robotron

SOFTWARE
DOKUMENTATION

T-PASCAL-Compiler

Bedienungsanleitung und Sprachbeschreibung

fuer

TPASCAL

VEB Robotron-Buchungsmaschinenwerk
Karl-Marx-Stadt

VEB Robotron Bueromaschinenwerk "Ernst Thaelmann"
Soemmerda
1987

Die vorliegende zweite Auflage der Dokumentation "Bedienungsanleitung und Sprachbeschreibung fuer TPASCAL" unter DCP 3.2. entspricht dem Stand 31.08.87 und unterliegt nicht dem Aenderungsdienst.

Nachdruck, jegliche Vervielfaeltigung oder Auszuege daraus sind unzulaessig.

Die Dokumentation wurde durch ein Kollektiv des

VEB Robotron
Bueromaschinenwerk
"Ernst Thaelmann"
Soemmerda

erarbeitet. Dabei wurden Teile der Dokumentation PASCMP und PASCAL 880/S des VEB Robotron Bueromaschinenwerk "Ernst Thaelmann" Soemmerda sowie weitere Materialien verwendet.

Bitte senden Sie uns Ihre Hinweise, Kritiken, Wuensche oder Forderungen zur Dokumentation zu.

VEB Robotron Bueromaschinenwerk "Ernst Thaelmann"
Soemmerda
Weissenseer Str. 52
Soemmerda
5230

III-12-12 Kv 805/88

I N H A L T S V E R Z E I C H N I S

Anhaenge

1. Einfuehrung

TPASCAL ist lauffaehig unter dem Betriebssystem DCP. Es ist ein leistungsfaehiges Programmiersystem fuer 16-Bitrechner und kompatibel zu TURBO-PASCAL.

Das System besteht aus folgenden Teilen:

Systemkern mit Editor, Compiler, Laufzeit-bibliothek, Interface (TPASCAL.COM) und Fehlertextfile (TPASCAL.TXT).

Diese Dokumentation beinhaltet die Sprachbeschreibung der Programiersprache TPASCAL und die Bedienanleitung fuer Editor, Compiler und Installierungsprogramm. Sie sollte aber nicht als Lehrbuch verstanden werden.

TURBO-PASCAL ist ein eingetragenes Warenzeichen der Firma Borland International USA
MSDOS ist ein eingetragenes Warenzeichen der Firma Microsoft

2. Installation

Zum Programmentwicklungssystem TPASCAL gehoert neben dem eigentlichen Compiler (TPASC.COM) und der Fehlermeitteilungsdatei (TPASC.TXT) noch ein Installierungsprogramm (INSTP.COM) mit dessen Hilfe es moeglich ist, den Compiler verschiedenen Bildschirmen anzupassen und festzulegen von welchem Laufwerk und Pfad (falls benoetigt) die Fehlertexte gelesen werden sollen. Das Installierungsprogramm wird durch die Eingabe von INSTP gestartet. Es erscheint ein Grundmenue, aus welchem mit 'B' oder 'P' die jeweilige Funktion ausgewaehlt werden kann. Bei der Wahl B fuer Bildschirminstallierung besteht die Moeglichkeit aus 5 verschiedenen Modis einen auszuwaehlen.

- Monochromes Display (ohne Grafik)
- S/W - Display 40 * 25 (mit Grafik)
- S/W - Display 80 * 25 (mit Grafik)
- Farb - Display 40 * 25
- **Farb - Display 80 * 25** (Standart bei Auslieferung)

Bei der Wahl P fuer Pfad und Laufwerkinstallierung wird immer die letzte Installierung als Standart angeboten. Diese kann bei Bedarf einfach ueberschrieben werden (Standart bei Auslieferung A:TPASC.TXT). Die Editorkommandos lehnen sich an das Textprogramm an und werden deshalb nicht installiert.
(Vergl. 3.6.2.)

3. Systemkern

3.1. Start und Menue

Der Start erfolgt durch die Eingabe des Kommandos TPASCAL.

Danach erscheint das Titelbild

```
 ,-------------------------------------------------.
/                                                   \
| ************************************              |
| TPASCAL - Programmsystem  Version: 002            |
| Pascal-Kern 3.xx Betriebssystem:  DCPx            |
|                                                   |
| (c)  1985, 86,  87       Robotron  BWS            |
| ************************************              |
|                                                   |
| <Display Angabe>                                  |
|                                                   |
|                                                   |
|                                                   |
| Fehlertext laden (J/N) ?                          |
|                                                   |
 \-------------------------------------------------/
```

Die Frage ist mit J oder N zu beantworten. Bei J wird die Komponente TPASCAL.TXT in den Speicher geladen. Sie enthaelt den Fehlertext des Compilers. Dafuer wird aber bei auftretenden Fehlern ausser der Fehlernummer auch der Fehlertext ausgegeben.

Beispiel:

```
     Fehler5 Druecke <ESC>                bei N.
     Fehler5: ')' fehlt Druecke <ESC>     bei J.
```

Nach Beantwortung der Frage mit N erscheint das Grundmenue

```
 ,-------------------------------------------------.
/                                                   \
| Lauf - Werk : A                                   |
| Gueltiger  Pfad :\                                |
|                                                   |
| Aktivfile:                                        |
| Hauptfile:                                        |
|                                                   |
| Editor  Compiler - Optionen Test                  |
| Sichern Beenden    Dir                            |
|                                                   |
| Text :      0 Bytes                               |
| Frei : xxxxx Bytes                                |
|                                                   |
| >                                                 |
 \-------------------------------------------------/
```

Als letztes Zeichen erscheint das Promptzeichen >. Durch Eingabe eines Kommandobuchstabens wird die entsprechende Komponente des Systemkerns aufgerufen. Nach Abarbeitung dieser Komponente erscheint wieder das Promptzeichen und eine andere Funktion kann aufgerufen werden.
Wird nach dem Promptzeichen eine Taste gedrueckt, die keinem Kommandobuchstaben entspricht, dann wird der Bildschirm geloescht und das Grundmenue erscheint. Dieses Wiederaufrufen des Grundmenues sollte man grundsaetzlich durchfuehren, falls die aktuellen Werte im Menue gewuenscht werden.

Es gibt folgende Kommandobuchstaben:

L — Laufwerkzuweisung.

G — Pfadzuweisung.

A — Aktivfile laden.
Das File steht danach zum Editieren bereit.

H — Hauptfile spezifizieren.
Dieses Kommando muss gegeben werden, wenn das zu compilierende Programm Includefiles enthaelt.

E — Editieren.
Eingabe / Veraenderung des Aktivfiles.

C — Compilieren.
Compiliert wird das Hauptfile, wenn nicht angegeben, dann das Aktivfile.

T — Test.
Ausfuehren des compilierten Programmes, wenn dieses mit den Optionen T oder P (vergl. Nebenmenue) uebersetzt wurde.

D — Directory.
Anzeige der Directory im aktuellen Laufwerk sowie des freien Speicherplatzes auf der Diskette.

S — Sichern.
Schreiben des Aktivfiles auf Diskette.

O — Optionen.
Setzen von Compiler-Optionen ueber ein Nebenmenue.

B — Beenden.
Rueckkehr zum Laufzeitsystem DCP.

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

3.2. Pfad-Zuweisung

Nach Eingabe von G erscheint die Aufforderung zur Eingabe des Pfades. Es gelten dabei die bekannten Regeln des DCP.

Die Eingabe wird mit Pfadname: angefordert und mit <Name> quittiert.

Mit der Angabe von <Name> ist auch eine Bezugnahme auf vorherige Ebenen moeglich.

3.3. Laufwerkzuweisung

Nach Eingabe von L erscheint die Aufforderung zur Laufwerkzuweisung:

Neues LW: B<ENTER>

Mit B <ENTER> wird als aktuelles Laufwerk B zugewiesen. Wird nur <ENTER> gedrueckt, bleibt die alte Zuweisung erhalten.
Die Neuzuweisung eines Laufwerkes wird im Menue nicht sofort angezeigt. Dazu ist nach Erscheinen des Promptzeichens > nochmal <ENTER> notwendig.

3.4. Aktivfilezuweisung

Nach Eingabe von A erscheint die Aufforderung zur Eingabe des Namens fuer das Aktivfile. Es gelten die bekannten Regeln des DCP.
Die Eingabe wird mit

Aktivfilename:

angefordert und mit

Laden <Name>

quittiert. <Name> schliesst eine eventuelle Laufwerksangabe ein.

Damit wird das File <Name> von dem jeweiligen Laufwerk in den Arbeitsbereich geladen und kann danach mit E oder C weiterbearbeitet werden. Eine Namenserweiterung .PAS erfolgt standardmaessig,wenn kein Dateityp eingegeben wurde. Existiert das File nicht, erscheint die Ausschrift:

File neu

In diesem Falle kann danach mit E das File neu angelegt werden. Falls im Arbeitsbereich ein noch nicht gerettetes, bearbeitetes File steht, wird dies angezeigt durch:

Akt. <Name> noch sichern (J/N)?

Bei Antwort J wird das alte Aktivfile auf dem jeweiligen Laufwerk abgelegt, und das neue Aktivfile zugewiesen.
Bei Antwort N wird das im Arbeitsbereich (dem Editorpuffer) stehende alte Aktivfile durch das neue ueberschrieben.

3.5. Hauptfilezuweisung

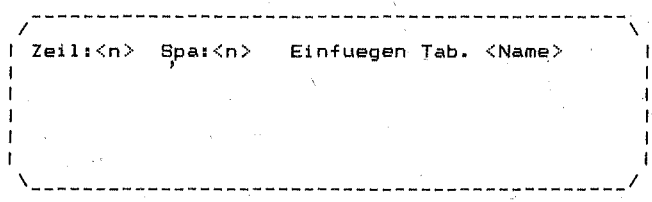
Nach Eingabe von H erscheint die Aufforderung zur Eingabe des Namens fuer das Hauptfile nach den bekannten Regeln fuer DCP. Die Namenserweiterung . PAS ist wieder Standard.
Die Benutzung eines Hauptfiles wird notwendig, wenn ein Pascal-Programm Includeanweisungen (vergl. Ziffer 4.1.4.6.) enthaelt. Der Compiler beginnt dann die Uebersetzung mit dem Hauptprogramm und, wenn er auf eine Includeanweisung trifft, laedt das durch die Includeanweisung benannte File in den Aktivfilebe-

reich. Die Includefiles werden Aktivfiles und an dieser Stelle vom Compiler uebersetzt. Treten bei der Compilierung Fehler auf, kann das fehlerhafte File wie ueblich korrigiert werden. Das Ablegen des geaenderten Files erfolgt automatisch und bei einem erneuten Compilerlauf wird auch das Hauptfile wieder geladen.

3.6. Editieren

3.6.1. Bildaufbau

Nach Eingabe von E erscheint der Text des Aktivfiles auf dem Bildschirm und der Editor wird gestartet. Der Text kann dann bearbeitet werden.
Existiert noch kein Aktivfilename, so erfolgt die Aufforderung zur Eingabe dieses Namens, danach wird das File geladen, oder, wenn es nicht gefunden wurde, die Ausschrift "File neu" ausgegeben und der Editor gestartet. Es erscheint das Bild:



Die erste Zeile bleibt als Statuszeile stets auf dem Bildschirm stehen. Es bedeuten:

Zeil <n>	n=Zeilenposition des Cursors
Spa <n>	n=Spaltenposition des Cursors
Einfuegen	Anzeige Einfuegen.Im Wechsel (CTRL V) mit Ersetzen
Tab	Zeigt automatisches Einruecken an
<Name>	Filename des gerade editierten Textes (eventuell einschliesslich Laufwerksangabe).

In den der Statuszeile folgenden Zeilen kann der Programmtext geschrieben werden. Jede Zeile ist durch <ENTER> abzuschliessen. In den untenstehenden Kommandos kann das zweite Control-Zeichen auch weggelassen werden, d.h., statt ^Q^L kann man auch ^QL druecken.
Fuer das Editieren koennen folgende Kommandos verwendet werden, die weitgehend mit denen von Textverarbeitungssystemen uebereinstimmen:

3.6.2. Cursorbewegungen

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.
Rollen 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.
Bildende	^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, z.B. nach ^QA/^QF -Kommando

Insert und Delete Kommandos:

Einfuege-Modus ein/aus	^V	Einfuegen oder ueberschreiben.
Loeschen Zeichen links	DEL	Wirkt ueber Zeilengrenzen.
Loeschen Zeichen	^G	Wirkt ueber Zeilengrenzen.
Loeschen Wort rechts	^T	Wirkt ueber Zeilengrenzen.
Zeile einfuegen	^N	Wenn der Cursor sich nicht am Zeilenanfang befindet, wird die 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 haben die Blockkommandos keine Wirkung. Es erfolgt keine Fehlermeldung.

Spezielle Kommandos:

Editieren Ende	^K^D	Rueckkehr zum PASCAL-Grundmenue. Es ist zu beachten, dass das Aktivfile noch nicht gesichert ist.
Tab	^I	Die Tabpositionen werden durch die Wortanfaenge der vorhergehenden Zeile bestimmt. Achtung! Im Einfuegemodus werden Tab eingefuegt!
Tab ein/aus	^Q^I	Bei <ENTER> springt Cursor in naechster Zeile nicht zur Spalte 1, sondern unter das erste Wort. Bei Tab ein wird Tab in Statuszeile angezeigt. Bei Tab aus ist diese Stelle leer.
Ruecksetzzeile	^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	^Q^F	Die Suchkette kann aus 30 Zeichen bestehen. Sie kann CTRL-Zeichen enthalten und wird durch <ENTER> beendet. Zeilenende kann durch ^M^J erzeugt werden. In der Suchkette kann ^A als Wildcard (Maskenzeichen) verwendet werden. Zusaetze: B Rueckwaerts suchen G Globales suchen (Anfang bis Ende) n Suchen des n. Auftretens U Ignorieren Gross/Kleinbuchstaben W Nur Worte suchen Zusaetze sind ohne Zwischenraum zu schreiben und mit <ENTER> zu beenden.

Suchen und Ersetzen	^Q^A	Suchkette und Zusaetze werden wie beim Suchen angegeben. Die Zeichenkette zum Ersetzen kann 30 Zeichen lang sein (auch CTRL-Zeichen). Abschluss durch <ENTER>. Zusaetze: B Rueckwaerts suchen G Globales suchen (von Anfang bis Ende) n n-maliges Ersetzen N Ersetzen ohne Fragen: (Ersatz (J/N)?) U Ignorieren Gross/Kleinbuchstaben W Nur Worte suchen Zusaetze ohne Zwischenraum schreiben und mit <ENTER> beeenden.
Suchen wiederholen	^L	Wiederholen des letzten ^Q^F oder ^Q^A Kommandos.
Eingabe CTRL-Zeichen	^P	Im Programmtext koennen CTRL-Zeichen durch durch Voranstellen von ^P eingegeben werden. Beispiel: ^G durch ^P^G.
Abort-Kommando	^U	Jedes Editor-Kommando kann durch ^U sofort abgebrochen werden.

3.6.3. Sonderbelegungen

Zur Vereinfachung der Editierung und zur Nutzung der Funktionstasten entsprechen folgende Belegungen den entsprechenden Ctrl-Kommandos.

^S	Linker Pfeil	Zeichen links
^D	Rechter Pfeil	Zeichen rechts
^A	Ctrl linker Pfeil	Wort links
^F	Ctrl rechter Pfeil	Wort rechts
^E	Pfeil nach oben	Zeile hoch
^X	Pfeil nach unten	Zeile tief
^R	PgUp	Blaettern hoch
^C	PgDn	Blaettern tief
^Q^S	Home	Nach erster Spalte
^Q^D	End	Nach Spalte hinter dem letzten Zeichen
^Q^E	Ctrl Home	Nach oberster Bildzeile
^Q^X	Ctrl End	Nach unterster Bildzeile
^Q^R	Ctrl PgUp	Nach erste Zeichen der Datei
^Q^C	Ctrl PgDn	Nach letzten Zeichen der Datei
^V	Ins	Einfuege-Modus ein/aus
^K^B	F7	Blockbeginn
^K^K	F8	Blockende
^I	TAB	Tabulierung

3.7. Compilieren

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

Existiert noch kein Aktivfile, erfolgt die Aufforderung zur Eingabe des Aktivfilenamens.
Wird das Aktivfile nicht auf der angegebenen Diskette gefunden, erscheint die Mitteilung "File neu" und es wird versucht, das leere File zu uebersetzen. Das Ergebnis ist die Meldung "Fehler91: Vorzeitiges Ende des Quell-Files Druecke <ESC>". Nach dem Druecken der Taste ESC wird der Editor aufgerufen und das neue File kann erstellt werden. Soll dies nicht geschehen, kann der Editorlauf durch ^KD beendet werden. Damit wird das Grundmenue erreicht. Das leere File wird nicht gerettet.

Die Compilierung und die Form des Objektcodes ist abhaengig von verschiedenen Compiler-Optionen. Es sind Standardwerte festgelegt, die eine zeitguenstige Uebersetzung, minimalen Objektcode und schnelle Abarbeitung des uebersetzten Programmes ergeben. Fuer bestimmte Faelle koennen diese Standardwerte veraendert werden (vergl. Ziffer 3.12.). Bei Auftreten eines Fehlers im Quelltext wird die Compilierung abgebrochen in der Form:

```
 ------------------------------------------------
/                                                \
| >C                                             |
|                                                |
|   Komp.                                        |
|   10 Zeilen                                    |
| Fehler5: ')' fehlt Druecke<ESC>                |
 \------------------------------------------------/
```

wobei der Fehlertext nur dann ausgeschrieben wird, wenn TPASCAL.TXT geladen wurde (Eingabe J bei "Fehlertext laden (J/N)?" nach dem Starten des Systemkerns). Nach Druecken der Taste ESC wird der Editor gestartet und der Cursor hinter den Fehler positioniert. Es kann sofort editiert und danach erneut compiliert werden. Bei fehlerfreier Uebersetzung erscheint z.B. folgendes Bild:

```
 ------------------------------------------------------------
/                                                            \
|   Komp.                                                    |
|   12 Zeilen                                                |
| Code:       xxxx Paragrafen ( 10 Bytes),zzzz Paragrafen frei|
| Daten:      xxxx Paragrafen ( 12 Bytes),zzzz Paragrafen frei|
| Stack/Heap: xxxx Paragrafen (999 Bytes)                    |
|                                                            |
|                                                            |
| >                                                          |
 \------------------------------------------------------------/
```

*** Compilieren ***

Es werden die Anzahl der uebersetzten Quelltextzeilen, die Anzahl der Bytes fuer den Programmcode, den Datenbereich und den jeweils noch verbleibenden freien Speicherbereich angeben. Je nach festgelegter Option (Kommando O) wird das erzeugte File gespeichert

```
    -im  Test          (Hauptspeicher) (Standard)
    -als *.COM-File    (Programm)
    -als *.CHN-File    (Modul)
(vergl. Ziffer 3.12.)
```

3.8. Testen

Nach Eingabe von T wird das uebersetzte Programm aktiviert und gestartet(nur bei Compiler-Option T). Wurde kein uebersetztes Programm gefunden, so uebersetzt automatisch der Compiler das Aktivfile und startet nach erfolgreicher Uebersetzung sofort auch das Programm.
Wurde noch kein Aktivfile angegeben, so wird zur Eingabe des Aktivfilenamens aufgefordert. Existiert auch das Aktivfile selbst noch nicht auf dem Datentraeger, so wird wie bei Kommando C der Editor gestartet.
Programme die mit der Option C oder H uebersetzt wurden, koennen nur auf der Betriebssystemebene getestet werden.

3.9. Sichern

Nach Eingabe von S wird das Aktivfile ueber das entsprechende Laufwerk auf Diskette geschrieben.
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.
Vor Operationen, die den Editorpuffer zerstoeren (Kommando A,B) wird fuer geaenderte Pufferinhalte das Sichern des Quelltextfiles ueberwacht und gegebenenfalls angeboten.

3.10. Directory-Anzeige

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

```
>D
Maske: A:*.PAS
Inhalt von C:\BEI\*.PAS
HANDEL   PAS : PROBE    PAS : PRO1   PAS : PRO2   PAS
PRO4     PAS : DRUCK    PAS

25K Bytes frei
>
```

Zum Abschluss wird der auf der Diskette noch freie Speicherbereich ausgegeben (hier 25K).

3.11. Beenden

Das Kommando B dient zum Verlassen des PASCAL-Systems und zur Rueckkehr zum DCP. Es erscheint das Prompt X> . Falls noch ein editiertes geladenes Aktivfile existiert, das nicht gesichert wurde, wird gefragt, ob das geschehen soll.

3.12. Compiler-Optionen

Mit diesem Kommando wird die Form des Ausgabefiles festgelegt oder kann nach einem bei der Abarbeitung eines COM-Files aufgetretenen Fehler dessen Stelle gefunden werden.

Nach der Eingabe des Kommandos O erscheint:

```
>O

  Code fuer -> Test
               *.COM
               *.CHN

  Befehlszeilen-Parameter:

  Finde Laufzeitfehler Zur.

  >
```

Die Stelle des Pfeiles zeigt das aktuelle Ziel des Compilates. Es kann durch Eingabe von T,C oder H veraendert werden. Mit F wird der Suchprozess zur Auffindung eines Fehlers im Quellprogramm gestartet und mit Z kann zum Grundmenue zurueckgekehrt werden.

Die Moeglichkeiten im einzelnen:

T: (Standard) Der Code wird im Hauptspeicher erzeugt und das Programm kann durch Test gestartet werden.

*** Compiler-Optionen ***

Unter dem Namen des Aktiv- bzw. Hauptfiles wird ein lauffaehiges Programm (COM-File) erzeugt, das den Code und die PASCAL-Bibliothek enthaelt. Gestartet wird das COM-File durch normalen DCP-Aufruf.
Diese Programme koennen groesser als bei der Option T sein, da ihre Anfangsadresse im Speicher niedriger liegt und bei der Compilierung kein Platz fuer die Speicherung des Programmes benoetigt wird. Nichtgeladener Fehlertext vergroessert auch hier den Arbeitsbereich des Compilers. Bei der Wahl der COM-Optionen erscheinen weitere vier Zeilen auf dem Bildschirm.

```
minimum cOde segment size:    XXXX paragraphs (max.YYYY)
minimum Data segment size:    XXXX paragraphs (max.YYYY)
mInimum free dynamic memory: XXXX paragraphs
mAximum free dynamic memory: XXXX paragraphs
```

- Minimale Codesegmentgroesse

Fuer .COM Dateien, die Chain oder Execute benutzen, wird mit dem O-Befehl die minimale Groesse des CodeSegments festgelegt. Dabei muss man beachten, dass die minimal festzulegende Groesse sich nach dem laengsten Segment in einer durch Chain oder Execute aufgefuehrten Programme richtet. Bei der Compilierung muss man die minimale Code-Segmentgroesse des Hauptprogramms auf den gleichen Wert setzen, den das groesste Segment der zu verkettenden Chain-Programme besitzt. Durch die Statusanzeige, die am Ende jeder Compilierung angezeigt wird, erhaelt man die erforderlichen Werte. Die ausgegebenen Werte stellen die Anzahl der Paragraphen hexadezimal dar (ein Paragraph belegt 16 Byte).

- Minimale Datensegmentgroesse

Fuer .COM Dateien, die Chain oder Execute benutzen, wird mit dem D-Befehl die minimale Groesse des DatenSegments festgelegt. Bei der Festlegung gelten die gleichen Regeln wie oben.

- Minimaler freier dynamischer Speicher

Die fuer den Stack und Heap benoetigte minimale Speichergroesse wird durch diesen Wert angegeben. Der angegebene Wert stellt die Anzahl der Paragraphen hexadezimal dar (ein Paragraph belegt 16 Byte).

- Maximaler freier dynamischer Speicher

Der fuer den Stack und Heab maximal moegliche Speicher wird durch diesen Wert angegeben. Bei Programmen, die in einer Mehrplatzumgebung laufen, muss dieser Wert verwendet werden, um sicherzustellen, dass das Programm nicht den ganzen Raumspeicher belegt. Der angegebene Wert stellt die Anzahl der Paragraphen hexadezimal dar (1 Paragraph belegt 16 Byte).

H: Unter dem Namen des Aktiv- bzw. Hauptfiles wird ein aufrufbarer Modul (CHN-File) erzeugt. Er enthaelt den Programmcode, aber nicht die PASCAL-Bibliothek. Diese Programme koennen nur von einem anderen Pascalprogramm, das mit der COM-Option uebersetzt wurde, durch die CHAIN-Prozedur aufgerufen werden. Damit besteht also die Moeglichkeit, auch sehr grosse Pascal-Programmpakete zu erzeugen.

P: Mit dem Kommando P ist es moeglich, einen oder mehrere Parameter an das Programm zu uebergeben, wenn es im Testmodus abgearbeitet werden soll. Die hier uebergebenen Parameter entsprechen den Parametern, die in der DCP-Kommandozeile uebergeben werden koennen. Man hat dadurch die Moeglichkeit, diese Parameter im Programm mit den Funktionen ParamCount und ParamStr auszuwerten.

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

```
Laufzeit Fehler <nn>, PC = <Adresse>
Programmabbruch
```

oder

```
E/A-Fehler <nn>, PC = <Adresse>
Programmabbruch
```

Dabei ist <nn> die Fehlernummer und die Hexadezimalzahl <Adresse> gibt die Fehlerstelle im Code an.

Allgemeine Laufzeitfehler (Vergl. Anhang C.)

(•nn in der Meldung: Laufzeit Fehler nn, PC = <Adresse>)

nn	Bedeutung
01	Gleitkommaueberlauf
02	Division durch Null
03	Fehler im Argument von SQRT (<Null)
04	Fehler im Argument von LN (<=Null)
10	Falsche STRING-Laenge (auch in Ergibtanweisungen)
11	Fehlerhafter STRING-Index (ausserhalb 1 - 255)
90	Index ausserhalb des zulaessigen Bereiches
91	Ordinaler Typ ausserhalb des Wertebereiches (auch bei Teilbereichstypen)
92	Wert ausserhalb des INTEGER-Bereiches
FF	Halden/Kellerspeicher-Kollision

Ein/Ausgabe - Laufzeitfehler (Vergl. Anhang D.)

(nn in der Meldung: EA-Fehler nn, PC = <Adresse>)

nn	Bedeutung
01	File existiert nicht
02	File fuer Leseoperationen nicht vorbereitet
03	File fuer Schreiboperationen nicht vorbereitet
04	File nicht geoeffnet
10	Fehler im numerischen Format
20	Operation auf logischem Geraet nicht erlaubt
21	Im Direktmodus (Zielauswahl T) nicht erlaubt
22	ASSIGN fuer vordefinierte Filevariablen nicht erlaubt
90	Recordlaenge nicht vertraeglich
91	Position ausserhalb des Files,
99	Vorzeitiges Fileende
F0	Disketten-Schreibfehler
F1	Directory voll
F2	File zu gross
FF	File nicht unter Kontrolle

Um diese Stelle im Quellprogramm zu finden, ist dieses mit A oder H zu laden. Dann muss das F-Kommando gegeben werden. Es wird die Eingabe der Fehleradresse gefordert:

```
Eing.PC:1B56 <ENTER>
```

Die Eingabe der Fehleradresse (hier 1B56) fuehrt zum Suchprozess im Quellprogramm. Wurde die Fehlerstelle gefunden, wird dies mitgeteilt und zum Druecken der Taste ESC-> aufgefordert. Danach erscheint das Quellprogramm und der Cursor steht hinter dem Sprachelement, das den Fehler verursachte. Wird bei der Compilation eine Ueberlagerungsstruktur erzeugt (Overlay), so kann ein Fehler nicht eindeutig lokalisiert werden. Die Adresse wird in dem Programmteil gesucht, der in dem Ueberlagerungsbereich als erstes compiliert wurde.

4. Sprachbeschreibung

4.1. Grundelemente

4.1.1. Beschreibungsform

Die Beschreibung der Sprache erfolgt in der erweiterten BACKUS-NAUR-Form.

