Gerd Goldammer

PASCAL

für die Anwendung in der Wirtschaft

Die Wirtschaft

PASCAL

für die Anwendung in der Wirtschaft

Prof. Dr. sc. oec.
Gerd Goldammer

Verlag Die Wirtschaft Berlin

Lektor:
Petra Tredup

Redaktionsschluß: 1. Juni 1987

Goldammer, Gerd:
PASCAL für die Anwendung in der Wirtschaft / Gerd Goldammer. – 1. Aufl. – Berlin : Verl. Die Wirtschaft, 1987. – 191 S. : 295 Abb.

ISBN 3-349-00130-0

Am Friedrichshain 22, Berlin, 1055
Lizenz-Nr. 122, Druckgenehmigungs-Nr. 195/119/87
LSV 0394
Einbandgestaltung: Klaus Herrmann
Typografie: Verlag Die Wirtschaft/Herbert Hölz
Printed in the German Democratic Republic
Gesamtherstellung:
(140) Druckerei Neues Deutschland, Berlin
Bestell-Nr. 675 9 768
01500

Inhaltsverzeichnis

Vorwort

Das vorliegende Buch vermittelt die Programmiersprache PASCAL mit allen Erweiterungen, die sich für Mikrorechner durchgesetzt haben. Es dient der Aus- und Weiterbildung sowie dem Selbststudium auf dem Gebiet der angewandten Informatik. Dabei wird speziellen Forderungen für die Anwendung in der Wirtschaft Rechnung getragen. Dem dienen insbesondere typisch ökonomische Beispiele, eine erweiterte Darstellung zu Datenstrukturen, Files und nichtnumerischen Operationen. Die Beispiele sind auch für den Nichtökonomen, der sich in die jeweiligen Sachverhalte hineindenkt, verständlich.
Dem Leser sollten die Grundstruktur eines Computers und Begriffe wie Hauptspeicher, Bildschirm, Cursor, Tastatur, Drucker bekannt sein. Spezielle Vorkenntnisse werden nur bei der Darlegung der Maschinencode-Anschlüsse vorausgesetzt. Die Kenntnis der internen Darstellung von Daten im Kapitel 2 ist für das Verstehen der Arbeitsweise des Computers und seines Fehlerverhaltens unerläßlich. Diese Darlegungen müssen gegebenenfalls unter Hinzuziehung ausführlicherer Literatur erarbeitet werden. Wenn ein Fachwort nicht bekannt ist, hilft das Sachwortregister, zunächst an anderer Stelle nachzulesen.
Die Erfahrung zeigt, daß die erste Programmiersprache im allgemeinen nicht ausschließlich aus einem Buch erlernt werden kann. Eine besondere Rolle kommt deshalb auch der begleitenden praktischen Tätigkeit zu. Es wird dringend empfohlen, die Beispiele durchzuarbeiten und einige Übungsaufgaben zu lösen. Die gesamte Darstellung geht davon aus, daß der Leser sich wirklich mit dem "fremden" Programmtext der Beispiele auseinandersetzt. Die Übungsaufgaben beziehen sich mehr als sonst in der Literatur auf das Verstehen und Ändern dieser, also bestehender Programme. Das geschieht absichtlich, weil es viel mehr darauf ankommt, bekannte Programme anzupassen und in einen Rahmen zu fügen, als selbst neue Programme zu schreiben. Der Leser findet eine Fülle "fertiger" Programme in der PASCAL-Literatur oder in PASCAL-Bibliotheken. Er soll angeregt werden, sie seinen Bedürfnissen anzupassen, statt mit weitaus höherem Aufwand selbst völlig neu zu programmieren. Viele der hier verwendeten Programme und Moduln sind ebenfalls wiederverwendungsfähig, obwohl natürlich einige – und auch die Programmstruktur – dem Lehrzweck untergeordnet werden mußten.
Besitzt der Leser bereits Kenntnisse in einer anderen Programmiersprache, sollte er sich ebenfalls der Mühe unterziehen, einige Beispiele durchzuarbeiten. Beim Lösen der Übungsaufgaben ist es dann erforderlich, auf das "GOTO" zu verzichten, um nicht nur die Syntax von PASCAL, sondern auch einen modernen, auf softwaretechnologische Erfordernisse ausgerichteten Programmierstil zu erlernen.
Dieses Buch ist zugleich Arbeitsmaterial. Man kann es noch benutzen, wenn die Sprache

selbst längst beherrscht wird. Dazu dient der umfangreiche Anhang mit Tabellen, Übersichten und Diagrammen für die praktische Arbeit. Der Anhang enthält auch alle Details, die im laufenden Text im Interesse einer guten Verständlichkeit weggelassen wurden.
Angesichts der Vielfalt verschiedener technischer Realisierungen des PASCAL (PASCAL-Implementationen) in der DDR und ihrer Besonderheiten wurde unter praktischem Aspekt wie folgt verfahren:

- Elemente von PASCAL, die in allen Implementationen enthalten sind, gewissermaßen der PASCAL-Kern, werden durch geschlossene Linien als solche in den Syntaxdiagrammen sichtbar gemacht.
- Der ISO-Standard (International Standards Organization) und damit die Sprachelemente des ESER-Pascal (PASCAL/P) und des SKR-Pascal entsprechen dem PASCAL-Kern zuzüglich der Befehle GET und PUT sowie der entsprechenden Arbeit mit dem Filepuffer. Diese Befehle sind dargestellt, so daß auch der ISO-Standard abgrenzbar ist.
- Alle wesentlichen Funktionen und Prozeduren der vielen PASCAL-Implementationen sind aufgenommen und – wenn nicht überall implementiert – in geschweifte Klammern eingeschlossen. Die Aufzählung regt die Schaffung von Funktionen und Prozeduren an und ermöglicht die Beurteilung des "eigenen" PASCAL-Systems.

Aus praktischen Gründen und um die Darstellung nicht übermäßig auszudehnen, wurde sie auf das Betriebssystem SCP des VEB Robotron beschränkt. Die Besonderheiten der Arbeit mit PASCAL unter den Betriebssystemen OS/ES (TSO), MOOS 1600 bzw. MUTOS für Groß- und Kleinrechenanlagen sind nicht enthalten. Hier wird auf das Buch „Programmieren mit PASCAL" von G. Paulin und H. Schiemangk sowie auf Systemunterlagen des VEB Robotron verwiesen. Für 16-Bit-Rechner ändern sich betriebssystemabhängige Teile von PASCAL, der Maschinencode für INLINE sowie Funktionen und Prozeduren, die mit Adressen arbeiten. Das ist aber nicht gravierend und kann der jeweiligen Dokumentation entnommen werden.
Es war noch eine weitere praktische Anforderung zu erfüllen. Zunehmend sind Programme typisch, die im Dialog mit dem Nutzer arbeiten. Für die Programmierung dieser Kommunikation gibt es Regeln, die wesentlich über die Akzeptanz einer Lösung entscheiden. Alle Programme wurden deshalb als interaktive Programme ausgelegt und somit die Grundkenntnisse vermittelt. Man erkennt gleichzeitig, daß die Programmierung der Kommunikation mit dem Nutzer einen größeren Teil des Programms beansprucht als die Programmierung der eigentlichen Verarbeitung. Das ist durchaus typisch für Dialoganwendungen.
Schließlich bleibt zu vermerken, daß die 40 Programme und Unterprogramme dieses Buches mit dem Programmiersystem PASCAL-880/S des VEB Robotron entwickelt und getestet wurden. Für die vielen Programmausschnitte, die verschiedene PASCAL-Systeme beschreiben, war das leider nicht möglich. Diese Programmausschnitte wurden gerahmt.

Die Arbeit an einem Fachbuch ist aufwendiger, als man es dem späteren Erzeugnis ansieht. Ich bedanke mich bei allen Fachkollegen, die dieses Vorhaben unterstützt haben. Das sind besonders die Herren Prof. Dr. sc. Menzel, Prof. Dr. sc. H.-F. Meuche, Prof. Dr. sc. Picht, Dr. sc. Tschirschwitz und Prof. Dr. sc. Wagner. Natürlich hatte ich die vielfältige

Hilfe der Kollegen der Sektion Mathematik und Datenverarbeitung der Handelshochschule Leipzig. Sie haben mir die Konzentration auf die Aufgabe erleichtert und manchen Handgriff abgenommen. Das waren vor allem die Herren Dr. sc. Aust, Dr. sc. Kirsten, Frau G. Fuchs und Studenten der Fachrichtung Mathematische Methoden und Datenverarbeitung in der Wirtschaft. Den größten Dank schulde ich meiner Frau Erika. Sie hat mich tatkräftig unterstützt. Ihr widme ich diese Arbeit.

Gerd Goldammer

Leipzig, im Mai 1987

1. Einführung in die PASCAL-Programmierung

1.1. Grundstruktur von PASCAL-Programmen

Die Programmierung dient dazu, den Computer für eine bestimmte Anwendung herzurichten. In einem einfachen Falle soll die Kostenfunktion

Kosten = f(x) = 0.417621 x

bekannt sein, wobei x die Leistungsmenge ist. Im Dialog mit dem Computer soll einem Nutzer die Ermittlung der Kosten für verschiedene Leistungsmengen ermöglicht werden. Ein PASCAL-Programm dafür kann wie folgt aussehen:

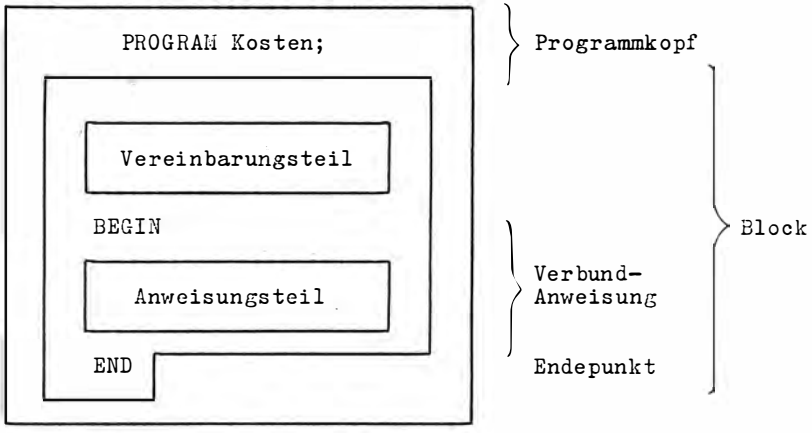
```
PROGRAM Kosten;
CONST Koeffizient = 0.417621;
VAR   Menge       : INTEGER;
BEGIN
  writeln('Kostenermittlung fuer verschiedene Leistungsmengen');
  write('Leistungsmenge: '); {Eingabeaufforderung}
  read(Menge); writeln;
  writeln('Die Kosten betragen ',Koeffizient * Menge:8:2,' M');
  write('Ende')
END.
```

Die Leistung dieses Programms ist natürlich sehr bescheiden. Das Problem auch. Sicher würde man einen Taschenrechner zu seiner Lösung vorziehen. Aber das wird sich in den folgenden Kapiteln ändern.
Dieses Programm soll ausgeführt werden. Die Kommunikation zwischen Nutzer und Computer erfolgt über Bildschirm und Tastatur.
Die Aufforderung WRITE (schreiben) im Programm richtet sich an den Computer, und dieser nutzt dafür den Bildschirm. READ bedeutet Lesen. Gelesen wird von der Tastatur, wenn der Nutzer die entsprechenden Tasten niederdrückt. Da auf dem Bildschirm zugleich auch ein Echo der Tastatureingabe dargestellt wird, läßt sich die Arbeitsweise des Programms exakt protokollieren. Nutzereingaben (eigentlich ihr Echo auf dem Bildschirm) werden hier unterstrichen.

```
Kostenermittlung fuer verschiedene Leistungsmengen
Leistungsmenge: 8500<ET>
Die Kosten betragen  3549.78 M
Ende
```

<ET> symbolisiert den Druck einer speziellen Taste (zum Beispiel ET1), mit der der Nutzer das Ende seiner Eingabe mitteilt. Es gibt kein Echo auf dem Bildschirm, oder es besteht darin, daß der Cursor (die Schreibmarke) an den Beginn der Folgezeile des Bildschirms gesetzt wird. Natürlich kann das Programm sofort erneut gestartet und die Kosten können für eine andere Leistungsmenge ermittelt werden.
Zunächst aber soll nicht die Arbeitsweise des Programms, sondern seine Struktur betrachtet werden. Es gibt verschiedene Klassifikationen. Hier wird zwischen Grobstruktur und Feinstruktur unterschieden.

Grobstruktur
Die für diese Klassifikation wichtigen Begriffe sind Programmkopf, Vereinbarungsteil, Anweisungsteil, Verbundanweisung und Block.



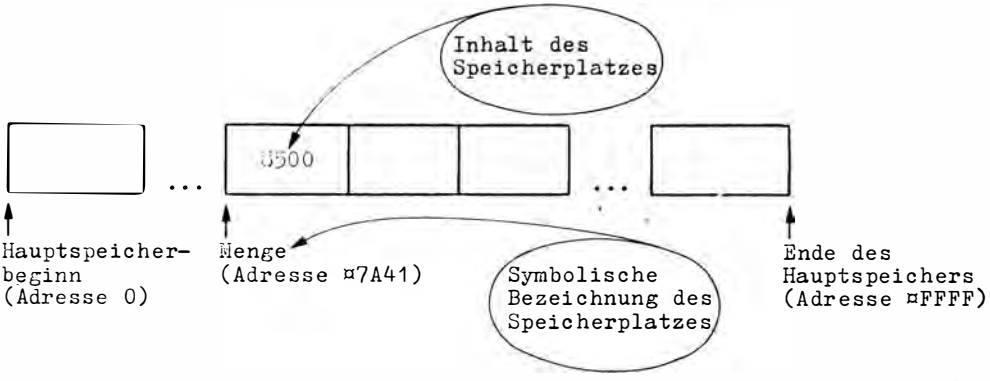
Der *Programmkopf* identifiziert das Programm von außen. Die Bezeichnung "Kosten" ist für das Programm selbst ohne Bedeutung. In älteren Sprachversionen und Versionen, die sich streng an den ISO-Standard halten, sind nach dem Programmbezeichner in Klammern externe Bezüge zu benutzten Ein- und Ausgabegeräten oder zu Datenobjekten (Speicherfiles) herzustellen. Als Standardgeräte (Gerätefiles) gelten dabei INPUT für ein Eingabegerät (bei Mikrorechnern die Tastatur) und OUTPUT für ein Ausgabegerät (bei Mikrorechnern der Bildschirm). Alle PASCAL-Versionen akzeptieren deshalb auch

```
PROGRAM Kosten(INPUT,OUTPUT);
```

Da diese Zuordnung von Geräten trivial ist und andere als diese Files für Mikrorechner ohnehin noch einmal im Programm spezifiziert werden müssen, wird in modernen PASCAL-Implementationen für Mikrorechner das Weglassen akzeptiert.
Der *Vereinbarungsteil* enthält im Programm KOSTEN lediglich die Befehle

```
.
CONST Koeffizient = 0.417621;
VAR   Menge       : INTEGER;
.
```

Der Befehl "CONST Koeffizient = 0.417621" bewirkt, daß dem Bezeichner "Koeffizient" die Konstante 0.417621 fest zugeordnet wird.
Überall, wo dieser Bezeichner später im Programmtext erscheint, steht er für diesen Wert. Während des Programmablaufs darf diese Belegung nicht verändert werden. Werden in einem Programm mehrere Konstanten definiert, braucht CONST (Constant) nicht jedesmal wiederholt zu werden. Die Syntax enthält Anhang C. Auf den Befehl "VAR Menge: INTEGER" hin wird Speicherplatz reserviert, der es ermöglicht, eine ganze Zahl aufzunehmen. Der Speicherplatzinhalt ist im Programm über einen Bezeichner "Menge" zugänglich. Das kann man sich wie folgt vorstellen:



"Menge" wird als Symbol für die konkrete Hauptspeicheradresse verwendet. Im allgemeinen ist es nicht erforderlich, die wirkliche Position im Speicher, die Adresse (hier im Beispiel die hexadezimale Zahl ¤7A41) festzustellen. Das übernimmt der Computer unter Nutzung einer Symboltabelle, die zwischen Bezeichner und Adresse vermittelt. Charakteristisch ist, daß die Adresse gleichbleibt, der Inhalt aber wechselt. "Menge" wird deshalb als Variable bezeichnet. Soll Speicherplatz für mehrere Variablen vereinbart werden, so können sie nach VAR (Variable) durch Komma getrennt aufgeführt werden.
Der Vereinbarungsteil ist natürlich bei größeren Programmen umfangreicher.
Er dient vor allem

- zur Übermittlung von Angaben über den Typ, das heißt den zulässigen Wertevorrat der im Programm erforderlichen Variablen und den dafür benötigten Speicherplatz (TYPE-Definition/VAR-Deklaration). Über diesen Teil wird das Typkonzept von PASCAL realisiert;
- der Vordefinition von Befehlsfolgen, die dann im Anweisungsteil über einen Bezeichner geschlossen zur Ausführung aufgerufen werden können (Routinen-Deklaration). Mit diesem Teil wird das Modulkonzept von PASCAL verwirklicht.

Außerdem besteht im Vereinbarungsteil die Möglichkeit, sogenannte Marken zu deklarieren, die später zur besonderen Adressierung von Befehlen im Anweisungsteil genutzt werden können (LABEL-Deklaration). Schließlich können Bezeichner wie im Beispiel für Konstanten festgelegt werden, so daß es im Programmtext genügt, den Bezeichner statt der Konstanten zu verwenden (CONST-Definition).
Der Vereinbarungsteil von PASCAL ist umfangreicher als beispielsweise der von PL/1-, FORTRAN- oder BASIC-Programmen. Wer diese Sprachen kennt und sich nunmehr PASCAL zuwendet, könnte das als Nachteil empfinden. Das ist es aber keineswegs. Ausführli-

che Vereinbarungen über Datenobjekte und Befehlsfolgen erleichtern die Lesbarkeit und damit erforderliche Änderungen eines Programms, ermöglichen während der Verarbeitung eine wirksame Fehlerkontrolle durch das Laufzeitsystem und steigern die Speichereffizienz. Die höheren Anforderungen von PASCAL an den Vereinbarungsteil sind wie die gesamte Sprache eine Schlußfolgerung aus Analysen anderer Programmiersprachen und tragen der wachsenden Forderung nach qualitätsgerechter Software Rechnung. Dieses Konzept von PASCAL lernt man sehr schnell schätzen.
Der *Anweisungsteil* besteht aus einer Folge von PASCAL-Anweisungen. Er ist durch die Textklammern BEGIN und END begrenzt. Im Beispiel sieht der Anweisungsteil ohne die Begrenzung durch BEGIN und END wie folgt aus:

```
writeln('Kostenermittlung fuer verschiedene Leistungsmengen');
write('Leistungsmenge: '); {Eingabeaufforderung}
read(Menge); writeln;
writeln('Die Kosten betragen ',Koeffizient*Menge:8:2,' M');
write('Ende')
```

Zwei aufeinanderfolgende Anweisungen werden durch Semikolon getrennt. Da END keine Anweisung ist, kann das Semikolon nach der Anweisung "write ('Ende')" entfallen. Wird es doch gesetzt, entsteht eine sogenannte Leeranweisung. Zu beachten ist, daß zwei Anweisungen auch auf einer Zeile stehen können.
Der Computer führt Anweisungen in der Reihenfolge ihrer Notation aus. Hier stellt sich das Programm zunächst vor, fordert danach eine Eingabe an und liest den Eingabewert. Anschließend wird das Ergebnis berechnet und mit Erläuterung ausgegeben. Danach teilt das Programm das Ende seiner Arbeit mit.
Eine Konstellation aus BEGIN, einer Folge von Anweisungen und END heißt in PASCAL Verbundanweisung. Verbundanweisungen sind auch als Teile einer Verbundanweisung erlaubt. So besteht ein PASCAL-Programm aus Programmkopf, Vereinbarungsteil, einer umfassenden Verbundanweisung, die selbst andere Anweisungen – darunter erneut eine Verbundanweisung – enthalten kann, und einem Punkt als Programmbegrenzer.
Über den Anweisungsteil wird das Steuerkonzept von PASCAL verwirklicht. Man unterscheidet hier nach einfachen und strukturierten Anweisungen und bei den strukturierten Anweisungen nach Sequenz (Folge), Selektion (Auswahl) und Iteration (Wiederholung). Besonders für die Iteration stehen im Vergleich zu anderen Programmiersprachen leistungsfähige Anweisungen zur Verfügung, die es ermöglichen, den Programmtext gut lesbar, vollständig prüfbar und änderungsfreundlich zu notieren. Das erleichtert das Testen eines Programms und erhöht seine Qualität.
Ein wichtiger Begriff ist auch der *Block*. Er besteht aus Vereinbarungsteil und Verbundanweisung. Im Gegensatz zu FORTRAN und BASIC ist PASCAL eine blockorientierte Sprache. Blockorientiert heißt, daß in den Vereinbarungsteil eines Blockes erneut ein Block eingelassen werden kann und in diesem eingelassenen Block Konstanten-, Typ-, Variablen- sowie Routinenvereinbarungen des übergeordneten Blockes gültig bleiben. Dadurch ermöglicht PASCAL eine zweckmäßige Modularisierung des Programmtextes und während der Verarbeitung den leistungsfähigen (rekursiven) Aufruf von Moduln durch sich selbst.

Gegenüber nichtblockorientierten Sprachen ergeben sich Vorteile hinsichtlich Sicherheit sowie Laufzeit- und Speicherplatzeffektivität. Eine detaillierte Übersicht über die Struktur von PASCAL-Programmen und das Datentypkonzept enthält Anhang A.

Feinstruktur
Betrachtet man den PASCAL-Text im Einführungsbeispiel hinsichtlich kleinerer Elemente, so ist noch eine andere Einteilung möglich. Sie ist zunächst nur theoretisch von Bedeutung. Deshalb kann bei Abschnitt 1.2. fortgesetzt und das Folgende zum späteren Nachschlagen verwendet werden. PASCAL nennt die nachstehenden Sprachbestandteile Morpheme.

Wortsymbole
Groß- und Kleinschreibung der Wortsymbole werden vom Computer nicht unterschieden. Die Wortsymbole sind im Beispiel PROGRAM, CONST, VAR, BEGIN und END.

Spezialsymbole
Im Programmbeispiel sind enthalten Semikolon, Gleichheitszeichen, Doppelpunkt, Apostroph (Hochkomma), Stern (als Multiplikationszeichen) sowie öffnende und schließende Klammern.

Bezeichner
Unterschieden werden vordefinierte und selbstdefinierte Bezeichner.
Vordefiniert sind durch PASCAL Typbezeichner (im Beispiel: "INTEGER"), Konstantenbezeichner (TRUE, FALSE als logische Konstanten, MAXINT als größte INTEGER-Zahl, NIL als Nullwert für Zeiger und implementationsabhängig PI mit dem Wert 3.1415926536) sowie Prozeduren und Funktionen (im Beispiel: "write", "writeln" und "read").
Selbstdefinierte Bezeichner legt der Programmierer fest. Dabei sind externe Bezeichner (im Beispiel: "Kosten") und interne Bezeichner (im Beispiel: "Koeffizient", "Menge") zu unterscheiden. Bezeichner bestehen aus Buchstaben und Ziffern, beginnend mit einem Buchstaben. Groß- und Kleinschreibung für Bezeichner werden vom Computer nicht unterschieden.
In einigen Implementationen kann außerdem das Unterstreichungszeichen genutzt werden. Die maximale Anzahl der signifikanten Zeichen ist verschieden. Für externe Bezeichner sind es oft sechs, für interne Bezeichner acht Zeichen. Es gibt aber auch Versionen, die bis zu 127 signifikante Zeichen erlauben.
Auf die richtige Bildung der Bezeichner mit mnemotechnischer (gedächtnisstützender) Beziehung zum Objekt selbst sollte große Sorgfalt verwendet werden. Sie trägt erheblich zur Selbstdokumentation (dem "sprechenden" Programmtext) von PASCAL-Programmen bei und ist ein wichtiger Vorzug von PASCAL zum Beispiel gegenüber BASIC.

Zahlen
Eine Zahl ist im Programmbeispiel mit 0.417621 enthalten. Das ist eine relle Zahl. Man beachte, daß entsprechend den internationalen Schreibweisen ein Punkt und nicht wie im Deutschen ein Komma zu schreiben ist. Für ganze Zahlen besteht in leistungsfähigen PASCAL-Implementationen auch die Möglichkeit, maximal vierstellige Hexadezimalzahlen unter Verwendung der Buchstaben A, B, C, D, E, F zu schreiben. Diesen Zahlen ist

das Sonderzeichen ¤ unmittelbar voranzustellen (zum Beispiel ¤FFFF = 65535). Vorzeichen sind in diesem Falle nicht erlaubt.

Zeichenketten
Das Programmbeispiel enthält mehrere Zeichenketten, darunter „Leistungsmenge:␣". Zeichenketten sind in Apostrophe eingeschlossen. Groß- und Kleinschreibung sowie Leerzeichen (hier mit "␣" gekennzeichnet) sind dabei signifikant. Apostrophe innerhalb von Zeichenketten sind doppelt zu schreiben. Zeichenketten dienen im allgemeinen der Verständigung mit dem Nutzer.

Kommentare
Das Programmbeispiel enthält mehrere Zeichenketten, darunter "Leistungsmenge: ␣". den in geschweifte Klammern eingeschlossen. Sie sind ausschließlich für den Programmierer bestimmt und sollen das Lesen des Programmtextes dokumentierend erleichtern. Der Compiler überliest diese Teile. Kommentare können überall dort stehen, wo Trennzeichen, vor allem Leerzeichen, erlaubt sind.
Kommentare sind gut von Direktiven zu unterscheiden, die dem Compiler oder anderen den Quelltext bearbeitenden Programmen (zum Beispiel Druckprogramme, Werkzeuge zur Dokumentation) übermittelt werden. Für Direktiven gelten strenge Sonderregeln. Nach der öffnenden geschweiften Klammer steht sofort (ohne Leerzeichen) ein Sonderzeichen und dann die eigentliche Direktive, zum Beispiel {¤I␣HANDEL.PAS}. In diesem Falle ist es eine Mitteilung für einen Compiler, an dieser Stelle den Quelltext des Files HANDEL.PAS einzufügen. Direktiven sind nicht standardisiert und folglich implementationsabhängig.

