils = string[10];
     Class  = array[1..30]  of Pupils;
     Scool  = array[1..100] of Class;
var
     J,P,Vacant    : Integer;
     ClassA,ClassB : Class;
     NewTownScool  : Scool;
```

Mit diesen Definitionen gelten folgende Zuweisungen:

```
     ClassA              := 'Peter';
     NewTownScool[5][21] := 'Peter Brown';
     NewTownScool[8,J]   := NewTownScool[7,J];
     ClassA[Vacant]      := ClassB[P];
```

10.3 **Zeichenfelder (Caracter Arrays)**

Zeichenfelder sind Felder mit einem Index und Komponenten vom Standard-Skalar-Typ Char. Zeichenfelder koennen auch als String mit fester Laenge aufgefasst werden.
In TURBO-Pascal duerfen Zeichenfelder auch in STRING-Ausdruecken auftreten. In diesen Faellen werden jeweils in einen String von der Laenge des Zeichenfeldes umgewandelt. Damit koennen Zeichenfelder in gleicher Weise wie String miteinander verglichen und manipuliert werden. Auch Stringkonstanten duerfen Zeichenfeldern zugewiesen werden, wenn sie die gleiche Laenge haben. STRING-Variable und -Werte, die aus STRING-Ausdruecken berechnet werden, koennen Zeichenfeldern nicht zugewiesen werden.

10.4 **Standard-Felder**

TURBO-Pascal stellt zwei Standard-Felder vom Typ Byte zur Verfuegung: **MEM** und **PORT**. Mit ihnen erhaelt man den Zugriff zum Speicher und zu den Datenports. Ihre Besprechung erfolgt in Anhang A und B.

11. Datensatztypen

Ein Datensatz ist eine Struktur, die aus einer festen Anzahl von Komponenten, den Datenfeldern, besteht. Die einzelnen Datenfelder koennen von unterschiedlichem Typ sein und jedes Datenfeld hat einen eigenen Datenfeldbezeichner, der dem Zugriff zu diesem Datenfeld dient.

11.1 **Datensatz-Definition**

Die Definition eines Datensatztyps besteht aus dem reservierten Wort **record**, dem eine Datenfeldliste folgt. Diese Liste wird durch das reservierte Wort **end** abgeschlossen. Sie besteht aus Datensatzabschnitten, die voneinander durch Semikolon getrennt sind. Jeder Datensatzabschnitt besteht aus einem oder mehreren Datenfeldbezeichnern, die durch Komma voneinander getrennt sind. Der letzte wird durch einen Doppelpunkt abgeschlossen. Diesem folgt ein Typbezeichner. Jeder Datensatzabschnitt spezifiziert demnach den Typ und die Bezeichner fuer einen oder mehrere Datenfelder.

Beispiele:

```
     type
          Date = record
                    Day   : 1..31;
                    Month : (Jan,Feb,Mrz,Apr,Mai,Jun,Jul,Aug,
                             Sep,Okt,Nov,Dez);
                    Year  : 1900..1999;
                 end;
     var
          Birth   : Date;
          WorkDay : array[1..5] of date;
```

In diesem Beispiel sind Day, Month und Year Datenfeldbezeichner. Ein Datenfeldbezeichner muss nur eindeutig in dem Datensatz selbst sein, indem er definiert wurde. Den Zugriff zu einem Datenfeld erhaelt man durch den Datenfeldbezeichner, dem man, getrennt durch einen Punkt, den Variablenbezeichner fuer diesen Datensatz voranstellt.

Beispiele:

```
     Birth.Month        := Jun;
     Birth.Year         := 1950;
     WorkDay[Current]   := WorkDay[Current-1];
```

Man beachte, dass wie bei den Feltypen auch ganze Datensaetze einander zugewiesen werden koennen, wenn sie vom gleichen Typ sind.
Da der Typ der Datensatzkomponenten nicht eingeschraenkt ist, kann er selbstverstaendlich auch ein Datensatztyp sein. Damit ist es moeglich, einen Datensatz von Datensaetzen zu bilden:

```
     type
          Name = record
                    FamilyName     : string[32];
                    ChristianName  : array[1..3] of string[16];
                 end;
          Rate = record
                    NormalRate,OverTime,NightTime,Weekend:Integer;
                 end;
          Date = record
                    Day   : 1..31;
                    Month :  (Jan,Feb,Mrz,Apr,Mai,Jun,Jul,Aug,
                             Sep,Okt,Nov,Dez);
                    Year  : 1900..1999;
```

```
            end;
       Person = record
                  Id   : Name;
                  Time : Date;
                end;
       Wages  = record
                  Individual : Person;
                  Cost       : Rate;
                end;
  var
       Salary,Fee  : Wages;
```

Mit diesen Definitionen sind folgende Zuweisungen erlaubt:

```
  Salary                          :=  Fee;
  Salary.Cost.OverTime            :=  950;
  Salary.Individual.Time          :=  Fee.Individual.Time;
  Salary.Individual.Id.FamilyName := 'Smith';
```

11.2 **WITH-Anweisung**

Die Verwendung von Datensaetzen in der obigen Weise ergibt meistens ziemlich weitschweifige Anweisungen. Die Schreibweise der Ergibtanweisungen wuerde einfacher sein, wenn die Datensatzfelder einfache Variablen waeren. Genau dies ermoeglicht die WITH-Anweisung: Sie "eroeffnet" einen Datensatz so, dass die Datenfeldbezeichner wie einfache Variablenbezeichner verwendet werden koennen.
Eine WITH-Anweisung besteht aus dem reservierten Wort **with**, dem eine Liste von Datensatzvariablen folgt, die durch Komma voneinander getrennt sind. Anschliessend folgt das reservierte Wort **do** und eine Anweisung.
In einer WITH-Anweisung wird ein Datenfeld nur durch seinen Datenfeldbezeichner gekennzeichnet, d.h. ohne Datensatzvariablenbezeichner.

```
  with Date do
       begin
            Day   := 23;
            Month := Feb;
            Year  := 1982;
       end;
```

Datensaetze koennen in einer WITH-Anweisung geschachtelt werden, d.h. Datensaetze von Datensaetzen koennen wie folgt eroeffnet werden:

```
  with Salary do with Individual do with Id do ...
```

oder kuerzer:

```
  with Salary,Individual,Id do
       begin
            FamilyName    := 'Smith';
            ChristianName := 'Jones';
       end;
```

Die maximale Tiefe dieser Schachtelung der WITH-Folgen, d.h. die maximale Anzahl der Datensaetze, die man in einem Block zur gleichen Zeit eroeffnen kann, haengt von der Implementierung ab. Eine ausfuehrliche Besprechung erfolgt im Anhang A und B.

11.3 **Definition von Datensatz-Varianten.**

Die Syntax eines Datensatztypes erlaubt auch die Definition von Datensatzteil-Varianten, d.h. von alternativen Datensatzstrukturen, die aus Datensatzteilen bestehen, deren Datenfeldanzahl und Datenfeldtypen unterschiedlich sind und deren Struktur jeweils vom Wert eines Kennzeichnungsfeldes abhaengen.
Ein Varianten-Datensatzteil besteht aus einem Kennzeichnungsfeld eines vorher definierten Typs, dem Marken folgen, die den moeglichen Werten des Kennzeichnungsfeldes entsprechen. Jede Marke fuehrt eine Datenfeldliste an, die den Typ der dieser Marke zugeordneten Datensatzteil-Variante definiert.
Als Beispiel gelte obige Typdefinition fuer Name und Date, sowie

```
type      Origin = (Citizen,Alien);
```

Dann ist es mit der folgenden Datensatztyp-Definition moeglich, fuer das Datenfeld CitizenChip zwei verschiedene Strukturen zu definieren, die nur von seinem Wert: Citizen oder Alien abhaengen:

```
type
  Name = record
          FamilyName       : string[32];
          ChristianNames   : array[1..3] of strings[16];
          HourRates        : array[1..4] of Integer;
         end;
  Person  = record
               PersonName       : Name;
               BirthDate        : Date;
               case CitizenShip : Origin of
                    Citizen: (BirthPlace :Name);
                    Alien  : (CountryOfOrigin : Name;
                              DateOfEntry     : Date;
                              PermittedUntil  : Date;
                              PortOfEntry     : Name);
             end;
```

Bei dieser Definition von Datensatzvarianten ist das Kennzeichnungsfeld ein explizites Datenfeld, das man wie jedes andere Datenfeld auswaehlen und mit Werten versehen kann. Es sind deshalb auch die folgenden Anweisungen exakt gueltig:

```
var  Passenger : Person;

Passenger.CitizenShip  := Citizen;
with Passenger,Person,Name do
     if CitizenShip = Alien then writeln(FamilyName);
```

Der feste Teil eines Datensatzes, d.h. der Teil, der gleiche Datenfelder enthaelt, muss stets vor dem variablen Teil liegen. Im obigen Beispiel sind PersonName und BirthDate die festen

Felder. Ein Datensatz darf nur einen variablen Teil haben. Im variablen Teil muessen die Klammern geschrieben werden, auch wenn sie nichts enthalten.
Die Nutzung eines Kennzeichnungsfeldes liegt in der Verantwortung des Programmierers und nicht bei TURBO-Pascal. Aus diesem Grunde kann man sich auf das Feld DateOfEntry im Typ Person auch beziehen, wenn das Kennzeichnungsfeld CitizenChip nicht den Wert Alien hat. Tatsaechlich kann man das Kennzeichnungsfeld ueberall weglassen, indem man den Typbezeichner streicht. Solche Datensatzdefinitionen sind bekannt als freie Vereinigungen (free unions). Im Gegensatz dazu heissen diejenigen mit Kennzeichnungsfeld gekennzeichnete Vereinigungen (discriminated unions). Die Verwendung freier Vereinigungen sind selten und sollten nur von erfahrenen Programmierern praktiziert werden.

12. Mengentypen

Eine Menge ist eine Sammlung verwandter Objekte, die man sich als eine Gesamtheit vorstellen kann. Jedes Objekt einer solchen Menge heisst Mitglied oder Element der Menge. Beispiele fuer Mengen sind:

1) Alle ganzen Zahlen von 0 bis 100.
2) Alle Buchstaben des Alphabets.
3) Alle Konstanten des Alphabets.

Zwei Mengen sind dann und nur dann gleich, wenn ihre Elemente die gleichen sind. Es gibt in dieser Definition keinen Ordnungsbegriff, sodass folgende Mengen gleich sind:

[1,3,5] [5,3,1] [3,5,1].

Wenn alle Elemente einer Menge auch Elemente einer anderen Menge sind, sagt man, dass diese Menge in der anderen Menge enthalten ist. Sie wird auch als Teilmenge der anderen Menge bezeichnet. Im obigen Beispiel ist 3) in 2) enthalten.
Es gibt drei Operationen, die den Zahlenoperationen entlehnt sind und auf Mengen angewandt werden koennen: die Vereinigung (+), der Durchschnitt (*) und das relative Komplement (-):

Die Vereinigung (oder Summe) zweier Mengen A und B, geschrieben A+B, ist die Menge aller Elemente, die entweder in A oder in B enthalten sind. Die Vereinigung von [1,3,5,7] und [2,3,4] ist [1,2,3,4,5,7].

Der Durchschnitt (oder Produkt) zweier Mengen A und B, geschrieben A*B, ist die Menge aller Elemente, die sowohl in A als auch in B enthalten sind. Der Druchschnitt von [1,3,4,5,7] und [2,3,4] ist [3,4].

Das relative Komplement (oder die Differenz) von B bezueglich A, geschrieben A-B, ist die Menge aller Elemente, die in A aber nicht in B enthalten sind. Die Differenz von [1,3,4,5,7] und [2,3,4] ist [1,5,7].

12.1 **Mengentyp-Definition**

Obgleich es in der Mathematik keine Einschraenkungen fuer die Elemente einer Menge gibt, gestattet Pascal nur eine eingeschraenkte Form der Definition von Mengen:
Die Elemente einer Menge muessen alle vom gleichen Typ sein, dem

Basistyp. Der Basistyp muss ein einfacher Typ sein, d.h. ein beliebiger Skalartyp ausser REAL. Die Definition eines Mengentyp besteht aus einem Mengentypbezeichner, dem ein Gleichheitszeichen und die reservierten Worte **set of** und ein einfacher Typ folgen.

Beispiele:

```
DaysOfMonth     =   set of 0..31;
WorkWeek        =   set of Mon..Fre;
Letter          =   set of 'A'..'Z';
Additiv Colors  =   set of [Red,Green,Blue];
Characters      =   set of Char;
```

In TURBO-Pascal betraegt die maximale Anzahl von Elementen einer Menge 256 und die Ordnungswerte des Basistyp muessen im Bereich 0..255 liegen.

12.2 **Mengenausdruecke**

Mengenwerte koennen aus anderen Mengenwerten durch Mengenausdruecke berechnet werden. Mengenausdruecke bestehen aus:

- Mengenkonstruktionen,
- Mengenoperatoren,
- Mengenkonstanten und
- Mengenvariablen.

12.2.1 Mengenkonstruktionen

Eine Mengenkonstruktion besteht aus einer oder mehreren Elementenspezifikationen, die durch Komma voneinander getrennt und in eckige Klammern eingeschlossen sind. Eine Elementenspezifikation ist ein Ausdruck vom gleichen Typ wie der Basistyp der Menge. Sie kann auch ein Bereich sein, der durch zwei solcher Ausdruecke dargestellt wird, getrennt durch zwei aufeinanderfolgende Punkte.

Beispiele:

```
['T','U','R','B','O']
['X','Y']
[X..Y]
[1..5]
['A'..'Z','a'..'z','0'..'9']
[1,3..10,12]
[]
```

Das letzte Beispiel stellt die leere Menge dar. Da sie keinen Ausdruck enthaelt, der ihren Basistyp festlegt, ist sie mit allen Mengentypen kompatibel. Die Menge [1..5] ist der Menge [1,2,3,4,5] equivalent. Wenn X>Y, dann bezeichnet [X..Y] die leere Menge.

12.2.2 Mengenoperationen

Die Mengenoperationen werden entsprechend ihrer Rangfolge in folgende drei Klassen eingeteilt:

1) * Mengendurchschnitt.

2) + Mengenvereinigung,
   - Mengendifferenz.

3) = Test auf Gleichheit,
   <> Test auf Ungleichheit,
   >= Wahr, wenn der zweite Operand im ersten enthalten ist,
   <= Wahr, wenn der erste Operand im zweiten enthalten ist,
   in Test auf Mitgliedschaft in einer Menge. Der zweite Operand ist ein Mengentyp und der erste ein Mengenausdruck vom gleichen Typ wie der Basistyp der Menge. Das Ergebnis ist wahr, wenn der erste Operand ein Element des zweiten Operanden ist, andernfalls ist es falsch.

Es gibt keinen Operator fuer ein exaktes Nichtenthaltensein. Aber man kann dies in der Form A*B = [] programmieren.

Mengenausdruecke sind oft brauchbar, um komplizierte Tests einfacher zu programmieren. Dafuer einige Beispiele:

```
fuer:     (CH='T') or (CH='U') or (CH='R') or (CH='B') or (CH='O')

besser:   CH in ['T','U','R','B','O']


fuer:     (CH>'0') and (CH<'9')

besser:   CH in ['0'..'9']
```

12.2.3 **Mengenzuweisungen**

Mengenvariablen wird das Ergebnis von Mengenausdruecken durch das Ergibtzeichen ':=' zugewiesen.

Beispiele:

```
type
     ASCII = set of 0..127;
var
     NoPrint,Print,AllChars : ASCII;
begin
     AllChars := [0..127];
     NoPrint  := [0..31,127];
     Print    := AllChars - NoPrint;
end;
```

13. Typisierte Konstante

Typisierte Konstante sind eine TURBO Spezialitaet. Eine typisier-

te Konstante kann man exakt wie eine Variable vom gleichen Typ verwenden. Sie koennen deshalb als initialisierte Variable eingesetzt werden. Denn ihr Wert ist von Anfang an definiert, waehrend eine Variable solange undefiniert ist, bis ihr ein Wert zugewiesen wurde.
Man sollte natuelich darauf achten, das einer typisierten Konstante keine Werte zugewiesen werden, denn ihr Wert sollte eben wirklich konstant sein.
Die Verwendung typisierter Konstanten hilft Speicherplatz sparen, wenn sie haeufig im Programm verwendet werden. Denn sie werden im Programmcode nur einmal gespeichert, im Gegensatz zu den untypisierten Konstanten, die im Code jedesmal gespeichert werden, wenn sie im Text vorkommen.
Typisierte Konstanten werden wie untypisierte Konstanten definiert (siehe 5.2.2), sie enthalten nur zusaetzlich auch ihren Typ.
Die Definition einer typisierten Konstanten besteht aus dem Konstantenbezeichner, einem Doppelpunkt, einem Typbezeichner, einem Gleichheitszeichen und dem wirklichen Konstantenwert.

13.1 **Unstrukturierte typisierte Konstante**

Eine unstrukturierte typisierte Konstante ist eine Konstante, die durch einen Skalartyp definiert wird.

Beispiele:

```
NumbersOfCars : Integer = 1267;
Interest : Real = 12.67;
Heading : string[7] = 'Section';
Xon : Char = ^Q;
```

Im Gegensatz zu den untypisierten Konstanten kann eine typisierte Konstante anstelle einer Variablen als Parameter in einer Prozedur oder Funktion verwendet werden. Eine typisierte Konstante ist tatsaechlich eine Variable mit einem konstanten Wert. Sie kann deshalb nicht in der Definition anderer Konstanten oder Typen verwendet werden. Deshalb ist im folgenden Beispiel die Verwendung der typisierten Konstanten Min und Max nicht erlaubt:

```
const
     Min : integer = 0;
     Max : integer = 50;
type
     Range : array[Min..Max] of integer;   {Falsch !}
```

13.2 **Strukturierte typisierte Konstante**

Strukturierte Konstante sind
- Feldkonstante,
- Datensatzkonstante und
- Mengenkonstante.

Sie werden häufig verwendet, um initialisierte Tabellen und Mengen fuer Tests, Konvertierungen, Abbildungsfunktionen usw. bereitzustellen. Die folgenden Abschnitte definieren jeden Typ im Einzelnen.

13.2.1 Feldkonstante

Die Definition einer typisierten Feldkonstanten besteht aus dem Konstantenbezeichner, einem Doppelpunkt, dem Typbezeichner eines vorher definierten Feldtyps, dem Gleichheitszeichen und dem konstanten Wert. Letzterer besteht aus einer Liste von Konstanten, die durch Komma getrennt und in Klammern eingeschlossen sind.

Beispiele:

```
     type
          Status         =    (Activ,Passiv,Wartend);
          StringRep      =    array[Status] of string[7];
     const
          Stat:StringRep =    ('aktiv','passiv','wartend');
```

Das Beispiel definiert die Feldkonstante Stat, die verwendet werden kann, um Werte vom Skalartyp Status in ihre entsprechende Stringdarstellung zu konvertieren:

```
     Stat[Aktiv]    = 'aktiv'
     Stat[Passiv]   = 'passiv'
     Stat[Wartend]  = 'wartend'
```

Der Komponententyp einer Feldkonstanten kann jeder Typ sein ausser einem Feld- oder Pointertyp. Charakterfeldtypen koennen sowohl als einzelne Char als auch als Strings definiert werden. Aus diesem Grunde ist es guenstiger statt:

```
     const
          Digits : array[0..9] of
               Char=('0','1','2','3','4','5','6','7','8','9');
```

besser:

```
     const
          Digits : array[0..9] of Char='0123456789';
```

zu schreiben.

13.2.2 Mehrdimensionale Feldkonstante

Eine typisierte mehrdimensionale Feldkonstante wird analog definiert, indem man jede Dimension in separate Klammernpaare einschliesst, die durch Komma voneinander getrennt sind. Die innerste Konstant entspricht der am weitesten rechts stehenden Dimension:

Beipiele:

```
     type
          Cube = array[0..1,0..1,0..1] od integer;
     const
          Maze : Cube = (((0,1),(2,3)),((4,5),(6,7)));
     begin
          writeln(Maze[0,0,0],' =0');
          writeln(Maze[0,0,1],' =1');
          writeln(Maze[0,1,0],' =2');
          writeln(Maze[0,1,1],' =3');
          writeln(Maze[1,0,0],' =4');
          writeln(Maze[1,0,1],' =5');
```

```
          writeln(Maze[1,1,0],' =6');
          writeln(Maze[1,1,1],' =7');
     end;
```

13.2.3 Datensatzkonstante

Die Definition einer typisierten Datensatzkonstanten besteht aus dem Konstantenbezeichner, einem Doppelpunkt, dem Typbezeichner eines vorherdefinierten Datensatztyps, einem Gleichheitszeichen und dem Konstantenwert. Letzterer ist eine Liste, die aus den Datenfeldkonstanten, getrennt durch Komma und eingeschlossen in runde Klammern, besteht.

Beipiele:

```
     type
          Point     = record
                         X,Y,Z : integer;
                      end;
          OS        = (CPM80,CPM86,MSDOS,UNIX);
          UI        = (CCP,SomethingElse,MenuMaster);
          Computer  = record
                         OperatingSystems : array[1..4] of OS;
                         UserInterfacs : UI;
                      end;
     const
          Origo     : Point   = (X:0; Y:0; Z:0);
          SuperComp : Computer=
                         (OperatingSystems:(CPM80,CPM86,MSDOS,UNIX);
                          UserInterface:MenuMaster);
          Planel : array[1..3] of Point =
                         ((X:1;  Y:4;  Z:5),
                          (X:10, Y:-78,Z:45),
                          (X:100,Y:10, Z:-75));
```

Die Feldkonstanten muessen in der gleichen Reihenfolge definiert werden, wie sie in der Datensatzdefinition auftreten. Wenn ein Datensatz Felder vom File- oder Pointertyp enthaelt, koennen typisierte Konstanten fuer diesen Datensatztyp nicht definiert werden. Wenn eine Datensatzkonstante eine Variante enthaelt, dann ist der Programmierer selbst dafuer verantwortlich, dass nur die Datenfelder der gueltigen Variante spezifiziert werden. Wenn die Variante ein Kennzeichnungsfeld enthaelt, dann muss auch ihr Wert spezifiziert werden.

13.2.4 Mengenkonstanten

Eine typisierte Mengenkonstante besteht aus einer oder mehreren Elementenspezifikationen, die durch Komma getrennt und in eckigen Klammern eingeschlossen sind. Eine Elementenspezifikation darf eine Konstante oder ein Bereichsausdruck sein, der aus zwei Konstanten, getrennt durch zwei Punkte, besteht.

Beipiele:

```
     type
          Up  = set of 'A'..'Z';
```

```
          Low = set of 'a'..'z';
     const
          UpperCase : Up  = ['A'..'Z'];
          Vocals    : Low = ['a','e','i','o','u'];
          Delimiter : set of Char =
                        [' '..'/',':'..'?','['..'`','{'..'~'];
```

14. Filetypen

Computerprogramme produzieren häufig so grosse Datenmengen, dass man sie nicht bis zu einer spaeteren Verwendung im gleichen oder anderen Programmen im Speicher belassen kann. Aus diesem Grunde speichert man solche Datenmengen auf externen Datentraegern, wie Magnetbandkassetten oder Disketten. Die Einheit einer solchen Datenmenge heisst File oder Datei.
Ein File besteht aus einer Folge von Komponenten gleichen Typs. Die Anzahl der Komponenten im File (die Filegroesse) wird nicht in der Filedefinition festgelegt. Stattdessen wird in Pascal der Zugriff zu den einzelnen Komponenten ueber einen Filepointer organisiert. Jedesmal, wenn eine Komponente des Files gelesen oder geschrieben wird, rueckt der Filepointer eine Komponente weiter, d.h. er weist auf die naechste Komponente. Da alle Komponenten eines File die gleiche Laenge haben, weil sie vom gleichen Typ sind, kann die Position einer bestimmten Komponente berechnet werden d.h. man kann ueber den Filepointer zu jeder Komponente des File zugreifen. Damit sind die Voraussetzungen gegeben, einen wahlfreien Zugriff zu den Komponenten des File aufzubauen.

14.1 **Filetyp-Definition**

Ein Filetyp wird durch die reservierten Worte **file of**, denen der Typbezeichner der Komponenten folgt, definiert. Eine Filevariable wird definiert durch einen Filevariablenbezeichner, einem Doppelpunkt und einen Filetyp. Dabei gibt es zwei Moeglichkeiten, jenachdem, ob der Filetyp mit einem Filetypbezeichner explizit im Typdefinitionsteil definiert wurde oder nicht:

```
     1) type   Komponente =  record
                              x,y,z : integer;
                             end;
               Filetyp    =  file of Komponente;
          var  FileVar    :  Filetyp;
oder
     2) type   Komponente =  record
                              x,y,z : integer;
                             end;
          var  FileVar    : file of Komponente
```

Beispiele:

```
     type
       ProductName = string[80];
       Product     = record
                       Name      : ProductName;
                       ItmNumber : Real;
                       InStock   : Real;
```