Folgende Symbole werden zur Beschreibung verwendet:

Zeichen/Zeichenfolgen ohne die Klammerung <>, ausser ::= und |	= Terminalsymbole; die Zeichen sind unveraendert in den PASCAL-Quelltext zu uebernehmen.
<Zeichenfolge>	= Nichtterminalsymbole; an anderer Stelle definiert; durch eine gueltige Konstruktion zu ersetzen.
|	= Alternative; ein Element muss gewaehlt werden.
{}	= Moegliche Wiederholung einer Konstruktion - kann auch weggelassen werden.
::=	= "..ist definiert durch.."
<leer>	= Leere Symbolfolge.

4.1.2. Grundsymbole

Das Grundvokabular 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
                     A|B|C|D|E|F      - (nur im Hex- Format)

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

4.1.3. Morpheme

4.1.3.1. Wortsymbole

Folgende Worte sind 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	OVERLAY	PACKED	PROCEDURE
PROGRAM	RECORD	REPEAT	SET	*SHL	*SHR	*STRING
THEN	TO	TYPE	UNT	VAR	WHILE	WITH
*XOR						

4.1.3.2. Standardbezeichner

TPASCAL verwendet eine Anzahl von Standardbezeichnern als Namen fuer Konstanten, Typen, Variablen, Prozeduren und Funktionen. Diese Standardbezeichner duerfen nicht umdefiniert werden. Als Standardbezeichner werden verwendet:

ABS	ADDR	APPEND	ARCTAN	ASSIGN
AUX	AUXINPTR	AUXOUTPTR	BLACK	BLINK
BLOCKREAD	BLOCKWRITE	BLUE	BOOLEAN	BROWN
BUFLEN	BW40	BW80	BYTE	C40
C80	CHAIN	CHAR	CHDIR	CHR
CLOSE	CLREOL	CLRSCR	CON	CONCAT
CONINPTR	CONOUTPTR	CONSTPTR	COPY	COS
CSEG	CYAN	DARKGRAY	DELAY	DELETE
DELLINE	DISPOSE	DSEG	EOF	EOLN
ERASE	EXECUTE	EXIT	EXP	FALSE
FILEPOS	FILESIZE	FILLCHAR	FLUSH	FRAC
FREEMEM	GETDIR	GETMEM	GOTOXY	GREEN
HALT	HEAPPTR	HI	INPUT	INSERT
INSLINE	INT	INTEGER	INTR	IORESULT
KBD	KEYPRESSED	LENGTH	LIGHTBLUE	LIGHTCYAN
LIGHTGRAY	LIGHTGREEN	LIGHTMAGENTA	LIGHTRED	LN
LO	LONGFILEPOS	LONGFILESIZE	LONGSEEK	LST
LSTOUTPTR	MAGENTA	MARK	MAXAVAIL	MAXINT
MEMAVAIL	MEMW	MKDIR	MOVE	MSDOS
NEW	NOSOUND	ODD	OFS	ORD
OUTPUT	OVRPATH	PARAMCOUNT	PARAMSTR	PI
PORT	PORTW	POS	PRED	PTR
RANDOM	RANDOMSIZE	READ	READLN	REAL
RECURPTR	RED	RELEASE	RENAME	RESET
REWRITE	RMDIR	ROUND	SEEK	SEEKEOF
SEEKEOLN	SEG	SIN	SIZEOF	SOUND
SQR	SQRT	SSEG	STACKPTR	STR
SUCC	SWAP	TEXT	TEXTCOLOR	TEXTBACKGROUND
TEXTMODE	TRUC	TRUE	TRUNC	UPCASE
USR	VAL	WHEREX	WHEREY	WHITE
WINDOW	WRITE	WRITELN	YELLOW	

4.1.3.3. Spezialsymbole

Folgende Spezialsymbole gelten:

Indexklammern:	[]
Ausdrucks- und Funktions-/Prozedur-klammern:	()
Zeigermarkierung	^
Kommentar- und Direktivklammern:	{}

Operatoren ohne Wortsymbole:

Arithmetische Operatoren:	* | - | + | /
Zuweisungsoperator:	:=
Vergleichsoperatoren:	<> | <= | >= | < | > | =
Teilbereichsbegrenzer:	..

Als Transkriptoren sind erlaubt:

Klammern:	(. .) gleichbedeutend mit [ ]
Kommentar:	(* *) gleichbedeutend mit { }

4.1.3.4. Begrenzer

Sprachelemente muessen durch wenigstens einen der folgenden Begrenzer getrennt werden:

<space>,
Zeilenende,
Kommentar.

4.1.3.5. Zeilenlaenge

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

4.1.4. Nutzerdefinierte Sprachelemente

4.1.4.1. Bezeichner

Bezeichner werden zur Bezeichnung von Marken, Konstanten, Typen, Variablen, Prozeduren und Funktionen verwendet. Ihre Zuordnung muss in ihrem Gueltigkeitsbereich (z.B. innerhalb einer Prozedur) eindeutig sein.

*** Bezeichner ***

Syntax:

```
<Bezeichner>      ::= <Buchstabe>
                      |<Buchstabe>{<weitere Zeichen>}
<weitere Zeichen>::= <Buchstabe>
                      |<Ziffer>
                      |<Unterstreichungszeichen>
```

Ein Bezeichner besteht also aus einem Buchstaben, dem Buchstaben, Ziffern und Unterstreichungsstrich folgen koennen. Die Laenge ist maximal 127 Zeichen, und alle Zeichen sind signifikant. Dadurch sind Programme moeglich, die sich in hohem Masse selbst dokumentieren.

Beispiel:

```
     Pascal
     Preis
     Art_Nummer
     3teWurzel          falsch! Ziffer am Anfang.
     zwei Worte         falsch! Leerzeichen unerlaubt.
```

Zwischen grossen und kleinen Buchstaben gibt es keinen Unterschied. So sind

```
          ArtikelNummer = ARTIKELNUMMER
```

identisch. Der linke Bezeichner ist leichter lesbar. Wortsymbole duerfen, Standardbezeichner sollten nicht als nutzerdefinierte Bezeichner verwendet werden.
Der Unterstreichungsstrich(_) ist in Bezeichnern zulaessig, wird aber vom Compiler ignoriert(z.B. Art_Nummer entspricht ArtNummer). Bezeichner koennen prinzipiell mit "@" beginnen.
Ein Bezeichner gilt immer als definiert im Block des jeweiligen Deklarationsteils. Im Hauptprogramm definierte Bezeichner sind immer global.

4.1.4.2. Zahlen

Zahlen sind Konstanten der Typen INTEGER, BYTE oder REAL. Integerzahlen sind ganze Zahlen, die dezimal und hexadezimal dargestellt werden koennen. Hexadezimale Integerzahlen werden durch vorangestellte $-Zeichen erklaert.
Integerzahlen haben einen Bereich von -32768 .. +32767.
Hexadezimalzahlen haben einen Bereich von $0000 .. $FFFF.
BYTE ist als Teilbereich 0..255 von INTEGER aufzufassen. Der Bereich der Realzahlen betraegt 1E-38 .. 1E+38 mit 11 signifikanten 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. Fuer Zahlen wird die uebliche Dezimaldarstellung genutzt. Unmittelbar vor einer Dezimalzahl darf ein Vorzeichen stehen.

Syntax:

```
<vzl. Zahl>:::=
        <natuerliche Zahl>
        | <vzl. Gleitkommazahl>
<vzl. Gleitkommazahl>:::=
        <natuerliche Zahl> . <Ziffernfolge>
        | <natuerliche Zahl> . <Ziffernfolge> E <Exponent>
        | <natuerliche Zahl> E <Exponent>
<Exponent> :::=
        <natuerliche Zahl>
        | <Vorzeichen><natuerliche Zahl>
<natuerliche Zahl>:::=
        <Ziffernfolge>
<Ziffernfolge>:::=
        <Ziffer> {<Ziffer>}
<Vorzeichen>:::= + | -
```

Der den Exponenten einleitende Buchstabe E steht fuer "10 hoch". Zahlen mit Dezimalpunkt haben vor dem Punkt mindestens eine Ziffer.

Beispiel:

```
     5
     62.12E+8
     0.691   (nicht .691)
     &3A
     &12G     Falsch! G keine Hexadezimalzahl.
     &12.3    Falsch! Punkt keine Hexadezimalzahl.
     -1.2345678901E+12
     1        erlaubt, ist aber eine Integerkonstante.
```

4.1.4.3. Zeichenketten

Zeichenketten sind Folgen von Zeichen, welche in Apostrophe eingeschlossen sind. Zeichenketten sind Daten vom Typ CHAR bzw. STRING.
Syntax:

```
<Zeichenkette>:::=
        ' <Zeichen> {<Zeichen>}
        | ''
<Zeichen>:::=
        <Buchstabe>
        | <Ziffer>
        | <Sonderzeichen>
```

Sollen innerhalb von Zeichenketten Apostrophe verwendet werden, dann ist das Apostroph zweimal zu schreiben.

Beispiel:

```
     'Zeichenkette
     'Mach''s moeglich !'
     '63'
     ''                                        {= leere Zeichenkette}
```

Eine Zeichenkette ist kompatibel mit einem ARRAY OF CHAR gleicher Laenge und mit allen String-Typen gleicher oder groesserer Laenge.

4.1.4.4. CTRL-Steuerzeichen

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

1) als #-Symbol, gefolgt von einer dezimalen oder hexadezimalen Zahl. Damit wird ein Zeichen mit dem entsprechenden Wert des Zeichensatzes definiert.
2) als ^-Symbol, gefolgt von einem Zeichen des Zeichensatzes. Damit wird das entsprechende CTRL-Zeichen definiert.

Beispiel:

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

Folgen von Steuerzeichen koennen ohne Begrenzer aneinandergereiht werden:

Beispiel:

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

Steuerzeichen koennen auch mit anderen Zeichenketten gemischt auftreten:

Beispiel:

```
'Fehler! ',^G^G^G,'Bitte, korrigieren! '
```

4.1.4.5. Kommentare

Kommentare dienen der Erlaeuterung von Anweisungen oder Programmteilen. Kommentare sind nur Bestandteil des Quellprogramms, d.h., sie belegen keinen Speicherplatz zur Programmlaufzeit.
Kommentare koennen an jeder beliebigen Stelle im Programm stehen. Sie werden in {...} bzw.(*...*) eingeschlossen und koennen Buchstaben, Ziffern und Sonderzeichen enthalten (ausser }).
Sie sollten benutzt werden zur Kennzeichnung von Programmteilen oder zur Erlaeuterung spezieller, nicht sofort interpretierbarer Befehle.

Beispiel:

```
Zwischenspeicher := CONOUTPTR;
CONOUTPTR :=LSTOUTPTR;  {Kanalumschaltung Bildschirm -->
                                                  Drucker}
writeln('Diese Ausgabe erfolgt ueber Drucker');
CONOUTPTR := Zwischenspeicher;       {Rueckschaltung}
```

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 in Kommentarklammern einschliessen und damit bei der Uebersetzung unberuecksichtigt lassen.

4.1.4.6. Compiler-Direktiven

Die Arbeitsweise des Compilers kann durch Direktiven gesteuert werden. Sie werden in den Quelltext als Kommentare mit einer speziellen Syntax eingefuegt. Die Direktiven koennen ueberall dort im Text stehen, wo Kommentare stehen koennen. Eine Compiler-Direktive besteht aus einer oeffnenden Kommentarklammer der unmittelbar ein $-Zeichen und dann die eigentliche Direktive oder eine Liste solcher Direktiven folgt, die durch Komma untereinander getrennt sind.

Beispiel:

```
{$I-}
{$I INCLUDE.PAS}
{$R-,B+,V-}
(*$S-*)
```

Achtung! Vor oder nach dem Zeichen $ ist kein Leerzeichen erlaubt. Ein + Zeichen indiziert eine Aktivierung der Compiler-Direktive und ein - Zeichen indiziert eine Deaktivierung der Compiler-Direktive.

Alle Compiler-Direktiven haben Standardwerte. Diese wurden so ausgewaehlt, dass die Ausfuehrungszeit der Programme schnell und die Programmgroesse minimal ist. Dies bedeutet beispielsweise, dass die Codeerzeugung fuer rekursive Prozeduren und Ueberpruefung der Indexbereiche standardmaessig abgeschaltet ist. Man sollte deshalb genau pruefen, ob in den Programmen die benoetigten Compiler-Direktiven richtig gesetzt wurden.

4.1.4.6.1. INCLUDE-Direktive

{$I <Filename>}

Das mit <Filename> bezeichnete File wird geladen und bearbeitet. Existiert es nicht, entsteht ein Compilerfehler. Include-Files duerfen nicht selbst wieder Include-Direktiven enthalten. Eine Include-Schachtelung ist also nicht erlaubt.

4.1.4.6.2. B-Compiler-Direktive

Standard: B+

Die B-Direktive steuert den Eingabe-/Ausgabe-Auswahlmodus. Wenn aktiv {$B+}, wird das CON:Geraet den Standard-Files Input und Output zugewiesen, d.h. dem Standard Input-Output-Kanal. Wenn passiv {$B-}, wird das TRM:Geraet zugewiesen. Diese Direktive ist global zum gesamten Block und kann im Programm nicht umdefiniert werden.

4.1.4.6.3. C-Compiler-Direktive

Standard: C+

Die C-Direktive steuert die Interpretation der Steuerzeichen bei Consol-I/O.

4.1.4.6.4. I-Compiler-Direktiven

Standard: I+

Die I-Direktive steuert die Behandlung der I/O-Fehler. Wenn aktiv {$I+}, werden alle I/O-Operationen auf Fehler ueberprueft. Wenn passiv {$I-}, ist es notwendig, dass der Programmierer selbst die I/O-Fehler mit der Standardfunktion IORESULT prueft. Folgt der I-Direktive ein Filename, so erkennt der Compiler auf INCLUDE-Direktive.

4.1.4.6.5. R-Compiler-Direktive

Standard: R-

Die R-Direktive steuert die Indexpruefung zur Laufzeit des Programms. Wenn aktiv {$R+}, werden alle Indexoperationen von ARRAYS geprueft, ob die Indizes in den definierten Grenzen liegen. Alle zugewiesenen Skalar- und Teilbereichs-Variablen werden geprueft, ob sie in den entsprechenden Bereichen liegen. Wenn passiv {$R-}, werden keine Pruefungen durchgefuehrt. Dann koennen Indexfehler zu falschen Programmmablaeufen fuehren. Waehrend der Programmentwicklung sollte man stets diese Direktive aktivieren. Nach Beseitigung aller Fehler kann man dann diese Direktive deaktivieren, um das Programmm schneller zu machen.

4.1.4.6.6. U-Compiler-Direktive

Standard: U-

Die U-Direktive steuert Nutzer-Unterbrechungen. Wenn aktiv {$U+}, kann der Nutzer zu jeder Zeit das Programm durch ^C unterbrechen. Wenn passiv {$U-}, hat ^C keine Wirkung. Bei

Aktivierung wird die Ausfuehrungszeit etwas verlangsamt. Es empfiehlt sich, waehrend der Programmentwicklung stets auch diese Direktive zu aktivieren.

4.1.4.6.7. V-Compiler-Direktive

Standard: V+

Die V-Direktive steuert die Type-Pruefung bei STRING-Variablen-Parametern. Wenn aktiv (V+), wird eine genaue Typ-Pruefung durchgefuehrt, d.h., die Laenge der aktuellen und formalen Parameter muss uebereinstimmen. Wenn passiv (V-), koennen bei aktuellen und formalen STRING-Parametern die Laengen abweichen.

4.1.4.6.8. G-Compiler-Direktive

Standart: G0

Mit der G-Direktive wird die Puffergroesse fuer Eingabedatei festgelegt. G0 legt CON: oder TRM: als Eingabedatei fest.

4.1.4.6.9. P-Compiler-Direktive

Standart: P0

Mit der P-Direktive wird die Puffergroesse fuer Ausgabedatei festgelegt. P0 legt CON: oder TRM: als Eingabedatei fest.

4.1.4.6.10. D-Compiler-Direktive

Standart: D+

Wird mit P oder G ein Puffer ungleich Null festgelegt, so kann mit der D-Direktive das Ueberpruefen von Standardgeraeten ausgeschaltet werden. Dieses ist notwendig, da Pascal bei Standardgeraeten die Pufferung ausschaltet.

4.1.4.6.11. K-Compiler-Direktive

Standart: K+

Die K-Direktive ueberprueft ob bei Unterprogrammaufruf genuegend Speicherplatz fuer lokale Variablen auf dem Stack vorhanden ist.

4.1.4.6.12. F-Compiler-Direktive

Standart: F16

Bei TPASCAL ist die Arbeit mit max. 16 Dateien voreingestellt. Man hat die Moeglichkeit, mit der Compilerdirektive F diesen

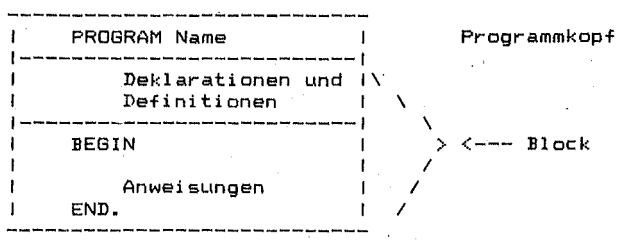
Wert zu aendern. Moechte man z.B. mit 20 Dateien gleichzeitig arbeiten, so muss man {$F20} an den Anfang des Programms (vor den Deklarationsteil) setzen.
Achtung: Bei Verwendung des Compilerbefehls F koennen Fehler auftreten, die auf zu viele Dateien verweisen. Wenn so ein Fehler auftritt sollte man den Wert fuer "Files = xx" in der Datei CONFIG.SYS aendern.

4.2. Programmstruktur/Programmrahmen

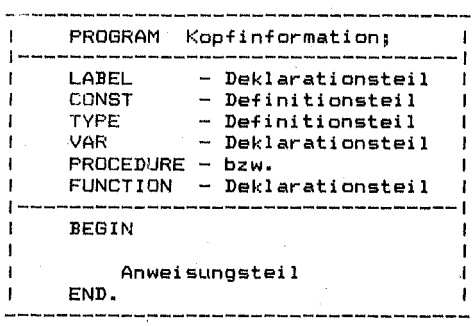
Die Programmsprache PASCAL ermoeglicht die modulare Programmierung. Aus diesem Grund sind zusammengehoerige Programmschritte zu Bloecken, Prozeduren oder Funktionen zusammengefasst.
Ein PASCAL-Programm besteht aus dem
-Programmkopf, dem der
-Programmblock folgt.
Der Programmblock selbst besteht aus dem
-Deklarationsteil und dem
-Anweisungsteil.

Die PASCAL-Syntax verlangt, dass alle Deklarationen bzw. Definitionen am Programmanfang stehen muessen. Im Deklarationsteil werden alle lokalen Objekte des Programms definiert, und im Anweisungsteil stehen alle Aktionen, die mit diesen Objekten ausgefuehrt werden sollen.

Grundaufbau eines PASCAL-Programms:



Im einzelnen besitzt ein PASCAL-Programm folgende Struktur:



*** Programmstruktur/Programmrahmen ***

Syntax:

```
<Programm>::=
        <Programmkopf> <Block> .
<Programmkopf>::=
        PROGRAM <Bezeichner> ((<Programmparameter>))
<Programmparameter>::=
        <Bezeichner> {,<Bezeichner>}
<Block>::=
        <Markendeklarationsteil>
        <Konstantendefinitionsteil>
        <Typdefinitionsteil>
        <Variablendeklarationsteil>
        <Prozedur-und Funktionsdeklarationsteil>
        <Anweisungsteil>
```

Die Programmparameter beschreiben Files, durch die das Programm mit seiner Umgebung verbunden ist. Werden Sie aufgefuehrt, sind sie im Hauptprogramm zu spezifizieren.

Die Standard-Dateinamen INPUT (Eingabe Tastatur) und OUTPUT (Ausgabe Bildschirm) brauchen nicht definiert werden. Die Angabe im Programmkopf kann entfallen, sollte jedoch aus Gruenden der Uebersichtlichkeit geschrieben werden.

Beispiel:

```
PROGRAM Test;                              {beide Beispiele }
PROGRAM Test (INPUT, OUTPUT);              {sind aequivalent}

PROGRAM Fakt (Drucker, Artikel, Kunde);
```

4.3. Deklarationen und Definitionen

4.3.1. Markendeklaration

Jede Anweisung in einem PASCAL-Programm kann mit einer Marke versehen werden. Die damit gekennzeichneten Anweisungen koennen mit der Sprunganweisung (GOTO) von jeder Stelle des Programmes aus erreicht werden.
Alle Marken (Label), welche in einem Programmblock verwendet werden sollen, muessen deklariert werden.

Syntax:

```
<Markendeklarationsteil>::=
        <leer>
        | LABEL <Marke> {,<Marke>}
<Marke>::=
        <natuerliche Zahl> | <Bezeichner>
```

Bei der Bildung von Markennamen ist sowohl die Verwendung von Zahlen als auch von Bezeichnern erlaubt.

Beispiel:

```
LABEL Ende,10,20,30;
```

4.3.2. Konstantendefinition

Durch die Konstantendefinition koennen Bezeichnern Werte zugeordnet werden.
Diese Konstantenbezeichner koennen im Programm als Synonym fuer die jeweilige Konstante verwendet werden. Es ist aber auch moeglich, Typenkonstanten zu definieren. Das sind Variablen, die mit CONST einen Anfangswert erhalten.

Syntax:

```
Konstantendefinition := <leer>
    |CONST <Bezeichnerdefinition>{;<Bezeichnerdefinition>
Bezeichnerdefinition::=
    <Konstantendefinition>
    | <Typkonstantendefinition>
Konstantendefinition::=
    <Konstantenbezeichner> = <Konstante>
Konstantenbezeichner::= Bezeichner
Typenkonstantendefinition::=
    <Typkonstantenbezeichner>:<Typ> = <Typkonstante>
Typkonstantenbezeichner::= Bezeichner
Typkonstante::=
    <Konstante>
    |(<Typkonstante>{,<Typkonstante>})
    |(<Recordteilbezeichner>:<Typkonstante>
                {;<Recordteilbezeichner>:<Typkonstante>})
    |Mengenkonstruktur
```

*** Konstantendefinition ***

Die Verwendung der Konstantendefinition hat folgende Vorteile:

- der Programmtext wird besser lesbar;
- bei Aenderungen von Konstanten (z.B. Feldgrenzen) muss nur die Definition, nicht aber die Konstante in den einzelnen Anweisungen geaendert werden.
- einfache Anfangswertbelegung fuer Variablen.

Bei der Konstantendefinition wird implizit auch der Typ deklariert. Der Wert der Konstanten legt den Typ fest.

Beispiel:

```
     CONST  Datum   = '12.12.85'      { STRING | ARRAY OF CHAR }
            Leerstr = '';             { CHAR | STRING }
            Index   = 130;            { INTEGER }
            Zg      = NIL;            { Zeiger }
```

Vordefinierte Konstante:

```
            PI      =  3.1415926536E+00 {REAL}
            FALSE   =  falsch           {BOOLEAN}
            TRUE    =  wahr             {BOOLEAN}
            MAXINT  =  32767            {INTEGER}
```

Typisierte Konstanten werden ausfuehrlich unter Ziffer 4.3.5. dargestellt.

4.3.3. Datentypen und TYPE-Definition

Jede Variable und Konstante eines PASCAL-Programms besitzt einen Datentyp, welcher die entsprechende Darstellungsform im Speicher und den Wertevorrat spezifiziert.

Grundsaetzlich ist zu unterscheiden zwischen
- Standard-Typen (sind vordefiniert) und
- benutzerdefinierten Typen.

Bei der Definition von Datentypen sind folgende Regeln zu beachten:

(1) Jede Variable kann nur einen Typ besitzen.
(2) Der Typ jeder Variablen muss vor der ersten Verwendung vereinbart werden.
(3) Bei Operationen mit Daten verschiedenen Typs sind Typ- und Zuweisungsvertraeglichkeit zu beachten.

Syntax:

```
<Typ>::=
        <einfacher Typ>
        | <strukturierter Typ>
        | <Zeigertyp>
```

4.3.3.1. TYPE-Definition

Die Festlegung des Datentyps erfolgt entweder direkt im Variablendefinitionsteil oder durch einen Typbezeichner. Der Nutzer hat die Moeglichkeit, festgelegte Typbezeichner anzuwenden oder mit Hilfe der Typdefinition eigene festzulegen.
Die Definition von Datentypen erfolgt in folgender Form:

Syntax:

```
<Typdefinitionsteil>::=
        <leer>
        | TYPE <Typdefinition> {;<Typdefinition>}
<Typdefinition>::=
        <Typbezeichner> = <Typ>
```

Eine Definition von Datentypen mit Type hat folgende Vorteile:
- Vereinfachung des Entwurfs eines PASCAL-Programms;
- Einsparung von Schreibaufwand bei Verwendung mehrerer Variablen des gleichen Typs;
- Hilfe beim Verhueten und Suchen von Fehlern;
- Herstellung von Typenvertraeglichkeit fuer Felder;
- Schaffung von Voraussetzungen zum Parameteraustausch mit Unterprogrammen fuer strukturierte Variablen.

4.3.3.2. Einfacher Typ

Syntax:

```
<einfacher Typ>::= <ordinaler Typ> | REAL

<ordinaler Typ>::= <ordinaler Standardtyp>
                 | <Aufzaehlungstyp>
                 | <Teilbereichstyp>
```

4.3.3.2.1. Ordinaler Typ

4.3.3.2.1.1. Ordinaler Standardtyp

Der ordinale Standardtyp bezeichnet eine endliche linear geordneter Menge von Werten.
Folgende ordinale Standardtypen sind in PASCAL realisiert:

Syntax:

```
<ordinaler Standardtyp>::=
        CHAR
        | BOOLEAN
        | INTEGER
        | BYTE
```

*** Ordinaler Standardtyp ***

Standardtyp	Groesse	Wertebereich
CHAR	1 Byte	Zeichensatz (chr(Ø)..chr(7F))
BOOLEAN	1 Byte	TRUE | FALSE
INTEGER	2 Byte	-32768 .. +32767 | Ø0000..ØFFFF
BYTE	1 Byte	Ø .. 255

CHAR
CHAR definiert einen Datentyp als Elemente des Zeichensatzes. CHAR-Variablen koennen Werte zwischen CHR(Ø) und CHR(127) annehmen. Ihre Wirkung (Steuerzeichen und druckbare Zeichen) richtet sich nach der Codevereinbarung .

BOOLEAN
Der Datentyp BOOLEAN repraesentiert Wahrheitswerte, welche durch TRUE ("wahr") und FALSE ("falsch") ausgedrueckt werden.

INTEGER
Der Datentyp INTEGER definiert eine Untermenge der ganzen Zahlen. Dabei wird zunaechst das nieder-, dann das hoeherwertige Byte abgelegt:

```
                 ------------- -------------
      367    = | ØLLØ LLLL | | ØØØØ ØØØL |      (6F Ø1)
                 ------------- -------------
                                |
                                |-> Vorzeichen  Ø = Plus
                                               L = Minus
```

Der Wertebereich umfasst -32768 ... +32767 (MAXINT=32767). INTEGER-Konstanten koennen hexadezimale Zahlen sein (z.B. Ø3A, Ø016F). Ihr Wertebereich reicht dann von Ø0000 bis ØFFFF. Ein Ueberlauf zwischen positiven und negativen Bereich wird nicht ueberwacht.

BYTE
Der Datentyp BYTE belegt ein Byte im Speicher und ist zuweisungsvertaeglich zum Typ INTEGER.

4.3.3.2.1.2. Aufzaehlungstyp

Ein Aufzaehlungstyp definiert eine geordnete Menge von Werten durch Aufzaehlung der Bezeichner, die als Konstanten deren Werte ausdruecken.