Marken
Marken sind in den meisten PASCAL-Implementationen maximal vierstellige positive Zahlen. In einigen Fällen sind auch Bezeichner erlaubt. Marken dienen der besonderen Kennzeichnung von Anweisungen, denen sie getrennt durch einen Doppelpunkt vorangestellt werden. Mit Hilfe einer GOTO-Anweisung können diese Anweisungen während der Ausführung des Programms unter Umgehung der Notationsfolge erreicht werden. Marken (und GOTO-Anweisungen), wenn sie nicht in Ausnahmefällen zur vorzeitigen Beendigung einer tiefen Schachtelung dienen, vermindern die Qualität eines Programms.

Wortsymbole, Bezeichner, Zahlen, Zeichenketten und Marken müssen sich im PASCAL-Text von ihrer unmittelbaren Umgebung abheben. Zum Beispiel kann man nicht die Bezeichner "Koeffizient" und "Menge" im Text ohne Trennung, also als "KoeffizientMenge" schreiben. Das wäre ein neuer Bezeichner. Als Trennzeichen in PASCAL gelten Spezialsymbole, Kommentare und beliebig viele Leerzeichen, in der praktischen Anwendung natürlich vor allem das Leerzeichen, weil es, außer innerhalb der Morpheme, überall eingefügt werden kann. Da aber auch Spezialsymbole und Kommentare trennen, sind in deren direkter Umgebung keine Leerzeichen erforderlich, natürlich aber möglich (im Programmbeispiel sind "Koeffizient␣ = ␣0.417621" und "Koeffizient=0.417621" völlig gleichwertig). Die Möglichkeiten, die sich aus gleichwertiger Groß- und Kleinschreibung, dem Einfügen von Leerzeichen und Kommentaren ergeben, sollen in PASCAL dazu genutzt werden, gut lesbare Texte zu schreiben.

1.2. Programmentwicklung in PASCAL

Ein Softwareerzeugnis, das für den Dauerbetrieb – also den oftmaligen Gebrauch durch verschiedene Nutzer – bestimmt ist, durchläuft die Phasen Aufgabenstellung, Spezifikation, Entwurf, Implementierung, Erprobung und Dauerbetrieb. In den Dauerbetrieb eingelagert sind Pflege, Wartung und Änderungen des Erzeugnisses, die einen erneuten Durchlauf durch die Phasen bedingen. Für Wartung, Pflege und Änderung in ökonomischen Anwendungen sind etwa 60 % des Gesamtaufwandes zu kalkulieren. Gerade hier zeigen sich die Vorteile von PASCAL mit transparenten, selbstdokumentierenden und änderungsfreundlichen Programmtexten. Von PASCAL werden unmittelbar Entwurf und Implementierug beeinflußt. Die Implementierung besteht aus Notieren und Editieren des PASCAL-Programms sowie aus Compilieren/Linken und Testen.

Entwerfen

Bei Softwareerzeugnissen größeren Umfangs, die für eine breite Nutzung bestimmt sind, kann nicht sofort "drauflos" programmiert werden. Die späteren Änderungen wären zu groß, oder die Qualität des Erzeugnisses würde den Anforderungen der Laufzeit- und Speichereffizienz, vor allem aber der Änderbarkeit nicht entsprechen. Aber auch bei kleineren Programmierarbeiten kann ein Entwurf nützlich sein. Für den PASCAL-gerechten Entwurf kommen eine Reihe von Methoden und Darstellungsmitteln in Betracht. Grundsätzlich sollte man bei aller Vielfalt folgende Anforderungen stellen:

- Der Entwurf sollte entsprechend dem Grundkonzept von PASCAL strukturorientiert erfolgen. Große Bedeutung kommt deshalb strukturorientierten Darstellungsmitteln zu, weil durch sie bereits eine gute Programmstruktur erzwungen werden kann. Ablauforientierte Darstellungsmittel wie der Programmablaufplan sind weniger geeignet.
- Der Entwurf muß in seinem Auflösungsgrad deutlich geringer sein als der spätere PASCAL-Text. Als Regel kann gelten, daß jedes Element des Entwurfs in der Verfeinerung durch bis zu zehn, im Mittel etwa fünf PASCAL-Befehle zu ersetzen ist. Daraus folgt, daß für Programme mit weniger als 100 Befehlen kein Entwurf erforderlich ist. PASCAL ist dann Entwurfs- und Implementierungsmittel. Diese Schwelle erhöht sich für den geübten Programmierer.
- Die Darstellung des Entwurfs sollte sich schrittweise verfeinern lassen, ohne daß jeweils der gesamte Entwurf neu anzufertigen ist.

Auf keinen Fall sollte sich der Entwurf verselbständigen. PASCAL selbst ist mit seiner klaren Sprache hervorragend als Ausdrucksmittel algorithmischen Denkens geeignet und sollte als solches auch genutzt werden.

Zur Auswahl und Handhabung von Darstellungsmitteln für den Entwurf sei auf Literatur verwiesen.[1] Hier wird für größere Aufgaben eine Strukturtabelle genutzt. Das ist einfach eine Gliederung. Sie entsteht, als würde man nicht ein Softwareerzeugnis herstellen, sondern eine Abhandlung schreiben wie einen Aufsatz, eine Diplomarbeit oder ähnliches. Diese Gliederung wird nur für den späteren Anweisungsteil sowie für eingelagerte Routinen aufgestellt. Ein Datenentwurf als Grundlage für den Vereinbarungsteil erweist sich hier nicht als erforderlich.

1 Vgl. Schumann, H.; Gerisch, M.: Softwareentwurf. Berlin: VEB Verlag Technik 1984.

Die Gliederung, also die Strukturtabelle, wird als Kommentar in den Anweisungsteil geschrieben. Für das Programmbeispiel erhielte man als Entwurf:

```
PROGRAM Kosten;

BEGIN
{ 1.Eroeffnung          }
{ 2.Eingabeaufforderung}
{ 3.Eingabe             }
{ 4.Berechnung          }
{ 5.Ausgabe             }
{ 6.Endemitteilung      }
END.
```

Dieser Entwurf läßt sich editieren und sogar compilieren. Durch schrittweise Verfeinerung können nun die PASCAL-Befehle im Vereinbarungs- und Anweisungsteil formuliert und mit dem Editor eingefügt werden. Die Entwurfskommentare bleiben dabei erhalten und dokumentieren damit zugleich das PASCAL-Programm. Stören später die Ziffern, können sie durch andere Zeichen, zum Beispiel den Bindestrich, ersetzt werden. Inhaltlich wird zum verwendeten Beispiel angemerkt, daß die Ziffern 1, 2 und 6 für interaktive Programme unbedingt verwendet werden sollten.
Das gewählte Beispiel ist natürlich zur Demonstration der Entwurfstechnik ungünstig. Einerseits ist die Aufgabenstellung zu klein; man würde keinen Entwurf benötigen. Andererseits ist lediglich eine Sequenz darzustellen ohne eine schrittweise Verfeinerung. Deshalb wird später, im Zusammenhang mit größeren Aufgaben, darauf zurückgekommen.

Notieren

Notieren ist das Schreiben des PASCAL-Textes außerhalb des Rechners. Die Notation ist zum Beispiel im Gegensatz zu FORTRAN formatfrei. Es sollten folgende Regeln beachtet werden, die das spätere Lesen und Ändern erleichtern:

- Ein PASCAL-Befehl belegt im allgemeinen eine Zeile. Bei längeren Befehlen besteht die Möglichkeit des Übergangs auf eine neue Zeile überall dort, wo Trennzeichen erlaubt sind. Die Zeilentrennung hat keinen Einfluß auf die Arbeitsweise des Programms.
- Reservierte Worte und vordefinierte Typ- sowie Konstantenbezeichner werden groß geschrieben. Vordefinierte Funktionen und Prozeduren, die auch als solche benutzt werden, werden klein geschrieben. Bezugnahmen auf Funktionen oder Prozeduren im Erläuterungstext dieses Buches werden zur Abhebung von der sonstigen Schreibweise auch groß geschrieben.
- Selbstdefinierte Bezeichner werden "normal", also groß und klein geschrieben.
- Je Schachtelungstiefe wird zwei Zeichen eingerückt und zurückgesetzt. Eine Schachtelung erfolgt innerhalb der Begrenzung BEGIN/END, REPEAT/UNTIL, CASE/END und wenn Befehle auf der Folgezeile fortgesetzt werden.

Im allgemeinen stehen Druckprogramme (LISTER) für PASCAL-Texte zur Verfügung, die eine gute Lesbarkeit durch Hervorhebung vordefinierter Worte, Anzeige der Schachtelungstiefe, Zeilennumerierung und anderes unterstützen und auch die "vergessene" Großschreibung übernehmen. Auch das Einrücken wird durchgeführt, wenn die entsprechenden Leerzeichen nicht editiert wurden. Die PASCAL-Texte in den folgenden Kapiteln wurden mit der D-Option im Systemservice des Programmiersystems PASCAL 880/S,

Version 1.1., gedruckt. Die Unterstreichung der Worte wird nur bei der Ausgabe zur besseren Lesbarkeit vorgenommen. Sie ist nicht Bestandteil des Programmtextes.

Editieren
Das Editieren dient der Eingabe des PASCAL-Textes in den Computer und dem Anlegen und Ändern des Quelltextfiles, der sogenannten PASCAL-Quelle. Die konkrete Vorgehensweise hängt ab vom verfügbaren Textmanipulationsprogramm, dem Editor. Unterschieden werden bildschirm- und zeilenorientierte Editoren. Die vielen Vorteile der Bildschirmarbeit werden nur durch bildschirmorientierte Editoren voll genutzt. Man unterscheidet grundlegende Editorkommandos
- zur Positionierung des Cursors
- zum Einfügen, Ersetzen und Löschen von Text ab der Cursorposition
- zum Suchen und Ersetzen von Zeichenketten
- zur Markierung und Handhabung von zusammenhängenden Textteilen.

Eine Übersicht über Editorkommandos, die in mehreren Editoren gleich sind, enthält Anhang D.

Kompilieren/Linken
Kompilieren und Linken (gleichbedeutend mit Übersetzen und Verbinden) dienen der Überführung des PASCAL-Quelltextes in ein lauffähiges Programm (den Maschinencode) oder einen während der Laufzeit zu interpretierenden einfachen Zwischencode (P-Code). Die entsprechenden Programme heißen Compiler und Linker (Linkage Editor). Es gibt auch PASCAL-Systeme, in denen Kompilieren und Linken geschlossene Vorgänge sind. In diesem Falle übernimmt der Compiler zusätzlich das Linken.
Beim Übersetzen und Verbinden werden zugleich, ausgelöst durch die PASCAL-Befehle, Befehlsfolgen einer Laufzeitbibliothek in das Programm eingefügt.
Die konkreten Kompilier- bzw. Linkkommandos sind abhängig von der PASCAL-Implementation.
Im allgemeinen kann oder muß der Compiler durch Direktiven zusätzlich gesteuert werden. Das ist erforderlich zum Beispiel
- bei Anwendung der Rekursion
- bei besonderen Anforderungen an die Behandlung von Ein- und Ausgabefehlern
- bei einer Forderung nach Überwachung von Indexgrenzen, die allgemein nicht vorgesehen ist, weil sie die Verarbeitung erheblich verlangsamen würde
- beim Einfügen von Quelltexten, die in anderen Files gespeichert sind.

Ausgewählte Direktiven enthält Anhang F. Beim Kompilieren und Linken wird der PASCAL-Text auf Fehler untersucht. Erkennbar sind in dieser Phase vor allem Verstöße gegen die Regeln der Sprache PASCAL (Syntaxfehler), Fehler bei der Verbindung verschiedener Programmeinheiten untereinander sowie mit der Laufzeitbibliothek und Fehler, die mit der Aufteilung des Speicherplatzes verbunden sind. Die Fehler werden mit Fehlernummer und nach Aufforderung auch mit Fehlertext angezeigt. Obwohl es hier ebenfalls ISO-Empfehlungen gibt, sind die Fehlermitteilungen weitgehend von der Implementation abhängig. Sie sind der Dokumentation zu entnehmen. Die Fehler werden durch Änderungen des Quelltextes unter Nutzung des Editors beseitigt.

Testen

Ein Programm, das die Phasen Kompilieren/Linken fehlerfrei passiert hat, kann gestartet werden. Dabei können folgende Fehler erkannt und angezeigt werden:

- Ein- und Ausgabefehler, die sich aus der konkreten Verbindung von Programm und externen Files ergeben (zum Beispiel Diskette voll, File nicht vorhanden);
- Laufzeitfehler, die sich aus der konkreten Belegung der Variablen während des Programmablaufs ergeben (zum Beispiel, wenn eine im Nenner stehende Variable den Wert Null besitzt, Indizes außerhalb des definierten Bereiches benutzt werden und anderes).

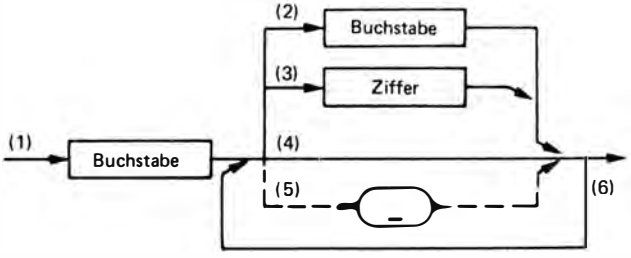
Nicht angezeigt werden Fehler, die ein Abweichen von der Aufgabenstellung bewirken, also falsche Ergebnisse dem Inhalt oder der Form nach. Diese Fehler müssen vom Programmierer selbst festgestellt und auch lokalisiert werden. Das ist das wichtigste beim Testen.

Eine Methode zur Lokalisierung solcher Fehler ist das Einfügen von Testzeilen zur Ausgabe aktueller Werte der kritischen Variablen an den entsprechenden Stellen des Quelltextes. Ist der Fehler gefunden, werden die Testzeilen wieder aus dem Quelltext herausgenommen. In der Programmierumgebung einiger Versionen existieren Debugger (Fehlersuchprogramme), die das Protokollieren der sich verändernden Speicherinhalte ermöglichen.

1.3. Metasprachliche Notation

Eine Programmiersprache soll leicht verständlich dargestellt werden. Früher oder später entsteht aber das Bedürfnis, für einen Befehl exakt zu wissen, was möglich und was untersagt ist. Spätestens ist dieser Zustand erreicht, wenn der Compiler einen Fehler anzeigt, dessen Ursache mehr als ein Schreibfehler des Programmierers ist. Damit ist es notwendig, eine Sprache (PASCAL) vollständig zu beschreiben. Eine entsprechende Beschreibungssprache heißt Metasprache.

Zur Beschreibung von PASCAL haben sich sogenannte Syntaxdiagramme durchgesetzt. Bei der Erläuterung der Feinstruktur von PASCAL-Programmen wurde in Abschnitt 1.1. der Begriff "Bezeichner" erläutert. Er soll hier durch ein Syntaxdiagramm exakt beschrieben werden. Damit wird zugleich die verbale Fassung präzisiert.



Die Ziffern an den Linien dienen lediglich der folgenden Erläuterung. Sie werden sonst nicht geschrieben.

In solchen Diagrammen werden hier und später unterschieden

- *Terminalsymbole.* Sie werden durch gerundete Sinnbilder gerahmt. Hier ist es das

Unterstreichungszeichen "_". Terminalsymbol heißt, daß das Symbol unverändert in den PASCAL-Quelltext zu übernehmen ist;

- *Nichtterminalsymbole.* Sie werden in eckige Sinnbilder eingeschlossen. Hier sind das "Buchstabe" und "Ziffer". Nichtterminalsymbol heißt, daß dieses Symbol im Quelltext durch ein gültiges Gebilde zu ersetzen ist, Buchstabe zum Beispiel durch A, B, ... Z oder a, b, ... z und Ziffer durch 0, 1, ... 9. Nichtterminalsymbole sind also letztlich Symbole, die an anderer Stelle, eventuell auch nicht in einem Syntaxdiagramm, definiert werden;
- *gerichtete Verbindungen.* Pfeile, die eine mögliche Aufeinanderfolge der Symbole festlegen, repräsentieren gewissermaßen die Bildungsregel der zulässigen Kombinationen. Hier wurde gegenüber den sonst üblichen Darstellungen in Pfeile mit geschlossener und mit unterbrochener (gestrichelter) Linie unterschieden. Pfeile mit geschlossener Linie sind für alle analysierten PASCAL-Implementationen gültig und können uneingeschränkt benutzt werden. Ob Pfeilen mit gestrichelter Linie gefolgt werden kann, ist implementationsabhängig. Der Programmierer muß sich vor der Benutzung in der jeweiligen Dokumentation informieren. Beim Durchlauf durch das Syntaxdiagramm ist natürlich die Richtung der Pfeile zu beachten.

Mit Hilfe des Syntaxdiagramms kann die Gültigkeit verwendeter Morpheme und Befehle genau geprüft werden. Die Richtigkeit der folgenden Bezeichner kann mit Durchläufen durch das obenstehende Syntaxdiagramm "Bezeichner" exakt nachgewiesen werden.

Bezeichner	Bildung	zulaessig
Kette	(1) (2) (6) (2) (6) (2) (6) (2)	ja
Satz 1	(1) (2) (6) (2) (6) (2) Fehler	nein(Leerzeichen)
Q_Wurzel	(1) (5) (6) (2) (6) (2) (6) (2) (6) (2) (6) (2) (6) (2)	ja(implementationsabhaengig)
1.Versuch	Fehler	nein(erste Zahl)

Anhang C enthält den in sich geschlossenen Satz (jedes Nichtterminalsymbol ist an anderer Stelle definiert) der PASCAL-Syntaxdiagramme. Diese sind außerdem so gefaßt, daß der Kreis O zugleich ein Trennzeichen charakterisiert, vor und nach dem beliebig viele Leerzeichen und Kommentare eingefügt werden können.
Im Text der Kapitel werden keine Syntaxdiagramme verwendet. Der Text soll gut lesbar sein. Außerdem werden viele Morpheme und Befehle schrittweise erläutert. Syntaxdiagramme müssen aber stets vollständig sein.

Aufgaben und Übungen

1. Editieren, compilieren/linken und starten Sie das Programmbeispiel KOSTEN mit einem verfügbaren PASCAL-System auf Ihrem Computer!

2. Prüfen und korrigieren Sie die Morpheme (Anhang B), die Editorkommandos (Anhang D) und die Compilerdirektiven (Anhang F) anhand der Systemunterlagen-Dokumentation Ihres PASCAL-Systems!

2. Sequentielle Ausführung von Anweisungen

2.1. Einfache Variablen für Zahlen und Zeichen

Ein Programm soll im Dialog mit dem Nutzer die Ermittlung optimaler Losgrößen bei verschiedenen Erzeugnissen und Mengen der Jahresproduktion ermöglichen. Die bekannte Losgrößenformel ist vereinfacht[2]:

$$\text{Optimale Losgröße} = \sqrt{\frac{2 \cdot \text{Rüstkosten} \cdot \text{Menge}}{\text{Lagerkosten} \cdot 12}}$$

Die Rüstkosten sind für alle Erzeugnisse gleich und betragen 300 Mark. Die Berechnungsergebnisse sind auf dem Drucker zu protokollieren.
Ein Entwurf ist angesichts des Umfangs der Aufgabe nicht erforderlich. Die Grobdarstellung aus Abschnitt 1.2. genügt den Anforderungen. Bei Berücksichtigung dieser Grundstruktur für interaktive Programme kann folgende Lösung entstehen:

```
PROGRAM Planung;
{Berechnung optimaler Losgroessen}
CONST Ruestkosten = 300;
VAR   Menge,Los    : INTEGER;
      Lagerkosten : REAL;
      Name         : STRING[20];
BEGIN
  writeln('Planung optimaler Losgroessen');
  write('Produktbezeichnung: ');
  readln(Name);
  write('Lagerkosten in Mark: ');
  read(Lagerkosten); writeln;
  write('Produktionsmenge: ');
  read(Menge); writeln;
  Los := round(sqrt((2 * Ruestkosten) * (Menge / Lagerkosten / 12)));
  writeln(LST,'Die optimale Losgroesse fuer ',Name,' betraegt');
  writeln(LST,Los);
  write('Ende')
END.
```

Bevor dieses Programm der Analyse unterzogen wird, soll seine Wirkung betrachtet werden. Das Bildschirmprotokoll enthält diesmal nur einen Teil der Gesamtkommunikation.

2 Vgl. Autorenkollektiv: Mathematik für Ökonomen, Band 1. 3., durchgesehene Auflage. Berlin: Verlag Die Wirtschaft 1987, S. 378ff.

Es sieht wie folgt aus (Nutzereingaben unterstrichen):

```
Planung optimaler Losgroessen
Produktbezeichnung: Etui 474 <ET>
Lagerkosten in Mark: 0.24<ET>
Produktionsmenge: 20000<ET>
Ende
```

Auf dem Drucker entsteht die folgende Ausgabe:

```
Die optimale Losgroesse fuer Etui 474 betraegt
2041
```

Die Vorteile der Druckausgabe sind offensichtlich: Das Ergebnis kann mitgenommen und später systematisch ausgewertet werden. Um die verschiedenen Produkte dann noch unterscheiden zu können, wurde ihre Textbezeichnung ein- und mit dem Ergebnis wieder ausgegeben.

Zunächst sollen lediglich die Datenobjekte im Programm untersucht werden, und zwar hinsichtlich ihrer Vereinbarung und ihrer Manipulation. Die Frage ist, warum überhaupt in den Programmiersprachen die verschiedenen Datenobjekte vereinbart werden müssen. Die Programmiersprachen verhalten sich hier auch etwas unterschiedlich. Für Taschenrechner ist überhaupt keine Vereinbarung erforderlich. Grundsätzlich dient die Vereinbarung der Unterscheidung von Daten nach Anzahl und Typ. Im Beispiel sind es

- zwei Speicherworte (Menge, Los) für ganzzahlige Werte (INTEGER)
- ein Speicherwort (Lagerkosten) für reelle Werte (REAL)
- ein Speicherbereich (Name) für einen maximal 20 Zeichen langen Text (STRING [20]).

Die Zeichen sind Elemente eines festgelegten Zeichensatzes. Einen mikrorechnertypischen Zeichensatz enthält Anhang G.

Dabei ist ein Speicherwort die kleinste Einheit, auf die durch den Programmierer mit den Mitteln der jeweiligen Programmiersprache zugegriffen werden kann. Mehrere Speicherworte bilden den Speicherbereich. Die Vereinbarung von Anzahl und Typ der Daten hat folgende Vorteile:

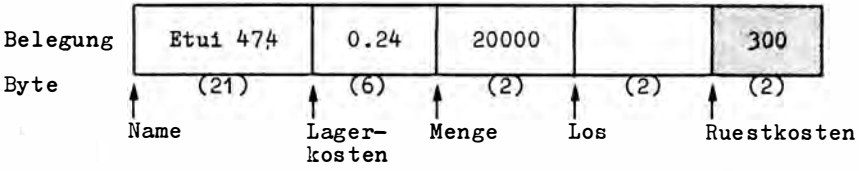
a) Der für Typ und Anzahl der Daten erforderliche Speicherplatz kann genau bereitgestellt werden. Der Datentyp REAL erfordert zum Beispiel vier bis sechs, der Datentyp INTEGER lediglich ein bis zwei Byte Speicherplatz. Trennt man die Datentypen nicht, so ist generell der Speicherplatz für REAL bereitzustellen (BASIC, auch Taschenrechner).

b) Bei Operationen mit Daten können die spezifischen Vorteile von INTEGER-Werten genutzt werden. Multiplikationen sind zum Beispiel implementationsabhängig bis zu zwangzigmal schneller als mit REAL-Werten. PASCAL stellt hier den speziellen Operator DIV für die Division zweier INTEGER-Werte, bitweise AND, OR, NOT und sogar Bitverschiebungen (SHL, SHR) zu Verfügung. Die Laufzeitgewinne sind beträchtlich.

c) Der Rechner selbst kann während der Programmierung und später während der Laufzeit zusätzliche Kontrollaufgaben übernehmen. Man spricht von Verträglichkeit der

Daten und Typen und der Typen untereinander. Die Sicherheit für das Ergebnis erhöht sich bedeutend.

Die Vereinbarungen im Programmbeispiel bewirken für die Daten die folgende Speicherplatzreservierung und nach dem Start die Speicherung der eingetragenen Werte:



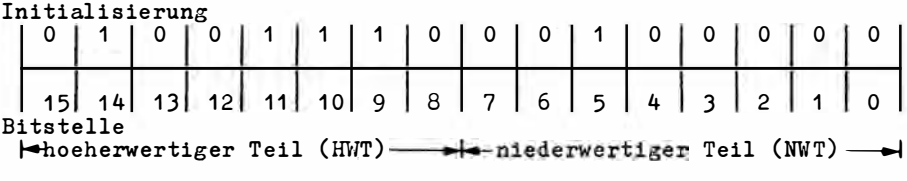
Die Anzahl der belegten Byte (für REAL verwenden einige Implementationen nur vier Byte) wurde in Klammern angegeben.

Eines der Grundprobleme für das Verständnis der Arbeitsweise eines Programms besteht darin, Konstanten und Variablen zu unterscheiden. Konstanten (hier "Ruestkosten") verändern ihren Wert während der gesamten Programmabarbeitung nicht. Der symbolische Bezeichner "Ruestkosten" steht im Programm für den Wert (300). Der Speicherplatz ist dem Programmierer nicht zugänglich (in der obigen Darstellung hervorgehoben). Anders ist es bei den Variablen "Name", "Lagerkosten", "Menge" und "Los". Hier wechselt der Inhalt der Belegung.

Die Eintragungen in die Darstellung, also "Etui 474", "0.24", "2000", werfen ein Problem auf, das für eine exakte Darstellung – aber auch für ein Verständnis der wirklichen Vorgänge im Computer – von Bedeutung ist. Dazu wird das Speicherwort "Menge" betrachtet. Als Speicherinhalt wurde der Wert 20000 angegeben. Aber es ist genau zu unterscheiden zwischen der Zeichenfolge "2", "0", "0", "0", "0", die hier zur Beschreibung des Inhalts benutzt wurde, und der internen Darstellung der Zahl 20000. In Wirklichkeit ist natürlich nicht die Folge dieser fünf Zeichen gespeichert. Dazu wären fünf Byte erforderlich, für jedes Zeichen ein Byte. Zwei Byte stehen aber nur zur Verfügung. Das und auch die anderen Darstellungen bedürfen deshalb der näheren Betrachtung.