```
                    MinStock  : Real;
                    Supplier  : Integer;
                   end;
     FProduct      = file of Product;

     var
       ProductFiles   :   FProduct;
       ProductNames   :   file of ProductName;
```

Der Komponententyp eines Files kann irgend ein Typ sein, jedoch kein Filetyp!, d.h. in obigem Beispiel ist
`var ProdFile : file of FProduct`
nicht erlaubt.

Man beachte auch, dass zwar
```
var P1File : FProduct;
    P2File : FProduct;
```
und
den gleichen Filetyp haben, aber nicht:
```
var PM1File : file of Product;
    PM2File : file of Product;
```
und
Weiterhin merke man sich: Filevariable duerfen weder in Ergibtanweisungen noch in Ausdruecken auftreten.

14.2 **Fileoperationen**

In den folgenden Abschnitten werden die in TURBO-Pascal vorhandenen Fileprozeduren beschrieben. Dabei werden folgende Kurzzeichen fuer die Parameter verwendet:

str	Stringausdruck, der einen gueltigen Filenamen in der bekannten Form '[A:]Dateinam.Typ' darstellen muss.
filvar	Filevariable.
var	Eine Variable oder mehrere Variable, die dann durch Komma getrennt sein muessen, und die alle den gleichen Komponententyp wie das File haben muessen.
num	Integerkonstante.

14.2.1 Assign

Syntax: `Assign(filvar,str)`

Durch diese Prozedur wird der in str enthaltene physische Filenamen der Filevariablen filvar zugewiesen. Danach beziehen sich alle auf filvar ausgeuebte Operationen auf das genannte physische Diskettenfile. Wurde der Filevariablen bereits ein physischer Filenamen zugewiesen, und mit der Filevariablen gearbeitet, darf ein erneutes ASSIGN auf sie nicht angewendet werden.

14.2.2 Rewrite

Syntax: `Rewrite(filvar)`

Mit dieser Prozedur wird auf der Diskette ein neues File mit dem filvar zugewiesenen physischen Filenamen aufgebaut. Der Filepointer wird dabei auf den Anfang des File, d.h. auf die Komponente mit der Nummer 0, gesetzt. Existiert auf der Diskette bereits der gleiche physische Filenamen, dann wird das zugehoerige File geloescht! Ein mit REWRITE erzeugtes File ist anfangs immer leer und enthaelt kein Element.

14.2.3 Reset

Syntax: Reset(filvar)

Das filvar zugewiesene File wird fuer die Verarbeitung vorbereitet und der Filepointer auf den Anfang des File, d.h. die Komponente mit der Nummer 0, gesetzt. Das zugewiesene File muss bereits existieren, sonst entsteht in I/O-Fehler.

14.2.4 Read

Syntax: Read(filvar,var)

Durch diese Prozedur werden die Variablen var nacheinander mit dem Inhalt der Komponenten des filvar zugeordneten Files gefuellt. Begonnen wird mit der Komponente, auf die der Filepointer zeigt. Nach jeder Zuweisung wird der Pointer auf die naechste Komponente eingestellt.

14.2.5 Write

Syntax: Write(filvar,var)

Durch diese Prozedur werden nacheinander der Inhalt der Variablen var in die Komponenten des filvar zugewiesenen Files geschrieben. Begonnen wird mit der Komponente, auf die der Filepointer zeigt. Nach jedem Schreibvorgang wird der Pointer auf die naechste Komponente eingestellt.

14.2.6 Seek

Syntax: Seek(filvar,num)

Der Filepointer des filvar zugeordneten Files wird durch diese Prozedur auf die Komponente mit der Nummer num-1 eingestellt (die 1.Komponente hat die Nummer 0!). Um ein File zu erweitern, braucht man nur den Filepointer auf die Komponente hinter der letzten Komponente des File einzustellen.

14.2.7 Flush

Syntax: Flush(filvar)

Diese Prozedur wird eigentlich nur in Multi-User-Systemen benoe-

tigt, in denen mehrere Nutzer zum gleichen Diskettenfile zugreifen koennen. Flush schreibt sofort den Update-Puffer auf die Diskette zurueck und sichert damit nach Updatefunktionen, dass die naechste Leseoperation wirklich als ein physisches Lesen ausgefuehrt wird. Flush darf niemals auf ein geschlossenes (CLOSE) File angewendet werden.

14.2.8 Close

Syntax: Close(filvar)

Diese Prozedur schliesst das filvar zugeordnete physische File und schreibt den aktuellen Filestatus in das Diskettenvezeichnis. In Multi-User-Systemen sollte man oefter diese Prozedur anwenden, auch wenn nur gelesen wurde.

14.2.9 Erase

Syntax: Erase(filvar)

Diese Prozedur loescht das filvar zugeordnete File im Diskettenverzeichnis. Wenn das File bereits eroeffnet wurde, d.h. RESET oder REWRITE ausgefuehrt wurde, sollte man stets CLOSE vor ERASE aufrufen.

14.2.10 Rename

Syntax: Rename(filvar,st)

Das filvar zugewiesene File erhaelt den in str enthaltenen neuen Filenamen. Der neue Name wird in das Diskettenverzeichnis eingetragen und die weiteren Operationen von filvar werden dann mit diesem File unter dem neuen Namen ausgefuehrt.
Man sollte RENAME niemals auf ein bereits eroeffnetes File anwenden. Ebenso sollte der Programmierer sichern, dass das mit str benannte File nicht bereits auf der Diskette existiert. Sonst entstehen doppelte Namen in der Direktory.

Die folgende Funktion gibt den Wert True zurueck, falls der im Parameter spezifizierte Filenamen bereits in der Direktory existiert, andernfalls ist der Wert False:

```
function Exist(Filename:Boolean):Boolean;
var
     File:file;
begin
     Assign(File,FileName);
     {$I-};
     Reset(File);
     {$I+};
     if Ioresult <> 0 then Exist := False;
                      else Exist := True;
end;
```

14.3 **Standardfilefunktionen**

TURBO-Pascal enthaelt die folgenden Standardfilefunktionen:

14.3.1 EOF

Syntax: Eof(filvar)

Diese Boolesche-Funktion gibt den Wert True zurueck, wenn der Filepointer das Fileende erreicht hat, d.h. hinter die letzte Komponente des filvar zugewiesenen Files weist. Andernfalls ist der zurueckgegebene Wert False.

14.3.2 FilePos

Syntax: FilePos(filvar)

Diese Integer-Funktion gibt den Wert der aktuellen Position des Filepointer zurueck. Die erste Komponente hat den Wert 0.

14.3.3 FileSize

Syntax: FileSize(filvar)

Diese Integer-Funktion gibt den Wert der Groesse des filvar zugeordneten Files zurueck, d.h. die Anzahl der Komponenten des Files. Wenn FileSize(filvar) gleich Null ist, dann ist das File leer.

14.4 **Filenutzung**

Bevor ein File benutzt werden kann, muss ASSIGN aufgerufen werden, um der Filevariablen ein physisches File zuzuordnen. Vor einer I/O-Operation sollte das File durch Aufruf von REWRITE oder RESET eroeffnet werden. Damit zeigt der Filepointer auf die erste Komponente des File, d.h. es ist FilePos(filvar)=0. Nach REWRITE ist stets FileSize(filvar)=0. Ein Diskettenfile kann nur durch Anfuegen von weiteren Komponenten hinter die letzte existierend Komponente des File erweitert werden. Den Filepointer kann man dafuer an das Fileende positionieren durch:

```
Seek(filvar,FileSize(filvar));
```

Nach Beendigung aller Input/Output-Operationen sollte man in einem Programm stets die eroeffneten Files durch CLOSE schliessen. Vergisst man dies, kann das zu Datenverlust fuehren, da das Diskettenverzeichnis dann eventuell nicht dem letzten Stand des Filestatus entspricht.

Das folgende Programm baut ein File mit dem Namen PRODUCTS.DTA auf und schreibt 100 Saetze vom Typ Product auf das File. Das

File ist fuer einen wahlfreien Zugriff vorbereitet (d.h. die Saetze koennen von jeder Stelle des File gelesen oder geschrieben werden).

```
program InitProductFile;
const
  MaxNumberOfProducts = 100;
type
  ProductName         = string[20];
  Product             = record
                          Name         : ProductName;
                          ItemNumber   : Integer;
                          InStock      : Real;
                          Supplier     : Integer;
                        end;
var
  ProductFile         : file of Product;
  ProductRec          : Product;
  I                   : Integer;
begin
  Assign(ProductFile,'F:PRODUCT.DTA');
  Rewrite(ProductFile);   {eroeffnet File und loescht alle Daten}
  with ProductRec do
    begin
      Name := ' '; InStock := 0; Supplier := 0;
      for I := 1 to MaxNumberOfProducts do
      begin
        ItemNumber := I;
        Write(ProductFile,ProductRec);
      end;
    end;
    Close(ProductFile);
  end.
```

Das folgende Programm demonstriert die Verwendung von SEEK fuer den wahlfreien Zugriff. Mit diesem Programm kann man in dem soeben aufgebauten File PRODUCT.DTA den Inhalt in die bereits existierenden Datensaetze (Komponenten) bringen oder den Inhalt spaeter aendern.

```
program UpdateProductFile;
const
  MaxNumberOfProducts = 100;
type
  ProductName         = string[20];
  Product             = record
                          Name         : ProductName;
                          ItemNumber   : Integer;
                          InStock      : Real;
                          Supplier     : Integer;
                        end;
var
```

```
  ProductFile         : file of Product;
  ProductRec          : Product;
  I,Pnr               : Integer;
begin
  Assign(ProductFile,'F:PRODUCT.DTA');
  Reset(ProductFile);
  ClrScr;
  Write('Enter product number (0 = stop):'); Readln(Pnr);
  while Pnr in [1..MaxNumberOfProducts] do
  begin
    Seek(ProductFile,Pnr-1); Read(ProductFile,ProductRec);
    with ProductRec do
    begin
      Write('Enter name of product(',name:20,') ');
      Readln(Name);
      Write('Enter number in stock (',InStock:20:0,') ');
      Readln(InStock);
      Write('Enter supplier number (',Supplier:20,') ');
      Readln(Supplier);
      ItemNumber := Pnr;
    end;
    Seek(ProductFile,Pnr-1);
    Write(ProductFile,ProductRec);
    ClrScr; Writeln;
    Write('Enter product number (0 = stop):'); Readln(Pnr);
  end;
  Close(ProductFile);
end.
```

14.5 **Textfiles**

Textfiles bestehen im Prinzip nicht wie alle anderen Files aus Folgen von Komponenten gleichen Typs. Obwohl die Basiskomponenten eines Textfiles alle vom Typ Char sind, besteht ihre Struktur doch aus Zeilen. Jede Zeile endet mit einer Zeilenendemarke: einer CR/LF-Folge. Das File selbst endet mit einer Fileendemarke: CTRL-Z. Da die Zeilenlaenge beliebig variieren kann, laesst sich die Position einer beliebigen Zeile in einem Textfile nicht berechnen. Aus diesem Grunde kann man Textfiles nur sequentiell verarbeiten und Ein- und Ausgaben duerfen nicht gleichzeitig auf ein Textfile erfolgen.

14.5.1 Textfileoperationen

Eine Textfilevariable wird erklaert, indem man sie dem Standard-typbezeichner Text zuweist:

```
var       Filvar : Text;
```

Als erste muss vor allen anderen Fileoperationen fuer ein Text-file zum Zuweisen des physischen Filenamens die Prozedure ASSIGN aufgerufen werden. Danach ist zur Eroeffnung des Textfile vor den I/O-Operationen entweder RESET oder REWRITE auszufuehren. REWRITE wird fuer den Aufbau eines neuen Textfiles verwendet. Danach

koennen nur Schreiboperationen folgen, da das File noch leer ist. RESET verwendet man, um ein bereits existierendes Textfile zu eroeffnen. Danach ist nur sequentielles Lesen erlaubt. Wenn ein neues File mit CLOSE geschlossen wird, wird automatisch eine Fileendemarke(EOF): CTRL-Z an das File angehaengt.

Zeichenweise I/O-Operationen werden fuer Textfiles mit den Standardprozeduren READ und WRITE ausgefuehrt. Zeilen werden mit den speziellen Textfileoperationen READLN, WRITELN und EOLN behandelt, dabei ist F eine beliebige Filevariable:

Readln(F)	Springt zum Beginn der naechsten Textzeile, d.h. ueberspringt alle Zeichen bis und einschliesslich der naechsten CR/LF-Folge.
Writeln(F)	Schreibt die Zeilenendemarke, d.h. die CR/LF-Folge auf das Textfile.
Eoln(F)	Ist eine Boolesche-Funktion, die den Wert True zurueckgibt, wenn das Ende der aktuellen Zeile erreicht ist, d.h. der Filepointer auf das CR-Zeichen der CR/LF-Folge zeigt. Wenn Eof(F) True ist, dann ist auch Eoln(F) True.

Wird die Eof-Funktion auf ein Textfile angewendet, dann gibt diese Funktion den Wert True zurueck, wenn der Filepointer die Fileendemarke CTRL-Z erreicht hat.
Auf Textfiles sind folgende Prozeduren bzw. Funktionen nicht anwendbar:

SEEK, FLUSH, FilePos und FileSize.

Das folgende Beispielprogramm liest ein Textfile von einer Diskette und druckt es auf einen vorher definierten Drucker LST aus. Die Worte, die in ^S eingeschlossen sind, werden unterstrichen gedruckt.