Syntax:

```
<Aufzaehlungstyp>::=
        (<Bezeichner> {,<Bezeichner>})
```

*** Aufzaehlungstyp ***

Beispiel:
```
TYPE Material = (Grisuten, Baumwolle, Wolle, Polyesterseide)
TYPE Wochtage = (Montag, Dienstag, Mittwoch, Donnerstag,
                 Freitag, Sonnabend, Sonntag);
```

4.3.3.2.1.3. Teilbereichstyp

Durch die Angabe des kleinsten und des groessten Wertes eines ordinalen Typs kann ein Typ als Teilbereich eines ordinalen Typs definiert werden:

Syntax:

```
<Teilbereichstyp>::=
      <Konstante>..<Konstante>
```

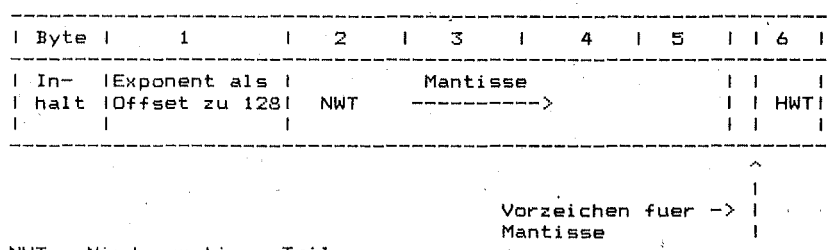
Die erste Konstante legt die untere Grenze fest; ihr Wert darf nicht groesser als die obere Grenze sein.

Beispiel:
```
TYPE Intzahl   = 1 ..1000;
     Bereich   = -10 .. +10;
     Wochentag = (Montag,Dienstag,Mittwoch,Donnerstag,
                  Freitag,Samstag,Sonntag)
     Werktag   = Montag .. Freitag
```

Werktag ist ein Teilbereich des ordinalen Typs Wochentag.

4.3.3.2.2. REAL-Typ

Der Datentyp REAL ermoeglicht die Darstellung positiver und negativer gebrochener Zahlen. Eine REAL-Zahl belegt 6 Bytes. Das erste enthaelt den Exponenten und das Vorzeichen, das zweite bis sechste Byte die Mantisse. Das Vorzeichen der Mantisse ist in einem Bit des sechsten Bytes verschluesselt.



NWT - Niederwertiger Teil
HWT - Hoeherwertiger Teil
Die 6-Byte-REAL-Darstellung sichert eine Genauigkeit von 11 signifikanten Ziffern. Der Wertebereich liegt zwischen 1E-38 und 1E+38. Weitere Einzelheiten zur internen Darstellung enthaelt Anhang D.

4.3.3.3. Strukturierter Typ

Ein strukturierter Typ wird durch die Typen seiner Komponenten und durch die Methode der Strukturierung gekennzeichnet.

Syntax:

```
<strukturierter Typ>::=
        < strukturierter Typ>
        | PACKED < strukturierter Typ>
< strukturierter Typ> ::=
        <Feldtyp>
        | <Recordtyp>
        | <Filetyp>
        | <Mengentyp>
        | <Zeichenkettentyp>
```

PACKED wird vom Compiler akzeptiert, hat aber keine Wirkung. Einzelheiten zur internen Darstellung enthaelt Anhang D.

4.3.3.3.1. Feld-Typ

Ein Feld-Typ ist eine aus einer festen Anzahl von Komponenten bestehende Struktur. Diese Komponenten sind alle vom gleichen Typ. Die Komponenten des Feldes werden durch Indizes angegeben, deren Werte zum ordinalen Typ gehoeren. Sie werden in eckigen Klammern geschrieben und an den Bezeichner des Feldes angehaengt.

Syntax:

```
<Feldtyp>::=
        ARRAY [<Indextyp>{,<Indextyp>}] OF <Typ>
<Indextyp>::=
        <ordinaler Typ>
```

<Typ> ist ein beliebiger Datentyp. Damit gibt es Felder von Feldern, Felder von Felder von Feldern, Felder von Records usw. Ein Feld-Typ heisst n-dimensional, wenn n Indextypen spezifiziert sind.

Beispiel:

```
     TYPE Kette  = ARRAY [Anfang..Ende] OF ARRAY[1..10] OF CHAR;
          Satz   = ARRAY ['a'..'z'] OF BYTE;
          B100   = ARRAY [1..10, 1..20] OF 0.:99;
     VAR  Tabelle:Kette;
          Tabelle[Anfang][1]:='A';
```

Es besteht die Moeglichkeit, Felder zu kopieren, wenn sie als Ganzes vom gleichen Typ sind, d.h. mit der gleichen Typenvereinbarung eingefuehrt wurden.
Die Pruefung zulaessiger Indexausdruecke ist mit der Compiler-Direktive R moeglich. Standard ist {$R-}, bei gewuenschter Zulaessigkeitspruefung muss {$R+} gesetzt werden (vergl. Ziffer 4.1.4.6.).

4,3.3.3.2. Record-Typ

Ein Record-Typ ist eine Struktur, welche aus einer festen Anzahl Komponenten gleicher oder unterschiedlicher Typen besteht. Fuer jede Komponente wird ein Bezeichner und ein Typ festgelegt. Ein Record-Typ kann mehrere Varianten haben. Dabei kann eine bestimmte Komponente als Kennzeichen verwendet werden, durch deren Wert festgelegt wird, welche Struktur zu einer gegebenen Zeit verwendet werden soll.
Jede Variante wird durch eine Kennzeichenkonstante charakterisiert. Jede dieser Konstanten ist ein Wert des Typs der Kennzeichenvariablen. Der Zugriff zu einer Recordkomponente wird erreicht, indem der Variablenbezeichner mit dem Recordkomponentenbezeichner, getrennt durch einen Punkt, angegeben wird.
Im Fall, dass Datensaetze vom gleichen Typ sind, ist es moeglich, diese einander zuzuweisen und somit einen Datensatz von Datensaetzen zu bilden.

Syntax:

```
<Recordtyp>::=
        RECORD <Recordkomponentenliste> END

<Recordkomponentenliste>::=
        <fester Teil>
        | <fester Teil> ; <varianter Teil>
        | <varianter Teil>

<fester Teil>::=
        <Recordkomponente> {;<Recordkomponente>}

<Recordkomponente>::=
        <Recordkomponentenbezeichner>
                        {,<Recordkomponentenbezeichner>} : <Typ>
        | <leer>

<varianter Teil>::=
        CASE <Kennzeichenvariable> <ordinaler Typ> OF
        <Variante>{;Variante>}

<Kennzeichenvariable>::=
        <Bezeichner> :<ordinaler Typ>
        | <ordinaler Typ>
Bei fehlenden Bezeichner spricht man von freie Varianten

<Variante>::=
        <Kennzeichenkonstantenliste>:(<Recordkomponentenliste>)
        | <leer>

<Kennzeichenkonstantenliste>::=
        <Konstante> {,<Konstante>}
```

*** Record-Typ ***

Beispiel:

```
TYPE Ta    = RECORD
                Anr    : STRING [16];
                Bez    : STRING [30];
                Preise : ARRAY [1..5, 1..10] OF REAL;
                Best   : REAL;
                Kz     : CHAR
             END;
     Tdat  = RECORD
                Tag : 1..31;
                Mon : 1..12;
                Jhr : INTEGER
             END;
     Tn    = RECORD
                Name,Vorname : ARRAY [1..25] OF CHAR;
                Alter        : 0..120;
                Verh         : BOOLEAN
             END;
     Form  = (Rechteck,Kreis,Dreieck);

     Tv    =  RECORD
                 x,y      : REAL;
                 Flaeche  : REAL;
                 Case S   : Form OF
                Dreieck   : (Seite : REAL;
                             Neigung,Wink1,Wink2:Winkel);
                Kreis     : (Radius : REAL);
                Rechteck  : (Seite1,Seite2 : REAL)
              END;
                {S ist eine Variable vom Typ Form}
```

4.3.3.3.3. File-Typ

Mit der Definition eines File-Typs wird eine Struktur festgelegt, die aus einer Folge von Komponenten gleichen Typs besteht. Die Anzahl der Komponenten (Groesse des Files) wird durch die Definition nicht festgelegt.

Syntax:

```
<Filetyp>::= FILE
           |FILE OF <Typ>
           |TEXT
```

Die erste Definition spezifiziert ein File beliebigen Typs (typlos, ungetypt), die zweite ein Binaerfile und die dritte ein Textfile(TYPE TEXT = FILE OF CHAR).
Fuer Programmverkettungen werden typlose Dateien vereinbart. Eine Konstruktion in der Form TYPE X = FILE OF FILE OF ... ist nicht zulaessig.

Beispiel:

```
TYPE Arfile = FILE OF Ta;
     Ket1   = FILE;
     Quelle = TEXT;
```

4.3.3.3.4. Mengen-Typ

Unter einer Menge versteht man in PASCAL die Zusammenfassung mehrerer Objekte des gleichen Typs. Zu einer Menge koennen maximal 256 Elemente gehoeren. Die Ordnungswerte des Typs der Objekte liegen folglich im Bereich 0..255.

Syntax:

```
<Mengentyp>::=
        SET OF <ordinaler Typ>
```

Jedes Element des Satzes wird in einem Bit gespeichert. Ist das jeweilige Element in der Menge enthalten, ist das Bit gesetzt, sonst nicht.

Es werden jeweils soviel Bytes reserviert, wie zur Darstellung der Elemente benoetigt werden (maximal 32).

Beispiel:

```
     TYPE Tsp   = (Skat,Halma,Dame,Schach)
     TYPE Spiel = SET OF Tsp;
     TYPE Park  = SET OF (Trabant, Wartburg, Lada, Skoda);
```

Weitere Einzelheiten enthaelt Anhang D.

4.3.3.3.5. Dynamischer Zeichenkettentyp

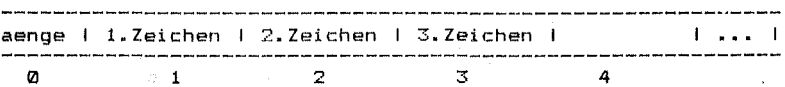
Syntax:

```
<Zeichenkettentyp>::=
          STRING [<natuerliche Zahl>]
        | STRING [<Konstantenbezeichner>]
```

Mit dem Typ STRING wird eine Zeichenkette durch die Angabe der Anzahl maximal moeglicher Zeichen definiert.

```
    STRING [<n>]  = Zeichenkette fuer max.  n Zeichen
                    (n = 1..255)
```

Eine Variable des Typs STRING[<n>] belegt n+1 Byte. Im ersten Byte wird die aktuelle Laenge der Variablen gespeichert. Die einzelnen Zeichen der Zeichenkette sind indizierbar.
Die Speicherung erfolgt in folgender Form:



Wenn die Anzahl der Zeichen einer Kette kleiner ist als die definierte Laenge, dann sind die am Ende im Speicher stehenden Bytes undefiniert (sie werden nicht geloescht).

Beispiel:
```
TYPE Ts = STRING[20];
```

Weitere Einzelheiten enthaelt Anhang D.

4.3.3.3.6. Standardfelder

Die Standardfelder MEM und MEMW werden eingesetzt, um den Zugriff zum Speicher zu realisieren. Mit dem Standardfeld MEM wird ein Byte und mit MEMW ein Wort (zwei Byte, LSB zuerst) lokalisiert. Der Index des Feldes besteht aus Segmentadresse und Offset.

Beispiel:
```
ABC := MEM[0000:0080]
```

Die Standardfelder PORT und PORTW werden benutzt um den Zugriff zu den Datenports zu realisieren. Der Index des Feldes ist vom Type Integer. Die Komponenten von PORT sind vom Type Byte, die von PORTW vom Type Integer;

Beispiel:
```
Port[28]:=15;
Abc:=Port[33];
```

Standardfelder sollten nur von erfahrenen Programmierern benutzt werden, da unmittelbare, nichtkontrollierbare Eingriffe in das Laufzeitsystem erfolgen koennen.

4.3.3.4. Zeigertyp

Der Zugriff zu dynamischen Variablen erfolgt mit Hilfe des Wertes eines Zeigers. Dieser Zeiger wird waehrend der Erzeugung einer dynamischen Variablen bereitgestellt.
Der Zeigertyp besteht aus einer theoretisch unbegrenzten Menge von Werten, die auf Elemente eines Typs weisen (vergl.4.9. Zeiger und Listen).

Syntax:

```
<Zeigertyp>::=
        ^<Typbezeichner>
```

Beispiel:
```
TYPE      Zgt       = ^Element;
          Element   = RECORD
                        Wert1 : REAL;
                        Wert2 : REAL;
                        Wert3 : INTEGER;
                        Next  : Zgt
                      END;
```

*** Zeigertyp ***

Anmerkung

Die Variable vom Typ Zgt ist hier ein Zeiger auf ein Objekt vom Typ Element.
Die Bezugnahme auf eine noch nicht definierte Struktur (hier Element) ist in diesen Ausnahmefall moeglich.

4.3.3.5. Typumwandlung und Bereichspruefung

Typumwandlungen werden auf konventionelle Art mit Konvertierungsfunktionen oder mit Retyping ermoeglicht. Bereichspruefungen fuer Skalar- und Teilbereichsvariablen sind mit der Compiler-Derektive {$R+} realisierbar. Standard ist dabei {$R-}, d.h. bei gewuenschter Bereichspruefung muss der Schalter {$R+} im Programmtext gesetzt werden.

4.3.3.5.1. Retyping

Die Typbezeichner CHAR, BYTE, INTEGER und BOOLEAN sowie Typbezeichner des Aufzaehlungstypes sind gleichzeitig als Funktionsbezeichner zur Konvertierung verwendbar.
Hierbei bedeutet das lediglich, dass zum Beispiel
"y = INTEGER('A')" und "y = ord('A')" sowie "x = CHAR(78)" und "x = chr(78)" voellig gleich sind (x ist hier vom Typ CHAR, y vom Typ INTEGER oder BYTE).
Diese Vorgehensweise heisst Retyping.
REAL und STRING sind nicht fuer das RETYPING zugelassen.

Beispiel:

```
TYPE    Monat =(Jan,Feb,Maerz,April,Mai,Juni,Juli,Aug,Sept,
                Okt,Nov,Dez);
        Farbe =(Rot,Gelb,Gruen);
```

Die Anwendung von Retyping auf die obigen Definitionsbeispiele ermoeglicht (Vergleich jeweils TRUE)

```
Monat(11)       = Nov
INTEGER(Gelb)   = 2
#41             = BYTE ('A')
```

4.3.3.5.2. Pseudofunktionen

Die Pseudofunktionen der Konvertierung dienen zur Herstellung der Vertraeglichkeit eines Skalartyps in einen anderen (Pseudo, weil in Wirklichkeit keine Operationen stattfinden).
Pseudofunktionen der Konvertierung sind ORD , PTR und CHR.

4.3.3.5.2.1. ORD-Funktion

ord (<Ausdruck>)

Die Funktion liefert den Ordinalwert (Typ INTEGER) des Ausdrucks.

Beispiel:
```
write (ord ('A'));          {= 65}
write (ord (67));           {= 67}
write (ord (chr(86)));      {= 86}
```

Mit ORD kann auch der INTEGER-Wert von Zeigern festgestellt werden.

4.3.3.5.2.2. CHR-Funktion

chr (<Ausdruck>)

Die Funktion liefert das Zeichen, dessen Ordinalwert dem Wert des Ausdrucks entspricht. Grundlage ist der jeweilige Zeichensatz.

Beispiel:
```
write(chr(66));                 {= B}
write(chr('C'));                {= C}
write(chr (ord('L')));          {= L}
```

Mit CHR ist auch ein Zugriff auf das dynamische Byte (Index=0) eines STRING moeglich. Es sollten jedoch die STRING-Funktionen-Prozeduren vorgezogen werden.

4.3.3.5.2.3. PTR-Funktion

ptr (<Ausdruck1>,<Ausdruck2>)

Mit der Pseudofunktion PTR ist es moeglich, die in einem Pointer stehende Adresse direkt zu steuern. PTR konvertiert dabei einen 32-bit Wert in einen Pointer. Ausdruck1 ist die Segmentadresse und Ausdruck2 das Offset.

Beispiel:
```
pointer:=ptr(cseg,$8000);
```

4.3.4. Variablendeklaration und Variablenzugriff

4.3.4.1. Deklaration von Variablen

Alle Variablen, die in PASCAL verwendet werden, muessen deklariert werden.
Bei der Deklaration wird einer Variablen ein Bezeichner und ein Typ zugeordnet.
Waehrend mit der TYPE-Definition nur ein Datentyp beschrieben wird, wird mit der Variablendeklaration Speicherplatz bereitgestellt.

*** Deklaration von Variablen ***

Syntax:

```
<Variablendeklarationsteil>::=
   <leer>
| VAR <Variablendeklaration> {;<Variablendeklaration>}

<Variablendeklaration>::=
   <Variablenbezeichner>{,<Variablenbezeichner>} : <Typ>
| <Variablenbezeichner>   :<Typ> ABSOLUTE <Adr1>:<adr2>

<Adresse>::=
          <Konstante>
         |<Variablenbezeichner>
```

Absolute Variablen werden durch das Schluesselwort ABSOLUTE gekennzeichnet. Die Variablen werden im Speicher an die durch <Adr1> und <Adr2> gekennzeichnete Adresse gelegt. Adr1 ist dabei die Segmentbasisadresse und Adr2 das Offset. Fuer <Adr1> koennen auch die Standardbezeichner Cseg und Dseg verwendet erden.
Diese Adresse sollte ausserhalb des PASCAL-Programms liegen. Der Programmierer ist fuer die Verwaltung selbst verantwortlich.

Beispiel:

```
     VAR  abc:  BYTE ABSOLUTE $0000:$8000;
          def:  STRING[127] ABSOLUTE $1000:$0080;
```

ABSOLUTE kann auch verwendet werden, um Variablen zu ueberlagern. Die eine Variable beginnt dann auf der gleichen Adresse wie die andere Variable. Dies ist leicht zu erreichen. Folgt in der Variablendefinition dem Wort ABSOLUTE der Bezeichner einer Variablen (oder eines Parameters), dann beginnt die neue Variable auf der Adresse dieser Variablen (oder des Parameters).

Beispiel:

```
     VAR Eins       : STRING[22];
         Zwei       : BYTE ABSOLUTE Eins;
```

In diesem Beispiel beginnt Zwei auf der gleichen Adresse wie Eins. Da aber an dieser Stelle die Laenge von Eins steht, enthaelt Zwei die aktuelle Laenge von Eins.
Es ist zu beachten, dass in einer absoluten Deklaration nur ein Bezeichner erklaert werden kann! Die folgende Konstruktion ist also nicht erlaubt:

```
Ident1, Ident2 : INTEGER ABSOLUTE $0000:0080;
```

Beispiel:

```
VAR      x : REAL;
         y : ARRAY [1..100] OF REAL;
       Art : TA;
       Mat : ARRAY [1..30, 1..50] OF INTEGER;
       Ans : ARRAY [1..5] OF ARRAY [1..30] OF CHAR;
     Ardat : FILE OF Ta;
      Zgrl : ^Element;
      Satz : RECORD
               R:REAL;
               I:INTEGER;
               M:ARRAY[1..10, 1..10, 1..100] OF INTEGER;
               B:BOOLEAN
             END;
       Tex : STRING [20];
```

4.3.4.2. Variablenzugriff

Der Zugriff zu den Werten der Variablen erfolgt durch ihre Auffuehrung im Programmtext. Es gibt folgende Moeglichkeiten:

Syntax:

```
<Variable>::=
        <vollstaendige Variable>
        | <indizierte Variable>
        | <Recordkomponentenvariable>
        | <dynamische Variable>
```

4.3.4.2.1. Vollstaendige Variable

Der Wert einer Variablen kann durch ihren Bezeichner aufgerufen werden. Es kann sich um einfache, strukturierte Variablen oder Zeiger handeln.

Beispiel:

```
Satz
Tex
x
```

4.3.4.2.2. Indizierte Variable

Die Komponente einer n-dimensionalen Feldvariablen wird durch die Angabe der Variablen bezeichnet, der ein n-dimensionaler Index folgt.

Syntax:

```
<indizierte Variable>:::=
        <Feldvariable> [<Index>{,<Index>}]
<Feldvariable>:::=
        <Variable>
<Index>:::=<ordinaler Ausdruck>
```

Die Typen der Indexausdruecke muessen mit den Indextypen vertraeglich sein, die in der Definition des Feld-Typs vereinbart wurden.

Beispiel:

```
Mat [5,6]
y [10]
y[I+14]
```

4.3.4.2.3. Recordkomponentenvariable

Die Komponente einer Recordvariablen wird bezeichnet durch die Angabe der Recordvariablen, welcher ein Punkt und der Bezeichner der Komponente folgt.

Syntax:

```
<Recordkomponentenvariable>:::=
        <Recordvariable> . <Recordkomponentenbezeichner>
```

Beispiel:

```
Art.Anr
Satz.r
Satz.m [i,j,k]
```

4.3.4.2.4. Dynamische Variable

Syntax:

```
<dynamische Variable>:::=
          <Zeigervariable>^
```

Wenn p eine an den Typ t gebundene Zeigervariable ist, dann bezeichnet p diese Variable und den Wert ihres Zeigers. Mit p^ wird die Variable des Typs t bezeichnet, auf die durch p verwiesen wird.

Beispiel:

```
Zgr1^                          {dynamische Variable}
Zgr1^.Wert1                    {Aufruf des Wertes einer }
Zgr2^.Nachf                    {dynamischen Recordkompo-}
                               {nentenvariablen}
```

Beispiele fuer Variablenzugriff

Nachfolgend wird an einigen Beispielen der Zugriff zu Variablen dargestellt.

Verwendet werden die Beispiele zur TYPE-Definition (Pkt.4.3.3.) und zur Variablen-Deklaration (Pkt. 4.3.4.).

Variablenzugriff	Bereitgestellte Daten	Typ
x	1 * REAL	REAL
y	100 * REAL	ARRAY OF REAL
y [66]	1 * REAL	REAL
Art	1 * 17 Byte STRING(CHAR)	RECORD bzw. FILE OF RECORD
	1 * 31 Byte STRING(CHAR)	
	50 * REAL	
	1 * REAL	
	1 * CHAR	
Art.Preise	50 * REAL	ARRAY
Art.Preise[i,j]	1 * REAL	REAL
Ardat .Kz	1 * CHAR	CHAR
Mat	1500 * INTEGER	ARRAY OF INTEGER
Mat[20,20]	1 * INTEGER	INTEGER
Ans	150 * CHAR	ARRAY OF ARRAY
Ans[2][3]	1 * CHAR	CHAR
Satz	1 * REAL	RECORD
	1 * INTEGER	
	10000 * INTEGER	
	1 * BOOLEAN	
Satz.m	10000 * INTEGER	ARRAY OF INTEGER
Satz.m[I,6,89]	1 * INTEGER	INTEGER
Zgr1^.Wert1	1 * REAL	REAL
Bs[12][8]	1 * CHAR	CHAR
k	1 * INTEGER	INTEGER
Zgr1	1 * INTEGER	Zeiger

4.3.5. Typisierte Konstante

Eine typisierte Konstante kann wie eine Variable verwendet werden. Sie ist als initialisierte Variable zu betrachten, deren Wert von Anfang an definiert ist. Die Verwendung typisierter Konstanten erspart Laufzeit, da die Anfangsbelegung bereits vom Compiler vorgenommen wird. Typisierte Konstanten werden wie normale Konstanten definiert; sie erhalten nur zusaetzlich auch ihren Typ. Man beachte, dass die definierten Werte nur beim Neustart der COM/CHN-Files zur Verfuegung stehen und dann ihren Wert aendern koennen.
Ein Wiederstart kann bereits andere Werte bringen. Die Syntax ist in Ziffer 4.3.2. definiert.

4.3.5.1. Einfache typisierte Konstante

Eine einfache typisierte Konstante wird wie eine einfache Variable definiert.

Beispiel:

```
CONST   Anzahl: INTEGER = 1267;
        Zahl: REAL = 12.67;
        Zeichen: CHAR = ^Q;
        Buchstabe:CHAR = #65;
```

Typisierte Konstanten duerfen anstelle einer Variablen als Parameter in Unterprogrammen verwendet werden. Eine typisierte Konstante stellt eine Variable mit einem definierten Wert dar. Sie kann somit nicht in der Definition anderer Konstanten oder Typen verwendet werden.

Beispiel:

```
CONST
    Unten : INTEGER = 0;
    Oben  : INTEGER = 50;
TYPE
    Feld: ARRAY [Unten..Oben] OF INTEGER;  {Unzulaessig !}
```

4.3.5.2. Strukturierte typisierte Konstante

Strukturierte typisierte Konstanten sind
- Feldkonstanten,
- Recordkonstanten und
- Mengenkonstanten.

4.3.5.2.1. Typisierte Feldkonstante

Beispiel:

```
TYPE
     Zustand        = (Kalt,Heiss,Warm);
     Feld           = ARRAY [ZUSTAND] OF STRING[5];
CONST
     Zust:Feld      =('Kalt','Heiss','Warm');
```

Im Beispiel wird die Feldkonstante Zustand definiert, die genutzt werden kann, um Werte vom Aufzaehlungstyp in ihre entsprechende Stringdarstellung zu konvertieren:

```
Zustand[Kalt]     = 'Kalt'
Zustand[Heiss]    = 'Heiss'
Zustand[Warm]     = 'Warm'
```

Jeder Typ, ausser einem Feld- oder Zeigertyp, stellt einen zulaessigen Komponententyp einer Feldkonstante dar. Bei Characterfeldtypen sind einzelne Zeichen und Zeichenketten erlaubt.

*** Typisierte Feldkonstante ***

Bei der Definition einer typisierten mehrdimensionalen Feldkonstante wird jede Dimension in separate Klammernpaare eingeschlossen, die durch Komma voneinander getrennt sind.
Dabei entspricht die innerste Konstante der am weitesten rechts stehenden Dimension.

Beispiel:

```
TYPE  Feld = ARRAY[0..1,0..1,0..1] OF INTEGER;

CONST Zahl : Feld = (((0,1),(2,3)),((4,5),(6,7)));

BEGIN
     writeln (Zahl[0,0,0],' =0');
     writeln (Zahl[0,0,1],' =1');
     writeln (Zahl[0,1,0],' =2');
     writeln (Zahl[0,1,1],' =3');
     writeln (Zahl[1,0,0],' =4');
     writeln (Zahl[1,0,1],' =5');
     writeln (Zahl[1,1,0],' =6');
     writeln (Zahl[1,1,1],' =7')
END;
```

4.3.5.2.2. Typisierte Recordkonstante

Beispiel:

```
TYPE Zahl        = RECORD
                     a,b,c : INTEGER
                   END;
     Farbe       = (Rot,Gelb,Gruen,Blau);
     Stoff       = (Wolle,Seide,Tweet);
     Kleid       = RECORD
                     Design  : ARRAY[1..4] OF Farbe;
                     Material: Stoff
                   END;

CONST Nummer: Zahl = (a:0, b:0, c:0);
      Modell: Kleid=
                 (Design:(Rot,Gelb,Gruen,Blau);
                  Material:Tweet);
      Matrix: ARRAY[1..3] OF Zahl=
                 ((a:1,  b:4,  c:5),
                  (a:13, b:8, c:55),
                  (a:200,b:16, c:-65));
```

Die Feldkonstanten sind in der gleichen Reihenfolge zu definieren, wie sie in der Recorddefinition auftreten. Im Fall, dass ein Datensatz Felder vom File- oder Zeigertyp enthaelt ist es nicht moeglich, typisierte Konstanten fuer diesen Recordtyp zu definieren. Wenn eine Recordkonstante Varianten enthaelt, so ist der Programmierer selbst dafuer verantwortlich, dass nur die Datenfelder der gueltigen Variable spezifiziert werden. Enthaelt die Variable ein Kennzeichnungsfeld, dann muss auch ihr Wert spezifiziert werden.

4.3.5.2.3. Typisierte Mengenkonstante

Eine typisierte Mengenkonstante wird aus einer oder mehreren Elementenspezifikationen, die durch Komma getrennt und in eckigen Klammern eingeschlossen sind, gebildet. Eine Elementenspezifikation kann eine Konstante oder ein Bezeichnerausdruck sein. Er besteht aus zwei Konstanten, getrennt durch zwei Punkte.