Variablen für Zahlen

Es ist bekannt, daß ein Rechenautomat intern nur die stabilen Zustände "0" und "1" unterscheiden kann, die einem Spannungspegel innerhalb und außerhalb eines festgelegten Bereiches entsprechen. Für die Darstellung des Wertes 20000 wie für jede ganze Zahl stehen zwei Byte zu je acht Bit, also insgesamt 16 initialisierbare Bitstellen zur Verfügung. Die interne Darstellung ist folgende:



In Wirklichkeit ist allerdings erst der nieder- und dann der höherwertige Teil gespeichert. Aber das genau zu erläutern wäre hier nur verwirrend. Man beachte auch die Zählung der Bits von 0 bis 15, statt von 1 bis 16.

Die allgemeine Formel für Zahlensysteme

$$x = \sum_{i=0}^{15} s \cdot B^i,$$

in der x der dezimale Zahlenwert, B die Basis des Zahlensystems, s die Ziffer und i die Bitposition ist, bestätigt $x = 1 \cdot 2^5 + 1 \cdot 2^9 + 1 \cdot 2^{10} + 1 \cdot 2^{11} + 1 \cdot 2^{14} = 20000$.
Natürlich ist die Handhabung einer 16stelligen Binärzahl zu umständlich. Mit Hilfe der Tabelle im Anhang E soll deshalb zu der bequemen hexadezimalen Interpretation der Binärzahl übergegangen werden. Die Zahlenbasis des Hexadezimalsystems ist die 16 (B=16 in der Formel für Zahlensysteme). Um 16 verschiedene Werte mit einer Ziffer darstellen zu können, werden neben 0 bis 9 auch A bis F zu gültigen Ziffern. Das Ablesen erfolgt für jeweils 4 Bit (Halbbyte). Man erhält aus der Tabelle (in aufsteigender Adreßfolge) für

	Halbbyte(binaer) 1	2	Halbbyte(hexadezimal) 1	2	Byte (hexadezimal)
HWT	0100	1110	¤4	¤E	¤4E
NWT	0010	0000	¤2	¤0	¤20

Um hexadezimale Darstellungen von dezimalen zu unterscheiden, wird den hexadezimalen das Sonderzeichen ¤ vorangestellt. Geht man mit den Werten ¤20 und ¤4E in den für die externe Darstellung gültigen Zeichensatz der Tabelle im Anhang G, so erhält man für ¤20 das Leerzeichen und für ¤4E das "N".
Eine Ausgabe des Speicherinhalts würde als "␣N", nicht aber als 20000 erfolgen. Die 20000 entspricht dagegen der Zeichenfolge ¤32, ¤30, ¤30, ¤30, ¤30. Die Feststellung hat Bedeutung in zwei Richtungen:

- Die Eintragung der Speicherinhalte ist in solchen Darstellungen symbolisch für INTEGER und – es soll vorweggenommen werden – auch für REAL.
- Der Computer muß, wenn zum Beispiel der Speicherinhalt von "Menge" auf den Bildschirm oder Drucker geschrieben werden soll, die Umwandlung vornehmen. Dieser Vorgang heißt Konvertierung.

  Konvertierung ist auch bei Eingabe von der Tastatur erforderlich, denn der Tastendruck erzeugt stets ein Bitmuster des Zeichensatzes.

Das Verständnis der Konvertierung bewirkt, daß Fehler bei der Wahl des Datentyps, später auch der Ein- und Ausgabe von Files vermieden werden können. Kennt man diese Zusammenhänge, so wird man zum Beispiel die Ziffern der ELN (Erzeugnislistennomenklatur) oder der HSL (Handelsschlüsselliste) als Zeichen des Zeichensatzes speichern und übertragen. Dadurch wird die Konvertierung vermieden und die Verarbeitung beschleunigt.
Es gibt noch eine andere Konsequenz aus der internen Darstellung ganzer Zahlen. Da 16 Bitstellen zur Verfügung stehen, können maximal $2^{16} = 65536$ verschiedene Möglichkeiten dargestellt werden. Darin ist die Null eingeschlossen. Da nun außerdem auch negative Zahlen erforderlich sind, wird dieser verfügbare Bereich in zwei Teile gespalten. Bei dem einen ist Bit 15 gesetzt (die Bitkombination 1000 0000 0000 0000 = ¤8000 entspricht dem Wert – 32 768, die Bitkombination 1111 1111 1111 1111 = ¤FFFF dem Wert

– 1), bei dem anderen Teil ist Bit 15 gelöscht (die Bitkombination 0000 0000 0000 0001 = ¤1 entspricht dem Wert 1, die Bitkombination 0111 1111 1111 1111 = ¤7FFF dem Wert 32 767). Mathematisch gesehen, ist eine negative Zahl z intern das Zweikomplement $2^{16} - z$. Die technische Realisierung des Zweikomplements erfolgt durch bitweise Negation und Addition von 1. Für den Programmierer folgt daraus, daß bei allen Operationen die Grenzen – 32768 bis 32767 einzuhalten sind. Die meisten Implementationen überwachen aus Gründen der Laufzeiteffektivität den Überlauf des positiven Bereichs nicht. Wird zum Beispiel mit der + 32769 operiert, erkennt der Computer auf – 2. Um diese Übergänge zwischen dem positiven und negativen Bereich dem Computer zu übertragen, kann man den INTEGER-Wert in einen REAL-Wert überführen und 65536.0 addieren.

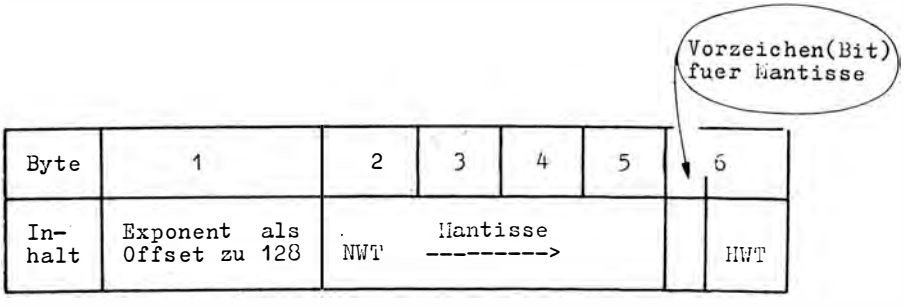
Etwas komplizierter ist die interne Handhabung *gebrochener Zahlen,* also des Typs REAL. Die Notwendigkeit der Darstellung solcher Zahlen ergibt sich aus den Anwendungsproblemen, wenn zum Beispiel Mark und Pfennige auszuweisen sind oder Zahlen den Bereich INTEGER überschreiten. Die Hauptschwierigkeit für REAL-Zahlen besteht darin, daß einer endlichen Anzahl von Bitstellen (zum Beispiel 6 Byte=48 Bit) eine eigentlich unendliche Anzahl verschiedener Zahlen gegenübersteht. Die unendliche Anzahl von gebrochenen Zahlen ergibt sich daraus, daß einer Stelle nach dem Komma immer wieder weitere Ziffern angehängt werden können. Diese Schwierigkeit wird in der Rechentechnik durch eine näherungsweise (approximierte) Darstellung gelöst. Dabei wird eine Zahl zunächst normalisiert. Das geschieht, indem sie in Mantisse und Exponent zerlegt wird. Die Mantisse hat sich dabei in einem definierten Bereich, zum Beispiel zwischen 1 und 10, zu befinden. Im Beispiel wurde die Zahl 0.24 auf den REAL-Speicherplatz "Lagerkosten" gelesen. Die Normalisierung ergäbe

+ 2.4E – 01,

wobei E für "10 hoch" steht. $2.4 * 10^{-1}$ ist natürlich wieder 0.24. Innerhalb des verfügbaren Speicherplatzes (zum Beispiel 6 Byte) werden nun vier Teile gesondert gespeichert:

- der Exponent (hier 01)
- das Vorzeichen des Exponenten (hier Minuszeichen)
- die approximierte Mantisse
- das Vorzeichen der Mantisse.

Wie diese Teile auf die verfügbaren Bytes verteilt werden, ist verschieden. Dafür existieren Empfehlungen (zum Beispiel die Konventionen AMD 9511 und IEEE), aber auch Abweichungen. Eine Möglichkeit ist folgende:



Wie man sieht, ist ein hexadezimaler Aufriß der Bytes wie bei INTEGER-Zahlen hier nicht erforderlich. Man sollte aber unbedingt folgendes beachten:

a) Die Anzahl der Bytes für die Mantisse (hier 5 Byte = 40 Bit, davon 1 Bit für das Vorzeichen und 39 Bit für den Wert) entscheidet über die Genauigkeit einer Darstellung. Üblich sind 3 Byte und 5 Byte. Bei einer 3-Byte-Darstellung der Mantisse beträgt die Genauigkeit 5 bis 6 signifikante Ziffern, bei einer 5-Byte-Darstellung 11 bis 12.
b) Die Anzahl der Bits für Exponenten (hier 8 Bit = 1 Byte für Vorzeichen und Wert) entscheidet über den Zahlenbereich. Üblich sind 8 Bit und 9 Bit. Bei einer 8-Bit-Darstellung des Exponenten beträgt der Zahlenbereich 1 E–38 bis 1 E+38, bei einer 9-Bit-Darstellung 1 E–77 bis 1 E+77.

Dabei ist immer zu berücksichtigen, daß lediglich eine Annäherung an die Werte erfolgt und deshalb Ungenauigkeiten auftreten können. Eine Möglichkeit, sich von der Arbeitsweise der konkreten PASCAL-Implementation zu überzeugen, bietet das im Anhang I dargestellte Programm Real_Test. Man bedenke, welche unangenehmen Folgen eine ungenaue Darstellung hat, wenn man auf Gleichheit einer Zahl mit dem Ergebnis einer Operation prüft.

Variablen für Zeichen

Ganz anders als für Zahlen vollzieht sich im Programmbeispiel die Darstellung in dem Speicherbereich, dessen symbolische Adresse mit "Name" vereinbart wurde. Als Typ wurde STRING [20] angegeben. STRING bedeutet, daß die maximal 20 zulässigen Werte Elemente eines festgelegten Zeichensatzes sind. Für solche Zeichensätze existieren Standards. Ein Beispiel ist der Zeichensatz druckbarer Zeichen im Anhang G. Dort entspricht dem Buchstaben "A" ein Bitmuster, das dezimal mit 65, hexadezimal mit ¤41 interpretiert werden kann, nämlich

01000001.

Der Zeichensatz ist mit den Ein- und Ausgabegeräten abgestimmt. Drückt man die Taste A (Großschreibung) der Tastatur, so wird genau dieses Bitmuster erzeugt und in den Speicher transportiert (zur sicheren Übertragung ist das Bitmuster "unterwegs" noch ergänzt). Umgekehrt schreibt ein Drucker ein "A", wenn er genau dieses Bitmuster erhält. Dasselbe Prinzip gilt für den Bildschirm. Strings können also Zeichen des jeweiligen Zeichensatzes aufnehmen. Soll nur ein Zeichen gespeichert werden, so ist die Variable zweckmäßiger als Typ CHAR (Character = Zeichen) festzulegen. Wird also zum Beispiel

```
.
VAR x : CHAR;
.
```

vereinbart, so kann x jeweils ein Zeichen des Zeichensatzes aufnehmen. Ein String ist einfach eine Kette von CHAR-Zeichen. Da die Anzahl der Zeichen eines Strings von 1 bis zu einer oberen Grenze (das sind 255) dynamisch ist, muß die aktuelle Länge vom Rechner festgehalten werden. Das geschieht im ersten Byte mit dem Index 0 (hier das Speicherwort Name [0] im Speicherbereich "Name"), dem sogenannten dynamischen Byte. Ist der String leer, enthält das dynamische Byte den Wert 0. Bei der aktuellen Belegung mit "Etui␣474" ist das dynamische Byte 8. Um die Probleme der Konvertierung noch ein-

mal zu untersetzen, wird nachstehend hexadezimal die Bytebelegung des Speicherbereichs "Name" angegeben (vergleiche Anhang G):

Byte	0	1	2	3	4	5	6	7	8	...	20
Inhalt	¤08	¤45	¤74	¤75	¤69	¤20	¤34	¤37	¤34	undefiniert	
Zeichen		E	t	u	i		4	7	4		

Der Belegung ¤08 entspricht kein druckbares Zeichen. Das dynamische Byte wird auch nach außen nicht dargestellt. Es dient ausschließlich der internen Steuerung der aktuellen Länge. Deshalb ist auch der Speicherbereich um ein Byte größer, als es der Index der Vereinbarung aussagt.
Die Besonderheit des dynamischen Byte macht deutlich, daß bei jeder Manipulation der aktuellen Länge eines Strings das dynamische Byte mit geändert werden muß. Bei der Eingabe wird das durch den Rechner selbst bei "readln" (nicht unbedingt auch bei "read") besorgt. Dem Programmierer stehen dafür die folgenden speziellen Prozeduren und Funktionen zur Verfügung:

CONCAT	für das Verbinden mehrerer Strings
COPY	für das Abbilden des Speicherinhalts ohne dessen Veränderung
DELETE	für das Löschen von Zeichen
INSERT	für das Einfügen einzelner Zeichen in einen String
LENGTH	zum Feststellen der aktuellen Länge
POS	für das Suchen von Zeichen oder Zeichenfolgen im String
VAL	für das Berechnen einer Zahl aus den Ziffern eines Strings
STR	für das Konvertieren einer Zahl in einen String.

Bei der Arbeit mit Strings ist zu beachten, daß diese nur unter Berücksichtigung der aktuellen Länge manipuliert werden können. Die Prozedur

```
.
insert('/1',Name,18);
.
```

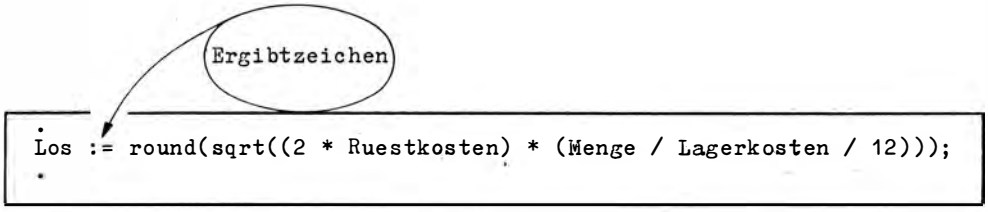
soll in den String "Name" ab Stelle 18 die Zeichen "/1" eintragen. Das führt im Programmbeispiel zu unübersehbaren Folgen, weil eine Einfügung des Textes '/1' infolge der aktuellen Länge nicht möglich ist. Der String hat im Beispiel nämlich nur acht Zeichen. Die Anwendung der STRING-Funktionen wird noch in vielen Beispielen gezeigt. Leider stehen der Datentyp STRING und die STRING-Funktionen im ISO-Standard nicht zur Verfügung. Da sie jedoch für die textintensive ökonomische Anwendung von großer Bedeutung sind, sollten sie als Unterprogramme bereitgestellt werden. Das bereitet keine Probleme.[3]

3 Vgl. Marty, R.: Methodik der Programmierung in PASCAL, Berlin – Heidelberg – New York: Springer Verlag 1983, S. 99ff.

2.2. Wertzuweisungen und arithmetische Ausdrücke

Ergibtanweisung

Im Programmbeispiel PLANUNG wird nun der Ausschnitt



betrachtet. Er besteht aus drei syntaktischen Teilen: der Zielvariablen "Los", dem Ergibtzeichen ":=" und einem komplizierten arithmetischen Ausdruck rechts vom Ergibtzeichen. Man spricht von einer Ergibtanweisung. Sie bewirkt, daß der Wert eines Ausdrucks oder einer Variablen rechts vom Ergibtzeichen der Variablen links vom Ergibtzeichen zugewiesen wird. Bei einem Ausdruck rechts vom Ergibtzeichen wird dieser erst ausgerechnet, bevor die Zuweisung erfolgt. Im Ausdruck bedeutet "*" Multiplikation und "/" Division. Dabei werden Zwischenergebnisse auf Hilfszellen zwischengespeichert. Im Beispiel beginnen die Operationen mit Ruestkosten = 300, Menge = 20000 und Lagerkosten = 0.24. SQRT ist eine vordefinierte arithmetische Funktion, die aus einem in Klammern angegebenen Argument die Wurzel zieht. ROUND ist eine Funktion, die das in Klammern angegebene Argument rundet, das heißt 0.5 addiert, und danach den gebrochenen Teil abschneidet. Die Ergibtanweisung löst folgenden Vorgang aus:

Schritt	Speicherinhalte			Operation
	Los	Hilfsworte nach Operation		
		Hilfswort_1	Hilfswort_2	
1	undefiniert	600		2 * Ruestkosten
2	undefiniert		83333.33	Menge / Lagerkosten
3	undefiniert		6944.44	Hilfswort_2/12
4	undefiniert		4166666.70	Hilfswort_1* Hilfswort_2
5	undefiniert		2041.24	SQRT
6	2041			ROUND

Die Hilfszellen sind dem Programmierer nicht zugänglich.

Aus der Arbeitsweise einer Ergibtanweisung können nunmehr einige Regeln abgeleitet werden:

a) Alle Variablen, die rechts vom Ergibtzeichen stehen, müssen zum Zeitpunkt der Ausführung der Ergibtanweisung einen Wert besitzen, man sagt: definiert sein. Das ist hier der Fall. "Ruestkosten" hat seinen Wert durch die CONST-Vereinbarung, "Menge" und "Lagerkosten" haben ihren durch die Eingabe (READ) erhalten. Die Verwendung undefinierter Variablen und damit rein zufälliger Bitkombinationen ist ein häufiger Programmierfehler. Die Folge ist in der Regel Speicherplatzüberlauf der Zielvariablen.

b) Links vom Ergibtzeichen kann kein Ausdruck stehen. Eine Wertzuweisung wäre un-

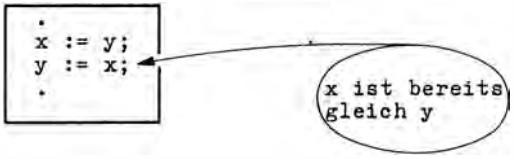
möglich, denn Ausdrücke haben selbst keinen Speicherplatz.

c) Die Reihenfolge, in der die Komponenten eines Ausdrucks notiert werden, beeinflußt die Speicherbelegung von Hilfszellen. Hier können Fehler auftreten. Der Befehl

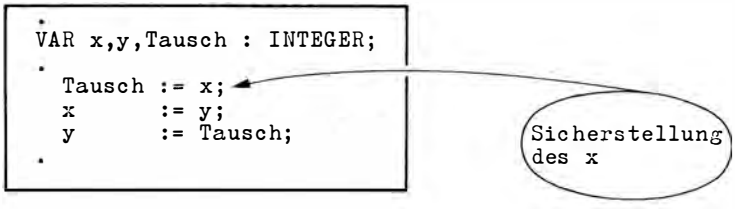
```
.
Los := round(sqrt((2 * Ruestkosten * Menge) / (Lagerkosten * 12)));
.
```

führt zu einem Laufzeitfehler. Es entsteht mit 12000000 ein Überlauf der INTEGER-Hilfszelle 1. In solchen Ausdrücken sollte man es außerdem möglichst nicht zu einer Subtraktion etwa gleich großer, vor allem sehr kleiner Zahlen kommen lassen.[4]

Natürlich hat man zu beachten, daß bei einer Wertzuweisung der bisherige Wert der Zielvariablen überschrieben wird und damit verloren ist. Die Anweisungsfolge



führt also keineswegs zum Vertauschen des Inhalts der beiden Speicherplätze, sondern der Inhalt "x" geht verloren. Beide Speicherplätze erhalten den Inhalt, den vorher "y" hatte. Für den Austausch ist also ein vom Programmierer bereitgestellter zusätzlicher Speicherplatz erforderlich. Eine Möglichkeit wäre



Der Inhalt von "x" wird mit dem Hilfsspeicherplatz "Tausch" vor dem Überschreiben gerettet.

Arithmetische Operatoren

Zur Durchführung arithmetischer Operationen an Konstanten und Variablen der Datentypen INTEGER und REAL (BYTE als Teilmenge von INTEGER im Bereich 0..255) stehen neben den Operatoren *, /, + und − noch zur Verfügung: DIV für die Division zweier INTEGER-Typen, MOD zur Bestimmung des Restes einer Division zweier INTEGER-Typen (zum Beispiel ist 12 MOD 5 = 2) sowie die (eigentlich logischen) Operatoren NOT (bitweise Negation), AND (bitweise Konjunktion), OR (bitweise Disjunktion) und XOR (bitweise Exklusion). In einigen Implementationen werden dafür auch Spezialsymbole (&, |, ?) zugelassen.

Wie die logischen Operatoren wirken, geht grundsätzlich aus der Tabelle auf Seite 44 hervor. Der Operator AND als arithmetischer Operator zum Beispiel prüft

4 Vgl. Ranft, J.: FORTRAN-Programmierung und numerische Methoden für Naturwissenschaft und Technik. Leipzig: BSB B. G. Teubner Verlagsgesellschaft 1972, S. 83ff.

paarweise die Bitbelegungen zweier INTEGER-Größen. Sind beide Bits gleich 1, wird als Ergebnis auf 1, in jedem anderen Falle auf 0 entschieden:

	Zahl	hexadezimal (HWT,NWT)	Bitmuster (HWT,NWT)
1.Operand	1986	¤712	0000 0111 0001 0010
2.Operand	2000	¤7D0	0000 0111 1101 0000
Ergebnis bei AND	1808	¤710	0000 0111 0001 0000

Operationen dieser Art werden sehr schnell ausgeführt. Ihre Anwendung ist aber auf Spezialfälle beschränkt. Ähnlich ist das mit NOT, OR und XOR.
Als besondere Operatoren sollen noch SHL (Verschiebung nach links) und SHR (Verschiebung nach rechts) genannt werden, die in neueren Implementationen enthalten sind. Der Ausdruck

```
.
Ergebnis := 2 SHL 7;
.
```

verschiebt das Bitmuster des INTEGER-Typs 2 um 7 Positionen nach links. Das Ergebnis ist 256:

Zahl dezimal	Zahl hexadezimal HWT	NWT	Zahl Bitmuster HWT	NWT
2	¤00	¤02	0000 0000	0000 0010
256	¤01	¤00	0000 0001	0000 0000

Entsprechend ist 256 SHR 7 wieder 2. Da das Umrechnen in Binär- und hexadezimale Darstellungen umständlich ist, sollte beim Verfolgen des Problems Anhang E benutzt werden. SHL und SHR sind besonders schnelle Operatoren zur Multiplikation (SHL) und Division (SHR) mit Potenzen zur Basis 2, denn

2 SHL 7 = $2 * 2^7$ = 2*128
256 SHR 7 = $256 / 2^7$ = 256 / 128

Entsprechend würde also die Operation 3 SHL 7 schneller zu dem Ergebnis 384 führen als 3 * 128. Die Anwendung dieser Operatoren ist Spezialfällen vorbehalten.
Zu beachten ist die Priorität, wenn keine Klammerung verwendet wird; sie lautet: Vorzeichen, Multiplikationsoperatoren, Additionsoperatoren.

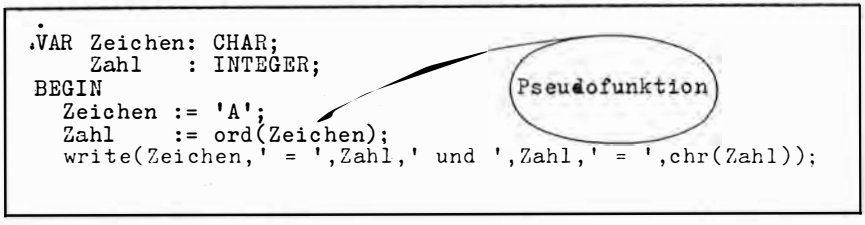
Typverträglichkeit
Es gäbe keinen Sinn und wäre sicher auf eine Unachtsamkeit zurückzuführen, wenn im Programmbeispiel der Befehl

```
.
Name := Menge;
.
```

enthalten wäre. Links steht eine STRING-Variable, dazu bestimmt, druckbare Zeichen des Zeichensatzes aufzunehmen, rechts eine INTEGER-Variable, die Werte zwischen – 32768 und 32767 darstellt. Keineswegs würde die Zeichenfolge 20000 Inhalt des Speicherplatzes "Name", sondern bestenfalls – wie bereits erläutert – die Zeichenfolge "␣N" entstehen. Das aber wäre eben sinnlos. Deshalb überwacht PASCAL bei Operationen (Name*Menge ergäbe auch keinen Sinn), bei Wertzuweisungen und anderen Gelegenheiten die Verträglichkeit der Datentypen. Bereits der Compiler würde im vorliegenden Falle auf Typkonflikt entscheiden.

Typverträglichkeit für einfache Variablen liegt zunächst nur vor, wenn diese vom gleichen Typ sind. Man spricht von Typgleichheit, also INTEGER und INTEGER, STRING und STRING, REAL und REAL. Dabei gilt BYTE im zulässigen Teilbereich als INTEGER. Da die Komponenten des Typs STRING (zum Beispiel Name [1], Name [2]) Elemente des gleichen Zeichensatzes sind wie Werte des Typs CHAR, sind beide komponentenweise verträglich. Eine Ausnahme von den strengen Verträglichkeitsregeln bildet die Wertzuweisung einer INTEGER-Variablen oder -Konstanten an eine REAL-Variable und die Verknüpfung beider Typen in Ausdrücken. Bei Wertzuweisungen dieser Art wird immer INTEGER in REAL umgewandelt (konvertiert), und bei Verknüpfungen ist das Ergebnis stets vom Typ REAL, aber nur in Richtung von INTEGER nach REAL. Man spricht von Zuweisungsverträglichkeit. Alle anderen Typen sind unverträglich, und es werden Fehler angezeigt.