```
program TextFileDemo;
var
  FileVar    : Text;
  Line,
  ExtraLine  : string[255];
  I          : integer;
  UnderLine  : boolean;
  FileName   : string[14];
begin
  UnderLine := False;
  Write('Enter name of file to list: ');
  Readln(FileName);
  Assign(FileVar,FileName);
  Reset(FileVar);
  while not Eof(FileVar) do
  begin
    Readln(FileVar,Line);
    I := 1; ExtraLine := ' ';
    for I := 1 to Length(Line) do
    begin
```

```
          if Line[I] <> ^S then
          begin
            Write(Lst,Line[I]);
            if UnderLine then ExtraLine := ExtraLine + '_'
                         else ExtraLine := ExtraLine + ' ';
          end
          else UnderLine := not UnderLine;
        end;
        Write(Lst,^M); Writeln(Lst,ExtraLine);
      end;             {while not Eof}
    end.
```

Erweiterungen der Prozeduren READ und WRITE bezueglich formatierter Ein- und Ausgaben werden in Abschnitt 14.6 beschrieben.

14.5.2 Logische Geraete

In TURBO-Pascal werden externe Geraete, wie Terminals, Drucker und Modems als logische Geraete bezeichnet. Sie werden genauso, wie Textfiles behandelt:

CON: Console. Ausgaben werden an das Consolausgabegeraet, normalerweise ein Bildschirmgeraet, gesendet. Eingaben werden vom Consoleingabegeraet, normalerweise eine Tastatur, gelesen. Im Gegensatz zum TRM-Geraet puffert das CON-Geraet die Eingaben. Dies bedeutet, dass READ oder READLN ueber CON eine ganze Zeile aus dem Zeilenpuffer einliest, und der Operator, bis zur Eingabe von CR ueber die ENTER-Taste, die Editiermoeglichkeiten des Systems fuer Eingaben nutzen kann. Genauere Eingaben ueber die Conoleingaben in Abschnitten 14.5.3 und 14.6.1.

TRM: Terminal. Ausgaben erfolgen an das Consolausgabegeraet, normalerweise ein Bildschirmgeraet. Eingaben werden vom Consoleingabegeraet, normalerweise der Tastatur, gelesen. Eingegebene Zeichen, ausser Controlzeichen, werden als Echo an das Consolausgabegeraet gesendet. Das einzige Controlzeichen, das als Echo gesendet wird, ist das Zeichen CR und zwar in Form der Folge CR/LF.

KBD: Keyboard. Eingaben werden von der Consoleingabe,normalerweise der Tastatur, gelesen. Ein Echo wird nicht gesendet.

LST: Listing. Die Ausgaben werden an des Systemlistgeraet, normalerweise ein Zeilendrucker, gesendet.

AUX: Auxiliary. Ausgaben werden an den Systemstanzer gesendet und Eingaben werden vom Systemleser gelesen. Normalerweise sind beide Lochbandgeraete.

USR: Usergeraet. Ausgaben gehen an das Nutzerausgabegeraet und Eingaben werden ueber die Nutzereingaberoutine gelesen. Weitere Einzelheiten siehe A.12.

Diese logischen Geraete koennen durch vorher definierte Files (siehe 14.5.3) oder wie ein Diskettenfile einer Filevariablen zugewiesen werden. Wurde ein File einem logischen Geraet zugewiesen, gibt es zwischen REWRITE und RESET keinen Unterschied. Die Prozedur CLOSE fuehrt dann keine Funktion aus und ERASE bringt einen I/O-Fehler.

Die Standardfunktionen Eof und Eoln arbeiten bei logischen Geraeten anders als bei Diskettenfiles. Bei einem Diskettenfile gibt Eof den Wert True zurueck, wenn das naechste Zeichen im File das Zeichen CTRL-Z ist, und Eoln gibt den Wert True zurueck, wenn das naechste Zeichen das Zeichen CR oder CTRL-Z ist, d.h. diese beiden Prozeduren sind dann praktisch vorausschauende Routinen.
Bei logischen Geraeten gibt es jedoch keine Moeglichkeit vorauszuschauen, welche Zeichen als naechste kommen werden. Aus diesem Grunde liefern Eof und Eoln bei logischen Geraeten das Ergebnis immer vom letzten behandelten Zeichen und nicht vom naechsten: Eof gibt True zurueck, wenn das letzte Zeichen CTRL-Z war und Eoln gibt True zurueck, wenn das letzte Zeichen CR oder CTRL-Z war.
Folgende Tabelle soll einen Ueberblick geben:

	Diskettenfiles	Logische Geraete
Eoln ist True	wenn aktuelles Zeichen CR und naechstes LF ist oder wenn naechstes Zeichen CTRL-Z ist.	wenn aktuelles Zeichen CR oder CTRL-Z ist.
Eof ist True	wenn naechstes Zeichen CTRL-Z ist.	wenn aktuelles Zeichen CTRL-Z ist.

Auch die Prozedur READLN arbeitet in beiden Faellen anders. Bei einem Diskettenfile liest READLN alle Zeichen bis einschliesslich der CR/LF-Folge, waehrend ein logisches Geraet nur bis einschliesslich dem ersten CR liest. Der Grund ist der gleiche wie oben, ein logisches Geraet kann erst CR erkennen, wenn es CR gelesen hat.

14.5.3 Standardfiles

Alternativ zur oben beschriebenen Zuweisung von Textfiles zu logischen Geraeten stellt TURBO-Pascal einige Standardtextfiles zur Verfuegung, die bereits logischen Geraeten zugewiesen sind und die so vorbereitet sind, dass sie unmittelbar benutzt werden koennen. Der Programmierer spart damit einerseits Speicherplatz und andererseits den Aufruf von RESET, REWRITE und Close.

Folgende Standardtextfiles sind implementiert:

Input	Primaeres Inputfile. Dieses File ist entweder dem CON- oder TRM-Geraet zugewiesen (s.u.).
Output	Primaeres Ausgabefile. Dieses File ist entweder dem CON- oder TRM-Geraet zugewiesen (s.u.).
Con	Zugewiesen dem Consolgeraet CON:.
Trm	Zugewiesen dem Terminalgeraet TRM:.

```
Kbd       Zugewiesen dem Keyboard KBD:.
Lst       Zugewiesen dem Listgeraet LST:.
Aux       Zugewiesen dem Auxiliarygeraet AUX:.
Usc       Zugewiesen dem Usergeraet USR:.
```

Bei der Verwendung dieser Standardtextfiles ist zu beachten, dass die Verwendung von RESET, REWRITE und CLOSE nicht nur nicht notwendig, sondern sogar verboten ist. Die Zuweisung des logischen Geraetes zu den Standardtextfiles Input und Output wird durch die Compilerdirektive $B bestimmt:

```
{$B+}  weist  CON: zu,
{$B-}  weist  TRM: zu.
```

Bei Zuweisung von CON: werden die Eingaben gepuffert und koennen in diesem Puffer bei der Eingabe editiert werden , aber fuer das Einlesen der Variablen gelten spezielle Gesetze (s.14.6.1). Bei Zuweisung von TRM: ist ein editieren der Eingaben nicht moeglich, das Einlesen der Variablen erfolgt aber nach den bekannten Regeln. Bei den Ausgabeoperationen gibt es fuer CON: und TRM: keinen Unterschied.
Die Compilerdirektive $B muss vor dem Programmblock stehen und darf als globale Direktive im Programmblock nicht geaendert werden. Wenn in einem Programm sowohl CON- als auch TRM-Geraete verwendet werden, ist die Direktive $B entsprechend dem am häufigsten verwendeten Geraet zu setzen und in den anderen I/O-Operationen ist das andere Geraet explizit anzugeben.

Beispiel:

```
{$B-}
program ReadAndWrite(input,output);
...
...
...
Readln(input,Var1);          Read von TRM:
Readln(Con,Var2);            Read von CON:
```

An den Stellen, wo auf dem Bildschirm kein automatisches Echo erscheinen soll, muss man das Standardtextfile Kbd zuweisen:

```
Read(Kbd,Var);
```

Da die Standardtextfiles Input und Output sehr häufig verwendet werden, wurde implementiert, dass sie automatisch zugewiesen werden, wenn kein Filebezeichner explizit angegeben wurde. Damit sind die folgenden Textfileoperationen equivalent:

```
Write(Ch)      Write(Output,Ch)
Read(Ch)       Read(Input,Ch)
Writeln        Writeln(Output)
Readln         Readeln(Output)
Eof            Eof(Input)
Eoln           Eoln
```

Das folgende Programm zeigt die Verwendung des Standardfiles Lst zur Ausgabe des ProductFile von 14.4 ueber den Drucker:

```
program ListProductFile;
```

```
const
  MaxNumberOfProducts = 100;
type
  ProductName         = string[20];
  Product             = record
                          Name         : ProductName;
                          ItemNumber   : Integer;
                          InStock      : Real;
                          Supplier     : Integer;
                        end;
var
  ProductFile         : file of Product;
  ProductRec          : Product;
  I                   : Integer;
begin
Assign(ProductFile,'PRODUCT.DTA');
Reset(ProductFile);
for I := 1 to MaxNumberOfProducts do
     begin
          Read(ProductFile,ProductRec);
          with ProductRec do
          begin
               if Name <> ' ' then
               Writeln(Lst,'Item:',ItemNumber:5,' ',Name:20,
                       'From: ',Supplier:5,
                       'Now in Stock: ',InStock:0:0);
          end;
     end;
     close(ProductFile);
end.
```

14.6 **Ein- und Ausgabe von Textfiles**

Die Ein- und Ausgabe von Daten in lesbarer Form wird mittels Textfiles so, wie in 14.5 beschrieben, ausgefuehrt. Ein Textfile kann irgendeinem Geraet, d.h. einem Diskfile oder einem Standard-I/O-Geraet zugewiesen werden. Die Ein- und Ausgabe kann ausgefuehrt werden mit den Standardprozeduren READ, READELN, WRITE und WRITELN, die eine spezielle Syntax fuer ihre Parameterliste verwenden, um eine maximale Flexibilitaet der Ein- und Ausgaben zu erreichen.
Die Parameter koennen im Einzelnen einen unterschiedliche Typen haben. In diesen Faellen erfolgt eine automatische Datenkonvertierung bei der Ein- und Ausgabe in und aus den Standard-Char-Typen des Textfiles.
Ist der erste Parameter einer I/O-Prozedur ein Variablenbezeichner eines Textfiles, dann bezieht sich die Ein- oder Ausgabe auf dieses File, andernfalls bezieht sie sich auf des Standartextfile Input oder Output (s.14.5.3).

14.6.1 READ-Prozedur

Die Read-Prozedure ermoeglicht die Eingabe von Zeichen, Strings und numerischen Daten.
Syntax:

```
          Read(Var1, Var2, .. , VarN)
oder      Read(File, Var1, .. , VarN)
```

wobei Var1, .. ,VarN Variable vom Typ Char, String, Integer oder Real sein koennen. Die erste Form liest Daten vom Standardfile Input, normalerweise dem Keyboard (Tastatur). Die zweite Form liest Eingaben vom Textfile File, welches fuer das Lesen vorbereitet werden muss.

Mit einer Variablen vom Typ Char liest Read vom File ein Zeichen und weist dieses der Variablen zu. Wenn das File ein Diskfile ist, wird Eoln True, wenn das naechste Zeichen CR oder CTRL-Z ist und Eof wird True, wenn das naechste Zeichen CTRL-Z ist.
Wenn das File ein logisches Geraet (einschliesslich Input und Output) ist, wird Eoln True, wenn das letzte Zeichen CR oder CTRL-Z war und Eof wird True, wenn das letzte Zeichen CTRL-Z war.

Mit einer Variablen vom Typ String liest Read soviele Zeichen, wie durch die maximale Laenge des String erlaubt sind, es sei denn Eoln oder Eof wurde vorher erreicht.
Eoln wird True, wenn das letzte gelesene Zeichen CR oder CTRL-Z war, und Eof wird True, wenn das letzte gelesene Zeichen CTRL-Z war.

Mit einer numerischen Variablen (Integer oder Real) erwartet Read eine Zeichenkette, die mit dem Format einer numerischen Konstanten des entsprechenden Typs, wie in Abschnitt 4.2. definiert, uebereinstimmt. Voranstehende Leerzeichen, HT, CR oder LF werden uebersprungen. Die Zeichenkette darf nicht laenger als 30 Zeichen sein und muss mit einem Leerzeichen, HT, CR oder CTRL-Z beendet sein. Wenn die Zeichenkette nicht mit dem Format uebereinstimmt, tritt ein I/O-Fehler auf. Im anderen Fall wird die numerische Zeichenkette in den entsprechenden Typ konvertiert und der Variablen zugewiesen.
Wenn von einem Diskfile gelesen wurde und die Eingabezeichenkette endet mit einem Leerzeichen oder HT, dann startet die naechste Read oder Readln Operation mit dem Zeichen, das unmittelbar diesem Leerzeichen oder HT folgt. Fuer beide, Diskfile oder logischem Geraet, wird Eoln True, wenn die Zeichenkette mit CR oder CTRL-Z endete und Eof wird True, wenn sie mit CTRL-Z endete. Ein Spezialfall der numerischen Eingabe tritt auf, wenn Eoln oder Eof bereits beim Beginn (beispielsweise, wenn die Eingabe nur CR war) True wird. In diesem Falle wird der Variablen kein neuer Wert zugewiesen und die Variable behaelt ihren alten Wert.

Wenn das Eingabefile CON: zugewiesen wurde, oder wenn das Standardfile im {$B+}-Modus verwendet wurde, gelten spezielle Gesetze fuer das Lesen der Variablen. Beim Aufruf von Read oder Readln wird die ganze Zeile von der Console in einen Puffer gebracht und das Einlesen der Variablen erfolgt aus diesem Puffer als Eingabequelle. Dadurch wird ein Editieren waehrend der Eingabe moeglich:

Backspace und DEL — Ruecksetzen des Cursors und Loeschen des dort stehenden Zeichens. Backspace wird durch die Taste --> oder CTRL-H erzeugt. DEL wird durch die Taste DEL oder RUBOUT erzeugt.

CTRL-X — Ruecksetzen des Cursor auf den Eingabebeginn und loschen aller eingegebenen Zeichen. CTRL-X wird durch die Taste (Pfeil nach unten) er-

zeugt.

Die Enter-Taste beendet die Eingabe. Das dabei eingegebene CR wird nicht als Echo auf dem Bildschirm ausgegeben.
Intern wird die Eingebezeile mit einem CTRL-Z am Ende gespeichert. Ist diese Eingabezeile kuerzer als die Variablen in der Parameterliste, werden die restlichen Variablen wie folgt behandelt: bei Char wird CTRL-Z eingetragen, bei String wird mit Leerzeichen aufgefuellt, und numerische Variable bleiben unveraendert.

Maximal koennen in eine Eingabezeile 127 Zeichen eingegeben werden. Man kann die Eingabezeile jedoch auch begrenzen. Dazu wird der vordeklarierten Variable Buflen eine Integerzahl aus dem Bereich 0 bis 127 zugewiesen.
Beispiel:

```
Write('Filename (max.14 Zeichen):');
Buflen := 14;
Readln(FileName);
```

Es ist zu beachten, dass die Zuweisungen zu Buflen nur fuer das unmittelbar darauffolgende Read wirken. Danach wird Buflen sofort wieder auf 127 gesetzt.

14.6.2 Readln-Prozedur

Die Readln-Prozedur ist mit der Read-Prozedur identisch, ausser dass nach dem Einlesen der letzten Variablen der Rest der Zeile uebersprungen wird, d.h. alle Zeichen bis und einschliesslich der naechsten CR/LF-Folge (oder dem naechsten CR bei einem logischen Geraet) werden uebersprungen.
Syntax:

```
Readln(Var1, Var2, .. , VarN)
oder    Readln(File, Var1, .. , VarN)
```

Nach einem Readln liest das naechste Read oder Readln vom Beginn der naechsten Zeile. Eoln ist immer False nach Readln, es sei denn, Eof ist True. Readln kann ebenso ohne Parameter aufgerufen werden:

```
Readln        oder      Readln(File)
```

In diesen Faellen wird die gesamte Zeile uebersprungen. Wenn Readln von der Console liest (mit dem Standardfile Input oder einem File, dem CON: zugewiesen wurde), wird im Gegensatz zu Read das beendende CR als Echo in der Form CR/LF-Folge auf den Bildschirm uebertragen.

14.6.3 Write-Prozedure

Die Write-Prozedure ermoeglicht die Ausgabe von Zeichen, Strings, Booleschen und numerischen Werten.
Syntax:

```
Write(Wp1, Wp2, .. , WpN)
```

oder Write(File,Wp1, .. , WpN)

wobei Wp1, .. , WpN die (Write-Parameter) Variablen vom Typ Char, String, Boolean, Integer oder Real sind. Wahlweise folgt diesen Parametern jeweils ein Doppelpunkt und ein Integerausdruck, der die Laenge des Ausgabefeldes angibt. In der ersten der oben angegebenen Formen erfolgt die Ausgabe der Variablen durch das Standardtextfile Output, das gewoehnlich ein Bildschirm ist. Im zweiten Fall werden die Variablen durch das Textfile File ausgegeben. Dieses File muss natuerlich fuer die Ausgabe vorbereitet werden.

Die Formate der Write-Parameter haengen vom Typ der Variablen ab. Im Folgenden werden die unterschiedlichen Formate und ihre Eigenschaften beschrieben. Dabei bezeichenen die Symbole:

I,m,n	Ausdruecke vom Typ	Integer
R	Ausdruecke vom Typ	Real
Ch	Ausdruecke vom Typ	Char
S	Ausdruecke vom Typ	String
B	Ausdruecke vom Typ	Boolean

Formatuebersicht

Ch — Ausgabe des Zeichen Ch.

Ch:n — Ausgabe des Zeichens Ch rechtsbuendig in einem n Zeichen langen Feld, d.h. vor Ch stehen n-1 Blanks.

S — Ausgabe des String S. Zeichenfelder (Arrays) koennen ebenfalls ausgegeben werden, wenn sie mit den Strings uebereinstimmen.

S:n — Ausgabe der Strings rechtsbuendig in einem n Zeichen langen Feld, d.h. vor S stehen n-Len(S) Blanks.

B — Ausgabe des Wortes True oder False.

B:n — Ausgabe des Wortes True oder False rechtsbuendig in einem n Zeichen langen Feld.

I — Ausgabe der Dezimaldarstellung von I.

I:n — Ausgabe der Dezimaldarstellung von I rechtsbuendig in einem n Zeichen langen Feld.

R — Ausgabe der Dezimaldarstellung von R rechtsbuendig in einem 18 Zeichen langen Feld als Gleitkommazahl in der Form:

```
R >= 0        __d.ddddddddddEtdd
R <  0        _-d.ddddddddddEtdd
```

Dabei bedeuten die Zeichen _ Blanks, d Ziffern und t endweder + oder -.

R:n — Ausgabe der Dezimaldarstellung von R rechtsbuendig in einem n Zeichen langen Feld als Gleitkommazahl in der Form:

```
      R >= 0        blanks d. digits Etdd
      R <  0        blanks-d. digits Etdd
```

Dabei bedeuten blanks keine oder mehrere Blanks, digits ein bis zehn Ziffern, d eine Ziffer und T entweder + oder -. Nach dem Dezimalpunkt wird mindestens eine Ziffer ausgegeben, d.h. n muss mindestens 7 (bzw.8) sein. Ist n groesser als 16 (bzw.17), so stehen vor der Zahl n-16 (bzw.n-17) Blanks.

R:n:m Ausgabe der Dezimaldarstellung von R rechtsbuendig in einem n Zeichen langen Feld als Festpunktzahl mit m Dezimalziffern. Dabei muss m im Bereich 0 <= m <= 24 liegen, sonst wird Gleitkommaformat verwendet. Das Feld wird vor der Zahl mit Blanks aufgefuellt.

14.6.4 Writeln-Prozedur

Die Writeln-Prozedur ist identisch mit der Write-Prozedur, ausser dass nach der letzten Variablen eine CR/LF-Folge ausgegeben wird.
Syntax:

```
         Writeln(Wp1, Wp2, .. , WpN)
oder     Writeln(File,Wp2, .. , WpN)
```

Wird die Prozedur ohne Write Parameter angegeben in der Form

```
         Writeln     oder       Writeln(File)
```

so wird nur die CR/LF-Folge ausgegeben.

14.7 **Nichttypisierte Files**

Nichttypisierte (untyped) Files sind Kanalein- und -ausgaben auf dem niedrigsten Niveau. Sie werden fuer den Direktzugriff zu einem Diskfile mit der Standardlaenge von 128 Bytes verwendet.
Bei Ein- und Ausgabeoperationen mit nichttypisierten Files werden die Daten direkt zwischen dem Diskfile und der Variablen uebertragen, ohne Platz fuer einen Sektorpuffer, wie bei den typisierten Files, zu benoetigen. Eine nichttypisierte Filevariable braucht daher weniger Platz als andere Filevariable. Wird eine Filevariable nur fuer die Prozeduren Erase, Rename oder andere I/O-Operationen verwendet, die eigentlich gar keine Eingaben oder Ausgaben sind, so sind nichttypisierte Filevariable vorzuziehen.

Ein nichttypisiertes File wird durch das reservierte Wort **File** definiert:
Beispiel:

```
          Var
               datafile : File
```

14.7.1 Operationen mit nichttypisierten Files

Fuer nichttypisierte Files sind alle Standard-Filebehandlungs-Prozeduren und -Funktionen, ausser Read, Write und Flush erlaubt. Read und Write werden durch zwei spezielle schnelle

Uebertragungsprozeduren ersetzt:
Syntax:

```
BlockRead(filvar,var,recs)
BlockWrite(filvar,var,recs)
```

dabei sind filvar ein Variablenbezeichner eines nichttypisierten Files, var irgendeine Variable und recs ein Integerausdruck, der die Anzahl der zu uebertragenden 128-Bytes-Saetze zwischen dem Diskfile und dem Speicher ist.

Die Uebertragung beginnt mit dem ersten Byte, das zur Variablen var gehoert. Der Programmierer hat selbst dafuer zu sorgen, dass hinter der Variablen genuegend Platz bereitgestellt ist, um alle Daten der Uebertragung ablegen zu koennen. Der Aufruf von BlockRead und BlockWrite rueckt auch den Filepointer um die entsprechende Anzahl von Saetzen weiter.
Ein File, das durch BlockRead oder BlockWrite bearbeitet werden soll, muss zuerst durch Assign und Rewrite oder Reset vorbereitet werden. Rewrite eroeffnet und baut ein neues File auf und Reset eroeffnet ein bereits existierendes File. Nach den Uebertragungs-prozeduren sollte mit Close ein sicherer Abschluss garantiert werden. Die Standardprozedure Seek und die Standardfunktionen Pos, FileSize und Eof arbeiten exakt genauso wie bei typisierten Files.

Das folgende Programm zeigt die Verwendung eines nichttypisierten Files. Es liest ein Diskfile und kopiert seinen Inhalt auf ein anderes Diskfile:

```
program Copy;

const
   BufSize     = 100;
   BufByteSize = 12800;

var
   Source,
   Destination     : File;
   SourceName,
   DestinationName : String[14];
   Buffer          : Array[1..BufByteSize] of Byte;
   NoOfRecsToRead,
   Remaining       : Integer;

begin
   Write('Enter Source file name:     ');
   Readln(SourceName);
   Assign(Source,SourceName);
   Reset(Source);
   Write('Enter destination file name:  ');
   Readln(DestinationName);
   Assign(Destination,DestinationName);
   Rewrite(Destination);
   Remaining := FileSize(Source);
   if Remaining <> 0 then
      begin
```

```
            repeat
               if BufSize <= Remaining then NoOfRecsToRead := BufSize
               else NoOfRecsToRead := Remaining;
               BlockRead(Source,Buffer,NoOfRecsToRead);
               BlockWrite(Destination,Buffer,NoOfRecsToRead);
               Remaining := Remaining - NoOfRecsToRead;
            until Remaining = 0;
            Close(Destination);
            Close(Source);
         end;
   end.
```

14.8 **Ein- und Ausgabepruefung**

Zur Generierung einer Ein- und Ausgabepruefung waehrend der Laufzeit eines Programmes wird die I-Compiler-Direktive verwendet. Der Standard-Status ist aktiv, d.h.{$I+}. Damit wird nach jeder I/O-Operation eine I/O-Pruefroutine aufgerufen und bei auftretenden I/O-Fehlern wird das Programm abgebrochen und eine Fehlermitteilung ausgegeben, die den Typ des Fehlers anzeigt.

Ist die I-Compiler-Direktive passiv, d.h. {$I-}, dann wird zur Laufzeit des Programmes keine I/O-Pruefung durchgefuehrt. Dann ist es notwendig, das Ergebnis von I/O-Operationen durch Verwendung der Standardfunktion IOresult zu ueberwachen. Denn in diesem Falle fuehren I/O-Fehler zwar nicht zum Programmstop, aber sie unterbinden jede weitere Ein- und Ausgabe, bis die Funktion IOresult aufgerufen wurde. Bei Ausfuehrung dieser Funktion wird die Fehlerbedingung zurueckgesetzt und es koennen wieder Ein- und Ausgaben durchgefuehrt werden. Nach dieser Ruecksetzung der Fehlerbedingung liefern spaeter folgende Aufrufe von IOresult den Wert 0, bis der naechste Fehler auftritt. Damit ist der Programmierer selbst in der Lage, geeignete Massnahmen bei auftretenden Fehlern vorzunehmen. Die Rueckgabe des Wertes 0 durch IOresult zeigt stets nach einer Ein- oder Ausgabe eine erfolgreiche Operation an, alle anderen zurueckgegebenen Werte weisen auf einen Fehler bei der letzten I/O-Operation hin. Im Anhang sind alle Fehlermitteilungen und ihre Nummern aufgelistet.

Die IOresult-Funktion ist fuer die Faelle sehr geeignet, wo bei einem Fehler ein Programmstop unzweckmaessig ist. So kann im folgenden Beispiel damit solange nach einem Filenamen gefragt werden, bis die Reset-Funktion ein erfolgreiches Ergebnis liefert, d.h. das File gefunden wurde.