Beispiel:

```
TYPE  Gross= SET OF 'A'..'Z';
      Klein= SET OF 'a'..'z';

CONST Grossbuchst :Gross  = ['A'..'Z'];
      Vokale      : Klein = ['a','e','i','o','u'];
      Zeichen     : SET OF CHAR =
                      [' '..'/',':'..'?','['..'`','{'..'~'];
```

4.3.6. Prozedur- und Funktionsdeklaration

Eine Prozedur- Funktionsvereinbarung definiert ein Unterprogramm innerhalb eines Programms oder einer anderen Prozedur/Funktion. Es ist gueltig fuer den gesamten Block, in dessen Vereinbarungsteil sie deklariert wurde. Einzelheiten enthalten die Ziffern 4.6. und 4.7. .

4.4. Operatoren und Ausdruecke

4.4.1. Operatoren

Operatoren werden zum Verknuepfen bzw. Vergleichen von Ausdruecken verwendet. Operatoren koennen in sechs Kategorien eingeteilt werden:

1) Minusvorzeichen
2) NOT Operator
3) Multiplikationsoperatoren: *,/,DIV,MOD,AND,SHL,SHR.
4) Additionsoperatoren: +,-,OR,XOR.
5) Vergleichsoperatoren: =,<>,<,>,<=,>=.
6) Mengenoperatoren.

Sind beide Operanden eines Multiplikations- oder Additionsoperators vom Typ INTEGER, dann ist auch das Ergebnis vom Typ INTEGER. Wenn einer oder beide Operatoren vom Typ REAL sind, dann ist auch das Ergebnis vom Typ REAL. Vergleichsoperatoren liefern immer den Typ BOOLEAN. Die Prioritaet der Operatoren wird in Ziffer 4.4.1.7. dargestellt.

4.4.1.1. Minuszeichen

Das Minuszeichen bezeichnet die Negation des Operanden, der vom Typ INTEGER oder REAL sein muss.

4.4.1.2. Operator NOT

Der Operator NOT kann auf Operanden vom Typ BOOLEAN angewendet werden und drueckt die Negation aus:

```
NOT TRUE  = FALSE
NOT FALSE = TRUE
```

PASCAL erlaubt auch die Anwendung des Operators NOT auf Operanden vom Typ INTEGER und BYTE. In diesem Falle erfolgt die Negation der einzelnen Bits.

Beispiel:

```
NOT 0     = -1
NOT $2345 = $DCBA
```

4.4.1.3. Multiplikationsoperatoren

```
<Multiplikationsoperator> ::=
          *
        | /
        | DIV
        | MOD
        | AND
        | SHL
        | SHR
```

*** Multiplikationsoperatoren ***

Operator	Operation	Typ der Operanden	Typ des Ergebnisses
*	Multiplikation	REAL,REAL	REAL
*	Multiplikation	INTEGER,INTEGER	INTEGER
*	Multiplikation	REAL, INTEGER	REAL
*	Durchschnitt	Mengen	Menge
/	Division	REAL,REAL	REAL
/	Division	REAL, INTEGER	REAL
DIV	Division	INTEGER	INTEGER
MOD	Rest (Modulus)	INTEGER,	INTEGER
AND	logisches UND	BOOLEAN	BOOLEAN
AND	arithmetisches UND	INTEGER	INTEGER
SHL	Shift links	INTEGER	INTEGER
SHR	Shift rechts	INTEGER	INTEGER

BYTE gilt als echte Teilmenge von INTEGER. Bei den Operationen muessen sich dann aber BYTE/BYTE gegenueberstehen.

Beispiel:

```
123*456          =    492 falsch, Ueberlauf der Integerzahl!
123 DIV 4        =    30
12 MOD 5         =    2
TRUE AND FALSE   =    FALSE
12  AND 22       =    4
2   SHL 7        =    256 {Linksverschiebung des Bitmuster
                          um 7 Positionen}
256 SHR 7        =    2
```

Die bitweise AND - Operation zeigt das folgende Schema:

	Zahl	hexadezimal (HWT,NWT)	Bitmuster (HWT,NWT)
1.Operand	12	80C	0000 0000 0000 1100
2.Operand	22	816	0000 0000 0001 0110
Ergebnis bei AND	4	804	0000 0000 0000 0100

4.4.1.4. Additionsoperatoren

Syntax:

```
<Additionsoperator>:::=
          +
        | -
        | OR
        | XOR
```

*** Additionsoperatoren ***

Operator	Operation	Typ der Operanden	Typ des Ergebnisses
+	Addition	REAL,REAL	REAL
+	Addition	INTEGER,INTEGER	INTEGER
+	Addition	INTEGER, REAL	REAL
+	Vereinigung	Mengen	Menge
-	Subtraktion	REAL,REAL	REAL
-	Subtraktion	INTEGER,INTEGER	INTEGER
-	Subtraktion	INTEGER, REAL	REAL
-	Differenz	Mengen	Menge
OR	logisches ODER	BOOLEAN	BOOLEAN
OR	arithmetisches ODER	INTEGER,INTEGER	INTEGER
XOR	logisches XOR	BOOLEAN,BOOLEAN	BOOLEAN
XOR	arithmetisches XOR	INTEGER,INTEGER	INTEGER

BYTE gilt wieder als Teilbereichstyp 0..255 von INTEGER.

Beispiel:

```
123 + 456        =    579
456 - 123.0      =    333.0
TRUE OR FALSE    =    TRUE
12 OR 22         =    30
TRUE XOR FALSE   =    TRUE
12 XOR 22        =    26
```

4.4.1.5. Vergleichsoperatoren

Syntax:

```
<Vergleichsoperator>  ::=
          =
        | <>
        | <
        | <=
        | >
        | >=
        | IN
```

*** Vergleichsoperatoren ***

Operator	Operation	Typ der Operanden	Typ d. Ergebnisses
=, <>	gleich, ungleich	einfacher Typ, Mengen, Zeiger, Zeichenketten	BOOLEAN
<, >	kleiner, groesser	einfacher Typ Zeichenkette,	BOOLEAN
<=, >=	kleiner gleich groesser gleich	einfacher Typ, Zeichenkette	BOOLEAN
<=	Inklusion "ist enthalten in"	Mengen	BOOLEAN
>=	Inklusion "enthaelt"	Mengen	BOOLEAN
IN	Enthaltensein	ordinaler Typ /Menge	BOOLEAN

Bei Vergleichen von Zeichenketten wird links begonnen, und die Zeichen werden byteweise entsprechend ihrer Ordnung im Zeichensatz verglichen. Kuerzere Zeichenketten werden durch Leerraeume (SPACE) ergaenzt.

Bei Vergleichen der Ordinalwerte von booleanschen Groessen gilt:
FALSE < TRUE
(ord(FALSE) = 0; ord(TRUE) = 1).

4.4.1.6. Mengenoperatoren

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.

Die Pruefung auf eine leere Durchschnittsmenge kann man in der Form A*B = [] fuer zwei Mengen programmieren, [] kennzeichnet eine leere Menge. Die Relationen < und > sind fuer Mengen nicht definiert.

Beispiel:

```
x:= [1,2];
y:= [2,3];

e:= x * y;      { e = [2] }

e:= x + y;      { e = [1,2,3] }

e:= x - y;      { e = [1] }
```

4.4.1.7. Prioritaet

In mehrgliedrigen Ausdruecken werden die einzelnen Operationen entsprechend ihrer Prioritaet ausgefuehrt:

NOT	{hoechste Prioritaet}
Multiplikationsoperatoren	
Additionsoperatoren	
Vergleichsoperatoren	{niedrigste Prioritaet}

Sind in einem Ausdruck mehrere Operatoren gleicher Prioritaet, dann werden diese von links beginnend abgearbeitet.
Die Prioritaet kann durch Setzen von Klammern veraendert werden. Dabei werden Klammern, von links bzw. von innen beginnend, zuerst aufgeloest.
Innerhalb der Klammer gelten wieder die o.g. Regeln.

Beispiel:

```
     5 + 6  * 10          =  65
    (5 + 6) * 10          = 110
    (5* (3+6) -8) +10     =  47
    (5+6) < (3*5)         = TRUE
     NOT (8 > 4)          = FALSE
```

Man beachte, dass in logischen Ausdruecken, z.B. (x>5) AND (y>10), die Klammern notwendig sind, um den durch die Prioritaet sonst entstehenden Typkonflikt zu vermeiden.

4.4.2. Ausdruecke

Ausdruecke sind Konstruktionen, welche die Regeln fuer das Rechnen mit Werten von Variablen und die Erzeugung neuer Werte durch Anwendung von Operatoren ausdruecken.

Ausdruecke bestehen aus Operanden (Variablen, Konstanten), Operatoren und Funktionen.

*** Ausdruecke ***

Syntax:

```
<Ausdruck>::=
        <einfacher Ausdruck>
        | <einfacher Ausdruck><Vergleichsoperator>
                                        <einfacher Ausdruck>

<einfacher Ausdruck> ::=
        <Term>
        | <einfacher Ausdruck><Additionsoperator><Term>

<Term>::=
        <Faktor>
        | <Faktor><Multiplikationsoperator><Faktor>

<Faktor>::=
        <Variable>
        | <vzl. Konstante>
        | <Funktionsaufruf>
        | (<Ausdruck>)
        | <NOT Operator><Faktor>
        | <Faktor>
        | <Menge>

<Menge>::=
        [ <Liste der Elemente> ] |  []

<Liste der Elemente>::=
        <Element> {,<Element>}

<Element>::=
        <Ausdruck>
        | <Ausdruck>..<Ausdruck>
```

Ausdruecke, die Elemente der gleichen Menge sind muessen alle vom gleichen Typ sein (= Basistyp der Menge).
[] kennzeichnet eine leere Menge und [x..y] bezeichnet die Menge aller Werte aus dem Intervall x bis y.

Beispiel:

```
100.76
x
x + y[12]
(x * ART.PREISE[1,4] / 100)
[Montag, Dienstag, Mittwoch]
(x < y[2]) AND (ZGR1^.WERT1 <> 0)
x = 12.3456
```

Wert des Operanden "a"	T	T	F	F
Wert des Operanden "b"	T	F	T	F
NOT a (Negation)	F	F	T	T
NOT b (Negation)	F	T	F	T
a AND b (Konjuktion)	T	F	F	F
a OR b (Disjunktion)	T	T	T	F
a XOR b (Exklusion)	F	T	T	F

4.4.3. Funktionsaufruf

Durch einen Funktionsaufruf wird eine Funktion aktiviert. Der Aufruf besteht aus dem Funktionsbezeichner und einer Liste aktueller Parameter.
Die aktuellen Parameter (Variablen und Ausdruecke) werden fuer die korrespondierenden formalen Parameter substituiert (vergl. Ziffer: 4.6.).

Das Auftreten eines Funktionsaufrufes im Programm bewirkt die Aktivierung der Funktion, durch die sie bezeichnet wird. Wenn die Funktion keine Standardfunktion ist, muss sie vor dem Aufruf definiert sein.

Syntax:

```
<Funktionsaufruf>::=
        <Funktionsbezeichner>
        | <Funktionsbezeichner> (<Parameter>{,<Parameter>})

<Parameter>::=
        <Ausdruck>
```

Funktionen oder Prozeduren sind als aktuelle Parameter nicht erlaubt.

Beispiel:

```
Volumen(Radius,Hoehe)
Durschn (x,y[j])
sin (x)
eof (f)
sqrt(x)
```

4.5. Anweisungen

4.5.1. Uebersicht

Anweisungen beschreiben auszufuehrende Operationen. Sie koennen durch Marken (Label) gekennzeichnet sein, auf die in Sprunganweisungen (GOTO) Bezug genommen wird. Davon sollte aus softwaretechnologischen Gruenden nur im Ausnahmefall Gebrauch gemacht werden.

Syntax:

```
<Anweisung>::=
        <Marke> : <nichtmarkierte Anweisung>
        | <nichtmarkierte Anweisung>

<Marke>::=
        <natuerliche Zahl> | <Bezeichner>

<nichtmarkierte Anweisung>::=
        <einfache Anweisung>
        | <strukturierte Anweisung>
```

Anweisungen werden durch Semikolon getrennt. Ein Semikolon vor END und UNTIL kann entfallen, da diese Wortsymbole noch zur Anweisung gehoeren. Wird es gesetzt, spezifiziert das Semikolon eine Leeranweisung.

4.5.2. Einfache Anweisungen

Eine einfache Anweisung ist eine Anweisung, in der keine andere Anweisung enthalten ist.

Syntax:

```
<einfache Anweisung>::=
        <Leeranweisung>
        | <Ergibtanweisung>
        | <Prozeduranweisung>
        | <Sprunganweisung>
```

4.5.2.1. Ergibt-Anweisung

Durch die Ergibt-Anweisung wird der rechts von := stehende Ausdruck der Variablen links von := zugewiesen. Innerhalb einer Funktion kann links der Funktionsbezeichner stehen.

Syntax:

```
<Ergibtanweisung>::=
        <Variable> := <Ausdruck>
        | <Funktionsbezeichner> := <Ausdruck>
```

Der Typ der Variablen bzw. der Funktion muss mit dem Typ des Ausdruckes zuweisungsvertraeglich sein.

*** Ergibt-Anweisung ***

Beispiel:

```
x:= Art.Best * Art.Preis [j,8];
y[66]:= x;
y[j]:= 47.88;
Mat [4,41]:= Satz.I;
Tex:= 'Zuweisung';
Zgr1^:= Zgr2^;
Zgr1:=Zgr2;
Zgr2:=NIL;
Zgr1^.Wert1:= 234.645;
x:= x + 10;
i:= succ(i);
```

4.5.2.2. Prozeduranweisung

Durch eine Prozeduranweisung wird die Aktivierung der Prozedur, die durch den Prozedurbezeichner gekennzeichnet ist, veranlasst.
Die Prozeduranweisung kann eine Liste von aktuellen Parametern enthalten, die fuer die korrespondierenden formalen Parameter substituiert werden. Diese formalen Parameter wurden in der Prozedurvereinbarung deklariert.
Die Korrespondenz ist durch die Stellung der Parameter in den Listen der aktuellen bzw. formalen Parameter gegeben.
Es werden Wert-, Variablen- und nichttypisierte Parameter unterschieden (vergl. Ziffer 4.6.2.2.).

Syntax:

```
<Prozeduranweisung>::=
        <Prozedurbezeichner> (<Parameter>{,<Parameter>})
        | <Prozedurbezeichner>

<Parameter>::=
        <Ausdruck>
```

Funktionen und Prozeduren sind als aktuelle Parameter nicht zugelassen.

Beispiel:

```
read  (x);
write ('Bildschirm-Ausgabe');
```

4.5.2.3. Sprunganweisung

Durch die Sprunganweisung wird erreicht, dass die Programmausfuehrung mit der Anweisung fortgesetzt wird, die durch die entsprechende Marke gekennzeichnet ist.

Syntax:

```
<Sprunganweisung>::=
        GOTO <Marke>
```

Der Gueltigkeitsbereich einer Marke ist der Anweisungsteil des Programmtextes, in welchem die Marke deklariert ist.

Beispiel:

```
PROGRAMM xyz;
LABEL 10;
VAR i,j: INTEGER;
BEGIN
10: read (i);
    j:= i * 21 + 5;
     .
     .
     .
    GOTO 10;

END.
```

Die Marken gelten nicht in Unterprogrammen des jeweiligen Blockes.

4.5.2.4. Leeranweisung

Die Leeranweisung enthaelt keinerlei Symbole und hat keine Wirkung.

```
<Leeranweisung>::=
        <leer>
```

Eine Leeranweisung kann ueberall im Programm stehen, wo eine Anweisung geschrieben werden kann.

Beispiel:

```
          .
          .
        IF x<0 THEN GOTO Stop;
        writeln('Das Ergebnis ist ',x);
Stop:END.
```

Die Leeranweisung befindet sich zwischen Doppelpunkt und END.

4.5.3. Strukturierte Anweisungen

Strukturierte Anweisungen sind aus mehreren Anweisungen zusammengesetzte Konstruktionen, die entweder
- nacheinander (Verbundanweisung),
- bedingt (bedingte Anweisungen) oder
- wiederholt (Zyklusanweisungen)

auszufuehren sind.

Syntax:

```
<strukturierte Anweisung>::=
        <Verbundanweisung>
        | <bedingte Anweisung>
        | <Zyklusanweisung>
        | <With-Anweisung>
```

4.5.3.1. Verbundanweisung

Durch die Verbundanweisung wird eine Folge von Anweisungen zusammengefasst. Die Ausfuehrung der geklammerten Anweisungen erfolgt in der gleichen Reihenfolge wie sie geschrieben sind.

Syntax:

```
<Verbundanweisung>::=
        BEGIN
        <Anweisung> {;<Anweisung>}
        END
```

Eine Verbundanweisung kann ueberall dort geschrieben werden, wo eine Anweisung stehen darf, aber eine Folge von Anweisungen erforderlich ist.

Beispiel:

```
BEGIN
  x:= 2.678;
  y[i] := x + 2.71;
END
```

4.5.3.2. Bedingte Anweisungen

Bedingte Anweisungen ermoeglichen die Steuerung der Programmausfuehrung in Abhaengigkeit von Bedingungen. Die Bedingung, als Alternative (IF-Anweisung) oder als Fallunterscheidung (CASE-Anweisung) formuliert, steuert die Auswahl der auszufuehrenden Anweisung.

Syntax:

```
<bedingte Anweisung>::=
        <CASE-Anweisung>
        | <IF-Anweisung>
```

4.5.3.2.1. IF-Anweisung

Durch die IF-Anweisung wird festgelegt, dass die nach THEN folgende Anweisung nur dann ausgefuehrt wird, wenn der boolesche Ausdruck nach IF den Wert TRUE hat. Wenn dieser Ausdruck den Wert FALSE annimmt, dann wird die nach ELSE folgende Anweisung abgearbeitet. Ist kein ELSE-Zweig vorhanden, wird die naechste Anweisung ausgefuehrt.

*** IF-Anweisung ***

Syntax:

```
<IF-Anweisung> ::=
        IF <Ausdruck> THEN <Anweisung> ELSE <Anweisung>
        | IF <Ausdruck> THEN <Anweisung>
```

Bei geschachtelten IF-Anweisungen ist zu beachten, dass ein ELSE-Zweig immer zur letzten IF-Anweisung der Schachtelung gehoert, die noch nicht durch einen ELSE-Zweig abgeschlossen wurde. Gegebenenfalls ist eine Leeranweisung erforderlich.

Nachfolgende IF-Anweisungen sind aequivalent:

```
  (1)   IF <Ausdruck1> THEN
          IF <Ausdruck2> THEN <Anweisung1>
                         ELSE <Anweisung2>;

  (2)   IF <Ausdruck1> THEN BEGIN
          IF <Ausdruck2> THEN <Anweisung1>
                         ELSE <Anweisung2>
        END;
```

Vor ELSE darf kein Semikolon stehen, da sonst die IF-Anweisung vorzeitig abgeschlossen wird.

Beispiel:

```
     (a) IF  x < 2.74 THEN y[i]:= x;
     (b) IF (x > 0) AND (x <= 100) THEN BEGIN
             y[i]:= x;
             x:= 0;
             i:= i+1
         END ELSE writeln('Fehler');
     (c) IF Zgr1^.Nachf <> NIL THEN x:= Zgr1^.Wert1;
```

4.5.3.2.2. CASE-Anweisung

Fuer Programmablaeufe, bei denen unter mehr als zwei Moeglichkeiten zu waehlen ist, steht in PASCAL die CASE-Anweisung zur Verfuegung. Diese Anweisung besteht aus einem Ausdruck (Selektor) und einer Liste von Anweisungen, von denen jede durch eine Liste von Fallkonstanten vom Typ des Selektors markiert ist.
Die CASE-Anweisung legt fest, dass die Anweisung ausgefuehrt wird, bei der eine Fallkonstante mit dem Ausdruck (Selektor) uebereinstimmt.

Syntax:

```
<CASE-Anweisung>::=
        CASE <Ausdruck> OF
        <Auswahllistenelement> {;<Auswahllistenelement>}
        ELSE  <Anweisung>
        END
        |
        CASE <Ausdruck> OF
        <Auswahllistenelement> {;<Auswahllistenelement>}
        END
```

*** CASE-Anweisung ***

```
<Auswahllistenelement>::=
        <Fallkonstantenliste>: <Anweisung>
        | <leer>
<Fallkonstantenliste:=
        <Fallkonstante> {,<Fallkonstante>}
```

Der Ausdruck muss vom ordinalen Typ sein. Entspricht der Wert des Ausdrucks keiner Fallkonstante, dann wird die Anweisung nach ELSE (wenn vorhanden), sonst die nach CASE folgende Anweisung ausgefuehrt.

Beispiel:

```
(a)
{Programmauswahl entsprechend eines Programm-Kennzeichens}
VAR Programmkennzeichen : CHAR;
BEGIN
 .
read (Programmkennzeichen);
CASE Programmkennzeichen OF
 'D','d' : Datenerfassung;
 'F','f' : Fakturierung;
 'B','b' : Buchung;
 'S','s' : Statististik
 ELSE writeln ('Falsches Programm-Kennzeichen!')
END;
 .

(b)
{Summierung von Betraegen zur Quartalssumme}
{(Qs1-Qs4), in Abhaengigkeit von der Monats-Nummer (MNR)}
VAR Qs1, Qs2, Qs3, Qs4,
    Betrag: REAL;
    Monat : INTEGER;
BEGIN
 .
CASE Monat OF
 1,2,3:   Qs1:= Qs1+ Betrag;
 4,5,6:   Qs2:= Qs2+ Betrag;
 7,8,9:   Qs3:= Qs3+ Betrag;
 10,11,12:Qs4:= Qs4+ Betrag
 ELSE writeln ('Ungueltige Monats-Nummer!')
END;
```

4.5.3.3. Zyklusanweisungen

Zyklusanweisungen ermoeglichen die wiederholte Ausfuehrung von bestimmten Anweisungsfolgen.
Wenn die Anzahl der Wiederholungen vorher bekannt ist, dann ist die FOR-Anweisung die schnellste Konstruktion, um dieses Problem zu programmieren. Andernfalls sollte die REPEAT- bzw. WHILE-Anweisung verwendet werden.

Syntax:

```
<Zyklusanweisung>::=
        <WHILE-Anweisung>
        | <REPEAT-Anweisung>
        | <FOR-Anweisung>
```

4.5.3.3.1. WHILE-Anweisung

Die WHILE-Anweisung dient zum Aufbau von Schleifen, die die Ausfuehrung einer Anweisung bereits abweisen, wenn die Bedingung am Anfang nicht erfuellt ist.

Syntax:

```
<WHILE-Anweisung>::=
        WHILE <Ausdruck> DO <Anweisung>
```

Die Anweisung nach DO wird solange (While) wiederholt wie der boolesche Ausdruck nach WHILE den Wert TRUE liefert.
Bei FALSE wird die Schleife verlassen.
Die Anweisung nach DO wird nicht ausgefuehrt, wenn bereits beim Schleifeneintritt der Ausdruck den Wert FALSE liefert.
Der Ausdruck muss im Schleifenkoerper beeinflusst werden, sonst erfolgt eine unendliche Ausfuehrung der Anweisung nach DO.

Beispiel:

```
     (a)
     i:= 1; x:= 0
     WHILE (x < 10000.00) AND (i <= 100) DO BEGIN
     i:= i+1;
     x:= x+y[i]
     END;
     (b)
     {Streichen fuehrender Leerzeichen in einer Zeichenkette}
     WHILE(pos(' ',Kette)=1)AND(Kette<>'')
          DO delete (Kette,1,1);
```

Die WHILE-Anweisung ist in der Ausfuehrung langsamer als die FOR- und die REPEAT-Anweisung.

4.5.3.3.2. REPEAT-Anweisung

Mit der REPEAT-Anweisung besteht die Moeglichkeit zur Programmierung von Schleifen, die in jedem Falle mindestens einmal durchlaufen werden.

Syntax:

```
<REPEAT-Anweisung>::=
        REPEAT
          <Anweisung> {;<Anweisung>}
        UNTIL <Ausdruck>
```

*** REPEAT-Anweisung ***

Die zwischen REPEAT und UNTIL stehenden Anweisungen werden wiederholt, bis (until) der Ausdruck nach UNTIL den Wert TRUE liefert. Im Gegensatz zur WHILE-Anweisung wird die REPEAT-Schleife also verlassen, wenn der boolesche Ausdruck den Wert TRUE liefert. Bei FALSE erfolgt eine weitere Wiederholung. Die REPEAT-Anweisung wird mindestens einmal ausgefuehrt. Der Ausdruck muss im Schleifenkoerper beeinflusst werden, sonst erfolgt eine unendliche Ausfuehrung der Anweisungen zwischen REPEAT und UNTIL.
Eine Klammerung von mehreren Anweisungen im Schleifenkoerper durch BEGIN und END ist nicht notwendig.

Beispiel:

```
     {Erzwingen einer gueltigen Antwort}
     REPEAT
          write ('Waehlen Sie (J/N):');
          readln (Antwort);
          Antwort:=upcase(Antwort)
     UNTIL (Antwort='J') OR (Antwort='N');
```

Die REPEAT-Anweisung ist in der Ausfuehrung schneller als die WHILE-, aber langsamer als die FOR-Anweisung.

4.5.3.3.3. FOR-Anweisung

Die FOR-Anweisung wird zur Programmierung von Zaehlschleifen verwendet.

Syntax:

```
<FOR-Anweisung> ::=
        FOR <Laufvariable> := <Anfangswert> TO <Endwert>DO
                                                <Anweisung>
        |
        FOR <Laufvariable> := <Anfangswert> DOWNTO <Endwert>DO
                                                <Anweisung>

<Laufvariable>::=
        <Variable ordinalen Typs>
<Anfangswert>::=
        <Ausdruck vom ordinalen Typ>
<Endwert>::=
        <Ausdruck vom ordinalen Typ>
```

Beim Schleifeneintritt bekommt die Laufvariable den Anfangswert zugewiesen. Vor Ausfuehrung der Anweisung nach DO wird der Wert der Laufvariablen mit dem vorgegebenen Endwert verglichen. Bei Ueberschreitung des Endwertes wird die Schleife verlassen, ansonsten wird die Anweisung ausgefuehrt.
Nach Ausfuehrung der Anweisung wird die Laufvariable um 1 erhoeht (bei TO) bzw. um 1 verringert (bei DOWNTO).
Ist der Endwert bei Schleifeneintritt bereits ueberschritten (bei TO) bzw. unterschritten (bei DOWNTO), dann wird die

Schleifenanweisung nicht ausgefuehrt.
Die Laufvariable, der Anfangswert und der Endwert muessen vom gleichen ordinalen Typ sein. Sie koennen im Schleifenkoerper wertmaessig genutzt, duerfen aber nicht veraendert werden.
Der Wert der Laufvariablen nach vollstaendigem Durchlauf der Schleife bei Schleifenaustritt ist undefiniert. Fuer Laufvariablen duerfen nur lokale Variablen verwendet werden.

Beispiel:

```
(a) VAR Summe  : ARRAY [1..100] OF REAL;
        Artikel: INTEGER;
      .
      .
      .
    FOR Artikel:=1 TO 100 DO
    writeln ('ART: ',Artikel:3,'=',Summe[Artikel]:8:2);

(b) VAR Kette : STRING[40];
        i     : INTEGER;
      .
      .
    writeln (Kette);
    FOR i:= 1 TO length (Kette) DO write ('-');
    writeln;

(c) FOR j:= 1 TO n DO BEGIN
    x:= 0;
    FOR k:= 1 TO n DO x:= x+y[k]*k
    END;

(d) VAR c: (rot, gelb, gruen, blau);
    .
    .
    FOR c:= rot TO blau DO Proz(c);

(e) x:= 0; j:= 100;
    FOR i:= j DOWNTO 1 DO BEGIN
    x:= x+y[i];
    IF  x > 1000.0 THEN exit       {vorzeitiger Austritt}
    END;
```

4.5.3.4. WITH-Anweisung

Innerhalb der WITH-Anweisung koennen die Recordkomponentenvariablen, die durch die WITH-Klausel spezifiziert sind, allein durch den Recordkomponentenbezeichner angegeben werden, d.h. ohne die Angabe der Recordvariablen voranzustellen.