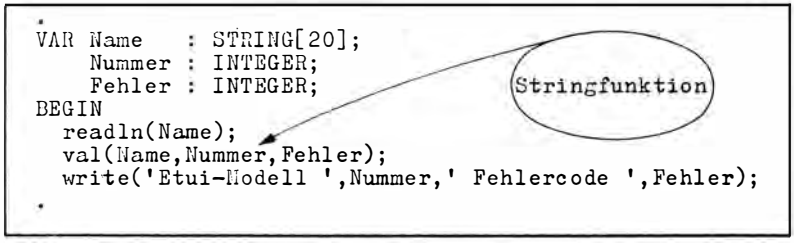
Diese Typverträglichkeitsregeln dienen, wie bereits erläutert, der Sicherheit. Bei bestimmten Problemen kann es aber erforderlich sein, Beziehungen zwischen so verschiedenen Typen herzustellen. Dafür enthält PASCAL die Pseudofunktionen CHR, ORD und PTR sowie die Routinen ROUND, TRUNC, VAL und STR. CHR, ORD und PTR sind Pseudofunktionen, weil bei ihrer Anwendung keine Operation ausgeführt wird. Es wird lediglich die Verträglichkeit hergestellt. Der Compiler erkennt nicht auf Fehler, weil er sich dem erklärten Willen des Programmierers unterordnet. CHR macht INTEGER für CHAR verträglich, ORD tut das mit CHAR für INTEGER. Der folgende Programmausschnitt zeigt die Anwendung:



Die Ausschrift würde lauten
"A = 65 und 65 = A".

Wichtig ist die Funktion CHR vor allem auch bei der Verwendung von Steuerzeichen. Das wird im nächsten Abschnitt gezeigt.
Sofort verständlich ist sicher, daß ein REAL-Wert, also gegebenenfalls eine gebrochene Zahl, nicht ohne weiteres einem Speicherplatz des Typs INTEGER zugewiesen werden kann. Es entsteht die Frage, ob der gebrochene Teil einer REAL-Zahl, der dabei in jedem Falle verlorengeht, einfach abgeschnitten oder aber gerundet werden soll. In PASCAL

wird diese Entscheidung durch die erforderliche Anwendung von ROUND (runden) oder TRUNC (abschneiden) erzwungen. Das Programmbeispiel PLANUNG zeigt die Anwendung von ROUND.
Die Konvertierung von Strings in Zahlen ist in interaktiven Programmen von Bedeutung. Sie erfolgt mit der Prozedur VAL. Die Anwendung zeigt folgender Programmausschnitt:



Wird auf den Speicherplatz "Name" der Text "474" eingelesen, so lautet die Ausschrift: Etui-Modell␣474 Fehlercode 0.
VAL ermittelt aus den Zeichen 474 die Zahl 474. Wird dabei kein Fehler entdeckt, wird die Variable "Fehler" mit dem Rückkehrcode 0 belegt. Ist der Rückkehrcode verschieden von 0, so zeigt er auf die Position des ersten fehlerhaften Zeichens. VAL ist vor allem von praktischer Bedeutung für nutzerfreundliche interaktive Programme. Die Eingaben vom Nutzer werden zunächst auf einen Speicherbereich des Typs STRING gelesen, um eine gewisse Robustheit des Programms gegenüber Fehlern zu sichern. So kann der Nutzer zum Beispiel bei der Eingabe gebrochener Zahlen sowohl den (ungewohnten) Punkt als auch das Komma benutzen. VAL unterstützt diese für die Softwarequalität interaktiver Programme wichtige Vorgehensweise. Für STR gibt es weniger Anwendungen, weil die Prozeduren WRITE und WRITELN die Konvertierung von Zahlen in Zeichenfolgen mit übernehmen können.
Neuere PASCAL-Implementationen ermöglichen zur Konvertierung die aus MODULA-2 bekannte Technik des Retyping. Dabei sind die Typbezeichner CHAR, INTEGER und BOOLEAN (wird im folgenden Kapitel behandelt) gleichzeitig als Funktionsbezeichner zur Konvertierung verwendbar. Hier bedeutet das lediglich, daß zum Beispiel "y = integer ('A')"und "y = ord ('A')" sowie "x = char (78)" und "x = chr (78)" völlig gleich sind (x ist hier vom Typ CHAR, y vom Typ INTEGER oder BYTE). Später wird das noch praktische Bedeutung gewinnen. REAL und auch STRING sind nicht für das Retyping zugelassen.

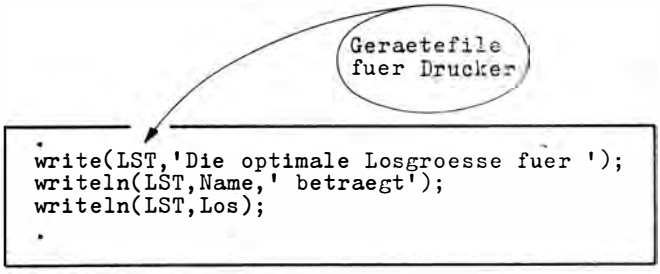
2.3. Ein- und Ausgabe zur Kommunikation

Ohne Ein- und Ausgabe wäre ein Programm für die Anwendung nutzlos. Die Funktion eines Anwenderprogramms besteht gerade darin, Eingabedaten in Ausgabedaten zu überführen. Das geschieht natürlich zielgerichtet entsprechend einer Aufgabe und ist bei jedem Programm anders. Die Verwertung der Ausgabedaten ist der eigentliche Anwendungszweck jedes Computers.
Aus der Sicht des Computers handelt es sich bei der Eingabe um einen Strom von Daten aus der Umgebung in den Computer hinein, bei der Ausgabe um einen Strom von Daten

aus dem Computer in die Umgebung. Einen solchen Strom von Daten nennt man File, auch Datei oder Massiv. Files unterteilen sich in

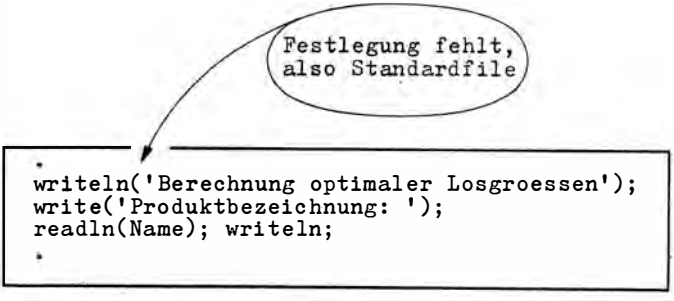
a) Files für die Kommunikation, vor allem den Datenaustausch mit dem Menschen. Da sich dieser über Kommunikationsgeräte, vor allem Tastatur, Bildschirm und Drucker vollzieht, spricht man von Gerätefiles;
b) Files zur Zwischenspeicherung der Daten, die nicht ständig im Hauptspeicher vorhanden sein können. Kennzeichnend ist, daß diese ausgelagerten Daten zur Auswertung wieder in den Computer müssen. Man spricht deshalb von Speicherfiles. Für Mikrorechner sind das in der Regel Diskettenfiles bzw. Magnetbandfiles.

Speicherfiles werden erst im Kapitel 6 behandelt. Gerätefiles wurden bereits benutzt. Im Programmbeispiel PLANUNG wurde das Ergebnis mit den Befehlen



übermittelt. Der Drucker schreibt den bereits im Abschnitt 2.1. dargestellten Text. Die Filezuordnung der Prozeduren WRITE bzw. WRITELN ist mit dem ersten Parameter der WRITE-Liste, hier LST, erfolgt. LST ist ein vordefinierter Gerätebezeichner, der das Gerätefile Drucker festlegt. Die verschiedenen PASCAL-Implementationen verhalten sich hier unterschiedlich. In einigen Fällen muß der Programmierer diesen Parameter selbst definieren (vgl. ASSIGN, REWRITE), dann sind aber die Gerätebezeichner vordefiniert (zum Beispiel 'LST:' für Drucker).

Nicht ganz so offensichtlich ist die Gerätezuordnung in dem folgenden Ausschnitt des Programms PLANUNG:



Die Ausgabeprozeduren WRITELN bzw. WRITE und die Eingabeprozedur READLN enthalten keine Parameter, die ein Gerätefile zuordnen. Natürlich ist eine Ein- und Ausgabe ohne Geräte unmöglich. Trotzdem akzeptieren alle PASCAL-Implementationen zur Vereinfachung häufiger Anforderungen, daß auf die Filezuordnung auch verzichtet werden kann. Dann wird allerdings eine Standardzuordnung von READ/READLN zu einem Stan-

dardfile INPUT und von WRITE/WRITELN zu OUTPUT vorgenommen. Welche Geräte nun INPUT und OUTPUT bezeichnen, ist implementationsabhängig.
Typisch für Mikrorechner ist, daß INPUT der Tastatur und OUTPUT dem Bildschirm zugeordnet ist. So vollzieht sich also die Ausgabe im dargestellten Programmausschnitt über den Bildschirm und die Eingabe über die Tastatur, ohne daß das ausdrücklich an den Befehlen zu erkennen ist (vgl. Protokoll der Ein- und Ausgabe in Abschnitt 2.1.).
In allen Programmen, besonders aber in den interaktiven (die unmittelbar mit dem Anwender kommunizieren), ist der Ein- und Ausgabe größte Aufmerksamkeit zu schenken. Es sollte unbedingt ein Eröffnungsbild geschrieben und es sollten deutliche Eingabeaufforderungen benutzt werden. Die Eingabeaufforderung (das sogenannte Prompting) sollte auch einen Hinweis enthalten, wie ein Nutzer die Kommunikation beenden kann. Das war in den bisherigen Programmbeispielen nicht erforderlich und mit den bisher behandelten programmtechnischen Mitteln auch nicht realisierbar. Dies wird in einem folgenden Kapitel behandelt. Wichtig ist auch, daß dem Nutzer das Ende des Dialogs mitgeteilt wird, um das entsprechende Eingabekommando sichtbar zu quittieren. Fehler des Nutzers sollten diesem mitgeteilt werden. Ein optisches oder akustisches Begleitsignal für die Fehlermitteilung erweist sich ebenfalls als günstig. Üblich ist auch, eine sogenannte HELP-Funktion zu programmieren. Meist nach Eingabe eines "?" erhält der Nutzer Hinweise, die ihm – dem ungeübten Nutzer – die ordnungsgemäße Weiterarbeit ermöglichen.
Auch für die Gestaltung der Ein- und Ausgabe sind einige Regeln zu beachten. Zum Beispiel sollte die übliche Groß- und Kleinschreibung (also nicht nur Großbuchstaben) benutzt, sollten nicht mehr als 30 % des Bildschirms beschrieben und wenig Abkürzungen verwendet werden. Bei der Druckerausgabe sind Fettdruck, Sperrdruck und Unterstreichungen wichtige Hervorhebungsmittel. Bei der Eingabe im Dialog muß man strengere Eingabekontrollen durchführen, aber auch großzügig Groß- und Kleinschreibung, gleichwertig Komma und Punkt in Zahlen zulassen und anderes.
PASCAL besitzt die programmtechnischen Mittel zur Erfüllung höchster Anforderungen. Auf Mikrorechnern stehen ein Zeichensatz für Groß- und Kleinschreibung, viele Sonderzeichen und auch Steuerzeichen (vgl. Anhang G) zur Verfügung. Viele PASCAL-Implementationen enthalten außerdem spezielle Prozeduren zur Steuerung der Schreibposition auf dem Bildschirm (Cursorsteuerung) und zur Druckbildsteuerung. Zunächst sollen die Möglichkeiten der Ausgabegestaltung mit WRITE/WRITELN behandelt werden, dann die mit Steuerzeichen und schließlich die mit nicht überall vorhandenen Prozeduren.

Ausgabegestaltung mit WRITE/WRITELN

Diese Prozeduren ermöglichen zunächst einen Übergang zur neuen Zeile auf Drucker und Bildschirm. Die Prozedur WRITE schreibt die in Klammern angegebenen Parameter (E/A-Liste). WRITELN arbeitet zunächst wie WRITE, führt aber anschließend eine Positionierung des Schreibkopfes beim Drucker bzw. des Cursors beim Bildschirm auf den Anfang der Folgezeile durch. Es ist zu beachten, daß WRITELN erst die Parameter schreibt und dann den Vorschub ausführt. WRITE läßt dagegen die Position von Schreibkopf bzw. Cursor nach dem Schreiben der Parameter unverändert. Die Verwendung von WRITELN bei der Ausgabe des Textes "Berechnung optimaler Losgroessen" im Programm PLANUNG bewirkt, daß die folgende Eingabeaufforderung "Produktbezeichnung: " auf die Folgezeile geschrieben wird. Für die Eingabeaufforderung wurde aber WRITE benutzt.

Dadurch steht der Cursor nach der Ausgabe des Textes "Produktbezeichnung: " noch in derselben Zeile. Die Aufforderung wirkt dadurch zwingender. Das Beispiel zeigt, daß die Prozedur WRITELN auch parameterlos verwendet werden kann. Sie bewirkt dann nur den Vorschub.
Eine wichtige Möglichkeit, innerhalb von WRITE/WRITELN das Ausgabebild zu formatieren, ist die Festlegung der Darstellungsbreite der einzelnen Parameter durch Angabe des Breiten- und Dezimalstellenformats. Zur Demonstration wird auf einen Ausgabebefehl im Einführungsbeispiel KOSTEN zurückgegriffen:

Die Zahl 8 (hier eine Konstante, es kann aber auch eine Variable oder ein Ausdruck sein) legt fest, daß der auszugebende Wert insgesamt in einer Breite von 8 Zeichen zu schreiben ist. Diese Breite versteht sich als Gesamtheit aller Zeichen, also zum Beispiel auch des Punktes in REAL-Zahlen und der Stellen nach dem Komma. Das Breitenformat kann auch hinter Zeichenketten angegeben werden. Es bewirkt, daß insgesamt unbedingt die vorgegebene Breite ausgefüllt wird. Ist das Ausgabeelement selbst kleiner, so wird links mit Leerzeichen aufgefüllt – also eine rechtsbündige Ausgabe realisiert. Mit "write (x:80)" kann also zum Beispiel der Inhalt des Speicherplatzes "x" (Zahl, Zeichen oder Zeichenketten) auf einem 80spaltigen Bildschirm ganz rechts ausgegeben werden.
Das Dezimalstellenformat, hier die Zahl 2 (es kann aber ebenfalls eine Variable oder ein Ausdruck sein), ist nur bei REAL-Ausgabeparametern erlaubt. Es steuert die Anzahl der Stellen nach dem Komma. Es wird gerundet. Das Dezimalstellenformat ist stets kleiner als das Breitenformat, denn es ist zugleich dessen Bestandteil. Das Dezimalstellenformat steuert eigentlich nicht nur die Anzahl der auszugebenden Dezimalstellen, sondern die Dezimaldarstellung überhaupt. Wird es nicht angegeben, so erfolgt die Ausgabe in der sogenannten Gleitkommadarstellung:

± Mantisse E ± Exponent,

zum Beispiel 0.24 E + 10 für 2.4. Das "E" steht für "10 hoch". Wie die Ausgabe wirklich aussieht, richtet sich noch danach, ob ein Breitenformat angegeben wurde. E und auch alle Vorzeichen sind mitzuzählen. Dem Breitenformat wird dabei durch eine längere oder kürzere Mantisse entsprochen. Es wird gerundet.
Wird das Breitenformat bei Realwerten nicht angegeben, ist implementationsabhängig eine Standardbreite vorgesehen. Läßt das Breitenformat keine ordnungsgemäße Ausgabe des jeweiligen Parameters zu, so wird in fast allen Implementationen eine Mindestbreite erzwungen.

Ausgabegestaltung mit Steuerzeichen

Steuerzeichen sind Bestandteil des gültigen Zeichensatzes, aber selbst nicht darstellbar, weder auf dem Bildschirm noch auf dem Drucker.

Werden sie mit Hilfe von WRITE/WRITELN trotzdem an die Ausgabegeräte gesendet, also Bestandteil des Ausgebefiles, so haben einige von ihnen steuernde Wirkung, andere sind für spezielle Aufgaben (zum Beispiel die Datenübertragung zu einem anderen Rechner) reserviert.

Die für die Ausgabe auf Drucker und Bildschirm wichtigen Steuerzeichen sind im Anhang G dargestellt. Ihre Benutzung in WRITE/WRITELN bringt aber eine Schwierigkeit mit sich. Sie lassen sich nicht direkt bezeichnen; "write ('12')"schreibt nicht das Steuerzeichen, sondern eine Zeichenkette, "write (12)" schreibt die Zahl. Der Versuch

```
.
VAR x : CHAR;
BEGIN
  x := 12;
  write(x);
.
```

Typkonflikt

scheitert am Typkonflikt. Die einzige Möglichkeit der Wertzuweisung von Steuerzeichen auf Variablen des Typs CHAR oder der direkten Ausgabe von Steuerzeichen führt deshalb über die Pseudofunktion CHR. Einige Beispiele der Verwendung sollen aufgezählt werden, zunächst zur Steuerung der Bildschirmausgabe. Möglich ist unter anderem:

a) Löschen des Bildschirms und Cursorpositionierung in die linke obere Ecke (Cursor home). Der Befehl

Steuerzeichen

```
.
write(chr(12),'BLAISE PASCAL(1623 - 1662)');
.
```

löscht den Bildschirm und schreibt Namen und Lebenszeit des französischen Mathematikers, dessen Namen die Programmiersprache trägt, auf die ersten Positionen;

b) Rücksetzen des Cursors an den Anfang der laufenden Zeile. Die Befehlsfolge

```
.
writeln('Eingabe(maximal 5 Zeichen): ');
write('.....',chr(13));
readln(x);
.
```

unterstützt den Anwender bei der Einhaltung der geforderten maximalen Zeichenzahl, weil der Cursor in der Zeile zurückgesetzt und dadurch bei jedem Eingabezeichen genau ein Punkt überschrieben wird;

c) Positionierung des Cursors auf eine beliebige Position (ohne Löschen). Die Befehlsfolge

```
.
CONST Steuerzeichen = 27;
VAR   x,y,Escape : CHAR;
BEGIN
  Escape := chr(27);
  x := chr(5);
  y := chr(6);
  write(Escape,x,y,'PASCAL');
.
```

Typ CHAR

schreibt das Wort PASCAL ab 5. Zeile, 6. Spalte auf den Bildschirm;

d) Löschen des Restes der laufenden Zeile ab aktueller Cursorposition mit "write(chr (22))"
oder Löschen des Restes des Bildschirms ab aktueller Cursorposition mit "write(chr(20))";

e) Senden eines akustischen oder optischen Signals je nach Wirkung des Steuerzeichens (BELL). Das zeigt der folgende Ausschnitt:

```
.
Akustiksignal := chr(7);
write('Es folgt das akustische Signal!',Akustiksignal);
.
```

Für die Steuerung der Druckerausgabe ist von Bedeutung:

f) der Blattvorschub auf den Anfang der Folgeseite:

```
.
write(LST,chr(12));
.
```

Die Wirksamkeit dieses Steuerzeichens hängt zusätzlich von der Hardware ab und bereitet bei einigen Geräten Schwierigkeiten;

g) Unterstreichen eines Zeichens (hier 'A') bewirkt

```
.
x := 'A';
write(LST,x,chr(8),'_');
.
```

h) Dreifach-Fettdruck eines Zeichens (hier 'A') erzeugt die nachstehende Befehlsfolge

```
.
Backspace := chr(8);
x := 'A';
write(LST,x,Backspace,x,Backspace,x,Backspace);
.
```

Natürlich müssen die Steuerzeichen standardgerecht im Laufzeitsystem implementiert sein. Viele PASCAL-Implementationen unterstützen die Ausgabegestaltung durch *vordefinierte Prozeduren.* Wichtig sind die Prozeduren

PAGE Bildschirmlöschen und Cursor auf erste Position bzw. Blattvorschub ("page (LST)"). Die Wirkung entspricht dem Steuerzeichen 12.
CLRSCR Bildschirmlöschen und Cursor auf erste Position (Clear Screen)
GOTOXY Cursorpositionierung auf Spalte (1. Parameter) und Zeile (2. Parameter), zum Beispiel "gotoxy (5, 6)"
CLREOL Löschen des Restes der Bildschirmzeile ab aktueller Cursorposition (Clear End of Line).

Dabei gibt es viele implementationsabhängige Besonderheiten.

Eingabe mit READ/READLN

Die Prozeduren READ/READLN, zum Beispiel in der Anwendung

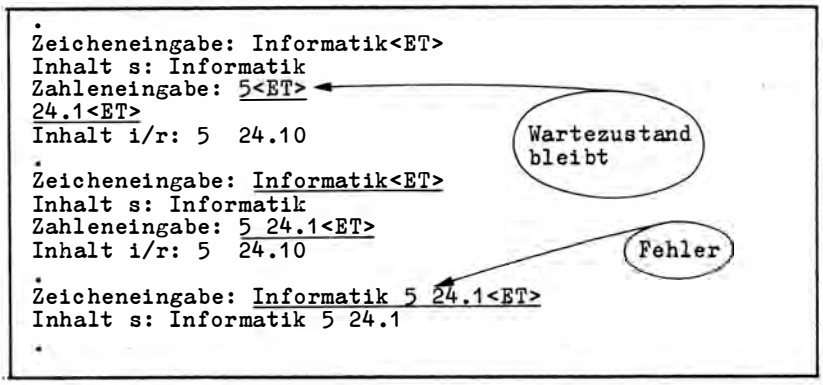
```
.
VAR r : REAL;
    s : STRING[20];
    i : INTEGER;
BEGIN
  write('Zeicheneingabe: ');
  readln(s);
  writeln('Inhalt s: ',s);
  write('Zahleneingabe: ');
  read(i,r); writeln;
  writeln('Inhalt i/r: ',i,r:7:2);
.
```

realisieren folgendes:

1. Der Computer wird in den Eingabemodus versetzt, das heißt, die Programmausführung ist unterbrochen, und es wird gewartet, bis die Eingabe erfolgt ist. Sind mehrere Parameter vorhanden (zum Beispiel "read (i, r)"), so wird der Wartezustand aufrechterhalten, bis alle ausgelösten Forderungen erfüllt sind. Im Beispiel "read (i, r)" wird mit der Eingabe "5" <ET> die Anforderung nicht erfüllt.
2. Beim Drücken einzelner Tasten der Tastatur wird das zur Taste (zum Zeichen) gehörende Bitmuster in einem speziellen Hauptspeicherbereich, dem Zeilenpuffer, zwischengespeichert. Dieser Bereich ist dem Programmierer nicht ohne weiteres zugänglich. Ob im Zeilenpuffer das Editieren möglich ist (also zum Beispiel die Korrektur einzelner Zeichen) und ob ein Echo auf dem Bildschirm dargestellt wird, ist beeinflußbar.
3. Beim Drücken der Endetaste, hier allgemein <ET>, sonst je nach Tastatur <ET1>, <RETURN>, <ENTER> usw., mit dem der Nutzer dem Rechner das Ende seiner aktuellen Eingabe mitteilt, oder wenn alle Eingabeaufforderungen erfüllt sind, wird der Wartezustand beendet. Man sollte wissen, daß beim Betätigen der Endetaste ein Leerzeichen in den Puffer geschrieben wird.
4. Der Pufferinhalt wird den angegebenen Variablen zugewiesen. Für Variablen des Typs INTEGER, BYTE und REAL erfolgt dabei eine Kontrolle auf Anzahl, syntaktische Richtigkeit (zum Beispiel Punkt bei REAL und nicht das Komma) und, wenn Compilerdirektiven gesetzt werden, auch auf die Einhaltung festgelegter Zahlenbereiche. Fehlerhafte Eingaben werden abgewiesen, und es wird zur Wiederholung der Eingabe aufgefordert. Sonst erfolgt bei diesen Datentypen die Umwandlung von Bitmustern des Zeichensatzes in Bitmuster für interne Zahlendarstellung (Konvertierung). Fehlerprü-

fungen für Eingaben von Werten des Typs CHAR und STRING sind nicht möglich, weil in jedem Falle alle Bitmuster, die die Tastatur erzeugt, Bestandteil des Zeichensatzes sind.

Aus dieser Arbeitsweise ergeben sich einige Besonderheiten, die beim Programmieren zu beachten sind. Zu ihrer Darstellung dient das folgende Bildschirmprotokoll des mehrfachen Dialogs mit dem vorigen Programmabschnitt (Nutzereingaben unterstrichen).



Man sieht folgendes:

a) Es ist möglich, mehrere Zahlen mit oder ohne Zwischenbenutzung der Endetaste einzugeben. Der Befehl "read (i, r)" und die Befehlsfolge "read (i); read (r)" sind in ihrer Wirkung völlig gleich. Die verschiedenen Zahlen müssen natürlich durch wenigstens ein Leerzeichen voneinander getrennt sein.

b) Die gemischte Eingabe von Text und Zahlen ohne Betätigung der Endetaste wird fehlerhaft. Der Computer kann dann nicht erkennen, wo der Text endet und die Zahl beginnt, denn alle Eingaben sind vor der Konvertierung Text. Deshalb sollte auch ein Befehl "read (s, i, r)" unbedingt vermieden werden.

Es gibt ein weiteres Problem, das sich in der Darstellung nicht sichtbar machen läßt, aber von erheblicher praktischer Bedeutung ist. Sein Verständnis bewirkt die richtige Verwendung von READ im Unterschied zu READLN. Es besteht keinesfalls nur darin, daß READLN die Benutzung der Endetaste erzwingt, READ aber nicht. READLN erzwingt vielmehr eine Zeilenstruktur der Eingabe, bei der der sogenannte "Rest der Zeile" überlesen und die Eingabe am Anfang der "nächsten Zeile" steht. Die "Zeile" wird durch alle aktuellen Eingabezeichen bis zum Betätigen der Endetaste gebildet. Den "Rest der Zeile" bilden alle Zeichen, die die Anforderung der Parameterliste hinter READLN übersteigen, und das von der Endetaste selbst geschriebene Leerzeichen (Zeilenendezeichen). Praktische Konsequenzen sind:

a) Zeichenketten sind stets mit READLN zu lesen. Bei "read (s)" im Beispiel und Eingabe der Zeichenfolge "Informatik" mit anschließender Betätigung der Endetaste ist der Speicherinhalt von s gleich "Informatik␣", enthält also am Ende ein zusätzliches Leerzeichen.

b) Bei der Eingabe von Zeichen oder Zeichenketten nach der Eingabe von Zahlen ist Vorsicht geboten. Die Befehlsfolge "read (i); read (s)" führt zum zusätzlichen Eintragen eines Leerzeichens (Zeilenendezeichen) in den String.
Richtig ist hier "readln (i); readln (s)".

c) Bei der Eingabe von Zahlen überliest READLN überschüssige Zeichen. Bei "read (i, r)" wird die Eingabezeile "5␣24.1␣45.4" als fehlerhaft abgewiesen, "readln (i, r)" überliest dagegen "␣45.4". Das kann gewollt sein, aber auch zu Fehlern führen.

d) READLN kann auch parameterlos verwendet werden. In diesem Fall wird die gesamte Eingabezeile überlesen. READLN bewirkt dann einfach nur einen Wartezustand, der durch Betätigen beliebiger (auch keiner) Tasten und der Endetaste aufgehoben wird.