```
program OPINFILE;

  var
  InFile       : File;
  InFileName   : String[14];
  OK           : Boolean;
     procedure OpenInFile;
     begin
        repeat
           Write('Enter name of input File: ');
           Readln(InFileName);
           Assign(InFile,InFileName);
```

```
          {$I-} Reset(InFile); {$I+}
          OK := (IOresult = 0);
          if not OK then Writeln('Cannot find file: ',InFileName);
       until OK;
    Close(InFile);
    end;
begin
OpenInFile;
end.
```

Wenn in einem Programm die I-Compiler-Direktive {$I-} passiv ist, sollten fuer die folgenden Standardprozeduren stets mittels der Funktion IOresult geeignete Fehlermassnahmen durchgefuehrt werden:

BlockRead	BlockWrite	Chain	Close
Erase	Execute	Flush	Read
Readln	Rename	Reset	Rewrite
Seek	Write	Writeln	

15. Pointertypen

Die bisher besprochenen Variablen sind statische Variablen, d.h. ihre Form und Groesse sind im Deklarationsteil festgelegt und sie bleiben auch waehrend der gesamten Ausfuehrung des Blockes, indem sie erklaert sind, existent. Programme benoetigen jedoch haeufig eine Datenstruktur, die sich in Form und Groesse waehrend der Ausfuehrung des Programmes aendern kann. Aus diesem Grunde wurden dynamische Variable eingefuehrt, die man erst dann generieren kann, wenn sie gebraucht werden, und die man nach ihrer Verwendung, wenn sie nicht mehr gebraucht werden, beseitigen kann.

Solche dynamischen Variablen werden nicht explizit in der Variablendefinition wie die statischen Variablen erklaert und man kann sich auf sie auch nicht direkt durch einen Bezeichner beziehen. Man verwendet fuer sie spezielle Variable, die nur jeweils die Speicheradresse der entsprechenden Variablen enthalten, die also zu diesen Variablen zeigen (point). Diese speziellen Variablen werden als Pointervariablen bezeichnet.

15.1 Definition einer Pointervariablen

Ein Pointertyp wird durch das Symbol ^ definiert. Diesem Symbol folgt der Typbezeichner der dynamischen Variablen, auf die sich die Pointervariablen dieses Typs beziehen.

Das folgende Beispiel zeigt, wie ein Satz und die auf ihn sich beziehenden Pointer definiert werden:

```
    type
         PersonPointer = ^PersonRecord;

         PersonRecord  = Record
```

```
                    Name   : String[50];
                    Job    : String[50];
                    Next   : PersonPointer;
                end;

var
     FirstPerson,LastPerson,NewPerson : PersonPointer;
```

Die Variablen FirstPerson, LastPerson und NewPerson sind Pointervariablen, die den Zugriff zu Saetzen vom Typ PersonRecord gestatten. Aus dem Beispiel ist auch ersichtlich, dass sich der Typbezeichner in einer Pointertypdefinition auf einen Bezeichner beziehen kann, der bis dahin noch nicht erklaert wurde.

15.2 **Zuweisungsvariable (New)**

Bevor es ueberhaupt einen Sinn hat, eine von diesen Pointervariablen zu verwenden, benoetigen wir natuerlich einige neue Variable, auf die sie zeigen. Die Generierung und Zuweisung zu solchen neuen Variablen von irgendeinem Typ erfolgt mit der Standardprozedur **New**. Die Prozedur hat einen Parameter, der eine Pointervariable von dem Typ ist, den wir generieren wollen.
Beispielsweise generiert

```
New(FirstPerson)
```

eine neue Variable vom Typ PersonRecord. Damit zeigt FirstPerson auf einen dynamisch erzeugten Satz von Typ PersonRecord.

Zuweisungen zwischen Pointervariablen kann man solange durchfuehren, solange sie vom gleichen Typ sind. Pointervariablen vom gleichen Typ koennen ebenso durch die Vergleichsoperatoren = und <> verglichen werden. Diese Operationen geben ein Boolesches Ergebnis (True oder False) zurueck.

Der Pointerwert **nil** gehoert jedem Pointertyp an. Dieser Wert verweist auf keine dynamische Variable und kann Pointervariablen zugewiesen werden, um anzuzeigen, dass sie keinen verwertbaren Zeiger enthalten. Natuerlich kann nil auch im Vergleich verwendet werden.

Variable, die man mit der Standardprozedur New erzeugt, werden in einer Stack-artigen Struktur aufgebaut. Man bezeichnet sie als **Heap**. Das TURBO-Pascal-System steuert den Heap durch Verwendung eines Heap-Pointers, der zu Programmbeginn auf die erste freie Adresse des Speichers weist. Bei jedem Aufruf von New wird der Heap-Pointer um soviele Bytes in Richtung Speicherende weitergestellt, als die dynamische Variable Bytes enthaelt.

15.3 **Mark und Release**

Wenn eine dynamische Variable nicht mehr im Programm benoetigt wird, kann man durch Verwendung der Standardprozeduren Mark und Release den Speicherplatz zurueckerhalten, der dieser Variablen zugewiesen wurde.

Die Prozedur Mark weist den Wert des Heap-Pointers einer Variablen zu:
Syntax:

```
Mark(Var);
```

wobei Var eine Pointervariable ist.

Die Prozedur Release setzt den Heap-Pointer auf die Adresse, die in seinem Argument enthalten ist:
Syntax:

```
Release(Var);
```

wobei Var eine Pointervariable ist, die vorher durch Mark gesetzt wurde.
Release gibt damit den gesamten Speicherplatz zurueck, der oberhalb der im Argument angegebenen Variablen liegt. Es ist natuerlich nicht moeglich, den Speicherplatz von Variablen zurueckzuerhalten, die in der Mitte des Heap liegen.

Mit der Standardfunktion **MemAvail** ist es moeglich zu einer beliebigen Zeit, die augenblickliche Groesse des vom Heap benutzten Speicherplatzes zu erhalten. Genaueres darueber im Anhang A.

15.4 **Verwendung der Pointer**

Nehmen wir an, in einem Programm sei die Prozedure New verwendet worden, um eine Reihe von Saetzen des Typs PersonRecord (wie im obigen Beispiel) aufzubauen. Dabei sei das Feld Next so verwendet worden, dass es auf den naechsten PersonRecord weist. Dann wuerden die folgenden Anweisungen durch die Liste dieser Saetze gehen und den Inhalt jedes Satzes ausschreiben (FirstPerson weist auf den ersten Satz in der Liste):

```
while FirstPerson <> nil do
begin
     Writeln(FirstPerson^.Name,'is a  ',FirstPerson^.Job);
     FirstPerson := FirstPerson^.Next;
end;
```

Hierbei kann man FirstPerson^.Name lesen als FirstPerson's Name, das ist das Feld Name in dem Satz, auf den FirstPerson zeigt.

Das folgende Beispiel demonstriert die Verwendung von Pointern bei einer Liste, die Namen und entsprechende Berufswuensche enthaelt. Die Namen und Berufswuensche werden nacheinander eingegeben. Die Eingabe der Liste wird durch Druecken nur der Enter-Taste bei der Eingabe des Namens beendet. Danach wird die gesamte Liste gedruckt. Nach dem Druck wird der Speicherplatz freigegeben, der von der Liste benutzt wurde. Die Speichervariable HeapTop wird nur fuer den Zweck benutzt, den ersten Wert des Heap-Pointers aufzuheben. Seine Definition als ^Integer (Pointer zu Integer) ist daher voellig willkuerlich.

```
program   Jobs;
type
   PersonPointer  = ^PersonRecord;
```

```
   PersonRecord   = record
                       Name  : String[50];
                       Job   : String[50];
                       Next  : PersonPointer;
                    end;

var
   HeapTop  : ^Integer;
   FirstPerson,LastPerson,NewPerson : PersonPointer;
   Name  : String[50];

begin
   FirstPerson := nil;
   Mark(HeapTop);
   repeat
      Write('Enter Name:   ');
      Readln(Name);
      if Name <> '' then
      begin
         New(NewPerson);
         NewPerson^.Name := Name;
         Write('Enter profession:  ');
         Readln(NewPerson^.Job);
         Writeln;
         if FirstPerson = nil then
            FirstPerson := NewPerson
         else
            LastPerson^.Next := NewPerson;
         LastPerson := NewPerson;
         LastPerson^.Next := nil;
      end;
   until Name = '';
   Writeln;
   while FirstPerson <> nil do
   begin
      Writeln(FirstPerson^.Name,' is a ',FirstPerson^.Job);
      FirstPerson := FirstPerson^.Next;
   end;
   Release(HeapTop);
end.
```

15.5 **Speicherplatzzuordnung**

Mit der Standardprozedur **GetMem** ist es moeglich Speicherplatz beliebiger Groesse aus dem Heap zuzuweisen. Waehrend New nur soviel Speicherplatz zuweist, wie der Typ der im Argument verwendeten Pointervariablen benoetigt, erlaubt GetMem dem Programmierer die Groesse des zugewiesenen Speicherplatzes selbst zu bestimmen. GetMem wird mit zwei Parametern aufgerufen:
Syntax:

```
GetMem(Pvar,I);
```

wobei Pvar irgendeine Pointervariable und I in Integerausdruck ist, der die Anzahl der zugewiesenen Bytes angibt.

16. Prozeduren und Funktionen

Ein Pascal-Programm besteht aus einem oder mehreren Bloecken. Jeder von ihnen kann wiederum aus Bloecken bestehen etc. Eine Prozedur, wie eine Funktion, ist einer von diesen Bloecken. Im allgemeinen kann man sie als Unterprogramme auffassen. Eine Prozedur ist also ein separater Teil eines Programmes, das von irgendwelchen anderen Stellen im Programm aus durch eine Prozeduranweisung aktiviert wird (s.7.1.2). Eine Funktion ist ganz aehnlich, nur berechnet sie etwas und gibt diesen berechneten Wert durch eine Variable zurueck, wenn ihr Bezeichner (der Funktionsaufruf) waehrend der Ausfuehrung des Programmes benutzt wird (s.6.2).

16.1 **Parameter**

Werte koennen den Prozeduren oder Funktionen durch Parameter uebergeben werden. Durch diese Parameter wird ein Substitutionsmechanismus bereitgestellt, der erlaubt, die Logik des Unterprogrammes mit verschiedenen Anfangswerten zuversehen, sodass es entsprechend verschiedene Ergebnisse produziert.
Die Prozeduranweisung oder der Funktionsbezeichner, die das entsprechende Unterprogramm aufrufen, koennen eine Liste von Parametern enthalten, die man als die aktuellen Parameter bezeichnet. Diese werden den formalen Paramtern uebergeben, die im Kopf des Unterprogrammes spezifiziert sind. Die Zuordnung der Parameter bei der Uebergabe erfolgt in der Reihenfolge ihres Auftretens in der Parameterliste. Pascal unterstuetzt zwei unterschiedliche Methoden der Parameteruebergabe: Uebergabe der Parameter durch Uebergabe eines Wertes (Wertuebergabe) und Uebergabe der Parameter durch Austausch der formalen Parameter durch die aktuellen Parameter (Referenz).

Werden Parameter durch Wertuebergabe uebertragen, stellt der formale Parameter eine selbstaendige logische Variable im Unterprogramm dar und alle Wertveraenderungen an diesem formalen Parameter im Unterprogramm haben keinen Einfluss auf den Wert des aktuellen Parameters nach Ausfuehrung des Unterprogrammes. Der aktuelle Parameter kann ein beliebiger Ausdruck, auch eine Variable, sein, muss aber den gleichen Typ wie der formale Parameter haben. Solche Parameter werden als Wert-Parameter bezeichnet und im Kopf des Unterprogrammes, wie im folgenden Beispiel, definiert. (Die folgenden Beispiele zeigen, dass die Prozedurvereinbarungen gegenueber den Funktionsvereinbarungen, die in 16.3.1 erklaert sind, ganz aehnlich sind, der Funktionskopf definiert nur noch zusaetzlich den Typ des Ergebnisses).

Beispiel Wertparameter:

```
procedure Example(Num1.Num2:Number; Str1,Str2:Txt);
```

Hierbei sind Number und Txt vordeklarierte Typen (z.B. Integer und String[255]) und Num1 , Num2, Str1 und Str2 sind formale Parameter, denen die Werte der aktuellen Parameter uebergeben werden. Die Typen der formalen und aktuellen Parameter muessen uebereinstimmen.

Bei der Definition des Kopfes ist zu beachten, dass der Typ der Parameter durch einen Bezeichner eines vordeklarierten Typs spezifiziert werden muss. Die folgende Konstruktion ist deshalb nicht erlaubt:

```
procedure Select(Model: array[1..500] of Integer);
```

Es ist notwendig den Typ von Model in der Typdefinition des Blockes zu erklaeren. Dann kann der Bezeichner des Typs in der Parametererklaerung des Kopfes verwendet werden:

```
type
     Range = array[1..500] of Integer;

procedure Select(Model: Range);
```

Wird ein Parameter durch Referenz ubergeben, stellt in der Tat der aktuelle Parameter den formalen Parameter waehrend der Ausfuehrung des Unterprogrammes dar. Jede im Unterprogramm vorgesehene Veraenderung des Inhaltes des formalen Parameters aendert waehrend der Ausfuehrung des Unterprogrammes in Wirklichkeit den Inhalt des aktuellen Parameters. Aus diesem Grunde muss der aktuelle Parameter stets eine Variable sein. Solche durch Referenz uebergebenen Parameter werden als Variablen-Parameter bezeichnet. Die Definition erfolgt wie im folgenden Beispiel:

Beispiel Variablenparameter:

```
procedure Example(var Num1,Num2:Number; Str1,Str2:Txt);
```

Hierbei sind Num1 und Num2 Variablenparameter und Str1 und Str2 Wertparameter.
Alle Adressrechnungen werden erst zur Zeit des Prozeduraufrufes ausgefuehrt. Ist eine Variable beispielsweise die Komponente eines Array, wird ihr Index erst zum Zeitpunkt des Unterprogrammaufrufes berechnet.

Beachte, File-Parameter muessen immer als Variablenparameter erklaert werden.

Wird eine grosse Datenstruktur, wie ein Array, einem Unterprogramm als Parameter uebergeben, dann spart die Verwendung eines Variablenparameters Abarbeitungszeit und Speicherplatz, da als einzige Information zwei Bytes an das Unterprogramm uebergeben werden, die die Adresse des aktuellen Parameters enthalten. Ein Wertparameter wuerde Speicherplatz fuer eine extra Kopie der gesamten Datenstruktur verlangen und ausserdem Zeit fuer das Kopieren benoetigen.

16.1.1 Reduzierung der Parametertyppruefung

Normalerweise muessen bei Verwendung von Variablenparametern die formalen und aktuellen Parameter exakt vom gleichen Typ sein. Dies bedeutet, wenn ein Variablenparameter vom Typ String verwendet wird, duerfen als aktuelle Parameter nur Strings mit exakt

der im Unterprogramm definierten Laenge zugewiesen werden. Diese Einschraenkung kann man durch Verwendung der V-Compiler-Direktive beseitigen. Fuer diese Direktive erzeugt der Standard-aktiv-Status {$V+} eine genaue Typpruefung, waehrend der passive Status {$V-} die Typpruefung soweit vermindert, dass aktuelle Parameter mit beliebiger Stringlaenge, unabhaengig von der definierten Laenge des formalen Parameters, uebertragen werden duerfen.

Beispiel:

```
program NSA;
{This program must be compiled with the $V-Directive}
{$V-}
type
     WorkString = string[255];

var
     Line1 : string[80];
     Line2 : string[100];

procedure     Encode(var LineToEncode : WorkString);
var I : Integer;
begin
     for I := 1 to Length(LineToEncode) do
     LineToEncode[I] := Chr(Ord(LineToEncode[i]) - 30);
end;
begin
     Line1 := 'This is s secret message';
     Encode(Line1);
     Line2 := 'Here is another (longer) secret message';
     Encode(Line2);
end.
```

16.1.2 Nichttypisierte Variablenparameter

Ist der Typ eines formalen Parameters nicht definiert. d.h. enthaelt der Parameterteil im Kopf des Unterprogrammes keine Typdefinition, dann wird der Parameter als nichttypisiert bezeichnet. Der aktuelle Parameter kann dann ein beliebiger Typ sein. Andererseits ist der nichttypisierte Parameter mit allen Typen inkompatibel. Aus diesem Grunde kann man nichttypisierte formale Parameter nur dort verwenden, wo der Datentyp ohne Bedeutung ist. Dies ist beispielsweise bei den Parametern von Addr, BlockRead, BlockWrite, FillChar oder Move und bei Adressspezifikationen von absoluten Variablen der Fall.

Im folgenden Beispiel wird bei der Prozedur SwitchVar die Verwendung nichttypisierter Parameter demonstriert. Sie uebertraegt den Inhalt der Variablen A1 nach A2 und von A2 nach A1.