Syntax:

```
<WITH-Anweisung>::=
        WITH <Recordvariablenliste> DO <Anweisung>
<Recordvariablenliste>::=
        <Recordvariable> {,<Recordvariable>}
```

*** WITH-Anweisung ***

Beispiel:

```
(a) TYPE Daten=RECORD
                 Adresse: STRING [100];
                 Konto:   STRING [15];
                 Umsatz:  REAL;
                 Datum:   STRING [8]
               END;
    VAR KUNDE: Daten;

 Die nachfolgenden Anweisungen sind aequivalent:
(1) Kunde.Adresse:= 'Lampen-Mueller, 50 Erfurt, Am Hang 4';
    Kunde.Konto  := '4444-46-1100';
    Kunde.Umsatz :=  6000.00;
    Kunde.Datum  := '12.12.84';
(2) WITH Kunde DO BEGIN
      Adresse:= 'Lampen-Mueller, 50 Erfurt, Am Hang 4');
      Konto  := '4444-46-1100';
      Umsatz :=  6000.00;
      Datum  := '12.12.84'
    END;
(b) TYPE Person=: RECORD
                   Mitarbeiter= RECORD
                                  Name, Ort, Str: STRING [20];
                                  Gdat: STRING [8]
                                  Verh: BOOLEAN
                                END;
                 .
                 .
    VAR Angest, Arb, Lehrl: Person;
    WITH Angest, Mitarb DO BEGIN
      Name:= 'Paul Meyer';
      Ort := '5230 Soemmerda';
      Str := 'Park-Str. 5';
      Gdat:= '12.10.46';
      Verh:= TRUE
    END;
```

4.6. Nutzerdefinition

4.6.1. Deklaration von Prozeduren und Funktionen

4.6.1.1. Prozedurkopf und -block

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.

Syntax:

```
<Prozedurkopf>::=
        PROCEDURE <Prozedurbezeichner> <Parameterliste>
      | PROCEDURE <Prozedurbezeichner>
```

<Parameterliste> wird in Ziffer 4.6.2.2. definiert.
Der Prozedurblock besteht aus einem Deklarationsteil und einem Anweisungsteil.

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 aeusseren Block definierte Konstante, Type, Variable, Prozedur oder Funktion verwenden.
Der Anweisungteil spezifiziert die Operationen, die ausgefuehrt werden sollen, wenn die Prozedur aufgerufen wird. Er hat die Form einer Verbundanweisung und endet also mit einem Semikolon. Wird der Prozedurbezeichner selbst innerhalb des Anweisungsteiles verwendet, wird die Prozedur rekursiv ausgefuehrt.

4.6.1.2. Funktionskopf und -block

Der Funktionskopf ist mit dem Prozedurkopf aequivalent, ausser dass der Funktionskopf mit dem reservierten Wort FUNCTION eroeffnet wird und dass auch der Typ des Ergebnisses mit definiert werden muss. Dies wird durch Anfuegung eines Doppelpunktes und eines Types an den Funktionskopf erreicht.

Syntax:

```
<Funktionskopf>::=
      FUNCTION <Funktionsbezeichner><Parameterliste>
                                              :<Ergebnistyp>
    | FUNCTION <Funktionsbezeichner>:<Ergebnistyp>
```

<Parameterliste wird in Ziffer 4.6.2.2. definiert.

Der Ergebnistyp einer Funktion muss ein einfacher Typ (d.h. INTEGER, REAL, BOOLEAN, CHAR), ein Stringtyp oder ein Zeigertyp sein.

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

Der Anweisungsteil einer Funktion ist eine Verbundanweisung. Innerhalb des Anweisungsteiles muss wenigstens eine Ergibtanweisung auftreten, die dem Funktionsbezeichner einen Wert zuweist. Die letzte 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.

Bei der Definition eines Funktionstyps ist zu beachten, dass ein als Parameter oder Ergebnistyp verwendeter strukturierter Typ vorher als Typbezeichner erklaert sein muss.
Aus diesem Grunde ist die folgende Konstruktion nicht erlaubt:

```
FUNCTION Kette(Zeile: Linie) : 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 Kette(Zeile: Linie) : 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.

4.6.2. Datenaustausch

4.6.2.1. Blockkonzept

PASCAL ist eine blockorientierte Sprache. Blockorientiert bedeutet, dass alle definierten und deklarierten Objekte, also Konstanten, Typen, Variablen und Unterprogramme in einem gesamten Block gueltig sind, in dem sie vereinbart (eingefuehrt) wurden. Eine Ausnahme bilden lediglich Marken. In den eingelagerten Prozeduren und Funktionen koennen also alle Objekte ohne eigene Deklaration benutzt werden, die im uebergeordneten Block enthalten sind. Solche Objekte nennt man deshalb global.
Eine Kollision wuerde entstehen, wenn im Vereinbarungsteil eines Unterprogramms ein Objekt unter einem Bezeichner deklariert wird, der in der uebergeordneten Programmeinheit bereits benutzt wurde. Entsprechend dem Blockkonzept waere er auch im Unterprogramm noch gueltig. PASCAL legt fest, dass in diesem Falle die (lokale) Deklaration im Unterprogramm die globale Gueltigkeit aufhebt, natuerlich nur lokal fuer den Block dieses Unterprogramms und auch genau nur fuer dieses Objekt.
Man beachte, dass sich die Gueltigkeit entsprechend dem Blockkonzept nur "nach innen", also vom Globalen zum Lokalen hin oeffnet. Die in einem Unterprogramm definierten und deklarierten Objekte - bei Variablen spricht man von lokalen Varia-

bleh - sind fuer die uebergeordnete Programmeinheit nicht gueltig, und es kann auch nicht auf sie zugegriffen werden. Das Blockkonzept geht aus dem folgenden Programmausschnitt hervor:

```
PROGRAM Hauptprogramm;
.
TYPE Feld = ARRAY[1..20] OF CHAR;
VAR  x,y  : Feld;
     i,j,k: INTEGER;
PROCEDURE Prozedur;
  VAR z : Feld;
      i : INTEGER;
  {Gueltig auch x,y,j,k}
   .
  END;
  FUNCTION Funktion : Typ;
  VAR a   : Feld;
      j,z : INTEGER;
  {Gueltig auch x,y,i,k}
   .
  END;
  BEGIN {Hauptprogramm}
  {Gueltig sind x,y,j,i,k}
  .
  END.
```

Die Gueltigkeit der globalen Variablen ist als Kommentar eingefuegt.
Das Blockkonzept regelt nicht nur die Gueltigkeit von Bezeichnern innerhalb des Programmtextes. Es legt auch fest, dass lokale Variablen beim Verlassen eines Unterprogramms ihren Wert verlieren.

4.6.2.2. 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 zu versehen, so dass 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 Parametern 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, Wertparameter) und Uebergabe der Parameter durch Substitution der Adressen (Referenz, Variablenparameter). Hier sind ausserdem typlose Parameter erlaubt.

Die Stacktechnik beim Parameteraustausch ist im Anhang D. beschrieben.
Syntax:

```
<Parameterliste>::=
    <formaler Parameter>{;<formaler Parameter>}

<formaler Parameter>::=
        VAR <Segment>
<Segment>::=
      <Parameter>{,<Parameter>}:<Typ>
    | <Parameter>{,<Parameter>
<Parameter>::= <Variablenbezeichner>
```

4.6.2.2.1. Variablenparameter

Bei Variablenparametern wird die Adresse des aktuellen Parameters an die Prozedur uebergeben (call by reference). Dabei arbeiten Prozedur und rufendes Programm mit der gleichen Variablen, so dass eine Uebermittlung von Ergebnissen moeglich ist, z.B.

```
PROCEDURE Test (VAR Fehler: BOOLEAN);
```

Kennzeichen fuer Variablenparameter ist das VAR im Segment.

4.6.2.2.2. Wertparameter

Bei Werteparametern wird der Wert des aktuellen Parameters (aus der Prozeduranweisung) in den formalen Parameter der Prozedur uebertragen (call by value).
Eine Rueckgabe von Ergebnissen ist nicht moeglich, z.B.

```
PROCEDURE Kombination (a: REAL; b: INTEGER);
```

Hier fehlt das VAR im Segment.

4.6.2.2.3. Ungetypte Variablenparameter

Ist der Typ eines Parameters nicht definiert, d.h., enthaelt der Parameterteil im Kopf des Unterprogrammes keine Typdefinition, dann wird der Parameter als ungetypt bezeichnet. Der aktuelle Parameter kann dann ein beliebiger Typ sein. Aus diesem Grunde kann man ungetypte 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 Adress-Spezifikationen von absoluten Variablen der Fall.
Im folgenden Beispiel wird bei der Prozedur Schalter die Verwendung ungetypter Parameter demonstriert. Sie uebertraegt den Inhalt der Variablen a1 nach a2 und von a2 nach a1.

*** Ungetypte Variablenparameter ***

Beispiel:

```
PROCEDURE Schalter(VAR a1p,a2p;·Anzahl : INTEGER);
TYPE
     a = ARRAY[1..Max] OF BYTE;
VAR
     a1      : a ABSOLUTE a1p;
     a2      : a ABSOLUTE a2p;
     Temp    : BYTE;
     Zaehler : INTEGER;
BEGIN
     FOR Zaehler := 1 TO Anzahl DO
     BEGIN
          Temp         := a1[Zaehler];
          a1[Zaehler] := a2[Zaehler];
          a2[Zaehler] := Temp;
     END;
END;
```

Definiert man:

```
TYPE Matrix = ARRAY[1..50,1..25] OF REAL;
VAR  TestMatrix,BestMatrix : Matrix;
```

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

```
SCHALTER(TestMatrix,BestMatrix,Umfang(TestMatrix));
```

4.6.3. FORWARD-Deklaration

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, usw., d.h. nur mit dem Namen. Die FORWARD-Deklaration ist nicht fuer OVERLAY-Unterprogramme erlaubt.

Beispiel:

```
PROCEDURE xyz(VAR a:REAL; b:CHAR); FORWARD;
```

4.6.4. EXTERNAL-Deklaration

Das reservierte Wort EXTERNAL wird zur Definition externer Prozeduren und Funktionen verwendet. Typisch ist die Verwendung fuer in Maschinencode geschriebene Prozeduren oder Funktionen. Ein externes Unterprogramm hat keinen Block, d.h. keinen Deklarationsteil und keinen Anweisungsteil. Es wird nur der Unterprogrammkopf spezifiziert, dem unmittelbar das reservierte Wort EXTERNAL und der Dateiname folgen.

Beispiel:

```
PROCEDURE DiskReset; EXTERNAL 'demo';
FUNCTION  Iostatus : BOOLEAN; 'stat';
```

Die externe Datei kann auch fuer mehrere Unterprogramme Codes enthalten. Das erste Programm wird wie im obigen Beispiel deklariert. Die weiteren Unterprogramme werden durch den Bezeichner des ersten Unterprogramms und durch eine in eckige Klammern geschriebene integer Konstante als Offset deklariert.

Beispiel:

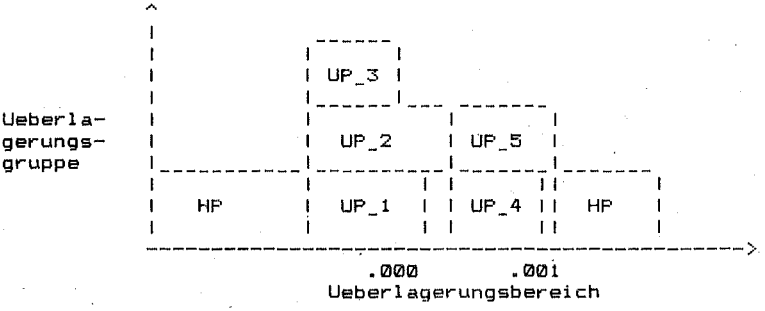
```
PROCEDURE DiskReset; EXTERNAL 'demo';
PROCEDURE Disk1; EXTERNAL Diskreset[15];
PROCEDURE Disk2; EXTERNAL Diskreset[35];
```

Parameter koennen an externe Unterprogramme uebergeben werden. Die Syntax ist dabei exakt die gleiche wie bei normalen Prozedur- oder Funktionsaufrufen:

```
PROCEDURE Kreis(m1,m2,r:INTEGER); EXTERNAL 'demo';
FUNCTION  Sortier(Var L:LTYP; A:INTEGER); EXTERNAL 'abc';
```

4.6.5. Overlay-Strukturen

Bei Programmen, die die Kapazitaet des Hauptspeichers uebersteigen, kann eine Ueberlagerungsstruktur fuer die Unterprogramme erzeugt werden. Die Hauptspeicherbelegung in einem solchen Falle zeigt das folgende Schema.



Das Prinzip besteht darin, nichtaktive Unterprogramme einer Ueberlagerungsgruppe auf den externen Datenspeicher auszulagern. Erst im Falle ihres Aufrufes werden sie in den Hauptspeicher transportiert. Sie ueberschreiben ein eventuell vorher aktives Unterprogramm derselben Ueberlagerungsgruppe. Der Vorteil besteht darin, dass der Ueberlagerungsbereich fuer eine Ueberlagerungsgruppe nur so gross sein muss wie das groesste Unterprogramm dieser Gruppe. Die moegliche Einsparung an Hauptspeicherkapazitaet kann betraechtlich sein. Auf diese Weise ist es moeglich, grosse Programme auf Rechnern mit relativ kleiner Hauptspeicherkapazitaet auszufuehren.

Die technische Realisierung einer Ueberlagerungsstruktur erfolgt waehrend des einheitlichen Compiler-/Linkervorgangs. Dabei entsteht je Ueberlagerungsgruppe ein Ueberlagerungsfile, das sofort auf einen externen Datentraeger ausgelagert wird. Es enthaelt den Maschinencode aller zur Gruppe gehoerenden Unterprogramme. Die Ueberlagerungsfiles erhalten den Filenamen des Hauptprogramms und eine Nummer 000, 001 ... als Filenamenserweiterung.
Dem PASCAL-System muss mitgeteilt werden, dass eine Ueberlagerungsstruktur erforderlich ist und welche Unterprogramme eine Ueberlagerungsgruppe bilden sollen. Dabei ist zu beachten, dass sich Unterprogramme der gleichen Gruppe niemals untereinander rufen, aktivieren koennen. Sonst gibt es fuer die Bildung der Gruppen noch die folgenden Hinweise:

a) Annaehernd gleich grosse Unterprogramme sollten in eine Gruppe aufgenommen werden. Die Einsparung an Hauptspeicherkapazitaet ist dann besonders gross.
b) Haeufig aktive Programme sollten verschiedenen Gruppen zugeordnet werden. Dadurch wird der Zeitverzug, der durch das staendige Laden der aktivierten Unterprogramme entsteht, geringer.
c) In OVERLAY-Unterprogrammen ist keine Rekursion erlaubt.

Fuer den Aufbau einer Ueberlagerungsstruktur gilt folgender Verfahrensweg. Ein Unterprogramm, das Bestandteil einer Ueberlagerungsstruktur werden soll, erhaelt vor dem Schluesselwort PROCEDURE oder FUNCTION den Zusatz OVERLAY.
Alle aufeinanderfolgenden Unterprogramme mit dem Schluesselwort OVERLAY bilden eine Ueberlagerungsgruppe. Die Gruppe gilt als abgeschlossen, wenn ein folgendes Unterprogramm kein OVERLAY enthaelt. Folgt nach diesem Unterprogramm ohne OVERLAY wieder ein Unterprogramm mit OVERLAY, so wird eine neue Ueberlagerungsgruppe eroeffnet. Da die Reihenfolge der Unterprogrammdeklarationen, gegebenenfalls mit FORWARD, vom Programmierer frei gewaehlt werden kann und auch "leere" (Pseudo-) Unterprogramme deklariert werden koennen, ist auf diese Art eine einfache, aber vollstaendige Mitteilung an das TPASCAL moeglich. Overlay-Unterprogramme koennen auch geschachtelt werden. Da solche Programmeinheiten sich dann gegenseitig rufen, wird eine Ueberlagerungsgruppe eroeffnet.

Beispiel:

```
    :
  OVERLAY PROCEDURE UP_1;          {Overlay:  .000}
  BEGIN
    :
  END;
  OVERLAY PROCEDURE UP_2;
  BEGIN
    :
  END;
  OVERLAY PROCEDURE UP_3;
  BEGIN
    :
  END;
  PROCEDURE tab (anz:INTGER);      {Prozedur zur Trennung}
  BEGIN                            {oder zwei Overlay-Gebiete}
    :
  END;
  OVERLAY PROCEDURE UP_4           {Overlay:  .001}
  BEGIN
    :
  END;

  BEGIN
    :                              {Hauptprogramm}
END.
```

4.7. Standard-Prozeduren und -Funktionen (ohne Filearbeit und Pointer)

Die nachfolgenden Standardprozeduren bzw. -funktionen werden getrennt nach ihren Anwendungsbereichen dargestellt:
- STRING-Prozeduren und -funktionen
- arithmetische Standardfunktionen
- Skalarfunktionen
- Konvertierungsfunktionen (ohne Pseudofunktionen ORD, CHR, PTR und Retyping)
- bildschirmorientierte Prozeduren
- sonstige Prozeduren und Funktionen.

Nachfolgend werden folgende Abkuerzungen verwendet:

<Quelle>	= Zeichenkette oder ARRAY-OF-CHAR
<Ziel>	= Zeichenkette
<Anzahl>	= INTEGER-Ausdruck / BYTE-Ausdruck
<Kette>	= Zeichenkette(STRING)
<Zeichen>	= CHAR-Variable/ - Konstante
<Integer>	= INTEGER-Ausdruck
<Real>	= REAL-Ausdruck
<Ordinale>	= Ordinal-Typ (INTEGER,CHAR,BYTE,BOOLEAN)
<Position>	= INTEGER-Ausdruck / BYTE-Ausdruck
<x>	= INTEGER-Ausdruck / REAL-Ausdruck

4.7.1. STRING-Funktionen und -Prozeduren

4.7.1.1. CONCAT-Funktion

Syntax:

concat (<Quelle> {,<Quelle>})

Die Funktion CONCAT (Typ STRING) liefert einen STRING, der die Zusammenfuegung der STRING's enthaelt.
Wenn die gesamte Laenge 256 Bytes uebersteigt, entsteht ein Laufzeitfehler. Man kann mit dem "+"-Operator das gleiche erhalten. CONCAT sichert nur die Kompatibilitaet mit anderen Compilern. Die Quellen koennen STRING-Variablen, ARRAY-OF-CHAR-Variablen, STRING-Konstanten oder Zeichen (CHAR) sein.

Beispiel:

```
PROCEDURE Conc;
VAR A,B: STRING[30];
BEGIN
  A:= 'Sprachbeschreibung ';
  B:= '1987';
  writeln (concat (A,'T','PASCAL ',B))
END;       {Ausgabe: Sprachbeschreibung TPASCAL 1987}
```

4.7.1.2. COPY-Funktion

Syntax:

copy (<Kette>,<Position>,<Anzahl>)

Diese Funktion (Typ STRING) liefert aus dem STRING <Kette> ab <Position> einen Teil-STRING in der Laenge <Anzahl>.

Wenn <Position> groesser als Laenge <Kette> ist, besteht das Ergebnis aus der leeren Zeichenkette ''.
Wenn <Position> + <Anzahl> ausserhalb von <Kette> liegt, werden nur die innerhalb von <Kette> liegenden Zeichen zurueckgegeben. Liegt <Position> nicht in 1..255, so entsteht ein Laufzeitfehler.

Beispiel:

```
PROCEDURE Cop;
VAR A: STRING[80];
BEGIN
  A:='Zeichenkettenfeld';
  writeln (copy(A,8,5))                    {Ausgabe: kette}
END;
```

4.7.1.3. DELETE-Prozedur

Syntax:

delete (<Kette>,<Position>,<Anzahl>)

In <Kette> werden ab <Position> <Anzahl> Zeichen geloescht (und verdichtet).
Die Parameter <Position> und <Anzahl> sind vom Typ INTEGER.
Wenn <Position> groesser als die Laenge von <Kette> ist, wird kein Zeichen geloescht. Wenn <Position>+<Anzahl> ausserhalb der Zeichenkette liegt, werden nur die Zeichen geloescht, die ab <Position> innerhalb liegen. Liegt <Position> nicht in 1..255, wird ein Laufzeitfehler erzeugt.

Beispiel:

```
PROCEDURE Del;
VAR TX: STRING[50];
BEGIN
  TX:= 'Programmierung in PASCAL';
  delete (TX,9,9);
  writeln (TX)                  {Ausgabe: Programm PASCAL}
END;
```

4.7.1.4. INSERT-Prozedur

Syntax:

insert (<Quelle>,<Ziel>,<Position>)

*** INSERT-Prozedur ***

Die Prozedur INSERT fuegt in den Ziel-STRING <Ziel> an der Position <Position> den Quell-STRING <Quelle> ein.
Als Quelle sind Konstanten oder Variablen vom Typ STRING und CHAR zugelassen.
Ist <Position> groesser als die Laenge von <Ziel> wird <Quelle> an <Ziel> angefuegt. Wenn das Ergebnis laenger als die maximale Laenge von <Ziel> ist, werden die ueberstehenden Zeichen abgeschnitten und <Ziel> erhaelt nur die links stehenden. Wenn <Position> ausserhalb von 1..255 liegt, entsteht ein Laufzeitfehler.

Beispiel:

```
PROCEDURE Ins;
VAR A: STRING[80];
    B: STRING[20];
BEGIN
  A:= 'Kombinat ROBOTRON Soemmerda';
  B:= 'Bueromaschinenwerk ';
  insert (B,A,19);
  writeln (A)              {Ausgabe: Kombinat ROBOTRON Buero-}
END;                                 {maschinenwerk Soemmerda}
```

4.7.1.5. LENGTH-Funktion

Syntax:

```
length  (<Kette>)
```

Diese Funktion liefert die Laenge des STRING <Kette> als INTEGER-Wert.

Beispiel:

```
PROCEDURE Len;
VAR A: STRING[40];
BEGIN
  A:= 'Erfurt';
  writeln (length(A),'/',length('SOEMMERDA'));
END;                                              {Ausgabe: 6/9}
```

4.7.1.6. POS-Funktion

Syntax:

```
pos  (<Kette>,<Quelle>)
```

Diese Funktion liefert die Position des 1.Auftretens von <Kette> im STRING <Quelle> als INTEGER-Wert.
Wenn <Kette> nicht in <Quelle> gefunden wird, dann liefert die Funktion den Wert 0.
Fuer <Kette> ist eine Konstante oder Variable vom Typ STRING oder CHAR zugelassen. ARRAY-OF-CHAR wird wie ein String fester Laenge behandelt.

*** POS-Funktion ***

Beispiel:

```
PROCEDURE Posf;
VAR A,B: STRING[30];
BEGIN
  A:= 'Standardfunktion';
  B:= 'fun';
  writeln (pos(B,A),'/', pos('a',A),'/', pos('xy',A))
END;                                        {Ausgabe: 9/3/0}
```

4.7.1.7. STR-Prozedur

Syntax:

```
str     (<x>,<Kette>)
```

Die STR-Prozedur konvertiert den numerischen Wert von <x> (INTEGER bzw. REAL-Typ) in eine Zeichenkette und speichert sie in <Kette> ab. Die Konvertierung kann durch die von writeln/ write bekannten Formatparameter gesteuert werden.

Beispiel:

```
PROCEDUR strt;
VAR  i:INTEGER;
     j:REAL;
     zk1,zk2:STRING[10];
BEGIN
  i:=1234;
  j:=2.5E4;
  str(i:5,zk1);          {zk1 ='  1234'}
  str(j:10:0,zk2)        {zk2 ='      25000'}
END;
```

4.7.1.8. VAL-Prozedur

Syntax:

```
val    (<Kette>,<x>,<Code>);
```

Der STRING-Ausdruck <Kette> muss den Regeln einer numerischen Konstanten genuegen. Weder fuehrende noch nachfolgende Leerzeichen sind erlaubt. Die Prozedur VAL konvertiert die Konstante zu einem Wert vom gleichen Typ wie <x>(INTEGER-/REAL-Typ) und speichert diesen Wert in <x> ab. Wird kein Fehler festgestellt, ist der Wert der Variablen <Code>=0. Andernfalls erhaelt <Code> den Wert der Position des ersten fehlerhaften Zeichens in <Kette> und der Wert von <x> ist undefiniert.

Beispiel:

```
str1 = '234'
val(str1,I,Result);      { I = 234        Result = 0 }
str2 = ' 234'
val(str2,I,Result);      { I = undef.     Result = 1 }
str3 = '2.5E4'
val(str3,r,Result);      { r = 25000      Result = 0 }
```

4.7.1.9. Bibliotheksprozeduren

Mit TPASCAL kann die Directory-Baumstruktur manipuliert werden.

4.7.1.9.1. CHDIR-Prozedur

Syntax:

```
CHDIR (<String>);
```

Die aktuelle Directory wird in den durch <String> definierten Ausdruck geaendert.

4.7.1.9.2. MKDIR-Prozedur

Syntax:

```
MKDIR (<String>);
```

Mit dem durch <String> definierten Ausdruck ist es moeglich, eine neue Subdirectory anzulegen. Der <String> deklariert den Pfad und den Namen. Der letzte Name darf noch nicht vorhanden sein.

4.7.1.9.3. RMDIR-Prozedur

Syntax:

```
RMDIR (<String>);
```

Mit dem durch <String> definierten Ausdruck ist es moeglich, eine Subdirectory zu loeschen.

4.7.1.9.4. GETDIR-Prozedur

Syntax:

```
GETDIR (<Integer>, <String>);
```

Von dem durch <Integer> festgelegten Laufwerk wird in <String> die Bezeichnung der aktuellen Directory eingegeben. Dabei bedeutet 1=A, 2=B, 3=C usw...

4.7.2. Arithmetische Funktionen

4.7.2.1. ABS-Funktion

Syntax:

```
abs  (<x>)
```

*** ABS-Funktion ***

Die Funktion liefert den Absolutwert der INTEGER- oder REAL-Zahl <x>.

Das Ergebnis ist vom gleichen Typ wie das Argument.

Beispiel:
```
write (abs (-3.21));        {= 3.21}
write (abs (-127));         {= 127}
write (abs (18.1299));      {= 18.1299}
```

4.7.2.2. ARCTAN-Funktion

Syntax:

arctan (<x>)

Die Funktion liefert den Arcustangens von <x> als reelle Zahl.
Das Argument <x> ist im Bogenmass anzugeben.

Beispiel:
```
write (arctan (1));             {= 7.85398 E-01   (0.7854)}
write (arctan (1.222));         {= 8.84977 E-01   (0.8850)}
```

4.7.2.3. COS-Funktion

Syntax:

cos (<x>)

Die Funktion liefert den Cosinus von <x> als reelle Zahl.
Das Argument <x> ist im Bogenmass anzugeben.

Beispiel:
```
write (cos (1));                {= 5.40302 E-01   (0.5403)}
write (cos (1.4444));           {= 1.26061 E-01   (0.1261)}
```

4.7.2.4. EXP-Funktion

Syntax:

exp (<x>)

Die Funktion liefert die Exponentialfunktion e^x als reelle Zahl.

Beispiel:
```
write (exp (8));                {= 2.98094 E+03   (2980.94)}
write (exp (-12.5555));         {= 3.52547 E-06}
```

4.7.2.5. FRAC-Funktion

Syntax:

```
frac   (<Real>)
```

Die Funktion ermittelt den gebrochen Teil von <Real>.
Das Ergebnis ist vom Typ REAL.

Beispiel:

```
write(frac (123.37));                    {=0.37}
```

4.7.2.6. INT-Funktion

Syntax:

```
int    (<Ausdruck>)
```

Die Funktion ermittelt den ganzen Teil von <Ausdruck>.
Ausdruck ist vom Typ INTEGER oder REAL. Das Ergebnis ist je nach Argument vom Typ INTEGER oder REAL.

Beispiel:

```
write(int (5.27));          {= 5}
r:= int(5);                 {Wenn r=REAL dann 5.}
```

4.7.2.7. LN-Funktion

Syntax:

```
ln  (<x>)
```

Die Funktion liefert den natuerlichen Logarithmus von <x> als reelle Zahl.

Beispiel:

```
write (ln (127));
write (ln (18.5555));
```

4.7.2.8. SIN-Funktion

Syntax:

```
sin  (<x>)
```

Die Funktion liefert den Sinus von <x> als reelle Zahl. Das Argument <x> ist im Bogenmass anzugeben.

Beispiel:

```
write (sin (1));
write (sin (1.684));
```

4.7.2.9. SQR-Funktion

Syntax:

```
sqr  (<x>)
```

Die Funktion liefert das Quadrat von x. Das Argument kann vom Typ INTEGER oder REAL sein. Das Ergebnis ist gleich dem Typ von x.

Beispiel:

```
write (sqr (9.0000));
write (sqr (-12));
```

4.7.2.10. SQRT-Funktion

Syntax:

```
sqrt  (<x>)
```

Die Funktion liefert die Quadratwurzel der Zahl <x> (REAL, INTEGER). Das Ergebnis ist vom Typ REAL.

Beispiel:

```
write (sqrt (100));
```

4.7.3. Skalarfunktionen

4.7.3.1. PRED-Funktion

Syntax:

```
pred    (<Ordinale>)
```

Die Funktion pred liefert den Vorgaenger von <Ordinale>.
<Ordinale> ist vom ordinalen Typ.
i:= pred(i) ist schneller als i:= i - 1.

Beispiel:

```
write(pred('C');              { = B}
k:=5; write(pred(k));         { = 4}
```

4.7.3.2. SUCC-Funktion

Syntax:

```
succ   (<Ordinale>)
```

Die Funktion liefert den Nachfolger von <Ordinale>
<Ordinale> ist vom ordinalen Typ.
i:= succ(i) ist schneller als i:= i + 1.

Beispiel:
```
write(succ('H'));              { = I}
k:= 29; write(succ(k));        { = 30}
k:= succ(k);                   { k = 30}
```

4.7.3.3. ODD-Funktion

Syntax:

odd (<Integer>)

Die Funktion liefert den booleschen Wert des Ausdrucks <Integer> mod 2 <> 0, d.h., fuer geradzahlige INTEGER-Werte liefert die Funktion FALSE, fuer ungerade Werte TRUE.

Beispiel:
```
PROCEDUR od;
VAR i:INTEGER;
BEGIN
  readln(i);
  IF odd(i) THEN writeln('I = ungerade Zahl')
            ELSE writeln('I = gerade Zahl')
END;
```

4.7.4. Konvertierungsfunktionen (ohne Pseudofunktionen)

Die Pseudofunktionen der Konvertierung CHAR, ORD, PTR und das Retyping sind in Ziffer 4.3.3.5.2. dargestellt.

4.7.4.1. ROUND-Funktion

Syntax:

round (<Real>)

Die Funktion liefert die ganzzahlige Rundung (INTEGER) der reellen Zahl <Real>.
Die Funktion aehnelt der Funktion TRUNC. Das Ergebnis wird hier jedoch auf die naechste ganze Zahl auf- oder abgerundet.
Betraegt der gebrochene Teil genau 0.5, dann wird bei positiven Zahlen auf- und bei negativen Zahlen abgerundet.

Beispiel:
```
write (round (1.463));                 {= 1}
write (round (12.864));                {= 13}
write (round (-127.3468));             {= -127}
```

4.7.4.2. TRUNC-Funktion

Syntax:

```
trunc  (<Real>)
```

Die Funktion liefert den ganzzahligen Teil (INTEGER) der reellen Zahl <Real>. Der Realteil wird abgeschnitten.

Beispiel:

```
write (trunc (31.6781));                    {= 31}
write (trunc (-6.18));                      {= -6}
```

4.7.5. Bildschirmorientierte Prozeduren

4.7.5.1. CLREOL-Prozedur

Syntax:

```
clreol
```

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

4.7.5.2. CLRSCR-Prozedur

Syntax:

```
clrscr
```

Diese Prozedur loescht den Bildschirm und setzt den Cursor in die linke obere Ecke.

4.7.5.3. DELLINE-Prozedur

Syntax:

```
delline
```

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

4.7.5.4. INSLINE-Prozedur

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.

4.7.5.5. GOTOXY-Prozedur

Syntax:

```
gotoxy   (<xpos>,<ypos>)
```

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

4.7.5.6. WHEREX-Funktion

Syntax:

```
wherex;
```

Die Funktion Wherex liefert die aktuelle x-Koordinate des Cursors.

4.7.5.7. WHEREY-Funktion

Syntax:

```
wherey;
```

Die Funktion Wherey liefert die aktuelle y-Koordinate des Cursors.

4.7.5.8. WINDOW-Prozedur

Mit der Prozedur Window ist es moeglich jeden Bereich des Bildschirms als Fenster festzulegen.

Syntax:

```
window (<XPos1>, <YPos1>, <XPos2>, <YPos2>)
```

Diese Prozedur legt ein Fenster durch zwei Diagonalpunkte fest (Linksoben: <XPos1> und <YPos1>, Rechtsunten: <XPos2> und <YPos2>).
Beachten muss man die minimale Fenstergroesse von 2 Spalten und 2 Zeilen. Nach der Definition beziehen sich die Bildschirmkoordinaten relativ auf dieses Fenster.
Der Bereich ausserhalb steht nicht mehr zur Verfuegung. Das Fenster verhaelt sich so wie vor der Definition der ganze Bildschirm. Es ist somit moeglich Texte in diesem Fenster rollen zu lassen oder Zeilen einzufuegen oder zu loeschen.

4.7.5.9. TEXTMODE-Prozedur

Die Prozedur Textmode ermoeglicht das Arbeiten in vier unterschiedlichen Betriebsarten.

Beispiel:
```
Textmode (BW40);  schwarz-weiss Darstellung mit 40 Zeichen/Zeile
Textmode (BW80);  schwarz-weiss Darstellung mit 80 Zeichen/Zeile
Textmode (C40);   Farbdarstellung mit 40 Zeichen/Zeile
Textmode (C80);   Farbdarstellung mit 80 Zeichen/Zeile
Textmode;
```

Die verwendeten Konstanten sind vom System vordefiniert und haben folgenden Wert:

```
BW40 = 0
BW80 = 2
C40  = 1
C80  = 3
```

4.7.5.10. Farbdarstellung

4.7.5.10.1. Farbvarianten

In dieser Betriebsart kann jedes Zeichen aus einer Menge von 16 Farben gewaehlt werden. Der Hintergrund ist aus einer Menge von 8 Farben waehlbar. Fuer die Programmunterstuetzung sind die Integerkonstanten 0 bis 15 vordefinert.

0: Blank	(Schwarz)	8: DarkGray	(Dunkles Grau)
1: Blue	(Blau)	9: LightBlue	(Helles Blau)
2: Green	(Gruen)	10: LightGreen	(Helles Gruen)
3: Cyan	(Tuerkis)	11: LightCyan	(Helles Tuerkis)
4: Red	(Rot)	12: LightRed	(Helles Rot)
5: Magenta	(Violet)	13: LightMagenta	(Pink)
6: Brown	(Braun)	14: Yellow	(Gelb)
7: LightGray	(Helles Grau)	15: White	(Weiss)

Fuer die Hintergrunddarstellung kann man eine Farbe aus dem Bereich 0 bis 7 waehlen. Fuer die Zeichendarstellung sind alle Werte im Bereich von 0 bis 15 zugelassen.

4.7.5.10.2. TEXTCOLOR-Prozedur

Syntax:

```
textcolor (<Integer>);
```

Mit der Prozedur Textcolor kann man die Farbe der Zeichen festlegen.
Es gibt die Moeglichkeit diese Zeichen blinken zu lassen. Dieses wird durch die Addition von 16 (vordefinierte Konstante Blink) zum Farbwert erreicht.

Beispiel:

```
Textcolor (4);              Festlegung der Farbe Rot fuer das
                            Zeichen
Textcolor (White);          Festlegung der Farbe Weiss fuer
                            das Zeichen
Textcolor  (Blue + Blink);Festlegung als blaues blinkendes
                            Zeichen
```

4.7.5.10.3. TEXTBACKGROUND-Prozedur

Syntax:

```
textbackground (<Integer>);
```

Mit der Prozedur Textbackground kann man die Farbe des Hintergrundes festlegen.

Beispiel:

```
Textbackground (3);  Festlegung der Farbe Tuerkis fuer den
                     Hintergrund
Textbackground (Red);Festlegung der Farbe Rot fuer den
                     Hintergrund
```

4.7.6. Sonstige Funktionen und Prozeduren

4.7.6.1. ADDR-Funktion

Syntax:

```
addr  (<abc>)
```

Die Funktion liefert die Speicheradresse des ersten Bytes der Variablen <abc>. Der Wert ist ein 32-bit Zeiger, bestehend aus Segmentadresse und Offset.

Beispiel:

```
PROCEDURE Addr_demo;
VAR Satz: RECORD  J: INTEGER;
                  B: BOOLEAN
          END;
    Adres: INTEGER;
        R: REAL;
        S: ARRAY [1..100] OF CHAR;
BEGIN
  writeln (addr (Satz));
  writeln (addr (Satz.B));
  writeln (addr (S))
END;
```

4.7.6.2. OFS-Funktion

Syntax:

```
ofs (<objekt>)
```

*** OFS-Funktion ***

Die Ofs-Funktion liefert das Offset als Integerwert vom ersten Byte der Variablen, Prozedur oder Funktion, die im entsprechenden Speichersegment verwendet wird.

4.7.6.3. SEG-Funktion

Syntax:

```
seg (<objekt>)
```

Die seg-Funktion liefert die Adresse des Segments als Integerwert vom ersten Byte der Variablen, Prozedur oder Funktion die als <objekt> angegeben ist.

4.7.6.4. CSEG-Funktion

Syntax:

```
cseg
```

Die cseg-Funktion liefert einen Integerwert von der Basisadresse des Codesegments.

4.7.6.5. DSEG-Funktion

Syntax:

```
dseg
```

Die dseg-Funktion liefert einen Integerwert von der Basisadresse des Datensegments.

4.7.6.6. SSEG-Funktion

Syntax:

```
sseg
```

Die sseg-Funktion liefert einen Integerwert von der Stacksegmentadresse.

4.7.6.7. DELAY-Prozedur

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 unterschiedlichen Hardware etwas davon abweichen.

4.7.6.8. CHAIN- und EXECUTE-Prozedur

Syntax:

```
chain    <Filevariable>
execute  <Filevariable>
```

Die Prozeduren CHAIN und EXECUTE erlauben von einem Programm aus die Aktivierung anderer Programmfiles. Eine Verkettung von Programmen macht sich erforderlich, wenn Programme groesser sind, als der verfuegbare Speicherplatz und OVERLAY-Strukturen ungeeignet sind.
<Filevariable> ist die Filevariable eines ungetypten Files. Sie muss vorher mittels ASSIGN einem Diskettenfile zugewiesen sein, aber nicht eroeffnet(RESET / REWRITE) werden.
Die Prozedur CHAIN wird verwendet, um ein CHN-File abzuarbeiten, welches vorher mit der Compiler-Option H compiliert wurde (siehe auch Punkt 3.12.).
Das CHN-File wird an die Stelle im Speicher geladen und bei der Adresse gestartet, die das aktuelle Programm hat, d.h. die Adresse, die bei der Uebersetzung des aktuellen Programms angegeben wurde. Das neu gestartete Programm verwendet auch die bereits im Speicher stehende Pascalbibliothek. Aus diesem Grund muessen beide die gleiche Startadresse haben.
Die Prozedur EXECUTE wird verwendet, um ein COM-File abzuarbeiten dass einen abarbeitungsfaehigen Code enthaelt. Existiert das Diskfile nicht, tritt ein E/A-Fehler auf.
Die Programmgroesse hat bei der Verkettung keine Bedeutung, allerdings muessen auszutauschende Daten oberhalb des groessten Programms stehen, wenn eine Datenuebergabe erforderlich ist.
Dieser Datenaustausch kann auf drei Wegen ausgefuehrt werden:

a.) Gemeinsam benutzte globale Variablen (gleicher Vereinbarungsteil notwendig)
b.) Verwendung von absoluten Variablen (ABSOLUTE)
c.) Verwendung von Diskettenfiles.

Beispiel:

```
PROGRAM Eins;
{Programmierter Start des Programms Zwei}
VAR  Start    : FILE;
BEGIN
  ASSIGN(Start,'ZWEI.COM');
  execute(Start)
END.
```

Eine eventuelle erforderliche Rueckkehr nach CHAIN oder EXECUTE ins rufende Programm muss mit EXECUTE selbst organisiert werden.

4.7.6.9. FILLCHAR-Prozedur

Syntax:

fillchar (<Ziel>,<Anzahl>,<Zeichen>)

Uebertragung von <Anzahl> gleicher Zeichen <Zeichen> in einen Speicherbereich, beginnend ab dem ersten Byte.
Wenn <Anzahl> groesser ist als die Laenge von <Ziel>, dann werden die nachfolgenden Daten ueberschrieben.
<Zeichen> ist eine Variable oder Konstante vom Typ CHAR. Bei <Zeichen> kleiner 255 ist auch die BYTE-Schreibweise erlaubt.

Beispiel:

```
PROCEDUR fil;
VAR Puffer: ARRAY[1..200] OF CHAR;
 BEGIN
  fillchar (Puffer,200,' ');{in die Variable Puffer werden
 END;                          200 Leerzeichen uebertragen}
```

4.7.6.10. EXIT-Prozedur

Syntax:

exit

Diese Prozedur dient zum vorzeitigen Beenden einer Programmeinheit (Prozedur, Funktion oder des Hauptprogrammes).
EXIT in einem Hauptprogramm wirkt wie HALT.

Beispiel:

```
PROCEDURE lesen;
BEGIN
   assign(f,'DATEN.BAS');
   {$I-}
   reset(f);
   {$I+}
   IF ioresult <> 0 THEN BEGIN
                           writeln('Dateifehler!!!');
                           exit;
                         END;
END;
```

4.7.6.11. HALT-Prozedur

Syntax:

halt

Die Prozedur HALT bewirkt den Abbruch der Programmausfuehrung und die Rueckkehr in das Laufzeitsystem.

4.7.6.12. HI-Funktion

Syntax:

```
hi    (<Integer>)
```

Das niederwertige Byte des Ergebnisses enthaelt das hoeherwertige Byte des Wertes vom Integerausdruck <Integer>. Das hoeherwertige Byte des Ergebnisses ist Null. Das Ergebnis ist vom Typ Integer.

4.7.6.13. KEYPRESSED-Funktion

Syntax:

```
keypressed
```

Die Funktion gibt den Wert TRUE zurueck, wenn eine Taste auf der Konsole gedrueckt wurde. Das Ergebnis wird durch Aufruf der Konsol-Status-Routine des BIOS realisiert.

4.7.6.14. LO-Funktion

Syntax:

```
lo    (<Integer>)
```

Die Funktion gibt das niederwertige Byte des Wertes vom Integerausdruck <Integer> zurueck, wobei das hoeherwertige Byte auf Null gesetzt wird. Der Typ des Ergebnisses ist Integer.

4.7.6.15. OVRPATH-Prozedur

Waehrend der Laufzeit eines Programmes wird das Overlay im aktuellen Laufwerk und der aktuellen Directory erwartet. Mit der Prozedur OvrPath hat man die Moeglichkeit, den voreinstellten Wert zu aendern.

Syntax:

```
ovrpath (<String>);
```

Mit dem durch <String> festgelegten Ausdruck spezifisiert man den Pfad, indem das Overlay vom System gesucht wird. Eine eroeffnete Overlay-Datei wird immer in der selben Directory gesucht. Durch Angabe eines Punktes kann man festlegen, dass die aktuelle Directory genutzt werden soll.

4.7.6.16. MOVE-Prozedur

Syntax:

```
move    (<Quelle>,<Ziel>,<Anzahl>)
```

Diese Prozedur kopiert im Speicher eine bestimmte Anzahl von Bytes <Anzahl> von der Speicherstelle <Quelle> zur Speicherstelle <Ziel>. Hierbei sind <Quelle> und <Ziel> zwei Variablen von beliebigem Typ (auch Zeiger) und <Anzahl> ist ein Integerausdruck.

4.7.6.17. PARAMCOUNT-Funktion

Syntax:

```
paramcount
```

Diese Funktion ermittelt die Anzahl der Kommandozeilenparameter, d.h. die beim Start eines COM-Files durch Leerzeichen getrennt nach dem COM-File-Namen noch ausgegeben werden. Das Ergebnis ist vom Typ INTEGER.

Beispiel:

```
Kommando
>TEST ARTIKEL.DAT 15.9.87<ENTER>
paramcount liefert den Wert 2.
```

4.7.6.18. PARAMSTR-Funktion

Syntax:

```
paramstr    (<Integer>)
```

Diese Funktion stellt den <Integer>-ten Kommandozeilenparameter bereit. Das Ergebnis ist von Typ STRING. Im Beispiel fuer PARAMCOUNT gilt:

Beispiel:

```
paramstr(1) = 'ARTIKEL.DAT';
paramstr(2) = '15.9.87';
```

4.7.6.19. RANDOM-Funktion

Syntax:

```
random    oder    random    (<Integer>)
```

Gibt eine Zufallszahl zurueck, die groesser oder gleich Null und kleiner als Eins ist. Der Typ ist REAL.
Gibt eine Zufallszahl zurueck, die groesser oder gleich Null und kleiner als <Integer> ist. <Integer> und die Zufallszahl sind beide vom Typ INTEGER.

4.7.6.20. RANDOMSIZE

Syntax:

randomsize

Der Zufallszahlengenerator wird in einen definierten Anfangszustand versetzt.

4.7.6.21. SIZEOF-Funktion

Syntax:

sizeof (<Variable>)

Die Funktion liefert als INTEGER-Wert die Laenge von <Variable> in Bytes.
Fuer <Variable> ist jeder Variablenbezeichner zugelassen.

Beispiel:

```
PROCEDURE Size;
VAR B: ARRAY[1..10] OF CHAR;
    A: ARRAY[1..15] OF CHAR;
BEGIN
  A:= 'ABCDEFGHIJKLMNO';
  B:= '0123456789';
  writeln (sizeof(A),'/',sizeof(B));       {Ausgabe: 15/10}
  move (B,A,sizeof(B));
  writeln (A)                      {Ausgabe: 0123456789KLMNO}
END.
```

SIZEOF laesst sich auch guenstig mit FILLCHAR und MOVE verbinden.

4.7.6.22. SWAP-Funktion

Syntax:

swap (<Integer>)

Die Funktion vertauscht vom Wert des Integerausdruckes <Integer> das hoeher- und niederwertige Byte und gibt das Ergebnis als Integerzahl aus.

Beispiel:

```
swap(01234)            { = 03412}
```

4.7.6.23. UPCASE-Funktion

Syntax:

upcase (<Zeichen>)
Zeichen::= Konstante oder Variable vom Typ ['a'..'z']

Das Ergebnis ist der entsprechende Grossbuchstabe. Liegt <Zeichen> ausserhalb des Bereichs 'a'..'z', ist die Funktion wirkungslos.

Beispiel:

```
VAR T:STRING[20];
    i:INTEGER;

READLN (T);
FOR i:=1 TO LENGTH(T) DO UPCASE(T[i]);
```

4.7.6.24. SOUND-Prozedur

Mit der Standardprozedur Sound hat man die Moeglichkeit, den Lautsprecher des PC anzusprechen.

Syntax:

```
sound (<Integer>);
```

Der Integerausdruck gibt die Frequenz in Hz an und ist solange wirksam, bis er durch die Standardprozedur Nosound ausgeschalten wird.

Syntax:

```
nosound;
```

Beispiel:

```
Begin
Sound (1000);       Das Beispiel erzeugt einen Ton
delay ( 250);       von  1 KHz und  einer   Laenge
Nosound;            von 250 msec.
End.
```

4.8. Operationen mit Mengen

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

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

Mengenoperationen wurden in Ziffer 4.4.1.5. dargestellt.

4.8.1. 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.

Beispiel:

```
['T','F','A','S','C','A','L']
['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] aequivalent. Wenn X>Y, dann bezeichnet [X..Y] eine leere Menge.

4.8.2. Mengenzuweisungen

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

Beispiel:

```
  .
TYPE  Attribut = (braun,grau,karo,beige,rot);
VAR   Farbe:SET OF Attribut;
BEGIN
  Farbe:= [braun];
  .
  .
```

4.9. Zeiger und Listen

4.9.1. Dynamische Variablen

Dynamische Variablen werden waehrend der Programmausfuehrung geschaffen und wieder vernichtet.

Beispiel:

```
TYPE  Zeiger   =  ^Struktur;            {^ = Zeigertyp}
                                  {Struktur noch nicht def.}
   Struktur = RECORD
     Nummer   : STRING[6];
     Name     : STRING[40];
     Menge    : INTEGER;
     Naechster: Zeiger
   END;
VAR  Artikel : Zeiger;
BEGIN
  .

{Speicheradresse durch den Programmierer}
Artikel :=  ptr(cseg,$8000);
{Speicheradresse durch das System}
new(Artikel);
```

Die obige Vereinbarung reserviert zunaechst nur 2 Bytes Speicherplatz fuer "Artikel" als Zeiger auf eine Struktur. Die Zeigervariable unterscheidet sich wesentlich von anderen Variablen. Sie enthaelt eine Speicheradresse einer einfachen oder strukturierten Variablen. Der Speicherplatz fuer einfache oder strukturierte Variablen (dynamische Variablen) wird geschaffen, wenn der Zeigervariablen eine (freie) Adresse zugewiesen wird. Das kann direkt oder mit NEW geschehen. Der Zugriff zum Inhalt der Adresse erfolgt mit ^.

4.9.2. New und Dispose

Mit der Prozedur NEW ist es moeglich, Speicherplatz fuer Variablen vom definierten Typ zu reservieren.

Syntax:

```
new (<Zeiger>)
```

Beispiel:

```
new(Artikel);
```

Artikel zeigt im Beispiel auf einen dynamisch erzeugten Satz vom Typ Struktur. Auf diese dynamische Variable wird wie folgt zugegriffen:

```
readln(Artikel^.Nummer);
readln(Artikel^.Name);
```

Der erneute Aufruf von NEW fuehrt zu neuer Speicherplatzreservierung.

Die Gesamtheit des belegten Speicherbereiches, der nicht zusammenhaengend sein muss, wird als Halde(HEAP) bezeichnet.
Der Zeigerwert NIL gehoert jedem Zeigertyp an. Er zeigt auf keine dynamische Variable und wird Zeigervariablen zugewiesen, um anzuzeigen, dass sie keine verwertbare Adresse enthalten.

Die Freigabe des Speicherplatzes von geloeschten Elementen einer Liste erfolgt mit der Prozedur DISPOSE.

Syntax:

```
dispose (<Zeiger>)
```

Hiermit wird bewirkt, dass der Speicherplatz, auf welchen der <Zeiger> zeigt, fuer weitere Belegungen verwendet werden kann.

Zur Verwaltung dynamischer Variablen werden noch folgende Funktionen bereitgestellt:

```
memavail
maxavail
```

Die Funktion MEMAVAIL liefert den fuer dynamische Variablen verfuegbaren Speicherraum als INTEGER-Wert (Anzahl der Bytes).
Die Funktion MAXAVAIL liefert den fuer dynamische Variablen verfuegbaren Speicherraum (ausschliesslich geloeschte dynamische Variablen) als INTEGER-Wert.

4.9.3. Mark und Release

Es gibt statt DISPOSE eine weitere Moeglichkeit zur Freigabe des Speicherplatzes dynamischer Variablen. Das sind die Standardprozeduren MARK und RELEASE .

Syntax:

```
mark  (<Zeiger>)
```

Mit MARK kann auf einer Zeigervariablen der aktuelle Stand der Halde festgehalten werden, mit dem Ziel der spaeteren Freigabe ab dieser Position.

Syntax:

```
release (<Zeiger>)
```

Mit RELEASE kann eine Halde ab der Position freigegeben werden, die vorher mit MARK fixiert wurde. Dabei darf die Pointervariable zwischen den Rufen MARK und RELEASE nicht veraendert werden. Mit RELEASE wird der Zustand wiederhergestellt, der zum Zeitpunkt des vorhergehenden Prozedurrufes MARK existierte.

In einem Programm muss zwischen der Freigabemethode DISPOSE (nach Wirth) und MARK/RELEASE (nach Bowles) gewaehlt werden. Sie sind unvertraeglich.

4.9.4. GETMEM und FREEMEM

Es gibt noch eine weitere Methode der dynamischen Verwaltung von Speicherplatz . NEW repraesentiert stets den Speicherplatz, der fuer die Struktur erforderlich ist, auf den der in der Prozedur angegebene Zeiger zeigt. Das kann hinderlich sein. Deshalb gibt es die Moeglichkeit, die Groesse des dynamisch reservierten Speicherplatzes selbst zu bestimmen.

Syntax:

```
getmem  (<Zeiger>,<Anzahl>)
<Anzahl>::= Ausdruck des Typs INTEGER
```

<Anzahl> gibt den Speicherplatz in Bytes an. Entsprechend kann der Speicherplatz wieder freigegeben werden.

Syntax:

```
freemem  (<Zeiger>,<Anzahl>)
```

Unter Nutzung von MAXAVAIL/MEMAVAIL ermoeglicht das eine dem Problem und der Speicherkapazitaet angepasste dynamische Datenverwaltung.

4.9.5. Programmierung dynamischer Listen

Beispiel:
(1) Deklaration:

```
TYPE Zeiger    = ^Objekt;
     Objekt = RECORD
                Wert: REAL;
                Naechst: Zeiger
              END;
VAR Z,Anf,P,Q: Zeiger;
        Wert1: REAL;
        Wert2: REAL;
```

(2) Initialisieren einer dynamischen Liste:

```
Anf:= NIL
                              Anf --> NIL     {= leere Liste}
```

Eine Liste hat immer einen Zeiger (z.B. Anf), welcher auf den Anfang der Liste zeigt.