In den meisten PASCAL-Implementationen kann man die Editiermöglichkeit und auch das Bildschirmecho der Eingabezeile beeinflussen. Wie WRITE/WRITELN dem Standardfile OUTPUT sind READ/READLN dem Standardfile INPUT zugewiesen; "read (s)" und "read (INPUT,s)" sind also in der Wirkung völlig gleich. Bei Mikrorechnern ist dem File INPUT die Tastatur zugewiesen, die als Konsoleingabegerät CON (auch 'CON:') festgelegt ist. Im allgemeinen arbeitet das logische Gerät CON mit editierbarem Zeilenpuffer. Wählt man dagegen TRM, wobei TRM nicht Standard ist und deshalb als erster Parameter (zum Beispiel "read (TRM, s)") aufgeführt werden muß, kann im Zeilenpuffer nicht editiert werden. Das Gerät KBD (also zum Beispiel in "read(KBD, s)") unterdrückt das Echo der Eingabe auf dem Bildschirm. Das ist vor allem bei der Eingabe von Codeworten nützlich.
Die Eingabe von Steuerzeichen ist auch möglich, wenn sich der Rechner im Wartezustand (Eingabemodus) befindet. Allerdings können dabei Steuerzeichen nicht direkt erzeugt werden. Die Tasten schaffen stets nur Bitmuster für druckbare Zeichen. Deshalb besitzen die Tastaturen eine sogenannte CTRL-Taste (CTRL = Control). Sie trägt direkt die Aufschrift CTRL oder wird indirekt erzeugt (zum Beispiel mit der Taste ET2). Steuerzeichen werden erzeugt, indem zunächst die CTRL-Taste und unmittelbar danach (bei einigen Tastaturen auch gleichzeitig) eine normale Zeichentaste betätigt wird. So bewirkt die Tastenfolge <CTRL> <X>, daß alle Zeichen des aktuellen Zeichenpuffers gelöscht werden. Dieser Effekt ist sichtbar, denn es erfolgt ein Echo auf dem Bildschirm. Mit der Tastenfolge <CTRL> <S> kann man die Bildschirmausgabe unterbrechen und – bei erneuter Betätigung dieser Folge – fortsetzen. Eine besondere Rolle spielt auch die Tastenfolge <CTRL> <Z> als Möglichkeit einer Endemitteilung. Darauf wird bei der Behandlung der vordefinierten Funktionen EOF (End of File) und EOLN (End of Line) zurückgekommen.

Übungsaufgaben

1. Verändern Sie das Programm PLANUNG so, daß a) die Rüstkosten variabel sind und vom Nutzer ebenfalls eingegeben werden müssen, b) die Endemitteilung von einem akustischen Signal begleitet wird! Das neue Programm ist zu testen!

2. Die optimale Nutzungsdauer eines Arbeitsmittels in Jahren, das heißt eine solche Dauer, für die minimale Kosten je Nutzungseinheit entstehen, kann nach der Formel

$$\sqrt{\frac{2 \cdot \text{Gesamtabschreibung}}{\text{Reparaturkoeffizient}}}$$

errechnet werden.
Der Reparaturkoeffizient läßt sich als Koeffizient der linearen Trendfunktion aus einer Zeitreihe der Reparaturkosten gewinnen (zum Beispiel im Bereich 0.15 bis 0.40). Entwickeln Sie ein interaktives Programm, das nach Eingabe der Gesamtabschreibung und des Reparaturkoeffizienten die optimale Nutzungsdauer ermittelt und mitteilt!

3. Selektive Ausführung von Anweisungen

3.1. Vergleiche, logische Variablen und logische Ausdrücke

Die rein sequentielle Ausführung von Anweisungen genügt den Anforderungen nur in wenigen Fällen. Nimmt man zum Beispiel an, während des Dialogs mit dem Programm PLANUNG entscheidet der Nutzer, daß eigentlich keine direkt zurechenbaren Lagerkosten für das Produkt entstehen, gibt er folgerichtig auf die Frage "Lagerkosten in Mark:" eine Null ein. Die Wirkungen wären fatal. Das Programm versucht nämlich eine Division durch Null. Da das nicht möglich ist, wird die Programmverarbeitung durch das Laufzeitsystem abgebrochen – das Programm "stürzt ab". Es ist ein ungeschriebenes Gesetz, vor jeder Division zu prüfen und unbedingt eine Division durch Null zu verhindern. Aber das ist nur ein Gesichtspunkt. Jedes wirkliche Problem steckt voller "Wenn" und "Aber". Programme müssen den Anforderungen genügen, die man an Algorithmen stellt[5] und deshalb für jede mögliche Situation eine eindeutige Lösungsvorschrift bereithalten. Dieser grundlegenden Forderung zu genügen ist eines der schwierigsten Probleme der Programmierung überhaupt. PASCAL hält dafür spezielle programmtechnische Mittel bereit. Zu ihnen gehören Vergleiche, logische Variablen, logische Ausdrücke und später auch Mengen. Zur Demonstration wird ein interaktives Programm benutzt, das entsprechend der Gebührenordnung der DEUTSCHEN POST die Telefongebühren im Inlandfernverkehr ermittelt. Die Entfernungszonen sind örtlich festgelegt. Ermäßigungen gibt es zwischen 22.00 und 7.00 Uhr sowie an Sonnabenden, Sonn- und Feiertagen. Der PASCAL-Text ist folgender:

```
PROGRAM Gebuehren;
{Berechnung von Telefongebuehren nach Tarif der Deutschen Post}
VAR Dauer          : INTEGER;
    Zone,Antwort   : CHAR;
    Gebuehr        : REAL;
    Ermaessigung   : BOOLEAN;
BEGIN
  writeln('Berechnung von Telefongebuehren');
  write('Dauer des Gespraechs in Minuten: ');
```

Logische Variable

Fortsetzung S. 42

5 Vgl. Kämmerer, W.: Einführung in mathematische Methoden der Kybernetik. Berlin: Akademie-Verlag 1971, S. 413.

```
  read(Dauer); writeln;
  write('Zone(1/2/3): ');
  read(Zone); writeln;
  write('Ermaessigung (J/N): ');
  read(Antwort); writeln;
  Ermaessigung := (Antwort = 'J') OR (Antwort = 'j');
  IF Zone = '1' THEN Gebuehr := 0.15
  ELSE IF Zone = '2' THEN Gebuehr := 0.45
       ELSE Gebuehr := 0.90;
  Gebuehr := Gebuehr * Dauer;
  IF Ermaessigung THEN Gebuehr := Gebuehr * 2 / 3;
  writeln('Die Gebuehr betraegt ',Gebuehr:5:2,' Mark');
  write('Ende');
END.
```

Bei der Programmausführung entsteht zum Beispiel das folgende Protokoll (Eingaben unterstrichen):

```
Berechnung von Telefongebuehren
Dauer des Gespraechs in Minuten: 4<ET>
Zone(1/2/3): 2<ET>
Ermaessigung (J/N): n<ET>
Die Gebuehr betraegt 1.80 Mark
Ende
```

Neu ist zunächst die Vereinbarung "Ermaessigung:BOOLEAN". Alle "Wenn" und "Aber" prüfen im Grunde eine oder mehrere Bedingungen. Jede Bedingung kann entweder erfüllt oder nicht erfüllt sein. In der Fachsprache sagt man für erfüllt TRUE (wahr) und für nicht erfüllt FALSE (falsch).

Logische Variablen

Speichert man das Ergebnis einer Bedingungsprüfung, das also lediglich TRUE oder FALSE sein und damit nur zwei Zustände annehmen kann, benötigt man nur ein Bit. So ist es eigentlich auch. Aber die kleinste adressierbare Einheit ist ein Byte. Deshalb muß man 8 Bit = 1 Byte als Speicherplatz reservieren, aber stets nur ein Bit davon auswerten. Das teilt man dem Rechner durch den reservierten Typbezeichner BOOLEAN (nach dem Begründer der mathematischen Logik G. Boole, 1815 bis 1869) mit. Einer so definierten logischen Variablen kann man nun zuweisen

a) die Konstantenbezeichner TRUE oder FALSE. Diese Werte werden von PASCAL direkt verstanden, zum Beispiel
   "Ermaessigung := FALSE"
b) den Wert eines Vergleichs. Eine Möglichkeit wäre
   "Ermaessigung := Antwort = 'J'"
   Die Variable "Ermaessigung" erhält den Wert TRUE, wenn der Wert der Variablen "Antwort" gleich 'J' ist, wenn also der Großbuchstabe J eingegeben wurde. Allerdings wäre eine solche Verfahrensweise im Programm nicht anwenderfreundlich. Wird nämlich bei der Eingabe die Großschaltung vergessen, so würde ein "j" wie eine Verneinung behandelt;
c) den Wert eines logischen Ausdrucks. Ein logischer Ausdruck ist im Beispiel "(Antwort = 'J') OR (Antwort = 'j')".

OR ist ein reserviertes PASCAL-Wort und verbindet hier zwei Vergleiche durch ein logisches ODER. Die Klammerung ist den Prioritätsregeln geschuldet. Darauf wird noch zurückgekommen. Im Beispiel ist der Wert des gesamten Ausdrucks TRUE, wenn einer der beiden Vergleiche TRUE ist. Das heißt praktisch, dem Anwender wird ermöglicht, auf die Frage (J/N) mit dem Großbuchstaben "J" oder auch mit dem Kleinbuchstaben "j" zu antworten. Das ist anwenderfreundlich.

Konstanten, Variablen und Ausdrücke können durch den Operator NOT negiert werden. Aus FALSE wird dann TRUE bzw. aus TRUE wird FALSE. Logische Werte können nicht direkt eingegeben werden. Implementationsabhängig besteht aber die Möglichkeit, sie direkt auszugeben. Der Computer schreibt dann die Worte "TRUE" oder "FALSE". Der Programmausschnitt

```
.
VAR Ermaessigung : BOOLEAN;
BEGIN
  Ermaessigung := TRUE;
  write(Ermaessigung);
  write('TRUE oder FALSE: ');
  read(Ermaessigung);
.
```

Fehler

führt in den meisten Implementationen zu Fehlern. Natürlich gibt es indirekte Möglichkeiten, logische Werte von außen festzulegen. Im Beispiel ist das über die Frage und Antwort geschehen. Man kann die rechnerinterne Darstellung von TRUE und FALSE mit der Funktion ORD prüfen:

```
.
VAR Test          : INTEGER;
    Ermaessigung  : BOOLEAN;
BEGIN
.
  Test := ord(Ermaessigung);
.
```

Es ergibt sich der Wert 1 für TRUE und 0 für FALSE.

Vergleiche

Allgemein besteht jeder Vergleich aus zwei Vergleichsoperanden und einem Vergleichsoperator. Vergleichsoperanden können Variablen und Konstanten aller bisher behandelten Typen, also INTEGER, BYTE, CHAR und REAL sein. Wenn der STRING-Typ implementiert ist, sind auch Strings als Vergleichsoperanden zugelassen. Sie können sogar unterschiedlich lang sein. Vergleichsoperatoren sind "=" (gleich), "<" (kleiner), ">" (größer), "<=" (kleiner oder gleich), ">=" (größer oder gleich) und "<>" (ungleich). Bei BYTE- und INTEGER-Typen ist die Vergleichsrelation offensichtlich. CHAR-Typen sind in der Reihenfolge des Zeichensatzes angeordnet. Deshalb sind zum Beispiel die Werte der folgenden Vergleiche stets TRUE:

'A'<>'a'
'a'>'A'
'A'<'B'

Bei Strings verschiedener Länge wird der kürzere String so behandelt, als sei er mit Leerzeichen aufgefüllt.
Vorsicht ist geboten, wenn zwei REAL-Typen auf Gleichheit geprüft werden. Die aus Abschnitt 2.1. bekannte Approximation der internen Darstellung gebrochener Zahlen sollte den Programmierer veranlassen, niemals auf Gleichheit zu prüfen, sondern stets auf Einhaltung einer Toleranz. Das gilt immer, wenn diese Werte aus verschiedenen Operationen entstanden sind. Sind x und y vom Typ REAL über READ durch Eingabe belegt und sind die eingegebenen Werte gleich, so ergibt unmittelbar nach der Eingabe eine Prüfung auf Gleichheit natürlich TRUE. Das ist aber nicht mehr so, wenn eine der Zahlen zum Beispiel mit 3.14159 multipliziert und danach wieder durch diese Zahl dividiert wird.
Wie logische Variablen selbst, können Vergleiche negiert werden. "NOT (Antwort = 'J')" wäre TRUE, wenn irgendein anderes Zeichen als 'J' eingetastet wurde.
Schließlich ist noch zu erwähnen, daß Vergleichsoperanden, die sich in einem Vergleich gegenüberstehen, typ- oder zuweisungsverträglich sein müssen. Es gelten die gleichen Grundregeln, wie im Abschnitt 2.2. für Ergibtanweisungen dargelegt. Auch Konvertierungen sind möglich, um Verträglichkeit zu erreichen. Vergleichsoperatoren haben die niedrigste Priorität. Daraus resultiert die Notwendigkeit der Klammerung, wenn im Ausdruck logische Operatoren vorhanden sind.

Logische Ausdrücke

Logische Ausdrücke sind Verknüpfungen logischer Variablen oder Vergleiche mit den logischen Operatoren AND, OR oder XOR. Jeder Operand kann dabei außerdem den logischen Präfix NOT besitzen. Die Wirkung der logischen Operatoren hängt jeweils von den aktuellen Werten der Variablen und Vergleiche ab, die diese zum Zeitpunkt der Befehlsausführung besitzen. Die Wirkung wird aus der folgenden Übergangstabelle ersichtlich (T für TRUE, F für FALSE).

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	(Konjunktion)	T	F	F	F
a OR b	(Disjunktion)	T	T	T	F
a XOR b	(Exklusion)	F	T	T	F

Das Programmbeispiel GEBUEHREN enthält den logischen Ausdruck

(Antwort = 'J') OR (Antwort = 'j').

Der Wert des Vergleiches "Antwort = 'J'" steht für den Operanden "a", der Wert des Vergleiches "Antwort = 'j'" für den Operanden "b" in der Übergangstabelle. Danach ergibt sich für den Wert des Gesamtausdrucks bereits TRUE, wenn (vgl. OR-Zeile in der Tabelle) einer dieser Vergleiche den Wert TRUE liefert, und FALSE, wenn weder der erste noch der zweite Vergleich TRUE ist. Dieser entsprechende Wert wird der Variablen "Ermaessigung" zugewiesen.
Das ist jener Effekt, der den Nutzer nicht unbedingt zur Beachtung der Groß- und Kleinschreibung bei seinem "ja" zwingt. Man sieht aber auch, daß sich das Programm nicht genau an seine eigene Ankündigung hält.

Es wird angefordert:

"Ermaessigung (J/N):"

Die Bedingungsprüfung beinhaltet aber die Prüfung auf "J", "j", und jede Abweichung davon wird als N aufgefaßt. Gibt man also auf die Frage zum Beispiel ein M (oder m) ein, so ist die Wirkung genauso, als hätte man N eingegeben. Diese kleine Vereinfachung ist in der Programmierung weit verbreitet. Trotzdem kann es Probleme geben, bei denen eine solche Vorgehensweise nicht zulässig ist, vor allem, wenn eine Eingabe erhebliche Konsequenzen für die Laufzeit, die Speichereffektivität oder sogar die Datensicherheit hat. Die programmtechnischen Mittel zur Sicherung einer exakten Vorgehensweise durch den Nutzer werden in Abschnitt 4.3. behandelt, und im Abschnitt 7.2. wird ein nützliches Unterprogramm angeboten.

3.2. Selektion bei zwei Möglichkeiten

Das eigentliche Anliegen einer Bedingungsprüfung besteht natürlich darin, in der Programmausführung bei TRUE anders zu verfahren als bei FALSE, wobei im einfachsten Falle bei FALSE überhaupt keine Operation erfolgen kann. Damit wird während der Verarbeitung in Abhängigkeit von Werten logischer Variablen oder auch direkt von Vergleichen selektiert, ausgewählt, welche der notierten Befehlsfolgen tatsächlich ausgeführt werden. Das erfolgt in PASCAL mit der IF-Anweisung. Im Programmbeispiel GEBUEHREN wird das erstemal durch die Befehlsfolge

```
.
IF Zone = '1' THEN Gebuehr := 0.15
ELSE IF Zone = '2' THEN Gebuehr := 0.45
     ELSE Gebuehr := 0.90;
.
```

selektiert. Die reservierten Worte IF (wenn), THEN (dann), ELSE (sonst) sind selbstsprechend, das heißt, THEN wird angewandt, wenn der Wert der logischen Variablen oder des Vergleichs TRUE ist, ELSE wird verwendet, wenn dieser Wert FALSE ist. THEN und ELSE schließen sich aus. Sie bilden Zweige (THEN-Zweig, ELSE-Zweig), von denen immer nur einer (alternativ) ausgeführt wird. Ihre Wirkungsweise soll etwas ausführlicher dargestellt werden. Dazu werden für die Nutzereingabe, die mit "Zone (1/2/3):␣" angefordert wird, vier Möglichkeiten kalkuliert. Sie können nach der Eingabe über die Belegung des Speicherplatzes "Zone" geprüft werden:

Eingabe einer "1" (für Zone 1)
Eingabe einer "2" (für Zone 2)
Eingabe einer "3" (für Zone 3)
Eingabe eines Zeichens, verschieden von "1", "2", "3", bei Betätigung einer falschen Taste.

Die Selektion durch die IF-Anweisung ist für diese vier Fälle folgende (die aktiven Teile der Anweisung sind unterstrichen, die nichtaktiven als Kommentar geklammert):

1. Eingabe der "1" (Zone = '1' ist TRUE)

```
IF Zone = '1' THEN Gebuehr := 0.15
{ELSE IF Zone = '2' THEN Gebuehr := 0.45
      ELSE Gebuehr := 0.90};
```

   Die Selektion ist logisch geschachtelt, denn das zweite IF als Ganzes ist Bestandteil des ELSE-Zweiges und nur aktiv, wenn Zone = '1' FALSE ist;
2. Eingabe der "2" (Zone = '1' ist FALSE, Zone = '2' ist TRUE)

```
IF Zone = '1' THEN {Gebuehr := 0.15}
ELSE IF Zone = '2' THEN Gebuehr := 0.45
      {ELSE Gebuehr := 0.90};
```

   Diesmal wird der ELSE-Zweig des ersten IF aktiv. Wegen Zone = '2' TRUE wird der THEN-Zweig ausgeführt;
3. Eingabe der "3" (Zone = '1' ist FALSE, Zone = '2' ist FALSE)

```
IF Zone = '1' THEN {Gebuehr := 0.15}
ELSE IF Zone = '2' THEN {Gebuehr := 0.45}
      ELSE Gebuehr := 0.90;
```

4. Eingabe verschieden von "1", "2" und "3" – also ein beliebiges anderes Zeichen (Zone = '1' ist FALSE, Zone = '2' ist FALSE)

```
IF Zone = '1' THEN {Gebuehr := 0.15}
ELSE IF Zone = '2' THEN {Gebuehr := 0.45}
      ELSE Gebuehr := 0.90;
```

Die vierte Möglichkeit realisiert die gleiche Befehlsverarbeitung wie die dritte. Hier sieht man wieder eine kleine Vereinfachung des Programmierers. Sie wäre für die praktische Anwendung sicher nicht zulässig, denn falsche Eingaben werden nicht mitgeteilt, sondern mit Zone 3 gleichgesetzt.
Das Programm GEBUEHREN zeigt, daß auf das Notieren des ELSE-Zweiges ganz verzichtet werden kann, wenn im "ELSE-Falle" keine Operation erforderlich ist.

```
.
Gebuehr := Dauer * Gebuehr;
IF Ermaessigung THEN Gebuehr := Gebuehr * 2 / 3;
.
```

Die Gebührenverringerung auf zwei Drittel kommt eben nur bei Ermäßigung zustande, sonst nicht. Zu beachten ist außerdem, daß hier hinter IF direkt eine logische Variable notiert ist. Der Vergleich ist durch die Ergibtzuweisung für "Ermaessigung" vorgezogen. Das ist zum Zwecke der Demonstration geschehen und würde sonst nur bevorzugt, wenn eine Mehrfachauswertung des Vergleichs erforderlich ist.

Die Befehlsfolge

```
.
IF (Antwort = 'J') OR (Antwort = 'j') THEN
  Gebuehr := Gebuehr * 2 / 3;
.
```

hätte hier den gleichen Dienst (bei einem Befehl weniger) getan. Auf zwei häufige Fehler bei der Anwendung der IF-Anweisung soll noch hingewiesen werden. Der erste ist syntaktisch. Sehr häufig kommt es vor, daß man nach Beendigung des THEN-Zweiges ein Semikolon setzt, obwohl noch der ELSE-Zweig folgt. Die Gefahr ist besonders groß, wenn man eine IF-Anweisung nachträglich durch einen ELSE-Zweig erweitert. Ein Semikolon vor ELSE führt in PASCAL zu einem Syntaxfehler. Das Semikolon dient der Trennung zweier Befehle. Der ELSE-Zweig ist aber Bestandteil der IF-Anweisung.
Ein anderer inhaltlicher Fehler resultiert aus logisch falscher Schachtelung. Es ist stets zu beachten, daß ein ELSE-Zweig immer zur letzten IF-Anweisung gehört. Deshalb war die ELSE-Einfügung im Programm sehr wichtig. Im Programm GEBUEHREN ist auch folgender Fehler möglich:

```
.
IF Zone = '1' THEN Gebuehr := 0.15;
IF Zone = '2' THEN Gebuehr := 0.45
ELSE Gebuehr := 0.90;
.
```

Fehler

Man muß nur prüfen, was geschieht, wenn eine "1" eingetastet wird. Die Gebühr wäre dann 0.90.
Abschließend soll noch darauf verwiesen werden, daß sowohl der THEN- als auch der ELSE-Zweig einer IF-Anweisung nur genau einen Befehl aufnehmen können. Oftmals sind aber je Zweig mehrere Befehle erforderlich. Dann ist (zusätzlich) eine Verbundanweisung zu verwenden, also die Befehlsfolge in die Textklammern BEGIN – END einzuschließen.

3.3. Aufzählungs- und Teilbereichstypen

PASCAL enthält als Spezialität gegenüber anderen Programmiersprachen die Möglichkeit, anwendereigene Datentypen festzulegen und mit diesen umzugehen. Zur Demonstration wird ein Programm benutzt, das im Dialog die Berechnung zu erstattender Fahrtkosten bei Benutzung privater PKW für Dienstreisen ermöglicht. Nach dem Reisekostenrecht sind die Kilometersätze betriebsindividuell und wurden hier mit 0.27, 0.21 und 0.24 festgelegt.

```
PROGRAM Fahrgeld;
{Berechnung von Kilometergeld nach Reisekostenrecht}
TYPE PKW = (Lada,Wartburg,Trabant,Moskwitsch,Skoda);
VAR  Fahrzeug                    : PKW;
     Typ,Personen,Kilometer      : INTEGER;
     Kilometersatz,Fahrkosten    : REAL;
BEGIN
  writeln('Kilometergeld fuer die Benutzung privater PKW');
```

Aufzaehlungstyp

Fortsetzung S. 48

```
  writeln;
  writeln('  Lada           = 1');
  writeln('  Wartburg       = 2');
  writeln('  Trabant        = 3');
  writeln('  Moskwitsch     = 4');
  writeln('  Skoda          = 5');  writeln;
  write('Waehlen Sie: ');
  read(Typ); writeln;
  Fahrzeug := PKW(Typ - 1);
  CASE Fahrzeug OF
    Lada,Wartburg    : Kilometersatz := 0.27;
    Trabant          : Kilometersatz := 0.21;
    Moskwitsch,Skoda : Kilometersatz := 0.24;
  ELSE BEGIN write('Fehlerhafte Eingabe'); halt END;
  END;
  write('Gefahrene Kilometer: ');
  read(Kilometer); writeln;
  write('Anzahl mitgefahrener Personen: ');
  read(Personen); writeln;
  Fahrtkosten := (Kilometersatz + Personen * 0.03) * Kilometer;
  writeln('Zu erstatten sind ',Fahrtkosten:6:2,' Mark');
  write('Ende')
END.
```

Retyping

Die Programmausführung ergibt zum Beispiel das folgende Protokoll (Eingaben unterstrichen):

```
Kilometergeld fuer die Benutzung privater PKW

  Lada           = 1
  Wartburg       = 2
  Trabant        = 3
  Moskwitsch     = 4
  Skoda          = 5

Waehlen Sie: 2<ET>
Gefahrene Kilometer: 302<ET>
Anzahl mitgefahrener Personen: 2<ET>
Zu erstatten sind  99.66 Mark
Ende
```

Aufzählungstypen

PASCAL versteht unter einem Datentyp die Menge der zulässigen Werte. Bei Zahlen, Zeichen und logischen Werten ist der Wertevorrat von vornherein gegeben. Weicht man davon ab, so muß der Wertevorrat selbst definiert, müssen die zulässigen Werte gewissermaßen aufgezählt werden. Das Aufzählen geschieht zweckmäßig mit einer TYPE-Definition. Die Syntax der TYPE-Definition ist dem Syntaxdiagramm zu entnehmen. Die Typvereinbarung im Beispiel ist

```
.
TYPE PKW = (Lada,Wartburg,Trabant,Moskwitsch,Skoda);
.
```

Dem Bezeichner PKW wird der nebenstehende Wertevorrat zugeordnet. Ähnlich hätte man im vorigen Abschnitt schreiben können "TYPE BOOLEAN = (TRUE, FALSE)".

Nur war das nicht erforderlich, weil es bereits bei der PASCAL-Implementation als gegeben, als vordefiniert berücksichtigt wurde. Rechnerintern sind die selbstdefinierten Werte je nach Wertevorrat durch ein Bitmuster der Länge 8 oder 16 dargestellt. Je Element wird ein Bit manipuliert. "Lada" erhielte dabei wie ein Konstantenbezeichner (bei anderer Typverträglichkeit) das Bitmuster 0000 0000, "Wartburg" das Bitmuster 0000 0001 usw., so daß der Vergleich Lada < Wartburg programmtechnisch TRUE ist. Später wird auch die Formulierung "von Lada bis Skoda", "Moskwitsch ist Nachfolger für Trabant" einen programmtechnischen Sinn erhalten. Aus diesen Darlegungen folgt auch, daß ein selbstdefinierter Typ mit einem Wertevorrat $2^8 = 256$ ein Byte und darüber hinaus (bis $2^{16} = 65\,535$) zwei Byte belegt. Die Namensgebung unterliegt den Regeln für Bezeichner.