```
procedure SwitchVar(var A1p,A2p; Size : Integer);
type
     A = array[1..MaxInt] of Byte;
var
     A1    : A absolute A1p;
```

```
          A2    : A absolute A2p;
          Tmp   : Byte;
          Count : Integer;
     begin
          for Count := 1 to Size do
          begin
               Tmp       := A1[Count];
               A1[Count] := A2[Count];
               A2[Count] := Tmp;
          end;
     end;
```

Definiert man:

```
     type
          Matrix = array[1..50,1..25] of Real;
     var
          TestMatrix,BestMatrix : Matrix;
```

dann kann man SwitchVar zum Austauschen des Inhaltes der beiden Matrizen verwenden. Der Prozeduraufruf lautet dann:

```
     SwitchVar(TestMatrix,BestMatrix,SizeOf(TestMatrix));
```

16.2 **Prozeduren**

Eine Prozedur kann entweder vordeklariert (Standard) oder nutzerdeklariert (durch den Programmierer vereinbart) sein. Vordeklarierte Proceduren sind Teile des TURBO-Pascal-Systems und koennen ohne weitere Vereinbarungen verwendet werden. Einer nutzerdeklarierten Prozedur kann der Name einer Standardprozedur gegeben werden, aber dann ist die Standardprozedur innerhalb des Gueltigkeitsbereiches der nutzerdeklarierten Prozedur nicht aufrufbar.

16.2.1 Prozedurvereinbarung

Eine Prozedurvereinbarung besteht aus einem Prozedurkopf und einem ihm folgenden Block. Dieser Block besteht aus einem Deklarationsteil und einem Anweisungsteil.

Der Prozedurkopf besteht aus dem reservierten Wort **procedure**, dem ein Bezeichner folgt, der als Name der Prozedur bezeichnet wird. Gewoehnlich folgt ihm eine formale Parameterliste, wie in 16.1. beschrieben.

Beispiele:

```
          procedure LogOn;
          procedure Position(X,Y : Integer);
          procedure Compute(var Data : Matrix; Scale : Real);
```

Der Deklarationsteil einer Prozedur hat die gleiche Form wie bei einem Programm. Alle in der formalen Parameterliste im Deklarationsteil erklaerten Bezeichner sind lokal zur Prozedur

und zu jeder Prozedur in ihr. Dieser Bereich heisst Gueltigkeitsbereich der Bezeichner. Ausserhalb dieses Bereiches sind sie nicht bekannt. Eine Prozedur kann jede in einem zu ihr äusseren Block definierte Konstante, Type, Variable, Prozedur oder Funktion verwenden.

Der Anweisungteil spezifiziert die Aktionen, die ausgefuehrt werden sollen, wenn die Prozedur aufgerufen wird. Er hat die Form einer Verbundanweisung (s.7.2.1). Wird der Prozedurbezeichner selbst innerhalb des Anweisungsteiles verwendet, wird die Prozedur rekursiv ausgefuehrt (bei CP/M80 muss dann beachtet werden, dass zu diesem Zeitpunkt bei der Compilierung die A-Compiler-Direktive passiv {$A-} gesetzt ist s.Anhang E).

Das naechste Beispiel zeigt ein Programm, das eine Prozedur verwendet und dieser Prozedur Parameter uebergibt. Da die aktuellen Parameter, die an die Prozedur uebergeben werden, Konstante (oder einfache Ausdruecke) sind, muss der formale Parameter als Wertparameter definiert werden:

```
program Box;
var
     I :Integer;
procedure DrawBox(X1,Y1,X2,Y2 : Integer);
     var  I : Integer;
     begin
          GotoXY(X1,Y1);
          for I := X1 to X2 do Write('-');
          for I := Y1+1 to Y2 do
          begin
               GotoXY(X1,I); Write('!');
               GotoXY(X2,I); Write('!');
          end;
          GotoXY(X1,Y2);
          for I := X1 to X2 do Write('-');
     end;
begin
     ClrScr;
     for I := 1 to 5 do DrawBox(I*4,I*2,10*I,4*I);
     DrawBox(1,1,80,23);
end.
```

Häufig sollen Veraenderungen des formalen Parameters in der Prozedur direkt auch den aktuellen Parameter betreffen. In diesen Faellen sind natuerlich Variablenparameter anzuwenden, wie im folgenden Beispiel:

```
procedure Switch(var A,B : Integer);
var Tmp : Integer;
begin
     Tmp := A; A := B; B := Tmp;
end;
```

Wenn diese Prozedur durch die Anweisung Switch(I,J); aufgerufen wird, werden die Werte von I und J ausgetauscht. Wuerde

stattdessen faelschlicherweise der Prozedurkopf durch

```
procedure Swap( A,B : Integer);
```

d.h. mit einem Wertparameter erklaert, dann werden durch die Anweisung Swap(I,J); die Werte von I und J nicht veraendert.

16.2.2 Standardprozeduren

TURBO-Pascal enthaelt eine Anzahl von Standardprozeduren:

1) Stringbehandlungsprozeduren (s. 9.5),
2) Filebehandlungsprozeduren (s. 14.2, 14.5.1, 14.7.1),
3) Zuordnungsprozeduren fuer dynamische Variable (s. 15.2, 15.5)
4) Input- und Output-Prozeduren (s. 14.6).

Zusaetzlich werden die folgenden Standardprozeduren bereitgestellt, vorausgesetzt, die entsprechenden Terminalkommandos sind installiert.

16.2.2.1 Delay

Syntax: Delay(Time)

Diese Prozedur erzeugt eine Warteschleife, die in ungefaehr soviel Millisekunden durchlaufen wird, wie im Argument angegeben ist. Die exakte Zeit kann wegen der unteschiedlichen Hardware etwas davon abweichen.

16.2.2.2 ClrEol

Syntax: ClrEol

Diese Prozedur loescht alle Zeichen ab Cursorposition bis zum Ende der Zeile, ohne die Cursorposition zu veraendern.

16.2.2.3 ClrScr

Syntax: ClrScr

Diese Prozedur loescht den Bildschirm und setzt den Cursor in die linke obere Ecke. (Bei einigen Bildschirmtypen koennen dabei auch eventuell vorhandene Videoattribute bzw. vom Nutzer gesetzte Attribute veraendert werden).

16.2.2.4 DelLine

Syntax: DelLine

Diese Prozedur loescht die Zeile, in der der Cursor steht und schiebt alle darunter stehenden Zeilen um eine Zeile nach oben.

16.2.2.5 InsLine

Syntax: InsLine

Diese Prozedur fuegt an der Cursorposition eine leere Zeile ein und schiebt alle Zeilen unterhalb um eine Zeile nach unten, die letzte Zeile wird weggerollt.

16.2.2.6 Init
Syntax: Init

Diese Prozedur sendet die Terminal-Initialisierungszeichenkette, die bei der Installierung von TURBO-Pascal definiert wurde, an den Bildschirm.

16.2.2.7 Exit
Syntax: Exit

Diese Prozedur sendet die Terminal-Reset-Zeichenkette, die bei der Installierung definiert wurde, an den Bildschirm.

16.2.2.8 GotoXY
Syntax: GotoXY(Xpos,Ypos)

Diese Prozedur setzt den Cursor an die Position auf dem Bildschirm, die durch die Integerausdruecke Xpos (horizontaler Wert oder Zeile) und Ypos (vertikaler Wert oder Spalte) angegeben werden. Die linke obere Ecke (Home-Position) ist (1,1).

16.2.2.9 LowVideo
Syntax: LowVideo

Diese Prozedur setzt im Bildschirm das Attribut, das bei der Installation al "Ende der Hellsteuerung" festgelegt wurde.

16.2.2.10 HighVideo
Syntax: HighVideo

Diese Prozedur setzt im Bildschirm das Attribut, das bei der Installation als "Hellsteuerung" definiert wurde.

16.2.2.11 Randomize
Syntax: Randomize

Diese Prozedur erzeugt mittels Zufallszahlgenerator eine Zufallszahl.

16.2.2.12 Move
Syntax: Move(var1,var2,num)

Diese Prozedur kopiert im Speicher eine bestimmte Anzahl von Bytes. Hierbei sind var1 und var2 zwei Variable von beliebigem Typ und num ist ein Integerausdruck. Die Prozedur kopiert einen Block von num Bytes von der Stelle des ersten Bytes von var1 zur Stelle des ersten Bytes von var2. Move behandelt automatisch bei der Uebertragung auftretende Ueberlappungen, sodass "moveright" und "moveleft" Prozeduren nicht benoetigt werden.

16.2.2.13 FillChar
Syntax: FillChar(var,num,value)

Diese Prozedur fuellt einen Bereich im Speicher mit eine, gegebenen Wert. Hierbei ist var eine Variable eines beliebigen Typs, num ist ein Integerausdruck und Value ist ein Ausdruck vom Typ Byte oder Char. Es werden durch die Prozedur num Bytes beginnend

ab dem ersten Byte von var mit dem Wert von value gefuellt.

16.3 **Funktionen**

Funktionen sind wie Prozeduren entweder (vordeklarierte) Standardfunktionen oder sie sind vom Programmierer definiert.

16.3.1 Funktionsvereinbarung

Eine Funktionsvereinbarung besteht aus einem Funktionskopf und einem Block, der aus einem Deklarationsteil und einem Anweisungsteil besteht.

Der Funktionskopf ist mit dem Prozedurkopf equivalent, ausser dass der Funktionskopf mit dem reservierten Wort **function** eroeffnet wird und auch den Typ des Ergebnisses mit definieren muss. Dies wird durch Anfuegung eines Doppelpunktes und eines Types an den Funktionskopf erreicht.

Beispiele:

```
funktion KeyHit : Boolean;
function Comput(var Value : Sample) : Real;
function Power(X,Y : Real) : Real;
```

Der Ergebnistyp einer Funktion muss ein Skalartyp (d.h. Integer, Real, Boolean, Char deklariert als Skalar- oder Teilbereich), ein Stringtyp oder ein Pointertyp sein.

Der Deklarationsteil einer Funktion ist der gleiche wie bei einer Prozedur.

Der Anweisungsteil einer Funktion ist eine Verbundanweisung, so wie in 7.2.1 beschrieben. Innerhalb des Anweisungsteiles muss wenigstens eine Ergibtanweisung auftreten, die dem Funktionsbezeichner einen Wert zuweist. Die allerletzte dieser Ergibtanweisungen zum Funktionbezeichner ergibt den Wert der Funktion. Wenn der Funktionsbezeichner selbst als Funktionsaufruf im Anweisungsteil der Funktion auftritt, dann wird die Funktion rekursiv aufgerufen. In diesem Falle muss zu diesem Zeitpunkt die A-Compiler-Direktive {$A-} passiv sein (s. Anhang E).

Das folgende Beispiel zeigt die Verwendung einer Funktion zur Berechnung der Summe eine Reihe ganzer Zahlen von I bis J:

```
function RowSum(I,J : Integer) : Integer;
     function SimpleRowSum(S : Integer) : Integer;
     begin
        SimpleRowSum := S * (S+1) div 2;
     end;
begin
     RowSum := SimpleRowSum(J) - SimpleRowSum(I-1);
end;
```

Die Funktion SimpleRowSum ist in die Funktion RowSum eingebettet.

SimpleRowSum ist deshalb nur im Gueltigkeitsbereich von RowSum zulaessig.

Das folgende Beispiel zeigt die Verwendung rekursiver Funktionen:

```
program Fact;
var number : Integer;
function factorial(value : Integer) : Real;
begin
   if value = 0 then factorial := 1
   else factorial := value * factorial(value-1);
end;
begin
   Write('input number: ');
   Readln(number);
   if number < 0 then Writeln('non valid input!')
   else Writeln(number,'! = ',factorial(number):8:0);
end.
```

Bei der Definition eines Funktiontyps ist zu beachten, dass der in der Definition verwendete Typ vorher als Typbezeichner erklaert sein muss. Aus diesem Grunde ist die folgende Konstruktion nicht erlaubt:

```
function LowCase(Line : UserLine) : string[80];
```

Man muss stattdessen vorher den Typ string[80] durch einen Bezeichner erklaeren und mit diesem dann den Typ des Funktionsergebnisses definieren:

```
type
   str80 = string[80];
function LowCase(Line : UserLine) : str80;
```

Wegen der Art der Implementation der Prozeduren Write und Writeln darf eine Funktion, die irgendeine der Standardprozeduren Read, Readln, Write oder Writeln verwendet, niemals durch einen Ausdruck in einer Write oder Writeln Anweisung aufgerufen werden.

16.3.2 Standardfunktionen

Die folgenden (vordeklarierten) Standardfunktionen sind in TURBO-Pascal implementiert:

1) Stringbehandlungsfunktionen (s. 9.5),
2) Filebehandlungsfunktionen (s. 14.2, 14.5.1),
3) Pointerbezogene Funktionen)s. 15.2, 15.3, 15.5)

und darueber hinaus folgende Funktionen:

16.3.2.1 **Arithmetische Funktionen**

16.3.2.1.1 Abs

Syntax: Abs(num)

Gibt den absoluten Wert von num zurueck. Das Argument num muss entweder Real oder Integer sein und das Ergebnis ist vom gleichen Typ, wie das Argument.

16.3.2.1.2 Arctan
Syntax: Arctan(num)

Gibt den Winkel im Bogenmass zurueck, dessen Tangens gleich num ist. Das Argument num muss entweder Real oder Integer sein, das Ergebnis ist Real.

16.3.2.1.3 Cos
Syntax: Cos(num)

Gibt den Cosinus von num zurueck. Das Argument num wird im Bogenmass ausgedrueckt und muss entweder Real oder Integer sein. Das Ergebnis ist Real.

16.3.2.1.4 Exp
Syntax: Exp(num)

Gibt die Exponentialfunktion von num zurueck, d.h. e^x. Das Argument num muss entweder Real oder Integer sein. Das Ergebnis ist Real.

16.3.2.1.5 Frac
Syntax: Frac(num)

Gibt den gebrochenen Teil von num zurueck, d.h. Frac(num) = num - Int(num). Das Argument num muss entweder Real oder Integer sein. Das Ergebnis ist Real.

16.3.2.1.6 Int
Syntax: Int(num)

Gibt den ganzen Teil von num zurueck, d.h. die groesste ganze Zahl, die kleiner oder gleich num ist, fall num >= 0 ist, oder die kleinste ganze Zahl, die groesser oder gleich num ist, falls num < 0 ist . Das Argument num muss entweder Real oder Integer sein. Das Ergebnis ist Real.

16.3.2.1.7 Ln
Syntax: Ln(num)

Gibt den natuerlichen Logarithmus von num zurueck. Das Argument num muss entweder Real oder Integer sein. Das Ergebnis ist Real.

16.3.2.1.8 Sin
Syntax: Sin(num)

Gibt den Sinus von num zurueck. Das Argument num muss im Bogenmass ausgedrueckt sein und sein Typ ist entweder Real oder Integer. Das Ergebnis ist Reql.

16.3.2.1.9 Sqr

Syntax: Sqr(num)

Gibt das Quadrat von num zurueck, d.h. num*num. Das Argument num muss entweder Real oder Integer sein. Das Ergebnis ist vom gleichen Typ wie das Argument.

16.3.2.1.10 Sqrt
Syntax: Sqrt(num)

Gibt die Quadratwurzel von num zurueck. Das Argument num muss entweder Real oder Integer sein. Das Ergebnis ist Real.

16.3.2.2 **Skalarfunktionen**

16.3.2.2.1 Pred
Syntax: Pred(num)

Gibt den Vorgaenger von num zurueck (falls er existiert). num ist ein beliebiger Skalartyp.

16.3.2.2.2 Suc
Syntax: Suc(num)

Gibt den Nacgfolger von num zurueck (falls er existiert). num ist ein beliebiger Skalartyp.

16.3.2.2.3 Odd
Syntax: Odd(num)

Gibt den Booleschen Wert True zurueck, wenn num eine ungerade Zahl ist und False, wenn num eine gerade Zahl ist. num muss vom Typ Integer sein.

16.3.2.3 **Konvertierungsfunktionen**

Die Konvertierungsfunktionen werden verwendet, um den Wert eines Skalartyps in den eines anderen Skalartyps umzuwandeln. Zusaetzlich zu den folgenden Funktionen kann man "Retyping" (s.8.3) fuer diese Zwecke verwenden.

16.3.2.3.1 Ord
Syntax: Ord(num)

Gibt die Ordnungszahl des Wertes von num in der durch den Typ von num definierten Menge zurueck. Ord(num) ist mit Integer(num) equivalent. num kann ein beliebiger Skalartyp sein. Das Ergebnis ist vom Typ Integer.,

16.3.2.3.2 Round
Syntax: Round(num)

Gibt den Wert von num gerundet zur naechsten ganzen Zahl wie folgt zurueck:
wenn num >= 0, dann ist Round(num) = Trunc(num+0.5),
wenn num < 0 , dann ist Round(num) = Trunc(num-0.5).
num muss vom Typ Real sein. Das Ergebnis ist vom Typ Integer.

16.3.2.3.3 Trunc
Syntax: Trunc(num)

Gibt fuer num >= 0 die groesste ganze Zahl zurueck, die kleiner oder gleich num ist. Wenn num < 0 ist, dann gibt diese Funktion die kleinste ganze Zahl zurueck, die groesser oder gleich num ist. num muss vom Typ Real sein und das Ergebnis ist vom Typ Integer.

16.3.2.4 **Verschiedene Standarfunktionen**

16.3.2.4.1 Hi
Syntax: Hi(I)

Das niederwertige Byte des Ergebnisses enthaelt das hoeherwertige Byte des Wertes vom Integerausdruckes I. Das hoeherwertige Byte des Ergebnisses ist Null. Das Ergebnis ist vom Typ Integer.

16.3.2.4.2 KeyPressed
Syntax: Keypressed

Gibt den Wert True zurueck, wenn eine Taste auf der Console gedrueckt wurde. Das Ergebnis wird durch Aufruf der Consol-Status-Routine des BIOS realisiert.

16.3.2.4.3 Lo
Syntax: Lo(I)

Gibt das niederwertige Byte des Wertes vom Integerausdruck I zurueck, wobei das hoeherwertige Byte auf Null gesetzt wird. Der Typ des Ergebnisses ist Integer.

16.3.2.4.4 Random
Syntax: Random

Gibt eine Zufallszahl zurueck, die groesser oder gleich Null und kleiner als Eins ist. Der Typ ist Real.

16.3.2.4.5 Random(I)
Syntax: Random(num)

Gibt eine Zufallszahl zurueck, die groesser oder gleich Null und kleiner als num ist. num und die Zufallszahl sind beide vom Typ Integer.

16.3.2.4.6 SizeOf
Syntax: SizeOf(name)

Gibt die Anzahl von Bytes zurueck, die von der Variablen oder dem Typ name belegt werden. Das Ergebnis ist vom Typ Integer.

16.3.2.4.7 Swap
Syntax: Swap(I)

Die Swapfunktion vertauscht vom Wert des Integerausdruckes I das hoeher- und niederwertige Byte und gibt das Ergebnis als Integerzahl aus. Beispiel Swap($1234) gibt $3412 zurueck (Werte zur Verdeutlichung in hexadezimaler Schreibweise).

16.4 **Vorwaerts Referenz**

Ein Unterprogramm wird vorwaerts deklariert, indem man seinen Kopf separat von seinem Block spezifiziert. Dieser separate Unterprogrammkopf ist exakt der gleiche, wie der eines normalen Unterprogrammes, ausser dass er mit dem reservierten Wort **forward** endet. Der Block selbst folgt spaeter innerhalb des gleichen Deklarationsteiles. Der Block beginnt mit einer Kopie des vorher definierten Kopfes ohne Parameter, Typen etc, d.h. nur mit dem Namen.

Beispiel:

```
program Catch22;
var
   X : Integer;
function Up(var I : Integer) : Integer; forward;
function Down(var I : Integer) : Integer;
begin
   I := I div 2; Writeln(I);
   if I <> 1 then I := Up(I);
end;
function Up;
begin
   while I mod 2 <> 0 do
   begin
      i := I*3+1; Writeln(I);
   end;
   I := Down(I);
end;
begin
   Write('Enter any integer: ');
   Readln(X);
   X := Up(X);
   Write('Ok Program stopped again.');
end.
```

Wird das Programm gestartet und eine 6 eingegeben, dann ergibt sich folgendes Bild:

```
Enter any integer : 6
3
10
5
16
8
4
2
1
Ok Program stopped again.
```

Das obige Programm ist eine kompliziertere Version des folgenden Programmes:

```
program Catch222;
var
   X : Integer;
begin
   Write('Enter any integer: ');
   Readln(X);
   while X <> 1 do
   begin
      if X mod 2 = 0 then X := X div 2
                     else X := X*3+1;
      Writeln(X);
   end;
   Write('Ok Program stopped again');
end.
```

Sie sind sicher ueberrascht, dass man bei diesem kleinen und sehr einfachen Progamm im Voraus nicht einschaetzen kann, wie lange es bei Eingabe einer beliebigen ganzen Zahl läuft.

17. Einfuegen von Programmteilen

Die Tatsache, dass der TURBO-Editor den gesamten Quelltext im Speicher bearbeitet, schraenkt die Groesse des Quelltextes ein. Diese Beschraenkung kann man durch Verwendung der I-Compiler-Direktive umgehen. Dazu teilt man den gesamten Quelltext in kleinere Einheiten. Eine Einheit, das Rahmenprogramm, bildet den Kern des Programmes und in diese werden die anderen Teile zur Zeit der Uebersetzung mittels I-Compiler-Direktive eingefuegt. Diese Include-Option gestattet ein uebersichtliches Programmieren. Es eroeffnet auch die Moeglichkeit der Verwendung einzelner Programmteile in anderen Programmen und damit kann man Bibiliotheken von Dateien schaffen, die fuer den flexiblen Aufbau verschiedener Programme zur Verfuegung stehen.

Die Syntax fuer die I-Compiler-Direktive ist

```
{$I filename}
```

wobei filename ein beliebiger erlaubter kompletter Dateiname ist. Blanks werden ignoriert und kleine Buchstaben in grosse umgewandelt. Wurde kein Typ spezifiziert, wird .PAS angehaengt. Diese Direktive muss allein in einer Zeile des Rahmenprogrammes definiert werden.
Beispiele:

```
{$I first.pas}
{$I STDPROC}
{$I COMPUTE.MOD}
```

Zur Demonstration der Include-Option nehmen wir an, in unserer Bibliothek exisitiere die Datei STUPCASE.FUN. Sie enthalte die Funktion StUpCase, die mit einem Zeichen oder String als Parameter aufgerufen wird und die den Wert des Parameters unter Umwandlung aller Kleinbuchstaben in Grossbuchstaben zurueckgibt.

Datei STUPCASE.FUN:

```
function StUpCase(St : ALlStrings) : AllStrings;
   var I : Integer;
   begin
      for I := 1 to Length(St) do
      St[I] := UpCase(St[I]);
      StUpCase := St;
   end;
```

In einem anderen Programm kann man dann diese Funktion zur Umwandlung von Klein- in Grossbuchstaben verwenden, indem man diese Datei mittels der Include-Option einfuegt, anstatt sie in das Programm einzukopieren:

```
program STUPCASE;
type
   InData     = string[80];
   AllStrings = string[255];
var
   Answer : InData;
   I : Integer;
{$I F:STUPCASE.FUN}
begin
   Writeln('Enter Name: ');
   Readln(Answer);
   Writeln(StUpCase(Answer));
end.
```

Diese Methode ist nicht nur einfacher und Speicherplatz sparender, sie ermoeglicht auch einen sicheren und einfachen Aenderungsdienst. Eine Verbesserung in einer solchen Routine wirkt dann sofort automatisch auf alle Programme, die diese Routine mittels Include einfuegen.

Man sollte auch beachten, dass TURBO-Pascal fuer die einzelnen Bestandteile des Deklarationsteiles keine feste Ordnung fordert und diese auch mehrfach auftreten koennen. Damit besteht die Moeglichkeit, bestimmte häufig verwendete Typdefinitionen in einer Bibliothek aufzunehmen und sie von dort aus in die einzelnen Programme einzufuegen.

Alle Compiler-Direktiven, ausser B und C, sind lokal zu dem File, in dem sie auftreten. Das heisst, wenn eine Compiler-Direktive in einem Include-File veraendert wird, wird sie nach Verlassen dieses Include-Files auf den urspruenglichen Wert zurueckgesetzt. Die Compiler-Direktiven B und C sind immer global. Eine Beschreibung aller Compiler-Direktiven steht im Anhang E.

Include-Files koennen nicht geschachtelt werden, d.h. ein Include-File kann immer nur von einem "Rahmenprogramm" aus aufgerufen und eingefuegt werden, niemals von einem eingefuegten Programm aus.