(3) Eintragen eines Listenelements an den Anfang einer Liste:

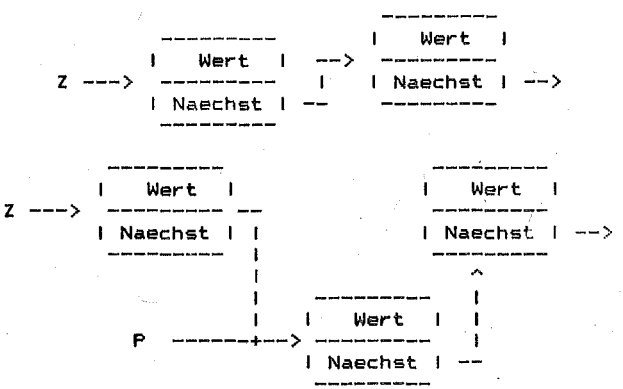
```
readln (Wert1);
new (Z);
Z^.Wert:= Wert1;
Z^.Naechst:= Anf;                    {oder Z^.Naechst:= NIL}
Anf:= Z;
```

(4) Suchen eines Elements (Wert1) in einer Liste:

```
PROCEDURE Such;
VAR Gefunden:Boolean;
BEGIN
  REPEAT
    readln (Wert2);
    Z:= Anf;
    WHILE Z <> NIL DO BEGIN
      Gefunden:=Z^.Wert = Wert2;
      IF Gefunden THEN Z:= NIL
      ELSE BEGIN
        Q:= Z;
        Z:= Z^.Naechst
      END;
    END;
    IF NOT Gefunden THEN writeln ('Nicht vorhanden!');
  UNTIL Gefunden;
  writeln('Wert gefunden');
END;
```

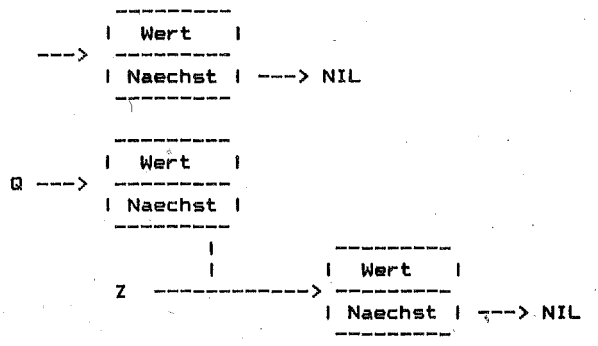
(5) Einfuegen eines neuen Elements in der Mitte der Liste:

```
new (P);
readln (P^.Wert);                    {neues Element eingeben}
Such;                          {Suchen eines Wertes (Wert2),  }
                               {nach welchem eingefuegt werden}
                               {soll.}
P^.Naechst:= Z^.Naechst;
Z^.Naechst:= P;
```



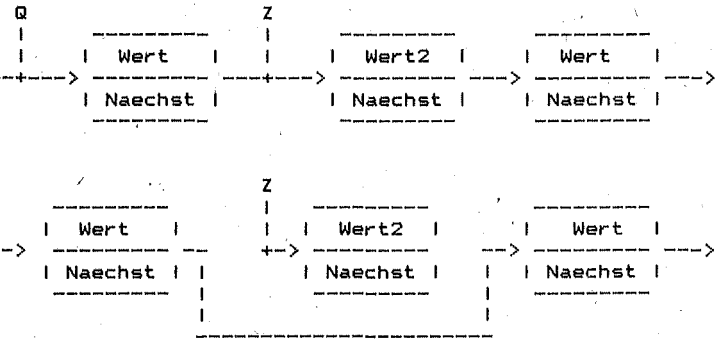
(6) Anfuegen eines neuen Elements am Ende der Liste:

```
new (Z);
readln (Z^.Wert);
P:= Anf;                             {neues Element eingeben}
WHILE P <> NIL DO
      BEGIN Q:= P;
            P:= Q^.Naechst
      END;
Z^.Naechst:= NIL;
Q^.Naechst:= Z;
```



(7) Loeschen eines Elements einer Liste:

```
Such;                          {Eingabe und Suchen des zu}
                               {loeschenden Elements}
Q^.Naechst:= Z^.Naechst;
dispose (Z);
```



4.10. Ein- und Ausgabe von Files

4.10.1. Begriffe

Ein File ist ein Datenbestand, welcher aus logisch gegliederten, gleichgrossen Filekomponenten besteht.
Der Zugriff erfolgt ueber einen Zeiger wahlweise direkt oder sequentiell.
PASCAL unterscheidet zwischen Geraete- und Diskettenfiles.
Diskettenfiles werden unter dem vom Nutzer vereinbarten Filenamen (Prozedur ASSIGN) auf Diskette abgelegt.
Der Name eines Diskettenfiles wird durch eine Zeichenkette dargestellt. Fuer <Filename> unter DCP gilt:

```
<Filename>::=<Pfad><Basisname>{.<Erweiterung>}
<Pfad>::=<\dirname>{\dirname}
<dirname>::=
     <Zeichen>{<Zeichen>}
<Basisname>::=
    <Zeichen>{<Zeichen>}
<Erweiterung>::=
    <Zeichen>{<Zeichen>}
<Zeichen>::= Element der Menge ['#'..'&','/','(',')','0'..'9',
             '@'..'Z','a'..'z']
```

Fuer den Basisnamen sind maximal 8, fuer die Erweiterung maximal 3 Zeichen zugelassen.

Durch den Typ eines Files (vergl. Ziffer 4.3.3.) wird die Groesse und das Format der Filekomponenten spezifiziert.

Folgender Standardtyp ist vordefiniert:

```
TEXT = FILE OF CHAR.
```

Fuer jedes File wird waehrend der Laufzeit ein File-Informationsblock angelegt (vergl. Ziffer D.1.5.). Dieser ist dem Programmierer nicht direkt zugaenglich.

Beispiel:

```
     PROGRAM xyz (Dateix);
     TYPE Dtx: RECORD a: INTEGER;
                      b: REAL
               END;
     VAR Dateix: FILE OF Dtx;
```

Durch diese Deklaration wird eine interne Filevariable Dateix erzeugt, die genau ein Element (Satz) des externen Diskettenfiles aufnehmen kann (1*INTEGER, 1*REAL).

In PASCAL stehen dem Nutzer folgende Zugriffsroutinen fuer ein File zur Verfuegung:

a) Sequentieller und wahlfreier Zugriff zu Binaerfiles
   - READ
   - WRITE

b) Sequentieller Zugriff zu Textfiles
   - READLN
   - WRITELN

c) Blockweiser Zugriff zu Binaer- und Textfiles
   - BLOCKREAD
   - BLOCKWRITE

Vor einem wahlfreien Zugriff muss das Filefenster mit der Prozedur SEEK positioniert werden.
Ausserdem stehen weitere Funktionen/Prozeduren zur Verfuegung, die die Filearbeit unterstuetzen.
Das Ende einer Textdatei wird durch das Steuerzeichen $1A gekennzeichnet.
Wird dieses Zeichen gelesen, dann liefert die Standardfunktion EOF = TRUE.

4.10.2. Fileoperationen fuer Binaerfiles

4.10.2.1. ASSIGN

Syntax:

```
assign(<Filevariable>,<Filename>)
```

Die Prozedur hat die Aufgabe, der Variablen <Filevariable> einen externen Filenamen zuzuweisen. Die Filevariable kann jeden beliebigen Typ annehmen.
Sollen DCP-Geraetenamen zugewiesen werden, dann muss die <Filevariable> vom Typ TEXT sein.
Ein erneutes Anwenden von ASSIGN auf eine Filevariable, welcher bereits ein physischer Filenamen zugeordnet wurde und mit welcher bereits gearbeitet wurde, ist unzulaessig.

INPUT, OUTPUT, LST, KBD, CON und TRM sind vordefinierte Textfiles (vergl. Ziffer 4.10.7.).

4.10.2.2. REWRITE

Syntax:

```
rewrite (<Filevariable>)
```

Die REWRITE-Prozedur erzeugt auf Diskette ein File mit dem Namen, welcher mit ASSIGN zugewiesen wurde.
Gleichzeitig wird die Datei fuer Schreiben freigegeben.
Der Filepointer wird dabei auf die Filekomponente mit der Nummer 0 gesetzt. Im Fall, dass auf der Diskette bereits der gleiche physische Name existiert, wird das dazugehoerige File ueberschrieben.
Zu Beginn enthaelt ein mit REWRITE erzeugtes File kein Element.
Die Funktion EOF ist TRUE.

*** REWRITE ***

Beispiel:

```
PROGRAM Ausgabe;
     .
     .
     .
BEGIN
 assign (Kunde,'A:Kunden.Dat');
 rewrite(Kunde);
       .
       .
```

4.10.2.3. RESET

Syntax:

reset (<Filvariable>)

Die RESET-Prozedur eroeffnet ein existierendes File. Bei direktem Zugriff ist die Datei offen zum Lesen und Schreiben. Der Filepointer wird auf die erste Filekomponente (mit der Nummer Null) gesetzt.

Beispiel:

```
PROGRAM Ausg (Kunde);
BEGIN
  assign(Kunde, 'A: Kunden.Dat');
  reset (Kunde);
    .
    .
    .
END.
```

4.10.2.4. APPEND

Syntax:

append (<Dateivariable>);

Die durch <Dateivariable> festgelegte Datei wird eroeffnet und der Dateizeiger befindet sich am Ende. Man hat somit nur die Moeglichkeit an die Textdatei neue Komponenten anzufuegen.

4.10.2.5. READ

Syntax:

read (<Filevariable>,<Variable>)

Diese Prozedur realisiert das Lesen einer Filekomponente.

Beispiel:

```
read(Kunde,Name);
```

Es wird, da die Zugriffsanzahl zu Diskettenfiles minimiert wird, nicht bei jedem READ auch wirklich von der Diskette gelesen, Anders ist das, wenn es sich um die Tastatur handelt.
Nach jedem Ruf von READ wird das Filefenster (Satzzeiger) um eine Position weitergestellt.

4.10.2.6. WRITE

Syntax:

```
write (<Filevariable>,<Variable>)
```

Diese Prozedur realisiert die Ausgabe der Inhalte der Variablen in die Filekomponente des Files, welchem <Filevariable> zugeordnet wurde.
Nach jedem Ruf von WRITE wird das Filefenster (Satzzeiger) um eine Position weitergestellt.

Beispiel:

```
write(Kunde,Name);
```

4.10.2.7. SEEK

Syntax:

```
seek (<Filvariable>,<Nummmer>)

Nummer::= Ausdruck
```

Das Filefenster des durch die <Filevariable> gekennzeichneten Files wird durch diese Prozedur auf die Komponente mit der <Nummer> eingestellt (die 1. Komponente hat die Nummer 0). Soll das File erweitert werden, so ist es moeglich, die letzte Filekomponente einzustellen (vergl. FILESIZE) und danach WRITE zu benutzen.
Auf diese Weise kann sehr einfach zum direkten Zugriff uebergegangen werden.

Beispiel:

```
seek(Kunde,20);
```

4.10.2.8. TRUNCATE

Syntax:

```
truncate (Filevariable)
```

Die Prozedur truncate schneidet alle Saetze ab der aktuellen Satzposition ab.

4.10.2.9. FLUSH

Die Flush-Prozedur hat bei DCP keine Wirkung.

4.10.2.10. CLOSE

Syntax:

```
close (<Filevariable>)
```

Mit dieser Prozedur wird der Disketten-FCB aktualisiert, indem die entsprechenden Bytes des Speicher-FCB kopiert werden. Das File, welchem die <Filevariable> zugeordnet wurde, wird geschlossen und der aktuelle Zustand in das Diskettenverzeichnis geschrieben. Wird die Prozedur close nicht aufgerufen, tritt ein Datenverlust ein, wenn die Datei geaendert wurde (WRITE).

Beispiel:

```
assign (Kunde,'Kunde.dat');
rewrite(Kunde);
    .
    .
    .

write(Kunde,Name);
close(Kunde);
```

4.10.2.11. ERASE

Syntax:

```
erase (<Filevariable>)
```

Die Prozedur ERASE loescht das File, welchem <Filevariable> zugeordnet wurde, im Diskettenverzeichnis. Sollte das File mit RESET oder REWRITE bereits eroeffnet sein, muss es vor ERASE mit CLOSE geschlossen werden.

Beispiel:

```
VAR x: FILE;
BEGIN
  assign (x,'Beispiel.Dat');
  erase  (x);
```

4.10.2.12. RENAME

Syntax:

```
rename (<Filevariable>,<Filename>)
```

Die Prozedur RENAME wird genutzt, um das <Filevariable> zugeordnete File umzubenennen. Der neue Name wird in das Diskettenverzeichnis eingetragen, und die weiteren Operationen von <Filevariable> werden dann mit diesem File unter dem neuen Namen ausgefuehrt. Nach der Eroeffnung des Files ist das Umbenennen nicht mehr erlaubt.
Es ist zu sichern, dass der neue Filename auf der Diskette nicht bereits exsistiert, um ein Entstehen doppelter Namen in der Directory zu vermeiden. Das kann geprueft werden, wenn die

Ein- und Ausgabeueberwachung des Systems mit {$I-} angeschaltet und mit dem neuen Namen eine Eroeffnung durch RESET versucht wird. Ist danach IORESULT gleich Null, so existiert das File bereits.

Beispiel:

```
VAR x:FILE;
BEGIN
  assign(x,'Alt.Dat');
  rename(x,'Neu.Dat');
  reset(x);
```

4.10.3. Filefunktionen fuer Binaerfiles

4.10.3.1. EOF

Syntax:

eof (<Filevariable>)

Im Fall, dass der Filepointer das Fileende erreicht hat, liefert die Funktion EOF den Wert TRUE. Andernfalls ist der zurueckgegebene Wert FALSE.

4.10.3.2. FILEPOS

Syntax:

filepos (<Filevariable>)

Mit dieser Funktion wird der Wert der aktuellen Position des Filefensters als ein INTEGER-Wert zurueckgegeben. Dabei besitzt die erste Komponente den Wert Null.

4.10.3.3. FILESIZE

Syntax:

filesize (<Filevariable>)

Mit dieser Funktion wird die Groesse des <Filevariable> zugeordneten Files zurueckgegeben. Es wird die Anzahl der Komponenten des Files bestimmt. Ergibt die Funktion den Wert Null, so ist das File leer.

4.10.4. Zusaetzliche Dateiroutinen

Mit den folgenden Routinen ist es moeglich, einen erweiterten Bereich von Records in DCP Platz zu geben.

Funktionen:	longfilesize	(<Filevariable>)
	longfileposition	(<Filevariable>)
Prozedur:	longseek	(<Filevariable>,<nummer>)

Die angegebenen Routinen sind aequivalent zu FileSize, FilePos,

und Seek, sie arbeiten aber mit Realzahlen.

4.10.5. Direktdateien unter DCP

Im Betriebssystem DCP hat man die Moeglichkeit mit TPASCAL nicht PASCAL-Dateien als Direktdateien zu lesen (File of Byte). Im Gegensatz zu PASCAL 880/S verwaltet TPASCAL keine Informationen am Anfang einer Datei.

4.10.6. Textfiles

4.10.6.1. Textfileoperationen

Textfiles sind Files aus Elementen des gueltigen Zeichensatzes (00 bis 07f). Nicht alle Bytes repraesentieren dabei druckbare Zeichen. Die Komponenten eines Textfiles sind Zeilen verschiedener Laenge, die durch Steuerzeichen CR/LF getrennt werden.
Eine Textfilevariable wird erklaert, indem man ihr den Standardtypbezeichner TEXT zuweist:

```
VAR <Filvariable> : TEXT;
```

Zeichenweise E/A-Operationen werden fuer Textfiles mit den Standardprozeduren READ und WRITE ausgefuehrt. Zeilen werden mit den speziellen Textfileoperationen READLN, WRITELN und EOLN behandelt. Es gilt:

readln(<Filevariable>)	Springt zum Beginn der naechsten Textzeile,d.h. ueberspringt alle Zeichen bis und einschliesslich der naechsten CR/LF-Folge.
writeln(<Filevariable>)	Schreibt die Zeilenendemarke, d.h. die CR/LF-Folge auf das Textfile.
eoln(<Filevariable>)	Ist eine Boolesche-Funktion, die den Wert TRUE zurueckgibt, wenn das Ende der aktuellen Zeile erreicht ist, d.h. wenn der Filepointer auf das CR-Zeichen der CR/LF-Folge zeigt. Ist EOF (Filevariable) gleich TRUE, so ist EOLN (Filevariable) auch TRUE.

Wendet man die EOF-Funktion auf ein Textfile an, dann liefert diese Funktion den Wert TRUE, wenn der Filepointer die Fileendemarke CTRL-Z erreicht hat. Auf Textfiles sind die Funktionen SEEK,FILEPOS und FILESIZE nicht anwendbar, da keine gleichgrossen Filekomponenten existieren.

Beispiel:

```
VAR Lst:TEXT;
BEGIN
  assign(Lst,'LST:');
  rewrite(Lst);
  write(Lst,...);
```

Diese Bedeutung von LST ist vordefiniert, so dass auf die Vereinbarung der Filevariablen mit TEXT, ASSIGN und REWRITE verzichtet werden kann (vergl. Ziffer 4.10.8.).

4.10.6.2. Puffergroesse

Die Puffergroesse ist bei Textdateien auf 128 Byte voreingestellt. Fuer die meisten Anwendungen ist dieser Bereich ideal. Bei Kopier- oder aehnlichen Programmen, die viele Ein/Ausgaben vornehmen, ist ein groesserer Puffer von Vorteil. Durch die Deklaration kann die Puffergroesse eingestellt werden.

Beispiel:

```
var   Textdatei: Text [$400];
```

Definieren einer File- Variablen mit einer Puffergroesse von 1 KByte Die Prozedur Flush bewirkt eine Entleerung des Dateipuffers bei der Arbeit mit Textdateien. Es wird mit Sicherheit erreicht, dass der Pufferinhalt auch in die Datei geschrieben wird.

4.10.7. Logische Geraete

Logische Geraete sind in PASCAL externe Geraete wie Terminals, Drucker und Modems. Sie werden wie Textfiles behandelt.

CON: Console. Ausgaben werden an ein Bildschirmgeraet gesendet und Eingaben werden von der Tastatur gelesen. READ oder READLN ueber CON lesen eine ganze Zeile aus dem Zeilenpuffer. Der Operator kann, bis zur Eingabe von CR ueber die ET-Taste, die Editiermoeglichkeiten des Systems fuer Eingaben nutzen.

TRM: Terminal. Ausgaben werden an ein Bildschirmgeraet gesendet und Eingaben werden von 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 Tastatur gelesen. Ein Echo erfolgt nicht.

LST: Listing. Die Ausgaben erfolgen an einen Drucker.

AUX: Auxiliary. Ausgaben werden an den Stanzer gesendet und Eingaben werden vom Leser gelesen. Normalerweise sind beide Lochband- oder Kassettenmagnetbandgeraete.

USR: Usergeraet. Ausgaben gehen an das Nutzerausgabegeraet, und Eingaben werden ueber die Nutzereingaberoutine gelesen.

Es ist moeglich, dass diese logischen Geraete durch vorher definierte Files oder wie ein Diskettenfile einer Filevariablen zugewiesen werden. Bei Zuweisung eines logischen Geraetes zu einem File existiert zwischen REWRITE und RESET kein Unterschied. Die Prozedur CLOSE fuehrt dann keine Funktion aus und ERASE liefert einen E/A-Fehler.

Die Standardfunktionen EOF und EOLN arbeiten bei logischen Geraeten anders als bei Diskettenfiles. Bei einem Diskettenfile liefert EOF den Wert TRUE zurueck, wenn das naechste Zeichen im File das Zeichen CTRL-Z ist. EOLN gibt den Wert TRUE zurueck, wenn das naechste Zeichen CR oder CTRL-Z ist.
Diese beiden Prozeduren sind vorausschauende Routinen.
Wird SEEKEOF oder SEEKEOLN statt EOF/EOLN verwendet, so werden Leerzeichen und Tabulatormarken (und bei SEEKEOF auch CR/LF) uebersprungen.
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 liefert TRUE, wenn das letzte Zeichen CTRL-Z war, und EOLN liefert TRUE, wenn das letzte Zeichen CR oder CTRL-Z war.

	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.

4.10.8. Standardfiles

TPASCAL stellt einige Standardtextfiles zur Verfuegung, die bereits logischen Geraeten zugewiesen sind und unmittelbar genutzt werden koennen. So ist es moeglich, Speicherplatz und den Aufruf von ASSIGN, RESET, REWRITE und CLOSE zu sparen.
Folgende Standardtextfiles sind implementiert:

INPUT	Primaeres Eingabefile. Dieses File ist entweder dem CON- oder TRM-Geraet zugewiesen.
OUTPUT	Primaeres Ausgabefile. Dieses File ist entweder dem CON- oder TRM-Geraet zugewiesen.
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:.
USR	Zugewiesen dem Usergeraet USR:.

Die Verwendung von RESET, REWRITE und CLOSE ist verboten. Die Zuweisung des logischen Geraetes zu den Standardtextfiles INPUT und OUTPUT erfolgt durch die Compilerdirektive $B.

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

Bei Zuweisung von CON: werden die Eingaben gepuffert und koennen in diesem Puffer bei der Eingabe editiert werden. Fuer das Einlesen der Variablen gelten spezielle Regeln. Bei Zuweisung

von TRM: ist ein Editieren der Eingaben nicht moeglich. Das Einlesen der Variablen erfolgt aber nach den bekannten Regeln. Bei den Ausgabeoperationen existieren fuer CON: und TRM: keine Unterschiede.
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 haeufigsten verwendeten Geraet zu setzen, und in den anderen E/A-Operationen ist das andere Geraet explizit anzugeben.

Beispiel:

```
{$B-}
PROGRAM Lesen/Schreiben(OUTPUT);
...
...
...
readln(INPUT,Var1);          Lesen von TRM:
readln(CON,Var2);            Lesen von CON:
```

An den Stellen, wo auf dem Bildschirm kein Echo der Eingabe erscheinen soll, muss man das Standardtextfile KBD zuweisen:

```
read(KBD,Ant);
```

Da die Standardtextfiles INPUT und OUTPUT sehr haeufig verwendet werden, wurde implementiert, dass sie automatisch zugewiesen werden, wenn kein Filebezeichner explizit angegeben wurde. Damit sind die folgenden Textfileoperationen aequivalent:

```
write(x)        write(OUTPUT,x)
read(x)         read(INPUT,x)
writeln         writeln(OUTPUT)
readln          readln(OUTPUT)
eof             eof(INPUT)
```

Das folgende Beispiel zeigt die Verwendung des Standardfiles LST.

Beispiel:

```
  .
  .
writeln(LST,'Ausgabe ueber Drucker');
  .
  .
```

4.10.9. Ein- und Ausgabe von Textfiles

Die Ein- und Ausgabe von Daten durch den Menschen in lesbarer Form wird mittels Textfiles, wie in Punkt 4.10.8. beschrieben, ausgefuehrt. Ein Textfile kann einem Diskfile oder einem Standard-E/A-Geraet zugewiesen werden. Die Ein- und Ausgaben werden ausgefuehrt mit den Standardprozeduren READ, READLN, WRITE und WRITELN.
Die Parameter koennen im einzelnen unterschiedliche Typen ha-

ben. 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 E/A-Prozedur ein Variablenbezeichner eines Textfiles, dann bezieht sich die Ein- oder Ausgabe auf dieses File. Im anderen Fall bezieht sie sich auf das Standardtextfile INPUT oder OUTPUT.

4.10.9.1. READ

Die READ-Prozedur ermoeglicht die Eingabe von Zeichen, Strings und numerischen Daten.

Syntax:

```
read  (<Variable>{,< Variable>})
read  (<Filevariable>,< Variable>{,<Variable>})
```

wobei die <Variable> vom Typ CHAR, STRING,INTEGER oder REAL sein koennen. Die erste Form liest Daten vom Standardfile INPUT. Die zweite Form liest Eingaben vom Textfile <File>, das fuer das Lesen vorbereitet werden muss oder vordefiniert ist. Mit einer Variablen vom Typ CHAR liest READ vom File ein Zeichen und weist dieses der Variablen zu. Im Fall, dass das File ein Diskettenfile ist, wird EOLN TRUE, wenn das naechste Zeichen CR oder CTRL-Z ist. EOF wird TRUE, wenn das naechste Zeichen CTRL-Z ist.
Mit einer Variablen vom Typ STRING liest READ so viele Zeichen wie durch die maximale Laenge des STRING erlaubt sind, es sei denn, EOLN oder EOF wurde vorher erreicht oder der Puffer mit buflen<n> auf n-Zeichen begrenzt.

Mit einer numerischen Variablen (INTEGER oder REAL) erwartet READ eine Zeichenkette, die mit dem Format einer numerischen Konstante des entsprechenden Typs 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. Im Fall, dass die Zeichenkette nicht mit dem Format uebereinstimmt, tritt ein E/A-Fehler auf. Im anderen Fall wird die numerische Zeichenkette in den entsprechenden Typ konvertiert und der Variablen zugewiesen. Im Fall, dass von einem Diskfile gelesen wurde und die Eingabezeichenkette mit einem Leerzeichen oder HT endet, dann startet die naechste READ- oder READLN-Operation mit dem Zeichen, das unmittelbar diesem Leerzeichen oder HT folgt. Fuer beide, Diskettenfile oder logischem Geraet, wird EOLN=TRUE, wenn die Zeichenkette mit CR oder CTRL-Z endete. EOF wird TRUE, wenn die Zeichenkette mit CTRL-Z endete. Ein Spezialfall der numerischen Eingabe tritt auf, wenn EOLN oder EOF bereits beim Beginn TRUE wird. In diesem Fall wird der Variablen kein neuer Wert zugewiesen. Die Variable behaelt ihren alten Wert.
Wenn das Eingabefile CON: zugewiesen wurde, oder wenn das Standardfile im {ØB+}-Modus verwendet wurde, gelten spezielle Regeln 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. Dies ermoeglicht das Editieren waehrend der Eingabe. Es bewirken:

Backspace und DEL	Ruecksetzen des Cursors und Loeschen des dort stehenden Zeichens. Backspace wird durch die Taste <-- oder CTRL-H, DEL durch die Taste DEL erzeugt.
CTRL-X	Ruecksetzen des Cursors auf den Eingabebeginn und Loeschen aller eingegebenen Zeichen.

Die ENTER-Taste beendet die Eingabe; das dabei eingegebene CR wird nicht als Echo auf dem Bildschirm ausgegeben.
Intern wird die Eingabezeile 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 Variablen bleiben unveraendert.

Maximal koennen in eine Eingabezeile 127 Zeichen eingegeben werden. Man kann die Eingabezeile, wie bereits beschrieben, begrenzen. Dazu wird der vordeklarierten Variablen BUFLEN, eine INTEGER-Zahl aus dem Bereich 1 bis 127 zugewiesen.

Beispiel:
```
write('Filename (max.10 Zeichen):');
buflen := 10;
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.

4.10.9.2. READLN

Der Unterschied zwischen READLN und READ besteht darin, dass nach dem Einlesen der letzten Variablen bei READLN der Rest der Zeile uebersprungen wird.

Syntax:

```
readln  (<Variable>{,<Variable>})
readln  (<Filevariable>{,< Variable>})
```

Nach einem READLN liest das naechste READ oder READLN vom Beginn der naechsten Zeile. EOLN ist immer FALSE nach READLN, ausser wenn EOF = TRUE ist. Es ist auch moeglich, READLN ohne Parameter aufzurufen.
In diesen Faellen wird die gesamte Zeile uebersprungen. Im Fall, dass READLN von der Console liest, wird im Gegensatz zu READ das beendende CR als Echo in der Form CR/LF-Folge auf den Bildschirm uebertragen.

4.10.9.3. WRITE

Mit WRITE ist die Ausgabe von Zeichen, Strings, booleschen und numerischen Werten moeglich.

Syntax:

```
write   (<Parameter>{,<Parameter>})
write   (<Filevariable>,<Parameter>{,<Parameter>})
```

Die Parameter sind Variablen vom Typ CHAR, STRING, BOOLEAN, INTEGER oder REAL. Wahlweise folgt diesen Parametern jeweils ein Doppelpunkt und ein INTEGER-Ausdruck, der die Laenge des Ausgabefeldes angibt. In der ersten der oben angegebenen Formen erfolgt die Ausgabe der Variablen durch das Standardfile OUTPUT. Im zweiten Fall werden die Variablen durch das Textfile File ausgegeben.

Die Formate der WRITE-Parameter haengen vom Typ der Variablen ab. Im folgenden werden die unterschiedlichen Formate und ihre Eigenschaften beschrieben. Dabei bezeichnen 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 Zeichens Ch.
Ch:n	Ausgabe des Zeichens Ch rechtsbuendig in einem n Zeichen langen Feld, d.h. vor Ch stehen n-1 Leerzeichen.
S	Ausgabe des STRING S. Felder (ARRAYs) koennen ebenfalls ausgegeben werden, wenn sie mit den STRINGs ueberein-stimmen und vom Typ CHAR sind.
S:n	Ausgabe der STRINGs rechtsbuendig in einem n Zeichen langen Feld, d.h. vor S stehen n-length(S) Leerzeichen.
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        __x.xxxxxxxxxxEtxx
R <  0        _-x.xxxxxxxxxxEtxx
```

Dabei bedeuten die Zeichen _ Leerzeichen, x Ziffern und t entweder +'oder -.

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

```
R >= 0        blanks x. Zahl Etxx
R <  0        blanks-x. Zahl Etxx
```

Dabei bedeuten blanks keine oder mehrere Leerzeichen, Zahl ein bis zehn Ziffern, x eine Ziffer und t entweder + oder -. Nach dem Dezimalpunkt wird mindestens eine Ziffer ausgegeben, d.h. n muss mindestens 7 sein. Ist n groesser als 16, so stehen vor der Zahl Leerzeichen.

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 Leerzeichen aufgefuellt.

4.10.9.4. WRITELN

Der Unterschied zwischen WRITE und WRITELN besteht darin, dass bei WRITELN nach der letzten Variablen eine CTRL-Z-Folge ausgegeben wird.

Syntax:

```
writeln  (<Parameter>{,<Parameter>})
writeln  (<Filevariable>,<Parameter>{,<Parameter>})
```

WRITELN oder WRITELN(Filevariable) bewirkt nur die Ausgabe einer CR/LF-Folge.