Die Nützlichkeit selbstdefinierter Datentypen liegt vor allem darin, sehr gut lesbare Programmtexte schreiben zu können. Im Programmbeispiel FAHRGELD wäre es durchaus möglich gewesen, die Ziffer 1 für "Lada", 2 für "Wartburg" usw. beizubehalten. Bei größeren Programmen ist aber dann nicht überall die Zuordnung zwischen Zahl und Problem sofort erkennbar.

Die Ein- und Ausgabe von Elementen dieses Typs ist natürlich nicht möglich, denn es sind dem Wesen nach programminterne Konstantenbezeichner. Es gibt aber viele Möglichkeiten, eine indirekte Ein- und Ausgabe zu realisieren. Im Programmbeispiel geschieht das durch Eingabe von Zahlen und anschließendes Retyping, also unter Verwendung des Typbezeichners als Pseudofunktion. Weil die Zahlendarstellung für die Elemente mit Null beginnt, die Zahlenzuordnung auf dem Bildschirm jedoch mit 1 (1 = Lada), ist das Retyping für Typ − 1 erforderlich.

Für das Verständnis der TYPE-Definition ist wichtig zu beachten, daß diese lediglich den Typ, also die Menge der zulässigen Werte, festlegt, selbst aber keinen Speicherplatz für diese Werte reserviert. Das geschieht im Programmbeispiel erst mit "VAR Fahrzeug: PKW". Diese Eigentümlichkeit von PASCAL ist vielleicht ungewohnt. Bedenkt man aber, daß der gewohnten Aussage "VAR x : BYTE" eine TYPE-Definition (gekürzt) TYPE BYTE = (0, 1, 2 ... 255) vorangegangen sein könnte, so ist dieses PASCAL-Konzept auch überzeugend.

Die Speicherplatzbelegung für "Fahrzeug" erfolgt mit dem Retyping PKW (Typ − 1). Wird also eine "2" für "Wartburg" eingegeben ("read(Typ)"), so liefert das Retyping das Element 1 (die Zählung beginnt mit Null) der definierten Typliste, hier realisiert sich also die Zuweisung

Fahrzeug := Wartburg.

Jeder Typbezeichner – das ist der Bezeichner links vom Gleichheitszeichen der TYPE-Definition, hier also PKW – kann, wenn die Implementierung das zuläßt, zugleich als Pseudofunktion für die Typanpassung genutzt werden. Sonst sind nur gleiche Aufzählungstypen untereinander verträglich.

Grundsätzlich ist in PASCAL auch eine direkte Vereinbarung von Speicherplatz für Aufzählungstypen möglich, im Beispiel durch

```
.
VAR Fahrzeug : (Lada,Wartburg,Trabant,Moskwitsch,Skoda);
.
```

Allerdings ist dann das Retyping unmöglich. Die TYPE-Definition wird später, vor allem bei der Vereinbarung strukturierter Datentypen und der Unterprogrammtechnik, noch sehr nützlich sein. Natürlich wäre im Beispiel FAHRGELD die Verwendung von Aufzählungstypen auch noch zweckmäßig, wenn das Retyping nicht möglich ist. Die Pseudofunktion ORD kann dann als Ersatz für Retyping eingesetzt werden, aber nur in einer Richtung, und das ist weit umständlicher.

Teilbereichstypen

Bevor das Beispiel FAHRGELD weiter analysiert wird, ist die Darlegung zu möglichen einfachen Datentypen in PASCAL mit den Teilbereichstypen abzuschließen. Dazu wird das Programm WOCHENTAG benutzt. Es ermöglicht die Eingabe eines Datums mit Tag, Monat, Jahr und antwortet mit dem Wochentag, auf den dieses Datum fallen wird bzw. schon gefallen ist. Für die Analyse dieses Programms benötigt man Grundkenntnisse des Gregorianischen Kalenders, der auf Vorschlag von Christoffel Clavius (1537–1612) der Zeiteinteilung seit 1583 zugrunde gelegt wurde. Danach beginnt das Jahrhundert mit einem Sonntag. Jedes vierte Jahr hat (außer die Jahrhundertwende selbst) einen Schalttag am 29. Februar.

Man hat also die Anzahl der Tage seit Beginn des laufenden Jahrhunderts zu berechnen. Der Rest einer Division durch 7 liefert den Wochentag (Sonntag = 1). Ein interaktives Programm, mit dem man für die Arbeitszeitplanung die Lage fester Feiertage – aber natürlich auch den persönlichen Geburtstag – ermitteln kann, ist folgendes:

```
PROGRAM Wochentag;
{Bestimmung des Wochentages nach dem Gregorianischen Kalender}
VAR Tag    : 1..31;
    Monat  : 1..12;
    Jahr   : 1583..MAXINT;
    Zeit   : INTEGER;
BEGIN
  writeln('Ermittlung des Wochenta es ab 1583');
  write('Datum (z.B. 19 6 1623): ');
  {¤R+} readln(Tag,Monat,Jahr); {¤R-}
  Zeit := Jahr MOD 100;                          {Jahrhundertbeginn}
  Zeit := Zeit {* (365 MOD 7)} + Zeit DIV 4; {Jahresanfang}
  Zeit := Zeit + (Monat-1) * 30 + Tag;           {Laufendes Jahr}
  IF ((Jahr MOD 4) = 0) AND (Monat <= 2) THEN Zeit := Zeit -1;
  CASE Monat OF
    2,6,7 : zeit := zeit + 1;                    {Monatskorrektur}
    3     : zeit := zeit - 1;
    8     : zeit := zeit + 2;
    9,10  : zeit := zeit + 3;
   11,12  : zeit := zeit + 4;
 END;
 writeln('Dieser Tag fiel/faellt auf einen');
 CASE (Zeit MOD 7) + 1 OF
   1 : writeln('Sonntag');
   2 : writeln('Montag');
   3 : writeln('Dienstag');
   4 : writeln('Mittwoch');
   5 : writeln('Donnerstag');
   6 : writeln('Freitag');
   7 : writeln('Sonnabend')
END;
  write('Ende');
END.
```

Teilbereichstyp

Die Programmausführung ergibt zum Beispiel das folgende Protokoll (Eingaben unterstrichen):

```
Ermittlung des Wochentages ab 1583
Datum(z.B. 19 6 1623): 19 6 1623<ET>
Dieser Tag fiel/faellt auf einen
Dienstag
Ende
```

Der 19. 6. 1623 ist der Geburtstag von Blaise Pascal.
Die Aufmerksamkeit gilt zunächst nur den Variablen "Tag", "Monat", "Jahr". Natürlich entsteht die Forderung, keine falschen Eingaben zuzulassen. Falsch wäre zum Beispiel 28. 14. 1984. Man muß also sichern, daß sich die Werte der Variablen "Tag" im Bereich 1 bis 31, Monat im Bereich 1 bis 12 und Jahr größer als 1582 befinden. Das wäre mit den bisherigen Mitteln möglich. Man würde die drei Variablen als Typ INTEGER vereinbaren und nach der Eingabe fragen:

```
.
IF (Tag < 1) OR (Tag > 31) OR (Monat < 1)
  OR (Monat > 12) OR (Jahr < 1583) THEN BEGIN
    write('Fehlerhafte Eingabe'); halt
END;
.
```

Abbruch

Ist der logische Ausdruck TRUE, erfolgt die Fehlerausschrift, und danach wird die Programmausführung abgebrochen (vordefinierte Prozedur HALT).
Einfacher als diese Fehlerbehandlung ist es, für die Überwachung das Datentypkonzept von PASCAL zu nutzen. Mit der Vereinbarung 1..31 wird der Wertevorrat für INTEGER auf den Teilbereich von 1 bis 31 eingeschränkt. Deshalb wird dieser Datentyp als Teilbereichstyp (Subrange-Typ) bezeichnet. Da das sogar ein Teilbereich von BYTE ist, wird auch nur 1 Byte Speicherplatz reserviert, was zusätzlich die Speicherplatzeffizienz erhöht. Für die Variable "Monat" geschieht das für den Bereich 1.. 12. Der Teilbereichstyp "Jahr" benutzt die vordefinierte Konstante MAXINT = 32767 als obere Grenze. Nach diesen Vereinbarungen überwacht der Rechner auch während der Laufzeit die Typverträglichkeit wie für INTEGER überhaupt.
Es muß aber hervorgehoben werden, daß eine standardmäßige Überwachung der Typverträglichkeit im ganzen Programm die Laufzeit erheblich verlängern würde. Deshalb sehen die PASCAL-Implementationen die Überwachung der Bereichsgrenzen nicht standardmäßig vor. Sie muß mit Compilerdirektiven ein- und, wenn möglich, auch wieder ausgeschaltet werden. Das geschieht im Programmbeispiel WOCHENTAG mit den Compilerdirektiven {¤R+} und {¤R−}.
Es soll noch angemerkt werden, daß Teilbereichstypen nicht nur für BYTE und INTEGER möglich sind. Zum Beispiel definiert

'a'..'z'	den Teilbereich Kleinbuchstaben des Zeichensatzes für CHAR
'A'..'Z'	den Teilbereich Großbuchstaben des Zeichensatzes
Wartburg..Trabant	den Teilbereich Inlandproduktion des Typs PKW im Beispiel FAHRGELD

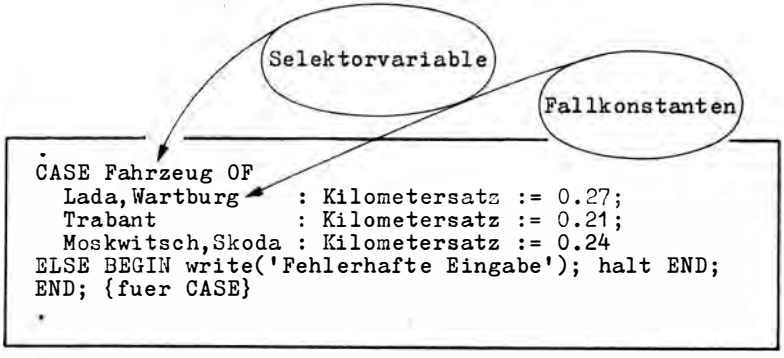
Teilbereiche des Typs REAL sind nicht zulässig, denn die Anzahl zulässiger Werte wäre unbegrenzt.
Damit sind die in PASCAL verfügbaren einfachen Datentypen behandelt. Es wird zunächst auf das Programm FAHRGELD zurückgekommen.

3.4. Selektion durch Fallunterscheidung

Im Programm FAHRGELD müssen für die Eingabe fünf und für die interne Verarbeitung drei Möglichkeiten vorgesehen werden. Es gibt nämlich fünf wichtige Fahrzeugtypbezeichnungen und drei verschiedene Kilometersätze. Natürlich wäre eine Lösung mit der aus Abschnitt 3.2. bekannten IF-Anweisung möglich:

```
.
IF (Fahrzeug = Lada) OR (Fahrzeug = Wartburg)
  THEN Kilometersatz := 0.27
ELSE IF Fahrzeug = Trabant THEN Kilometersatz := 0.21
     ELSE IF (Fahrzeug = Moskwitsch) OR (Fahrzeug = Skoda)
            THEN Kilometersatz := 0.24
          ELSE write('Fehlerhafte Eingabe');
.
```

Aber man sieht wohl, daß diese Struktur unübersichtlich ist. Wenn sich die Zahl der Möglichkeiten auf 10 oder 20 erhöht, ist eine Folge von IF-Anweisungen verwirrend und damit fehleranfällig.
PASCAL stellt deshalb eine spezielle Anweisung, die CASE-Anweisung, zur Verfügung. Die exakte Syntax ist Anhang C zu entnehmen.
Im Beispiel ist sie



Der ELSE-Zweig ist in einigen Implementationen nicht zulässig.
Die durch die CASE-Anweisung realisierte Auswahl wird auch als Fallunterscheidung bezeichnet. Nach dem reservierten Wort CASE steht ein Selektorausdruck. Das kann eine Variable vom Aufzählungstyp, jede andere ordinale Variable oder auch ein Ausdruck mit ordinalem Ergebnis sein. Ordinal bezeichnet den abzählbaren Wertevorrat und ist das Attribut aller einfachen Variablen vom Typ INTEGER, BYTE, CHAR, BOOLEAN – außer REAL. Ein String als Ganzes ist ebenfalls nicht zulässig, wohl aber eine STRING-Komponente, die ja vom Typ CHAR ist.

Je nach dem Wert der Selektorvariablen bzw. des Selektorausdrucks (der dann erst ermittelt wird) verzweigt die Programmausführung zu der Anweisung, die mit dem Wert (der Fallkonstanten) markiert ist, den der Selektor aktuell besitzt. Dabei kann eine Anweisung innerhalb des CASE mit mehreren Fallkonstanten markiert sein (zum Beispiel "Lada, Wartburg:"). Nach Ausführung dieser Anweisung (es kann natürlich auch eine Verbundanweisung sein, die mehrere andere Anweisungen klammert) gilt die CASE-Anweisung in ihrer Gesamtheit als ausgeführt. Existiert für den aktuellen Selektorwert keine Fallkonstante, so wird der ELSE-Zweig der CASE-Anweisung ausgeführt. Im Beispiel wird eine Fehlermitteilung ausgeschrieben und mit der vordefinierten Prozedur HALT ein Programmabbruch mit Rückkehr zum Betriebssystem veranlaßt. Besitzt die CASE-Anweisung keinen ELSE-Zweig und keine mit dem aktuellen Selektorwert übereinstimmende Fallkonstante, so wird die gesamte CASE-Anweisung (von CASE bis END) wie eine Leeranweisung behandelt und die Programmausführung mit der folgenden Anweisung fortgesetzt.
Um die Anwendung der CASE-Anweisung für den Fall zu zeigen, daß die Selektorvariable vom Typ CHAR ist, wird der folgende Programmabschnitt verwendet:

```
.
VAR Fahrzeug :CHAR;
BEGIN
  writeln('  Lada          = L');
  writeln('  Wartburg      = W');
  writeln('  Trabant       = T');
  writeln('  Moskwitsch    = M');
  writeln('  Skoda         = S');
  read(Fahrzeug); writeln;
  CASE Fahrzeug OF
    'L','l','W','w': Kilometersatz := 0.27;
    'T','t'        : Kilometersatz := 0.21;
    'M','m','S','s': Kilometersatz := 0.24
  ELSE BEGIN write('Fehlerhafte Eingabe'); halt END;
  END;
.
```

Man sieht, daß die CHAR-Fallkonstanten in Apostrophe einzuschließen sind.
Bei dieser kleinen Aufgabe scheint die Verwendung einer CHAR-Variablen für "Fahrzeug" vielleicht sogar einfacher als die Verwendung von Aufzählungstypen, aber nur, weil das Programm sehr klein und dadurch die Zuordnung der Fahrzeugtypen zur Fallkonstanten sofort sichtbar ist. Bei größeren Programmen ist das nicht mehr gegeben. Wie sich das Programm verhält, wenn Fallkonstanten in der CASE-Anweisung mehrfach notiert werden, ist implementationsabhängig. Bei den Fallkonstanten

'L', 'W': ...
'T', 'W': ...

würde im allgemeinen die Liste der Fallkonstanten von "oben nach unten" durchgemustert. Wird das erstemal Übereinstimmung zwischen Fallkonstante und Selektorwert festgestellt, so ist die Suche beendet, und die nachstehende Anweisung wird ausgeführt. Das zweite 'W' hätte also keine Wirkung. Die PASCAL-Compiler zeigen deshalb mehrfach verwendete Fallkonstanten nicht als Syntaxfehler an.
Jetzt soll auf das Beispiel WOCHENTAG zurückgekommen werden. Die CASE-Anweisung ist hier gleich zweimal enthalten.

Die Anweisung

```
.
CASE Monat OF
  2,6,7 : Basis := Basis + 1;
  3     : Basis := Basis - 1;
  8     : Basis := Basis + 2;
  9,10  : Basis := Basis + 3;
  11,12 : Basis := Basis + 4
END;
.
```

gleicht Differenzen aus. Sie ergeben sich aus der unterschiedlichen Länge der Monate. Verwendet man das Produkt 7 mal 30 für die Berechnung der Tage des laufenden Jahres bis zum Ende des Vormonats, so ergibt sich ein Fehler. Wird zum Beispiel das Datum 14 für Tag und 3 für Monat eingegeben, so ist die Anzahl der Tage des laufenden Jahres bis zum Ende des Vormonats 31 (Januar) + 28 (Februar) gleich 59. Das Produkt 2 mal 30 ist aber 60. Deshalb erfolgt die Korrektur (Fallkonstante 3) mit − 1. Für Berechnungen von Daten im Mai sind bis Ende April 31 + 28 + 31 + 30 = 120 Tage zu berücksichtigen. Das ist gleich 4 mal 30. So ist es auch im April. Deshalb sind die Fallkonstanten 4 und 5 nicht aufgeführt. Man achte darauf, daß bei Selektoren vom Typ BYTE oder INTEGER einfach die Zahlen als Fallkonstante aufgeführt werden. Die zweite CASE-Anweisung und die anderen Teile des Programms WOCHENTAG bringen nichts Neues.
Abschließend soll noch darauf hingewiesen werden, daß die CASE-Anweisung etwas mehr Speicherplatz verbraucht als die IF-Anweisung und auch langsamer ist als diese. Diese Fragen treten aber in der Softwaretechnologie immer mehr hinter Transparenz und Klarheit der Programmstruktur zurück, die bereits bei fünf Fallunterscheidungen besser ist.

Übungsaufgaben

1. Erweitern Sie das Programm GEBUEHREN so, daß Zuschlaggebühren für
   dringende Gespräche (doppelte Gebühr),
   Blitzgespräche (10fache Gebühr),
   berücksichtigt werden!

2. Entwickeln Sie ein interaktives Programm, das nach Eingabe der Kilometerzahl, der Zugart (P, E, S) und der Klasse (1, 2) den Fahrpreis für eine Dienstreise (ohne Ermäßigung) mit der Deutschen Reichsbahn ermittelt. Für die Zuschläge gilt:

	Eilzug(E)		Schnellzug(S)	
	1.Klasse	2.Klasse	1.Klasse	2.Klasse
Zone 1(bis 300 km)	3.00	1.50	6.00	3.00
Zone 2(ueber 300 km)	5.00	2.50	10.00	5.00

   Der Kilometertarif beträgt 11.6 Pfennig für die erste und 8 Pfennig für die zweite Klasse.

4. Iterative Ausführung von Anweisungen

4.1. Felder

Für viele Anwendungen ist es unzweckmäßig, jedem Datenobjekt innerhalb des Programms einen "individuellen" Namen zu geben. Es soll zum Beispiel die Summe von 20 Werten gebildet werden. Stehen diese Werte auf Speicherplätzen, von denen jeder einen anderen Namen hat, so entsteht eine Ergibtanweisung mit 20 Summanden. Das ist vielleicht noch möglich. In umfangreicherer Anwendung sind aber manchmal Tausende Werte zu addieren oder nach irgendeiner anderen Regel zu verarbeiten. Dann ist eine andere Vorgehensweise erforderlich.

Dazu wird als Beispiel die Sicherung eines Schlüssels betrachtet. Schlüssel sind Folgen von Zeichen, die in Kurzform Objekteigenschaften identifizieren und klassifizieren. Bekannte Schlüsselsysteme sind die Personenkennzahl, die Kontensystematik der Sparkasse, die Erzeugnisliste und die Handelsschlüsselliste. Schlüssel spielen bei der Verwaltung von Daten eine große Rolle und sind deshalb vor Fehlern besonders zu schützen. Eine gebräuchliche Methode ist das Anhängen einer Prüfziffer nach dem Divisionsverfahren.[6] Das geschieht wie folgt:

- Jede Schlüsselstelle wird mit einer Zahl F multipliziert, und die Summe der Produkte wird gebildet. F kann je Schlüsselstelle verschieden sein.
- Die Summe der Produkte wird durch eine Zahl D dividiert und die Prüfziffer als Rest dieser Division (Modulus) festgestellt.
- Die Prüfziffer wird an den Schlüssel angehängt und dient der Kontrolle vor jeder Verarbeitung.

Ein Beispiel ist folgendes (Bestellnummer aus dem Sortiment Waren täglicher Bedarf):

Zu sichernder Schlüssel	1	8	1	9	3	7	3	8
F (i)	8	7	6	5	4	3	2	1
Produkte	8	56	6	45	12	21	6	8
Summe der Produkte	162							
D	10							
Rest der Division	2							
Gesicherter Schlüssel	1	8	1	9	3	7	3	8 2

6 Vgl. Herausgeberkollektiv: Grundlagen der elektronischen Datenverarbeitung für Ökonomen. Berlin: Verlag Die Wirtschaft 1982, S. 83ff.

Das folgende Programm übernimmt die Ermittlung des gesicherten Schlüssels im Dialog:

```
PROGRAM Schluessel;
{Schluesselsicherung durch Pruefziffer}
CONST Divisor = 10; Laenge = 9;
VAR Schluessel      : ARRAY[1..Laenge] OF INTEGER;
    Produktsumme,i,j: INTEGER;
BEGIN
  writeln('Sicherung numerischer Schluessel durch Pruefziffern');
  write('Zu sichernder Schluessel: ');
  FOR i := 1 TO Laenge - 1 DO read(TRM,Schluessel[i]);
  Produktsumme:=0;
  FOR i := 1 TO Laenge - 1 DO
     Produktsumme:= Produktsumme + Schluessel[i] * (Laenge - i);
  Schluessel[Laenge]:= Produktsumme MOD Divisor;
  write('Gesicherter   Schluessel: ');
  FOR i:= 1 TO Laenge DO  write(Schluessel[i]);
  writeln;
  write('Ende');
END.
```

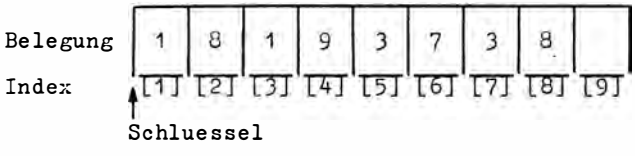
Bei der Ausführung des erläuterten Beispiels entsteht das folgende Protokoll:

```
Sicherung numerischer Schluessel durch Pruefziffern
Zu sichernder Schluessel: 1 8 1 9 3 7 3 8<ET>
Gesicherter   Schluessel: 181937382
Ende
```

Die Prüfziffer ist also 2.

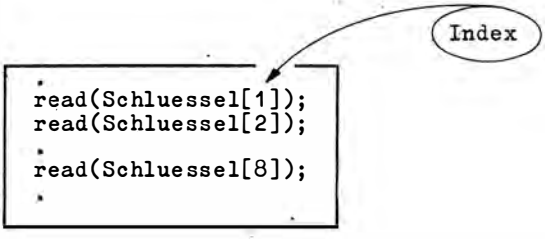
Interne Darstellung und Zugriff

Analysiert wird zunächst nur die Vereinbarung "VAR Schluessel: ARRAY [1..Laenge] OF INTEGER". Im Unterschied zu früheren Variablenvereinbarungen wird hier mit dem Bezeichner "Schluessel" nicht nur ein, sondern es werden neun (Länge ist durch die Konstantendefinition gleich 9) Speicherworte als zusammenhängender Bereich für die Aufnahme von INTEGER-Werten festgelegt. Einen solchen zusammenhängenden Bereich, bei dem alle Bereichskomponenten vom gleichen Typ sind (hier INTEGER) nennt man Feld oder ARRAY. Die entsprechenden Variablen (hier "Schluessel") heißen Feld- bzw. ARRAY-Variablen. Feldvariablen belegen einen Speicherbereich. Die Anordnung im Hauptspeicher zeigt die folgende Darstellung:

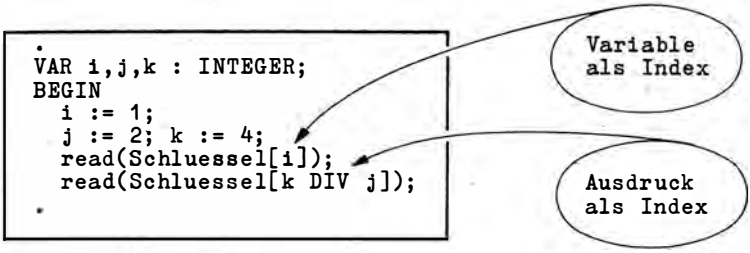


Als Inhalt der Speicherworte wurden die Komponenten des zu sichernden Schlüssels aus dem Beispiel eingetragen. Auf diese einzelnen Worte kann in einem Programm zugegriffen werden, indem man den Namen des Feldes verwendet und in eckige Klammern einen Index einschließt. In der Darstellung ist Schlüssel [1] = 1, Schlüssel [4] = 9 usw. Bezeichner des Feldes und Index identifizieren die Speicherworte genauso exakt, als hätte

jedes Wort einen eigenen Namen. Nur werden sich erhebliche Vorteile für die Verarbeitung ergeben. Auch kann auf Felder als Ganzes Bezug genommen werden. Die Eigenschaft, Teilkomponenten zu besitzen, auf die einzeln zurückgegriffen werden kann, unterscheidet Felder grundsätzlich von einfachen, elementaren Variablen. Felder gehören deshalb zum strukturierten Datentyp.
Zunächst wäre es durchaus zulässig, die acht Ziffern des zu sichernden Schlüssels durch die folgenden Anweisungen zu lesen:



und so die Eingabe zu ermöglichen. Im folgenden Abschnitt wird dazu eine iterative Schreibweise dargelegt, die das Ganze auf eine Anweisung reduziert.
Hier ist zunächst noch wichtig zu wissen, daß der Index in den eckigen Klammern auch eine Variable oder sogar ein INTEGER-wertiger Ausdruck sein kann. Die folgenden Anweisungen wären also zulässig:



Natürlich müssen die Werte der Indexausdrücke im vereinbarten Bereich liegen. Der Zugriff Schluessel [12] führt zum Fehler, wird aber vom Rechner nur angezeigt, wenn das über Compilerdirektiven gesondert vom Programmierer gefordert wird. Eine standardmäßige Überwachung würde die Laufzeit erheblich belasten.

Modifikationen der Feldvereinbarung
Natürlich sind Felder nicht nur für den Datentyp INTEGER, sondern für alle Datentypen, also auch CHAR, BYTE, STRING, REAL, Teilbereichs- und Aufzählungstypen möglich. Das Feld belegt dann jeweils den Speicherplatz, der sich aus der Multiplikation der Speicherbelegung für eine Variable mit der Anzahl der Komponenten ergibt.
Das Feld "Schluessel [1. .9] OF INTEGER" belegt 18 Byte.
Ein Feld "Schluessel [1. .9] OF CHAR" belegt 9 Byte usw.
In PASCAL gehört es zum Programmierstil, eine möglichst transparente Darstellung anzustreben. Dazu kann besonders auch die Anwendung der TYPE-Vereinbarung als Vorvereinbarung des Datentyps vor der eigentlichen Variablenvereinbarung genutzt werden. Eine Möglichkeit im Programm SCHLUESSEL wäre folgende (völlig gleichwertige) Notation:

```
.
CONST Laenge = 9;
TYPE  Feld   = ARRAY[1..Laenge] OF INTEGER;
VAR   Schluessel : Feld;
.
```

Wie schon gezeigt, wird die Definition der zulässigen Werte im Interesse besserer Lesbarkeit – bei mehreren Feldern auch zur vereinfachten Schreibweise – vorgezogen. Das gesonderte Vereinbaren der oberen Indexgrenze 9 in der Konstantenvereinbarung hat einen praktischen Vorteil. Ändert sich die Schlüssellänge im Beispiel, muß nur diese eine Stelle im Programm geändert werden. Das schützt vor Nebeneffekten, die sonst bei der Änderung bestehender Programme auftreten können. Keinesfalls ist das Feld dadurch ein dynamisches Feld. Aber eigentlich gibt es eine noch günstigere Notation, wenn man sich an die Möglichkeit der Verwendung von Teilbereichstypen erinnert:

```
.
TYPE Index = 1..9;
     Feld  = ARRAY[Index] OF INTEGER;
VAR  Schluessel : Feld;
.
```

Diese Darstellung bietet die Möglichkeit, darauf hinzuweisen, daß es für PASCAL keine Restriktionen für die Wahl des Indexbereiches gibt wie zum Beispiel in FORTRAN oder BASIC.

"TYPE Index = 0..8",
"TYPE Index = 'a'..'i'"

oder sogar ein negativer Bereich hätten den gleichen Zweck erfüllt. Im Beispiel FAHRGELD könnte man sogar auf die folgende Idee kommen:

```
.
TYPE PKW      = (Lada,Wartburg,Trabant,Moskwitsch,Skoda);
     Index    = Lada..Skoda;
VAR  Fuhrpark: ARRAY[Index] OF STRING[20];
.
```

Man wählt den Indexbereich natürlich so, daß klare, gut lesbare Programme notiert werden können.

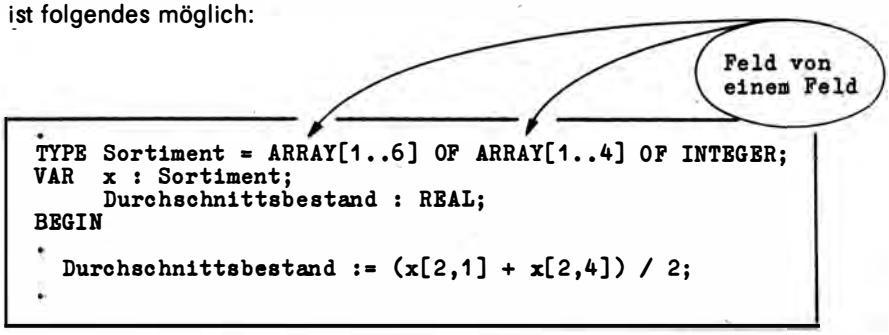
Felder von Feldern

Es wurde bereits darauf hingewiesen, daß Felder für jeden Datentyp vereinbart werden können, damit auch für Felder selbst. Die praktischen Erfordernisse liegen auf der Hand: Ein Produktions- oder Handelssortiment besteht aus einzelnen Artikeln (Feld 1, 1. Dimension), also aus Artikel 1, Artikel 2, Artikel 3 usw., und jeder Artikel hat Parameter (Feld 2, 2. Dimension) wie Anfangsbestand, Zugang, Abgang, Endbestand usw.
Felder von Feldern sind eine Möglichkeit, diese Struktur zu beschreiben.

In einer Tabelle kann man das so veranschaulichen:

	Anfangsbestand 1	Zugang 2	Abgang 3	Endbestand 4
Artikel 1	x[1,1]	x[1,2]	x[1,3]	x[1,4]
Artikel 2	x[2,1]	x[2,2]	x[2,3]	x[2,4]
Artikel 3	x[3,1]	x[3,2]	x[3,3]	x[3,4]
.	.	.	.	.
Artikel 6	x[6,1]	x[6,2]	x[6,3]	x[6,4]

In Verflechtungsbilanzen sind die Dimensionen sofort sichtbar. Die lineare Algebra nennt sie Matrizen.
Die Vereinbarung der Felder von Feldern kann durch eine TYPE-Definition vorgezogen werden oder natürlich direkt durch VAR-Deklaration erfolgen. Für das genannte Beispiel ist folgendes möglich:




```
TYPE Sortiment = ARRAY[1..6] OF ARRAY[1..4] OF INTEGER;
VAR  x : Sortiment;
     Durchschnittsbestand : REAL;
BEGIN
  .
  Durchschnittsbestand := (x[2,1] + x[2,4]) / 2;
  .
```

Dem Feld wird also noch ein "ARRAY OF" vorgesetzt. Zur Vereinfachung der Schreibweise ist auch

"TYPE Sortiment = ARRAY [1..6,1..4] OF INTEGER"

erlaubt.
Wie man sieht, erfolgt der Zugriff auf die Komponenten dieses Feldes mit

"x[2,1]",

also dem Feldbezeichner und zwei Indizes. Hier wird die zweite Komponente des ersten Feldes (also im Beispiel der zweite Artikel) und innerhalb dieser zweiten Komponente die erste Komponente (also der Anfangsbestand dieses Artikels) identifiziert. Völlig gleichwertig mit diesem Zugriff ist

"x[2][1]".

In allen Implementationen sind wenigstens dreidimensionale Felder erlaubt. In dreidimensionalen Feldern ist jedes Element eines zweidimensionalen Feldes selbst wieder ein Feld.
Bei der Vereinbarung von Feldern sollte man wissen, daß es Vor- und Nachteile gibt. Das ist in PASCAL wie in allen Programmiersprachen. Der Vorteil besteht in der Möglichkeit, sich wiederholende Vorgänge rationell und transparent zu notieren. Viele typische Operationen, vor allem Suchen, Sortieren, Zeitreihenanalysen usw., sind überhaupt nur mit

Feldern möglich. Auch ist Einsparung von Speicherplatz für die Symboltabelle möglich. Sie enthält nur den Feldbezeichner.
Ein Nachteil ist, daß bei jedem Zugriff eine Adreßrechnung (Indexrechnung) erforderlich ist. Bei eindimensionalen Feldern besteht sie in der Addition der Adresse aus der Symboltabelle und dem Produkt aus Wortlänge und Zugriffsindex minus Anfangsindex. Bei mehrdimensionalen Feldern ist die Indexrechnung für jeden Index gesondert erforderlich und ab zweitem Index durch Berücksichtigung der vorhergehenden Dimension sogar noch komplizierter. Bei nur eindimensionalen Feldern ist der entsprechende Zeitverzug unbedenklich. Bei zeitkritischen Problemen sollten mehrere Dimensionen auf eine Dimension zurückgeführt werden. Das nennt sich Skalieren.

Operationen mit Feldern und Typverträglichkeit
Für die Ein- und Ausgabe von Feldern, für Wertzuweisungen und für die Verwendung von Feldern in Ausdrücken gelten Besonderheiten. Diese Besonderheiten gibt es nur für Operationen mit Feldern als Ganzes. Komponenten dieser Felder, also indizierte Variablen, sind in fast allen Belangen (Ausnahmen gibt es für formale Parameter in Unterprogrammen) den einfachen Variablen gleichgesetzt. Das ist ein wichtiger Grundsatz. Aus ihm folgt, daß alle Operationen, die mit dem Feld als Ganzes nicht durchgeführt werden dürfen, auf komponentenweise Operationen zurückzuführen sind.Dazu werden in den folgenden Abschnitten mit FOR, REPEAT und WHILE leistungsstarke Anweisungen vorgestellt.
Die Besonderheiten der Arbeit mit Feldern sind also Einschränkungen. Dabei ist von Bedeutung, ob die Felder gepackt sind oder nicht und welchen Typs die Feldkomponenten sind.
Unter Packen versteht man ein Verdichten der internen Datendarstellung. Im Programmbeispiel FAHRGELD wurde der Typ PKW mit einem Wertevorrat der fünf Bezeichner "Lada", "Wartburg", "Trabant", "Moskwitsch", "Skoda" eingeführt. Zur Speicherung von acht Werten würde man ein Feld des Typs PKW mit acht Komponenten benötigen. Jede Komponente belegt ein Byte, das sind acht Byte insgesamt. Eigentlich würde aber 1 Bit genügen, um eine Komponente zu speichern. Also ist nur ein Byte wirklich erforderlich (damit können $2^3 = 8$ Zustände unterschieden werden). Das Verdichten auf den direkt erforderlichen Speicherplatz ist in den verschiedenen PASCAL-Implementationen sehr unterschiedlich. Die meisten Compiler für Mikrorechner akzeptieren in der TYPE-Definition das Wortsymbol PACKED, weil es Bestandteil des ISO-Standardvorschlages ist, reagieren aber nicht darauf. Spezielle Prozeduren PACK (zum Packen) und UNPACK (zum Entpakken) sind für PASCAL auf Mikrorechnern ebenfalls nicht implementiert. Grundsätzlich erfolgt überall dort standardmäßig ein Packen, wo das zweckmäßig ist. Man sollte wissen, daß Packen zwar Speicherplatz spart, aber die Zugriffszeit verlängert, denn ein Zugriff setzt im allgemeinen das Entpacken voraus.
Nun sollen die wesentlichen Einschränkungen für die Arbeit mit Feldern mikrorechnertypischer PASCAL-Implementationen aufgezählt werden:

a) Die Eingabe eines Feldes muß komponentenweise erfolgen. Die folgenden Befehle sind also fehlerhaft:

```
.
VAR Eingabe : ARRAY[1..20] OF CHAR;
BEGIN
  read(Eingabe);
.
```

Fehler

Die programmtechnischen Mittel zur Lösung des Problems werden im folgenden Abschnitt behandelt.

b) Felder als Ganzes sind zuweisungsverträglich, wenn sie den gleichen Typ besitzen. Einige PASCAL-Implementationen fordern streng, daß sie in der gleichen Typvereinbarung definiert werden:

```
.
TYPE Feld = ARRAY[1..20] OF REAL;
VAR  x,y  : Feld;
     z    : ARRAY[1..20] OF REAL;
BEGIN
  x := y;
  z := y;
.
```

zulaessig

Typkonflikt

c) Die Verwendung von Teilfeldern ist im allgemeinen nicht zugelassen, vor allem nicht, wenn b) gültig ist:

```
.
VAR x : ARRAY[1..10,1..10] OF INTEGER;
    y : ARRAY[1..10] OF INTEGER;
BEGIN
  y := x[2];
.
```

implementations-
abhaengig

d) Arithmetische und logische Operationen mit Feldern als Ganzes, einschließlich Vergleiche, sind nicht zugelassen:

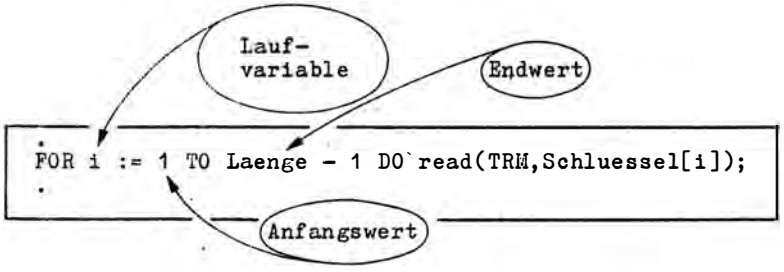
```
.
VAR x,y : ARRAY[1..30] OF BYTE;
BEGIN
.
IF x > y THEN write('x ist groesser als y');
.
```

Fehler

Eine Ausnahme von den Einschränkungen bildet die Ausgabe von Feldern des Typs CHAR. Die Ausgabe des gesamten Feldes wird auch vorgenommen, wenn nur der Feldbezeichner geschrieben wird.

4.2. Iteration bekannter Häufigkeit (FOR-Anweisung)

Die Vereinbarung von Feldern allein würde das Problem der Verarbeitung großer Datenmengen durch ein Programm nicht lösen. Eine Addition von 20 Werten hätte weiterhin eine umfangreiche Ergibtzuweisung zur Folge. Neben dem Variablennamen wären auch die eckigen Klammern und der Index zu notieren. Die Lösung bringt eine spezielle Anweisung, die zwar nur einmal notiert wird, aber für einen Zyklus, eine Schleife, eine Wiederholung – eine Iteration sorgt. Eine solche Anweisung ist die FOR-Anweisung. Sie wird im Programm SCHLUESSEL gleich zweimal verwendet, zunächst zur Realisierung der sonst nicht möglichen Eingabe eines Feldes:



Diese Anweisung, die FOR-Anweisung, gestattet die Eingabe von acht Werten, weil sie wiederholt ausgeführt wird.

Um die Anzahl der Wiederholungen steuern zu können, gibt es eine sogenannte Laufklausel, bestehend aus Laufvariable, Anfangswert und Endwert. Die Laufvariable muß eine einfache Variable vom ordinalen Typ sein. Endwert und Anfangswert sind Variablen, Konstanten (hier "1") oder Ausdrücke (hier "Laenge −1"). Laufvariable, Anfangswert und Endwert müssen typverträglich sein. Hier sind alle vom Typ INTEGER. Die FOR-Anweisung arbeitet in vier Schritten:

a) Der Laufvariable (hier "i") wird der Anfangswert (hier "1") zugewiesen.
b) Die Anweisung hinter DO (hier "read (TRM, Schluessel [i])") wird ausgeführt. Hinter DO kann immer nur eine Anweisung, gegebenenfalls eine Verbundanweisung stehen.
c) Der Laufvariablen wird der Nachfolger des aktuellen Wertes der Laufvariablen oder, wenn für "TO" in der Anweisung "DOWNTO" notiert ist, der Vorgänger zugewiesen. Bei Laufvariablen des Typs INTEGER läuft das auf die Erhöhung oder Verminderung um 1 hinaus.
d) Es wird geprüft, ob die Laufvariable den Endwert überschritten (bei TO) oder unterschritten (bei DOWNTO) hat. Ist das der Fall, wird die FOR-Anweisung verlassen und die nächste notierte Anweisung ausgeführt. Ist das nicht der Fall, wird zu b) zurückgekehrt und erneut in diese Schrittfolge eingerastet.

Abweichend vom Standardvorschlag behandeln neuere Implementationen die FOR-Anweisung wie eine Leeranweisung, wenn bereits am Anfang der Endwert überschritten (TO) oder unterschritten (DOWNTO) ist.

Im Beispiel SCHLUESSEL ergibt sich die nachstehende Speicherbelegung und Befehlsfolge:

Durchlauf	Speicherbelegung Laufvariable i	Endwert	Anweisung
1	1	8	read[1]
2	2	8	read[2]
.	.	.	.
8	8	8	read[8]
9	9	8	-

Obwohl die Anweisung "read [i]" nur einmal notiert ist, wird sie achtmal ausgeführt. Auf diese Weise füllt sich das Feld Schluessel: ARRAY [1 ... Laenge] in den ersten acht Komponenten mit den jeweils vom Nutzer eingegebenen Werten.
Etwas schwieriger ist die zweite FOR-Anweisung im Programm SCHLUESSEL:

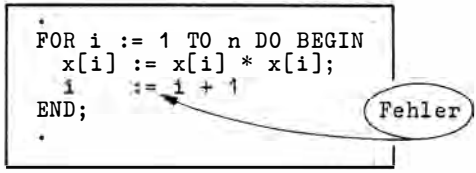
```
.
Produktsumme := 0;
FOR i := 1 TO Laenge - 1 DO
  Produktsumme := Produktsumme + Schluessel[i] * (Laenge - i);
.
```

Sie vollzieht die Multiplikation der Ziffern des Schlüssels mit den Faktoren und bildet die Summe der Produkte (das Skalarprodukt). Beim ersten Aufruf der Ergibtanweisung hätte die Variable "Produktsumme" keinen Wert. Sie wird deshalb unmittelbar vor der Schleife Null gesetzt. Nunmehr vollzieht sich zyklisch, iterativ folgendes:

Durchlauf (Zyklus)	Produktsumme links	Produktsumme rechts	i	Schluessel[i]	Laenge - i
1	8	0	1	1	8
2	64	8	2	8	7
3	70	64	3	1	6
4	115	70	4	9	5
5	127	115	5	3	4
6	148	127	6	7	3
7	154	148	7	3	2
8	162	154	8	8	1

Der Rest der Division 162 durch 10 ist 2.
Grundsätzlich ist die FOR-Anweisung anzuwenden, wenn die Anzahl der Wiederholungen vor dem Eintritt in die Anweisung bekannt ist oder aus Variableninhalten ermittelt werden kann. In diesem Falle ist sie unbedingt den noch zu behandelnden REPEAT- und WHILE-Anweisungen vorzuziehen. Sie ist schneller als diese.
Zur Erhöhung der Laufzeiteffizienz von FOR-Anweisungen verwenden die Compiler spezielle Techniken. So wird zum Beispiel die Anzahl der Durchläufe vor dem Eintritt in den eigentlichen Zyklus berechnet und auf einer Hilfszelle zwischengespeichert. Der Wert der Laufvariablen ist dann ein Offset (konstante Differenz) der absolvierten Durchläufe. Aus diesem Grunde sollte man wissen:

a) Der Wert der Laufvariablen darf in der FOR-Anweisung nicht geändert werden. Der Ausschnitt



ist keine Möglichkeit, für jede zweite Komponente eines Vektors das Quadrat zu bilden.

b) Nach dem Verlassen der FOR-Anweisung ist nicht sicher, daß die Laufvariable den Endwert besitzt. Sie ist deshalb als undefiniert anzusehen.

Für die Laufvariable ist noch anzumerken, daß sie ein beliebiger einfacher ordinaler Typ sein kann. Zulässig ist also auch

```
.
VAR i : CHAR;
BEGIN
  FOR i := 'a' TO 'z' DO write(i:2);
  writeln;
.
```

In diesem Falle wird das kleine Alphabet auf den Bildschirm geschrieben.

Eine große Rolle spielen FOR-Anweisungen und Felder in der Matrizenrechnung. Der folgende Programmausschnitt zeigt die Befehlsfolge zur Multiplikation einer Matrix "Links" von rechts mit einer Matrix "Rechts". Die Matrix "Links" hat bei dieser Verknüpfung stets ebensoviel Spalten, wie die Matrix "Rechts" Zeilen hat. Diese Reihigkeit wird mit "Gemeinsam" bezeichnet:

```
.
VAR Links     : ARRAY[1..Zeilen,1..Gemeinsam]  OF Typ;
    Rechts    : ARRAY[1..Gemeinsam,1..Spalten] OF Typ;
    Ergebnis  : ARRAY[1..Zeilen,1..Spalten]    OF Typ;
    i,j,k     : INTEGER;
BEGIN
.
  FOR i := 1 TO Zeilen DO
    FOR j := 1 TO Spalten DO BEGIN
      Ergebnis[i,j] := 0;
      FOR k := 1 TO Gemeinsam DO
        Ergebnis[i,j] := Ergebnis[i,j] + Links [i,k]
                                       * Rechts[k,j]
    END;
.
```

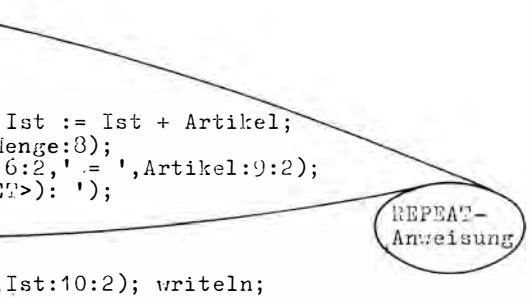
Typ ist durch eine TYPE-Definition mit INTEGER, BYTE oder REAL vordefiniert (oder kann in der Befehlsfolge hier für "Typ" eingesetzt werden). "Zeilen", "Spalten" und "Gemeinsam" sind vorher durch CONST festgelegt. Ein Unterprogramm INVERSE zur Berechnung einer Kehrmatrix enthält Anlage I.

Werden FOR-Anweisungen in Unterprogrammen benutzt, so muß die Laufvariable lokal zum Block deklariert sein (vgl. Abschnitt 5).

4.3. Offene Iteration nicht bekannter Häufigkeit (REPEAT-Anweisung)

In vielen Anwendungen ist die Anzahl der erforderlichen Zyklen nicht bekannt. Eine solche Situation kann auch für die Aufgaben zu den Programmen KOSTEN, PLANUNG, GEBUEHREN, FAHRGELD und auch SCHLUESSEL formuliert werden. Es ist besser, dem Buchhalter, Kassierer oder Betriebswirtschaftler die wiederholte Berechnung der Kosten, der optimalen Losgrößen, der Gebühren usw. zu ermöglichen, ohne daß immer wieder ein Neustart des Programms erforderlich ist. Natürlich muß man sich dann auf eine Endebedingung festlegen. Bei den Kosten ist das durch die Eingabe einer Null möglich.
Ein Sachverhalt, bei dem die Anzahl der Wiederholungen ebenfalls vor dem Eintritt in den Zyklus nicht bekannt sein kann, ist die Inventur. Die körperliche Aufnahme der Lagerbestände geschieht in der Regel nach Menge und Preis. Ausgehend vom wertmäßigen Soll-Bestand ist eine Kontrollrechnung für den wertmäßigen Ist-Bestand erforderlich. Dabei ist eine exakte Listenführung, die Inventurliste, erforderlich. Diese Aufgaben übernimmt das nachstehende Programm INVENTUR.

```
PROGRAM Inventur;
{Unterstuetzung einer Inventur}
VAR Bezeichnung   : STRING[20];
    Menge         : INTEGER;
    Preis,Artikel,
    Ist,Soll,
    Ergebnis      : REAL;
BEGIN
  writeln('Auswertung einer Inventur');
  write('Sortiment: ');
  readln(Bezeichnung); writeln;
  write('Sollbestand: ');
  read(Soll); writeln;
  writeln(LST);
  writeln(LST,'Inventurliste fuer das Sortiment ',Bezeichnung);
  writeln(LST,'Sollbestand: ',Soll:9:2);
  writeln(LST);
  Ist := 0;
  write('Artikel: ');
  readln(Bezeichnung);
  REPEAT
    write('Menge: ');
    read(Menge); writeln;
    write('Einzelpreis: ');
    read(Preis); writeln;
    Artikel := Menge * Preis; Ist := Ist + Artikel;
    write(LST,Bezeichnung:20,Menge:8);
    writeln(LST,' a'' ',Preis:6:2,' = ',Artikel:9:2);
    write('Artikel(Ende mit <ET>): ');
    readln(Bezeichnung);
  UNTIL Bezeichnung = '';
  writeln(LST);
  writeln(LST,'Istbestand':40,Ist:10:2); writeln;
  Ergebnis := Ist - Soll;
  IF Ergebnis >= 0 THEN writeln(LST,'Inventurplusdifferenz ',
                                Ergebnis:7:2)
  ELSE writeln(LST,'Inventurminusdifferenz ',ABS(Ergebnis):7:2);
  write('Ende');
END.
```



Dieses Programm soll (nach den erforderlichen Zwischenschritten des Editierens, Compilierens und Linkens) ausgeführt werden. Es ergibt sich der folgende Anfang des Protokolls:

```
Auswertung einer Inventur
Sortiment: Schreibwaren<ET>
Sollbestand: 4249.84<ET>
Artikel: Faserschreiber<ET>
Menge: 100<ET>
Einzelpreis: 8.50<ET>
Artikel(Ende mit <ET>): Feinschreiber<ET>
Menge: 26<ET>
Einzelpreis: 30.00<ET>
Artikel(Ende mit <ET>):
.
```

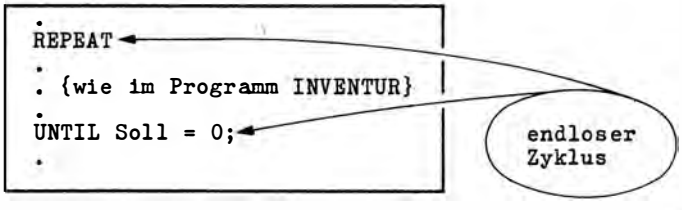
Bis zu diesem Zeitpunkt ist die folgende Inventurliste entstanden:

```
Inventurliste fuer das Sortiment Schreibwaren
Sollbestand   4249.84
      Faserschreiber      100 a   8.50 =    850.00
       Feinschreiber       26 a  30.00 =    780.00
.
```

Am Ende werden auch der Ist-Bestand und die Inventurdifferenz ausgeschrieben.
Die Aufmerksamkeit gilt der REPEAT-Anweisung. Ihre exakte Syntax ist den Syntaxdiagrammen zu entnehmen. Die Wirkung der REPEAT-Anweisung ist so, daß die zwischen REPEAT und UNTIL notierten Anweisungen zyklisch in der Reihenfolge ihrer Notation immer wieder ausgeführt werden, bis (until) die neben dem reservierten Wort UNTIL notierte Bedingungsprüfung erfüllt ist. Im Beispiel INVENTUR besteht die Endebedingung darin, daß auf die Eingabeaufforderung

"Artikel (Ende mit <ET>):"

mit der Endetaste geantwortet wird, also keine Eingabe erfolgt. Die STRING-Variable "Bezeichnung" ist dann leer. Der Vergleich mit dem Leerstring '' liefert TRUE, und die REPEAT-Anweisung wird verlassen.
Aus der Arbeitsweise der REPEAT-Anweisung folgt, daß während des Durchlaufs durch die zwischen REPEAT und UNTIL notierten Anweisungen eine Änderung des Wertes der Bedingungsprüfung möglich sein muß. Die Anweisungsfolge



würde zu einem Zyklus ohne Ende führen, wenn am Anfang ein Soll-Bestand größer als Null eingegeben wurde.
Bei der Programmierung eines Zyklus mit REPEAT/UNTIL ist darauf zu achten, daß die Anweisung, die die Endebedingung beeinflußt, möglichst unmittelbar vor UNTIL steht. Im Beispiel ist das die Anweisung "readln(Bezeichnung)". Um das zu erreichen, wurden sogar die Eingabeaufforderung und der Lesebefehl doppelt notiert, und zwar einmal unmittelbar vor dem REPEAT und dann noch einmal (mit geringer Modifikation der Textausschrift) kurz vor dem UNTIL.
Die folgende "Einsparung" führt zu einem Fehler:

```
.
Ist := 0;
REPEAT
  write('Artikel(Ende mit <ET>): ');
  readln(Bezeichnung);
  write('Menge: ');
  read(Menge); writeln;
.
. {wie im Programm INVENTUR}
.
  writeln(LST,' a'' ',Preis:6:2,' = ',Artikel:9:2);
UNTIL Bezeichnung = '';
.
```

Fehler: Abbruch muss vor UNTIL

Nach der Eingabe von < ET > folgt natürlich die Ausschrift "Menge:", und danach wird erneut eine Eingabe gefordert.
Es war aber der Abbruch beabsichtigt. Bei der Gestaltung eines solchen Zyklus kann man also Fehler machen.
Die Analyse der anderen Befehle des Programms INVENTUR ist eine Wiederholung bereits behandelter Probleme.
Wichtig ist noch, daß sich die REPEAT-Anweisung besonders eignet zur Gestaltung einer nutzerfreundlichen und sicheren Eingabe in interaktiven Programmen.
Der Rahmen dafür ist folgender:

```
.
REPEAT
.
. {Eingabe}
.
UNTIL <Eingabe> = <Richtig>;
.
```

< Eingabe > und < Richtig > wären im konkreten Falle zu spezifizieren. So kann es erforderlich sein, nach einer besonders wichtigen, aber fehleranfälligen Eingabe den Eingabewert zur Kontrolle noch einmal auf dem Bildschirm anzuzeigen und eine Bestätigung mit "J" (für "ja") oder "N" (für "nein") zu verlangen. Nur diese beiden Aussagen sind zugelassen.