4.10.10. Nichtgetypte Files

Nichtgetypte Files sind Kanalein- und -ausgaben auf niedrigstem Niveau. Es werden Saetze zu 128 Bytes verarbeitet.
Eine nichtgetypte Filevariable benoetigt weniger Speicherplatz als eine andere Filevariable, da die Daten bei E/A-Operationen direkt zwischen dem Diskettenfile und der Variablen uebertragen werden, ohne Platz fuer einen Sektorpuffer zu benoetigen.

Beispiel:

```
VAR Kunde: File;
```

Alle Standardfileprozeduren, also auch SEEK, ausser READ, WRITE und FLUSH, sind erlaubt. BLOCKREAD und BLOCKWRITE sind zwei

spezielle schnelle Uebertragungsprozeduren, die anstelle von READ und WRITE genutzt werden .

Syntax:

blockread (<Filevariable>,<Variable>,<n>)
blockwrite (<Filevariable>,<Variable>,<n>)

<Filevariable> entspricht dabei dem Variablenbezeichner eines ungetypten Files, <Variable> einer beliebigen Variable und n einem INTEGER-Ausdruck. N gibt die Anzahl der zu uebertragenden 128-Byte-Saetze zwischen Diskettenfile und Speicher an. <Variable> muss dafuer ausreichen. BLOCKREAD und BLOCKWRITE realisieren zusaetzlich die Weiterfuehrung des Filefensters um die entsprechende Anzahl von Saetzen.

Beispiel:

```
PROGRAM Kopieren;
VAR Quellfile,Zielfile : FILE;
    Filename           : STRING[12];
    Laufwerk           : CHAR;
    Puffer             : ARRAY[1..1024] OF BYTE;

BEGIN
  writeln('Kopieren eines Files');
  write('Filename: ');
  assign(Quellfile,Filename);
  reset(Quellfile);
  write('Ziellaufwerk: ');
  read(KBD,Laufwerk);
  assign(Zielfile,CONCAT(Laufwerk,':',Filename));
  rewrite(Zielfile);
  WHILE NOT eof(Quellfile) DO BEGIN
    blockread(Quellfile,Puffer,8);
    blockwrite(Zielfile,Puffer,8)
  END;
  close(Quellfile);
  close(Zielfile);
END.
```

4.10.11. Ein- und Ausgabepruefung

E/A-Pruefungen waehrend der Laufzeit eines Programmes sind durch die I-Compilerdirektiven moeglich.
Ist $I+ gesetzt, so werden Fehler in E/A-Operationen durch das Laufzeitsystem DCP auf die uebliche Weise behandelt. Ist $I- gesetzt, so sind die E/A-Operationen durch den Programmierer zu ueberwachen und Fehler entsprechend zu behandeln. Dazu dient die vordefinierte Funktion IORESULT. Sie liefert nach der E/A-Operation einen Fehlercode vom Typ INTEGER. Null ist fehlerfrei.

Beispiel:

```
PROGRAM bsp;
VAR  Kunde      : FILE;
     Dateiname  : STRING[14];
     Test       : BOOLEAN;
BEGIN
  REPEAT
    write('Eingabe des Namens der Datei:');
    readln(Dateiname);
    assign(Kunde,Dateiname);
    {$I-} reset(Kunde); {$I+}
    Test:= (ioresult = 0);
    IF NOT Test THEN writeln('Datei',Dateiname,'kann
                                  nicht eroeffnet werden!');
  UNTIL Test;
  close(Kunde);
END.
```

Bei folgenden Standardfunktionen kann es zweckmaessig sein, mit IORESULT die Fehlerbehandlung selbst zu uebernehmen.

BLOCKREAD	BLOCKWRITE	CHAIN	CLOSE
ERASE	EXECUTE	FLUSH	RENAME
RESET	REWRITE	SEEK	

4.11. Sonstige Sprachelemente und Besonderheiten

4.11.1. HEAP- und STACK-Manipulationen

Waehrend der Programmabarbeitung werden zwei stapelartige Strukturen verwaltet.
Fuer die Speicherung dynamischer Variablen benoetigt man den HEAP(die Halde). Diese wird durch die Standard Prozeduren NEW, MARK und RELEASE gesteuert. Zum Programmstart wird HeapPtr auf den Anfang des freien Speicherbereiches, also auf das erste Byte nach dem Objektcode gestellt.
Fuer die Speicherung von lokalen Variablen, Zwischenergebnissen und Parametern wird der STACK benoetigt. Der Heap bewegt sich vom ersten freien Speicherplatz in Richtung Stack und der Stack vom letzten freien Speicherplatz in Richtung Heap.
Die K-Direktive ueberprueft ob der Heap und der Stack Zusammenstossen (vergl. Ziffer 4.1.4.6.11.).

4.11.2. DCP Systemaufruf

Aus Kompatibilitaetsgruenden zu Turbopascal wurde der Systemaufruf MSDOS beibehalten. Das Betriebssystem DCP ist kompatibel zu dem internationalen Betriebssystem MSDOS .

Syntax:

```
msdos (<recsatz>)
```

Durch das System wird vor Aufruf von msdos das Laden des Registersatzes aus den Variablen des Recordsatz realisiert. Nach Beendigung des Systemaufrufs wird der aktuelle Registerstand wieder in die Variablen des Records zurueckgespeichert.

Beispiel:

```
Type abc = record
                    ax,bx,cx,dx,bp,di,si,ds,es,flagreg:integer;
           end;

Var  def : abc;
     g   : integer;

begin
   .
   ax:=$xxxxx;
   msdos (def);
   g:=hi(ax);
   .
end.
```

4.11.3. INLINE-Maschinencode

Das PASCAL-Programmiersystem stellt mit den INLINE-Anweisungen einen sehr brauchbaren Weg zum direkten Einfuegen von Maschinencode in den Pascal-Programmtext zur Verfuegung.
Eine INLINE-Anweisung besteht aus dem reservierten Wort INLINE und einer oder mehreren Codeelementen die durch Schraegstrich voneinander getrennt und in runden Klammern gesetzt sind. Ein Codeelement besteht aus ein oder mehreren Datenelementen, getrennt durch die Zeichen plus oder minus. Datenelemente koennen sein:

- integer Konstante
- Variablenbezeichner
- Prozedurbezeichner
- Funktionsbezeichner
- Kommandozaehlerreferenz (gekennzeichnet durch einen Stern)

Beispiel:

```
inline (20/$2222/abc+1/def-*+5);
```

Die Zeichen '<' und '>' koennen zur Festlegung der Codegroesse benutzt werden. Wenn ein Codesegment mit einem '<' Zeichen beginnt wird nur das niederwertige Byte des 16-bit Wertes codiert. Bei Verwendung des Zeichens '>' wird immer ein 16-bit Wert codiert. Dieses ist auch der Fall wenn das hoechste Byte null ist.

Beispiel:

```
inline (<$9988/>$77);
```

Durch diese Anweisung werden drei Bytes codiert: $88, $77, $00.

4.11.4. Nutzergeschriebene I/O-Driver

Fuer einige Anwendungen ist es fuer den Programmierer praktisch, seine eigenen Ein- und Ausgabedriver zu schreiben, d.h. Routinen, die Ein- und Ausgabe von Zeichen zu und von externen Geraeten liefern. Die folgenden Driver sind Teile des Programmiersystems und werden von Standard-I/O-Drivern verwendet (obgleich sie selbst nicht als Standard-Prozeduren oder Funktionen aufgerufen werden duerfen).

```
FUNCTION    CONST : BOOLEAN;
FUNCTION    CONIN : CHAR;
PROCEDURE   CONOUT  (Ch:CHAR);
PROCEDURE   LSTOUT  (Ch:CHAR);
PROCEDURE   AUXOUT  (Ch:CHAR);
FUNCTION    AUXIN : CHAR;
PROCEDURE   USROUT  (Ch:CHAR);
FUNCTION    USRIN : CHAR;
```

Die CONST-Routine wird durch die Funktion KEYPRESSED aufgerufen, die CONIN- und CONOUT-Routinen werden durch die CON:-, TRM:-, und KBD-Geraete verwendet; die LSTOUT wird durch das Geraet LST: verwendet; die Routinen AUXOUT und AUXIN werden durch das Geraet AUX: verwendet und die Routinen USROUT und USRIN werden durch das Geraet USR: verwendet. Standardmaessig verwenden diese Driver die entsprechenden Eintrittspunkte des DCP-Systems.
Diese Zuordnung kann jedoch vom Programmierer geaendert werden, indem er den folgenden Standardvariablen die Adresse eigener Driver-Prozeduren oder Driver-Funktionen zuweist:

Variable	enthaelt die Adresse der Funktion
CONSTPTR	ConSt Funktion
CONINPTR	ConIn Funktion
CONOUTPTR	ConOut Prozedur
LSTOUTPTR	LstOut Prozedur
AUXOUTPTR	AuxOut Prozedur
AUXINPTR	AuxIn Funktion
USROUTPTR	UsrOut Prozedur
USRINPTR	UsrIn Funktion

Eine vom Nutzer geschriebene Driver-Prozedur oder Driver-Funktion muss mit den oben beschriebenen Definitionen uebereinstimmen, d.h. ein CONST-Driver muss eine BOOLEAN-Funktion, ein CONIN-Driver muss eine CHAR-Funktion sein usw.

Anhang
A. Compilerdirektiven

Compilerdirektiven werden mit {$<Direktive>} an den Beginn einer Quelltextzeile geschrieben.

<Direktive> (Standard erstgenannt)	Wirkung
B+	Standardfile INPUT gleich CON
B-	Standardfile INPUT gleich TRM
C+	Eingabeinterpretation ^C Programmabbruch ^S Unterbrechung Bildschirmausgabe
C-	CTRL-Zeichen werden nicht interpretiert (beschleunigter Ablauf)
D+	Geraeteueberpruefung ein (wenn Datei ein Geraet ist,dann keine Pufferung)
D-	Geraeteueberpruefung aus
G0	Puffergroesse 0 (Eingabedatei CON: oder TRM:)
Gn	Puffergroesse n
I+	E/A-Fehlerbehandlung durch das Laufzeit-/PASCAL-System
I-	E/A-Fehlerbehandlung ueber IORESULT durch den Programmierer
K+	Stackpruefung ein
K-	Stackpruefung aus
P0	Puffergroesse 0 (Eingabedatei CON: oder TRM:)
Pn	Puffergroesse n
R-	Ohne Index- und Bereichsueberwachung waehrend der Laufzeit
R+	Index- und Bereichsueberwachung (langsamerer Ablauf)
U-	Keine Programmunterbrechung durch den Benutzer waehrend der Laufzeit
U+	Unterbrechung waehrend der Laufzeit mit ^C moeglich(langsamerer Ablauf)
V+	Stringlaenge bei Parameteruebergabe wird geprueft
V-	Stringlaenge kann verschieden sein
I <Name>	File <Name> wird an diese Stelle kopiert (<Name> = Include-Datei)

B. Fehlermeldungen Compiler

Die folgende Liste enthaelt die Fehlermitteilungen des Compilers. Wenn beim Compilieren ein Fehler auftritt, wird immer die Fehlernummer angezeigt. Der Text wird nur ausgegeben, wenn der Fehlertext geladen ist (Anwort "J" auf die erste Frage beim Start des Systemkerns). Die meisten der Fehlermitteilungen erklaeren sich von selbst, aber bei einigen sind noch kurze Erlaeuterungen angefuegt.

01: ';' fehlt

02: ':' fehlt

03: ',' fehlt

04: '(' fehlt

05: ')' fehlt

06: '=' fehlt

07: ':=' fehlt

08: '[' fehlt

09: ']' fehlt

10: '.' fehlt

11: '..' fehlt

12: BEGIN fehlt

13: DO fehlt

14: END fehlt

15: OF fehlt

17: THEN fehlt

18: DO oder DOWNTO fehlt

20: Boolscher Ausdruck erwartet

21: File-Variable erwartet

22: INTEGER/BYTE-Konstante erwartet

23: INTEGER/BYTE-Ausdruck erwartet

24: INTEGER/BYTE-Variable erwartet

25: INTEGER/BYTE- oder REAL-Konstante erwartet

26: INTEGER/BYTE- oder REAL-Ausdruck erwartet

27: INTEGER/BYTE- oder REAL-Variable erwartet

28: Zeiger-Variable erwartet

29: Record-Variable erwartet

30: Einfacher Typ erwartet

31: Einfacher Ausdruck erwartet

32: STRING-Konstante erwartet

33: STRING-Ausdruck erwartet

34: STRING-Variable erwartet

35: Textfile erwartet

36: Typ-Bezeichner erwartet

37: Ungetyptes File erwartet

40: Undefinierte Marke

41: Undefinierter Bezeichner oder Syntaxfehler
(Unbekannter Marken-, Konstanten-, Typ-, Variablen- oder Feldbezeichner oder Syntaxfehler in der Anweisung)

42: Undefinierter Zeigertyp in vorhergehenden Typdefinitionen

43: Doppelter Bezeichner oder doppelte Marke

44: Typ unvertraeglich
1. Inkompatible Typen von Variablen und Ausdruecken in einer Ergibtanweisung.
2. Inkompatible Typen von aktuellen und formalen Parametern bei einem Unterprogrammaufruf.
3. Der Typ eines Ausdruckes ist inkompatibel mit dem Indextyp in einer ARRAY-Ergibtanweisung.
4. Die Typen der Operanden eines Ausdruckes sind inkompatibel.

45: Konstante ausserhalb des zulaessigen Bereiches

46: Konstante und CASE Selektortyp unvertraeglich

47: Operanden- und Operatortyp unvertraeglich

48: Ungueltiger Ergebnis-Typ

49: Unzulaessige STRING-Laenge
(Die Laenge eines STRING muss im Bereich 1..255 liegen)

50: STRING-Konstantenlaenge unvertraeglich

51: Ungueltiger Teilbereichstyp
(Gueltige Basistypen sind alle Skalartypen, ausser REAL)

52: Untere > obere Grenze

53: Reserviertes Wort
(Das Wort kann nicht als Bezeichner verwendet werden)

54: Unzulaessige Wertzuweisung

55: STRING-Konstante ueberschreitet Zeile

56: Fehler in einer INTEGER/BYTE-Konstante

57: Fehler in einer REAL-Konstante

58: Unzulaessiges Zeichen im Bezeichner

60: Konstanten sind hier nicht erlaubt

61: Files oder Zeiger sind hier nicht erlaubt

62: Strukturierte Variablen sind hier nicht erlaubt

63: Textfiles sind hier nicht erlaubt

64: Textfiles oder ungetypte Files sind hier nicht erlaubt

65: Ungetypte Files sind hier nicht erlaubt

66: Eingabe/Ausgabe ist hier nicht erlaubt

67: Files erfordern VAR-Parameter

68: Filekomponenten duerfen keine Files sein

69: Unzulaessige Ordnung von Feldern

70: SET-Basistyp ausserhalb des zulaessigen Bereiches
Der Basistyp einer Menge muss ein ordinaler Typ mit nicht mehr als 256 Werten oder ein Teilbereich mit Grenzen im Bereich 0..255 sein.

71: Ungueltiges GOTO
Eine GOTO-Anweisung ausserhalb einer FOR-Schleife kann sich nicht auf eine Marke in dieser FOR-Schleife beziehen. Auch gelten Marken nicht in eingeschlossenen Unterprogrammen.

72: Marke nicht im gleichen Block

73: Undefiniertes FORWARD-Unterprogramm
Ein Unterprogramm wurde FORWARD-deklariert, aber der zugehoerige Programmkoerper existiert nicht.

74: INLINE-Fehler

75: Unzulaessiger Gebrauch von ABSOLUTE
1. Vor dem Doppelpunkt darf nur ein Bezeichner bei der Definition einer absoluten Variablen stehen.
2. Das Wort ABSOLUTE darf nicht innerhalb eines Satzes verwendet werden.

76: OVERLAY und FORWARD unvertraeglich

77: Unzulaessiges OVERLAY im Indirektmodus

90: File nicht gefunden
Das spezifizierte INCLUDE-File existiert nicht.

91: Vorzeitiges Ende des Quell-Files

92: Anlegen des OVERLAY-Files unmoeglich

93: Ungueltige Compilerdirektive

96: Unzulaessiges Schachteln von INCLUDE-Files

98: Speicherueberlauf

99: Compilerueberlauf
Es ist nicht genuegend Speicherplatz vorhanden, um das Programm uebersetzen zu koennen. Sie muessen Ihr Programm in kleinere Segmente teilen und INCLUDE-Files verwenden.

C. Fehlermeldungen Laufzeitsystem

C.1. Allgemeine Laufzeitfehler

Fehler fuehren zur Laufzeit eines Programms zum Abbruch und zur Anzeige der Mitteilung:

```
Laufzeit- Fehler <nn>, PC = <Adresse>
Programmabbruch
```

wobei nn die Fehlernummer und <Adresse> die Adresse im Programmcode ist, an der der Fehler auftrat. Diese Stelle kann bei geladenem Quellcode mit F im O-Menue des Systemkerns gesucht werden.

nn	Bedeutung
01	Gleitkommaueberlauf
02	Division durch Null
03	Fehler im Argument von SQRT (< Null)
04	Fehler im Argument von LN (<= Null)
10	Falsche STRING-Laenge (auch in Ergibtanweisungen) (1. Eine STRING-Kettung ergibt einen STRING, der mehr als 256 Zeichen hat. 2. Nur STRING's der Laenge 1 koennen in ein Zeichen konvertiert werden.)
11	Fehlerhafter STRING-Index (ausserhalb 1 - 255)
90	Index ausserhalb des zulaessigen Bereiches
91	Ordinaler Typ ausserhalb des Wertebereiches (auch bei Teilbereichstypen)
92	Wert ausserhalb des INTEGER-Bereiches
FF	Halden/Kellerspeicher-Kollision. Es wurde die Prozedur NEW oder ein rekursives Unterprogramm aufgerufen und es gibt zwischen Heap-Pointer und dem Rekursionsstack-Pointer keinen freien Speicherplatz mehr.

C.2. Ein/Ausgabe-Laufzeitfehler

Tritt waehrend der Laufzeit bei einer Ein- oder Ausgabeoperation ein Fehler auf und ist die Systemueberwachung aktiv (bei aktiver I-Compiler-Direktive), erfolgt ein Programmabbruch mit folgender Fehlermitteilung:

EA-Fehler <nn>, PC = <Adresse>

<nn> ist die EA-Fehlernummer und <Adresse> die Adresse im Programm, an der der Fehler auftrat.
Wenn die EA-Fehlerpruefung passiv ist {$I-}, dann erfolgt kein Programmabbruch. Man kann das Ergebnis der EA-Operation mittels der Funktion IORESULT abfragen und so entsprechende Massnahmen treffen. Ein Fehler kann bei geladenem Quelltext mit F im Ø - Menue des Systemkerns lokalisiert werden.

nn	Bedeutung
Ø1	Dieses File existiert nicht
Ø2	File fuer Leseoperationen nicht vorbereitet
Ø3	File fuer Schreiboperationen nicht vorbereitet
Ø4	File nicht geoeffnet
1Ø	Fehler im numerischen Format
2Ø	Operation auf logischem Geraet nicht erlaubt
21	Im Direktmodus (Zielauswahl H) nicht erlaubt
22	ASSIGN fuer vordefinierte Filevariablen nicht erlaubt
9Ø	Recordlaenge nicht vertraeglich
91	Position ausserhalb des Files
99	Vorzeitiges Fileende
FØ	Disketten-Schreibfehler (Diskette ist voll)
F1	Directory voll
F2	Fileumfang zu gross (Versuch mit WRITE einen 65536. Satz zu schreiben.)
FF	File nicht mehr unter Kontrolle (Versuch ein File mit CLOSE zu schliessen, das nicht mehr in der Directory steht, z.B. durch Diskettenwechsel.)

D. Interne Datenformate

Im folgenden bezeichnet @ das Offset des ersten Bytes einer Variablen eines entsprechenden Typs. Die Standardfunktion SEG kann man zur Ermittlung der Segmentbasisadresse fuer eine beliebige Variable verwenden.
Fuer die unterschiedlichen Variable sind andere Segmente vorgesehen.

- **globale Variable**
  das Offset bezieht sich auf das DS- Register
- **lokale Variablen**
  das Offset bezieht sich auf das BS-Register
- **typisierte Konstanten**
  das Offset bezieht sich auf das CS-Register

D.1. Basis-Datentypen

D.1.1. Skalare

In einem einzigen Byte werden folgende Skalare gespeichert:
- INTEGER-Teilbereiche, wenn beide Grenzen im Bereich 0...255 liegen,
- BOOLEAN,
- CHAR und
- deklarierte ordinale Datentypen mit weniger als 256 moeglichen Werten.

Dieses Byte enthaelt die Ordnungszahl der Variablen.
Folgende ordinale Datentypen werden in 2 Bytes gespeichert:
- INTEGER,
- INTEGER-Teilbereiche, wenn mindestens eine der Grenzen nicht im Bereich 0...255 liegt, und
- deklarierte ordinale Datentypen mit mehr als 256 moeglichen Werten.

Diese Bytes enthalten ein Zweierkomplement - 16-Bit-Wert, wobei der niederwertige Teil zuerst gespeichert wird (umgekehrtes Byte-Format).

D.1.2. REAL-Zahlen

REAL-Zahlen belegen 6 Bytes und stellen eine Gleitkommazahl mit 40-Bit-Mantisse und einem 8-Bit-Exponenten zur Basis 2 dar. Im ersten Byte wird der Exponent und in den naechsten 5 Bytes die Mantisse gespeichert mit dem niederwertigen Byte zuerst.

```
@       Exponent
@+1     niederwertiger Teil
.
.
@+5     hoeherwertiger Teil
```

Der Exponent verwendet ein Binaerformat mit einem Offset von $80, d.h., ein Exponent von $84 bedeutet, die Mantisse ist mit 2^($84-$80) = 2^4 = 16 zu multiplizieren. Der Gleitkommawert ist Null, wenn der Exponent Null ist.
Den Wert erhaelt man, indem man die 40-Bit-Integerzahl (ohne Vorzeichen) durch 2^40 dividiert. Die Mantisse wird immer normalisiert, d.h., das hoechstwertige Bit (Bit 7 des fuenften Byte) wird immer als 1 interpretiert. Das Vorzeichen der Mantisse wird jedoch auch in diesem Bit gespeichert. 1 bedeutet negatives und 0 positives Vorzeichen.

D.1.3. STRING

Ein STRING belegt im Speicher immer 1 Byte mehr als seine angegebene (maximale) Laenge. Das 1. Byte enthaelt die aktuelle Laenge des STRING; dieses wird auch als dynamisches Byte bezeichnet. Die folgenden Bytes enthalten die aktuellen Zeichen des STRING. Das 1. Byte steht auf der niedrigsten Adresse. In der folgenden Tabelle bezeichnet L die aktuelle Laenge und Max die maximale Laenge des STRING:

@	aktuelle Laenge: L
@+1	1. Zeichen
@+2	2. Zeichen
.	
.	
@+L	letztes Zeichen
@+L+1	nicht verwendet
.	
.	
@+Max	nicht verwendet

Der Bereich ab @+L+1 wird nicht geloescht.

D.1.4. Mengen

Ein Element einer Menge belegt ein Bit. Die maximale Anzahl von Elementen einer Menge betraegt 256, eine Mengevariable belegt also niemals mehr als 32 (=256/8) Bytes.
Die Bytes, deren Bits alle statisch Null sind (d.h. nicht verwendet werden), werden nicht gespeichert.
Die Zahl der Bytes, die von einer Mengenvariablen belegt wird, ist (Max DIV 8) - (Min DIV 8) + 1, wobei Max und Min die oberen und unteren Grenzen des Basistyps der Menge sind. Die Speicheradresse eines speziellen Elements E ist:

```
Elementeadresse = @ + (E DIV 8) - (Min DIV 8)
```

Die Bit-Adresse innerhalb des Bytes mit der Elementeadresse betraegt

```
BitAdresse = E MOD 8
```

wobei E die Ordnungszahl des Elementes ist.

D.1.5. File-Interface-Block

Jede Filevariable besitzt einen ihr zugeordneten File-Interface-Block. Im FIB stehen verschiedene Informationen ueber das Diskettenfile oder Geraet, das aktuell dem File zugeordnet ist.

(LSB = niedrigstes signifikantes Byte,
MSB = hoechstes signifikantes Byte):

			Datei geschlossen	Textdatei
@	Datei-Kennzeichen	(LSB)	0ffh	
@+1	Datei-Kennzeichen	(MSB)	0ffh	
@+2	Recordlaenge	(LSB)	00h	Flag-Byte
@+3	Recordlaenge	(MSB)	00h	Zeichenpuffer
@+4	Puffer-Offset	(LSB)		
@+5	Puffer-Offset	(MSB)		
@+6	Puffergroess	(LSB)		
@+7	Puffergroess	(MSB)		
@+8	Pufferzeiger	(LSB)		
@+9	Pufferzeiger	(LSB)		
@+10	Pufferende	(LSB)		
@+11	Pufferende	(MSB)		
@+12	erstes Byte des Dateipfad			
.				
.	max. 63 ASCII Zeichen String der durch			
.	Nullbyte beendet wird			
.				
@+75	letztes Byte des Dateipfad			

Bei Textdateien hat das Flag-Byte folgende Bedeutung:

@+2		Dateityp:
Bit 0..3	Dateityp	0 Diskettendatei
Bit 5	Read Zeichenflag	1 CON:
Bit 6	Outputflag	2 KBD:
Bit 7	Inputflag	3 LST:
		4 AUX:
		5 USR:

D.1.6. Zeiger

Ein Zeiger besteht aus 2 Bytes, die eine 16-Bit-Speicheradresse enthalten, die im umgekehrten Byte-Format gespeichert ist, d.h., der niederwertige Adressteil wird zuerst gespeichert. Der Wert NIL entspricht einem Wort mit dem Wert Null.

D.2. Strukturen

Datenstrukturen werden entsprechend den Basistypen aufgebaut, die verschiedene Struktur-Methoden verwenden. Es gibt drei unterschiedliche Strukturmethoden:

- ARRAY,
- RECORD und
- Diskettenfiles.

Die Strukturierung der Daten beeinflusst nicht das interne Format der Datentypen.

D.2.1. ARRAY

Die Komponenten mit der niedrigsten Indexadresse werden auf der niedrigsten Speicheradresse gespeichert. Ein mehrdimensionales ARRAY wird so abgespeichert, dass die am weitesten rechts stehende Dimension zuerst aufgebaut wird.

D.2.2. RECORD

Das erste Feld eines Records wird auf der niedrigsten Speicheradresse gespeichert. Die Laenge des Records ist gleich der Summe der Laenge der einzelnen Felder, wenn der RECORD keinen varianten Teil hat. Hat er einen varianten Teil, so wird die Gesamtzahl der belegten Bytes bestimmt durch die Laenge des festen Teils plus Laenge der maximalen Groesse des varianten Teils.
Jeder variante Teil beginnt an der gleichen Speicheradresse.

D.2.3. Diskettenfiles

Diskettenfiles unterscheiden sich von den anderen Datenstrukturen dadurch, dass ihre Daten nicht im internen Speicher, sondern in einem File auf einer externen Diskette gespeichert sind. Ein Diskettenfile wird bei der Uebertragung durch einen File-Interface-Block (FIB) gesteuert (siehe D.1.5.). Im allgemeinen gibt es zwei unterschiedliche Filetypen:
- Binaerfiles und
- Textfiles.

D.2.3.1. Binaerfiles

Ein Binaerfile besteht aus einer Folge von Saetzen gleicher Laenge und gleichem internen Format. Die Saetze werden kontinuierlich hintereinander gespeichert, um die Filespeicherung zu optimieren. Auf Binaerfiles kann sequentiell und wahlfrei zugegriffen werden.

D.2.3.2. Textfiles

Die Basiskomponenten eines Textfiles sind Zeichen (CHAR), und ausserdem wird jedes Textfile in Zeilen eingeteilt. Jede Zeile besteht aus einer beliebigen Zahl von Bytes und endet mit einer CR/LF-Folge (ØD/ØA). Das File wird durch das Zeichen CTRL Z (1A) beendet (EOF).

E. Stichwortverzeichnis

*** Stichwortverzeichnis ***