Eine programmtechnische Lösung dafür ist:

```
.
VAR x       : CHAR;
    Richtig : BOOLEAN;
BEGIN
  REPEAT
    write('Bestaetigen Sie (J/N): ');
    read(x);
    x := upcase(x);
    Richtig := (x = 'J') OR (x = 'N');
    IF NOT Richtig THEN write(chr(7));
  UNTIL Richtig;
.
```

Der Nutzer kann diesen Zyklus erst verlassen, wenn er sich klar für Ja oder Nein entschieden hat. Die vordefinierte Funktion UPCASE setzt das Bit 5 zurück und sorgt dafür, daß Kleinbuchstaben zu Großbuchstaben werden. UPCASE ist nur wirksam, wenn wirklich ein Kleinbuchstabe gespeichert ist.
Abschließend soll noch darauf hingewiesen werden, daß – wie die Notation der Bedingungsprüfung hinter UNTIL schon ausdrückt – die Anweisungen nach REPEAT erst ausgeführt werden und dann die Prüfung am Ende erfolgt. Die Anweisungsfolge wird damit wenigstens einmal durchlaufen. Sie ist eine offene Iteration. Man nennt REPEAT deshalb auch eine nichtabweisende Schleife.

4.4. Geschlossene Iteration nicht bekannter Häufigkeit (WHILE-Anweisung)

Es gibt eine Reihe von Problemen, bei deren Realisierung ein Zyklus überhaupt nicht durchlaufen werden darf, wenn eine bestimmte Bedingung nicht gegeben ist. Natürlich läßt sich das programmtechnisch realisieren, indem die REPEAT-Anweisung als Zweig einer IF-Anweisung geschrieben wird. In einigen PASCAL-Implementationen sind dadurch sogar geringfügige Laufzeitvorteile zu erzielen. Die günstigere Anweisung dafür ist jedoch die WHILE-Anweisung. Zu ihrer Demonstration wird das folgende interaktive Programm GELD benutzt. Es ermöglicht die Eingabe eines Geldbetrages in Mark. Ermittelt und auf dem Bildschirm angezeigt wird die Stückelung eines Betrages in 100, 50, 20, 10, 5, 2 und 1 Mark, die die Anzahl der Stücke minimal hält:

```
PROGRAM Geld;
{Stueckelung eines ganzzahligen Geldbetrages}
VAR Geldbetrag,
    Einheit,Stueck : INTEGER;
    Ende           : STRING[4];
BEGIN
  writeln('Stueckelung eines Geldbetrages(ohne Pfennige)');
  REPEAT
    write('Geldbetrag: ');
    read(Geldbetrag); writeln;
```

```
    IF Geldbetrag > 0 THEN BEGIN
      Einheit := 100;
      writeln('Die Stueckelung betraegt'); writeln;
      WHILE Geldbetrag > 0 DO BEGIN
        Stueck := Geldbetrag DIV Einheit;
        IF Stueck > 0 THEN writeln(Stueck:5,' a',' ',Einheit:4,' Mark');
        Geldbetrag := Geldbetrag MOD Einheit;
        Einheit := Einheit DIV 2;
        IF Einheit = 25 THEN Einheit := 20
      END;
    END
    ELSE writeln('Fehlerhafte Eingabe');
    writeln;
    writeln('Weiter mit <ET>');
    readln(Ende)
  UNTIL Ende <> '';
  write('Ende')
END.
```

WHILE-Anweisung

DO-Bereich

Die WHILE-Anweisung ist tief geschachtelt als Zweig einer IF-Anweisung innerhalb einer REPEAT-Anweisung notiert. Ihre Wirkung gleicht grundsätzlich der der REPEAT-Anweisung. Allerdings steht die Bedingungsprüfung (auch optisch) am Anfang des Zyklus, und die Anweisungen innerhalb des DO-Bereiches werden nur ausgeführt, solange (while) sie erfüllt ist – gegebenenfalls überhaupt nicht. Das "TRUE" für die Bedingungsprüfung in REPEAT und WHILE ist in der Wirkung grundverschieden. TRUE führt in REPEAT zum Abbruch der Schleife, in WHILE zu ihrer Durchführung. Die Ausführung des Programms GELD zeigt das folgende Protokoll, wenn ein Geldbetrag größer Null eingegeben wurde:

```
Stueckelung eines Geldbetrages(ohne Pfennige)
Geldbetrag: 1493<ET>
Die Stueckelung betraegt

   14 a  100 Mark
    1 a   50 Mark
    2 a   20 Mark
    1 a    2 Mark
    1 a    1 Mark

Weiter mit <ET>
nein<ET>
Ende
```

Im Programm GELD wird ein Geldbetrag kleiner oder gleich Null bereits vor der WHILE-Anweisung abgefangen. Aber auch ohne IF-Anweisung würde sie bei Geldbetrag = 0 nicht durchlaufen. Hier dient die WHILE-Anweisung zur Beendigung des Zyklus, wenn der eingegebene Betrag bereits vollständig gestückelt ist. Im übrigen gilt alles, was über die Veränderung der Abbruchbedingung im Zyklus und ihre Stellung im Zyklus als letzte Anweisung geschrieben wurde.

FORTRAN- und BASIC-Programmierer, die nur FOR-ähnliche Zyklen kennen, müssen beachten, daß in REPEAT- und WHILE-Anweisungen keine "sich selbst" erhöhende Laufvariable zur Verfügung steht. Das ist nur bei der FOR-Anweisung der Fall. In PASCAL gibt es zusätzlich die vordefinierten Funktionen SUCC (Nachfolger) und PRED (Vorgänger), die das einfach und schnell ermöglichen.

Schließlich sei noch darauf hingewiesen, daß zwischen REPEAT und UNTIL beliebig viele

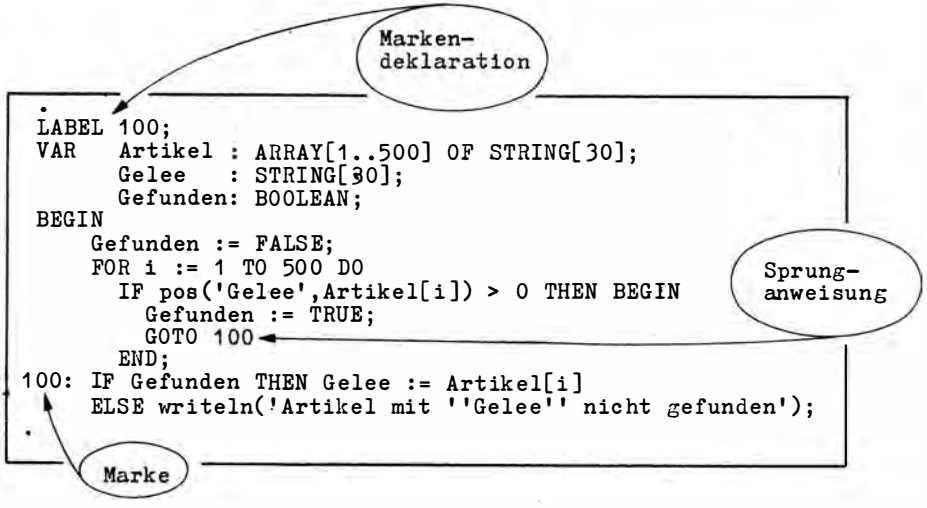
Anweisungen notiert werden können. Die Schlüsselworte haben zugleich eine Verbundfunktion. Hinter DO in der WHILE-Anweisung darf dagegen nur eine Anweisung stehen. Gegebenenfalls ist, wie im Programm GELD, eine Verbundanweisung zu benutzen.

4.5. Vorzeitiges Beenden der Iteration im Ausnahmefall (GOTO-Anweisung)

Es gibt Situationen, in denen eine Iteration vorzeitig verlassen werden muß: Die vorgesehene Anzahl von Durchläufen bei der FOR-Anweisung ist noch nicht erreicht, oder die Bedingungsprüfung der REPEAT- oder WHILE-Anweisung sieht noch keine Beendigung des Zyklus vor. Dennoch könnte beispielsweise ein Fehler in den Daten (zum Beispiel eine Null für eine Variable, wenn diese im Nenner steht) den vorzeitigen Abbruch erfordern. Natürlich kann man jede dieser Sonderbedingungen in die Bedingungsprüfungen von REPEAT- oder WHILE-Anweisungen aufnehmen. Man sollte sich auch darum bemühen. Aber dabei wird die Befehlsstruktur immer tiefer geschachtelt, und das kann zu einem nicht vertretbaren Aufwand bei der Notation und auch zur Unübersichtlichkeit führen. Es geht vom Problem her keineswegs nur um Fehlerkontrollen, sondern auch um die verarbeitungsintensiven Suchprozesse. In einem Feld

ARRAY [1..x] OF STRING [30],

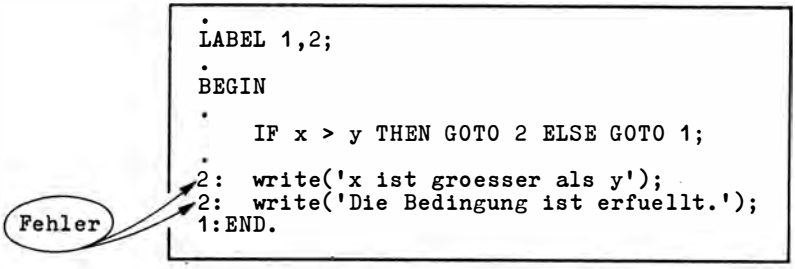
das die Artikelbezeichnungen eines Sortiments enthält, soll ein Artikel gesucht werden, der den Namensbestandteil "Gelee" hat. Um die Suche zu beschleunigen, ist die laufzeitgünstige FOR-Anweisung zu benutzen. Der folgende Programmausschnitt zeigt die Lösung unter Verwendung einer Marke:




```
.
LABEL 100;
VAR   Artikel : ARRAY[1..500] OF STRING[30];
      Gelee   : STRING[30];
      Gefunden: BOOLEAN;
BEGIN
    Gefunden := FALSE;
    FOR i := 1 TO 500 DO
      IF pos('Gelee',Artikel[i]) > 0 THEN BEGIN
        Gefunden := TRUE;
        GOTO 100
      END;
100: IF Gefunden THEN Gelee := Artikel[i]
    ELSE writeln('Artikel mit ''Gelee'' nicht gefunden');
.
```

Wie bereits im Abschnitt 1.1. im Zusammenhang mit der Feinstruktur von PASCAL-Programmen dargelegt, dienen Marken der besonderen Kennzeichnung von Anweisungen. Das geschieht, um sie von jeder anderen Stelle des Anweisungsteils eines Programms un-

abhängig von der notierten Reihenfolge der Anweisungen erreichen zu können. Mit der Anweisung "GOTO 100" wird die normale Anweisungsfolge verlassen und bei der Anweisung fortgesetzt, der die Marke 100 vorangestellt ist. Danach gilt wieder die übliche Reihenfolge, es sei denn, es kommt erneut eine GOTO-Anweisung.
Man sieht, daß Marken im Vereinbarungsteil eines Programms mit dem Schlüsselwort LABEL zu deklarieren sind. Syntaktisch gesehen, können in einem Programm beliebig viele Marken verwendet werden. Sie sind dann in der LABEL-Deklaration durch Komma zu trennen. Die exakte Syntax ist dem Syntaxdiagramm zu entnehmen. Zu beachten im Programmausschnitt ist auch die vordefinierte Stringfunktion POS. Wird 'Gelee' im Artikel [i] gefunden, liefert POS einen Wert größer Null.
Der folgende Programmausschnitt zeigt die fehlerhafte Verwendung von Marken:




```
.
LABEL 1,2;
.
BEGIN
.
    IF x > y THEN GOTO 2 ELSE GOTO 1;
.
2:  write('x ist groesser als y');
2:  write('Die Bedingung ist erfuellt.');
1:END.
```

Natürlich können niemals zwei Anweisungen dieselbe Marke tragen, denn dann ist die GOTO-Anweisung nicht eindeutig. Die Compiler zeigen diesen Fehler als Syntaxfehler an. Der Programmausschnitt enthält aber eine unter Umständen nützliche Markenanwendung, nämlich einen Sprung an das Programmende "GOTO 1".
Die Notation der Markierung "1:END." erscheint dabei etwas ungewöhnlich, denn eigentlich werden Anweisungen markiert, und END ist keine Anweisung. Trotzdem ist sie optisch günstig. Sie ist auch zulässig, denn PASCAL erlaubt sogenannte Leeranweisungen, und deshalb gilt hier nicht END, sondern die Leeranweisung vor END als markiert. Sichtbar machen läßt sich das mit der (ungünstigen) Darstellung "1:;END".
Optisch noch günstiger ist diese Schreibweise, wenn die PASCAL-Implementation auch Buchstaben als Marken zuläßt. Dann ist folgendes möglich:

```
.
LABEL Stop;
.
BEGIN
.
  IF x = 'Ende' THEN GOTO Stop;
.
Stop : END.
```

Allerdings gibt es in den verschiedenen PASCAL-Implementationen für das vorzeitige Beenden spezielle vordefinierte Prozeduren.

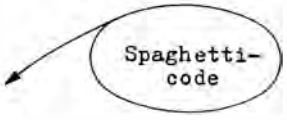
Das sind

HALT für den Programmabbruch mit Rückkehr in das Betriebssystem (wurde im Programm FAHRGELD angewendet) und

EXIT für das Verlassen der aktuellen Programmeinheit und die Rückkehr in die übergeordnete Programmeinheit. EXIT in einem Hauptprogramm wirkt wie HALT.

GOTO-Anweisungen führen dazu, daß jede Anweisung im Programm von jeder Stelle aus auf direktem Wege erreicht werden kann. In Programmiersprachen mit weniger leistungsfähigen Iterationsanweisungen (FORTRAN, BASIC) ist die GOTO-Anweisung ein unerläßliches Instrument zur Steuerung der Ablauffolge.
Sie birgt aber in hohem Maße, besonders für den nicht geübten Programmierer, die Gefahr einer unübersichtlichen, fehleranfälligen Gesamtstruktur des Programms in sich. Programmabschnitte sollen stets in der Ablauffolge nur einen Ein- und einen Austrittspunkt haben, auch um Nebenwirkungen bei Programmänderungen sofort zu erkennen. PASCAL ermöglicht, durch seine Steuerstrukturen FOR, REPEAT und WHILE grundsätzlich ohne GOTO-Anweisungen auszukommen. Ist der Programmierer die strukturierte Herangehensweise gewöhnt – und diese beginnt bei größeren Programmen mit dem Entwurf –, so wird er im allgemeinen auf GOTO verzichten. In der Softwaretechnologie werden GOTO-freie, zumindest GOTO-arme Programme als ein Qualitätsmerkmal angesehen. Eigentlich geht es aber nicht um das GOTO, sondern um die Programmstruktur. Auch mit GOTO sind gut strukturierte Programme möglich, aber ohne GOTO werden sie erzwungen. Man muß auch die Programmiersprache beachten, denn eine Forderung, mit FORTRAN oder BASIC GOTO-arm zu programmieren, ist nicht realisierbar.
Die Forderung nach strukturierten Programmen ist keineswegs nur theoretisch. Es sind Fälle bekannt, in denen wenig strukturierte, unübersichtliche Programme zu erheblichen Kosten geführt haben. PASCAL garantiert natürlich keine strukturierten Programme. Der GOTO-Befehl ist anwendbar wie in FORTRAN oder BASIC. Die Gültigkeit von Marken und damit GOTO ist allerdings meistens auf den aktuellen Block beschränkt. Das Programm SPAGHETTI (siehe Seite 73) ist wie das Programm GEBUEHREN ausführbar. Es arbeitet mit demselben Ergebnis, aber unübersichtlicher Struktur.
Solche Strukturen nennt man scherzhaft Spaghetti-Code. Das Durcheinander ist hier besonders sichtbar, weil es auf kleinstem Raum angeordnet ist. Größeren Programmen sieht man die Spaghetti-Struktur nicht so deutlich an – die Wirkung beim Testen und später beim Ändern ist aber ebenso. Natürlich ist das Programm GEBUEHREN dem Programm SPAGHETTI vorzuziehen.

```
PROGRAM Spaghetti;
{Falsche Verwendung von "GOTO" fuer Programm GEBUEHREN}
LABEL 1,2,3,4,5;
VAR   d   : INTEGER;
      z,a : CHAR;
      g   : REAL;
      e   : BOOLEAN;
BEGIN
  writeln('Berechnung von Telefongebuehren');
  write('Dauer des Gespraechs in Minuten: ');
  read(d); writeln;
  write('Zone (1/2/3): ');
  read(z); writeln;
  write('Ermaessigung (J/N): ');
  read(a); writeln;
  e := (a = 'J') OR (a = 'j');
  IF z = '1' THEN GOTO 1;
  IF z = '2' THEN GOTO 2;
  g := 0.90;
4:g := g * d;
  GOTO 3;
1:g := 0.15;
  GOTO 4;
2:g := 0.45;
  GOTO 4;
3:IF NOT e THEN GOTO 5;
  g := g * 2 / 3;
5:writeln('Die Gebuehr betraegt ',g:5:2,' Mark');
  write('Ende');
END.
```



Übungsaufgaben:

1. Entwickeln Sie ein interaktives Programm, das die Eingabe von drei Ortsnamen als Zielorte einer Rundreise anfordert! Die verschiedenen Möglichkeiten für die Rundreise sind zu ermitteln und anzuzeigen (das sind 3! = 6 Anordnungen bei verschiedenen Elementen).

2. Entwickeln Sie ein interaktives Programm, das nach Eingabe der Materialart M (max. 10 Zeichen), des optimalen Lieferzyklus L (max. 90 Tage), des durchschnittlichen Tagesverbrauchs T (max. 800 Stück), des Sicherheitsvorrats S (max. 30000 Stück) und des Durchlaufvorrats D (max. 10000) die Materialvorratsnormen V für verschiedene Materialarten ermittelt und anzeigt! Die Berechnung erfolgt wiederholt nach der Formel

   $V = L * T * 0.5 + S + D,$

   bis der Nutzer sie mit <ET> statt einer Eingabe abbricht.

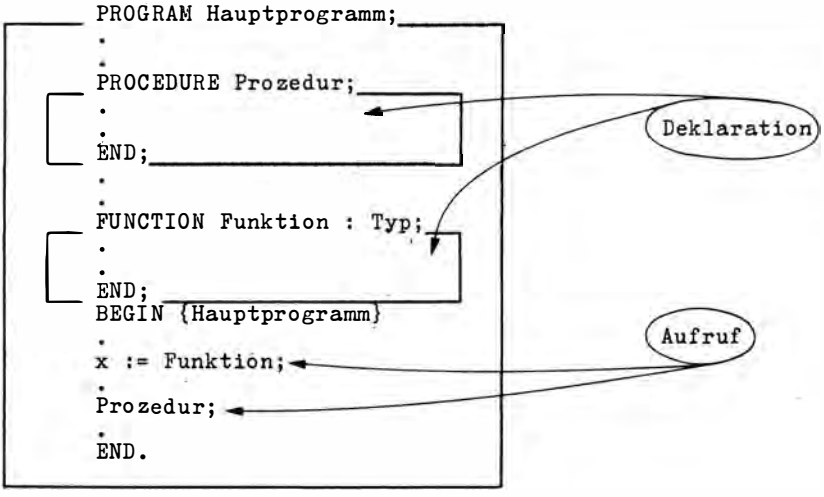
3. Ändern Sie das Programm PLANUNG aus dem Abschnitt 2.1. so, daß die Berechnung der optimalen Losgröße für beliebig viele Produkte (auch mit verschiedenen Rüstkosten) erfolgen kann!

5. Entwicklung und Nutzung von Unterprogrammen

5.1. Klassifikation und Schnittstellen

Auch bei Einhaltung aller Regeln für gute Programmstrukturen geht die Übersicht verloren, wenn das Programm mehr als 60 bis 80 Befehlszeilen umfaßt. Gleichzeitig nehmen Befehlsfolgen zu, die eigentlich nur andere Variablen benutzen, sonst aber bereits programmierten Abläufen ähnlich sind. Will man in solchen Fällen das GOTO vermeiden, wäre man nach den bisherigen Darlegungen gezwungen, diese Abläufe immer wieder zu programmieren. Einen Ausweg aus dieser Situation stellen Unterprogramme dar.
Unterprogramme sind Befehlsfolgen, die unter einem Namen zusammengefaßt werden und unter diesem geschlossen zur Ausführung aufgerufen werden können.
Die Zusammenfassung von Befehlsfolgen unter einem Namen, die Deklaration eines Unterprogramms, geschieht im Vereinbarungsteil eines Programms als PROCEDURE oder FUNCTION. Die Aktivierung (der Aufruf) erfolgt vom Anweisungsteil aus und ist beliebig oft möglich. Da Unterprogramme auch in Unterprogrammen deklariert werden können, spricht man bei der Deklaration statt von "Programm" besser von "übergeordneter Programmeinheit". Auch können Unterprogramme von anderen Unterprogrammen aufgerufen werden. Deshalb ist es exakter, sie beim Aufruf rufende und gerufene Programmeinheiten zu nennen. Die allgemeine Vorgehensweise ist auf Seite 75 oben gezeigt.

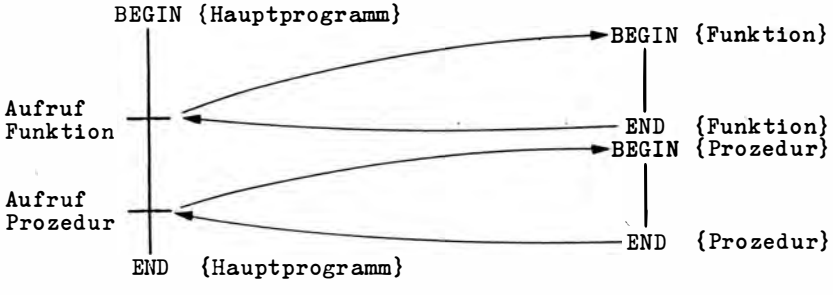
Unterprogramme werden also im Vereinbarungsteil einer übergeordneten Programmeinheit deklariert. Sie selbst bestehen aus einem Kopf, der statt mit PROGRAM in diesem Falle mit PROCEDURE oder FUNCTION beginnt, und einem Block. Der Block besteht aus dem (lokalen) Vereinbarungsteil des Unterprogramms und einer Verbundanweisung. Nach der Verbundanweisung wird nicht wie im Hauptprogramm ein Punkt, sondern ein Semikolon notiert. In den Block eines Unterprogramms können erneut Unterprogramme eingelagert werden.
Ein Unterprogramm kann von dem gesamten Block aus aufgerufen werden, in dessen Vereinbarungsteil es vorher deklariert wurde. Im Programmrahmen HAUPTPROGRAMM geschieht das vom Anweisungsteil des Hauptprogramms aus. Aber das Unterprogramm FUNKTION könnte intern ebenfalls die Leistung des Unterprogramms PROZEDUR in Anspruch nehmen, da es im gleichen Block geschieht. Umgekehrt ist es auch möglich, daß PROZEDUR das Unterprogramm FUNKTION intern nutzt. Allerdings gibt es dabei eine Schwierigkeit. Zum Zeitpunkt der Benutzung des Bezeichners FUNKTION in PROZEDUR



wäre FUNKTION noch nicht deklariert, weil es erst nach PROZEDUR notiert wurde. PASCAL verlangt in einem solchen Falle, getreu dem Prinzip der Verwendung vorher definierter Namen, daß der Kopf des zweiten Unterprogramms vor dem ersten Unterprogramm mit dem Zusatz FORWARD nocheinmal aufgeführt wird, also

```
.
FUNCTION Funktion : Typ; FORWARD;
.
```

FORWARD steht für den Block des Unterprogramms. Später muß dieses Unterprogramm trotzdem vollständig deklariert werden.
Enthielte das Unterprogramm PROZEDUR auch ein Unterprogramm, so wäre dieses nicht aufrufbar vom Unterprogramm FUNKTION und auch nicht vom Hauptprogramm. Der Vereinbarungsteil von PROZEDUR und damit auch das dort eingelagerte Unterprogramm gelten lokal als eigener, geschlossener Block.
Wichtig für das Verständnis der Unterprogrammtechnik ist es zu wissen, daß nach der Ausführung der Befehlsfolge des Unterprogramms zu der Stelle zurückgekehrt wird, von der der Aufruf erfolgte. Im Beispiel läßt sich das so demonstrieren:



Unterprogramme sind in folgenden Richtungen von grundlegender Bedeutung:

a) Unterprogramme dienen der Zerlegung großer Programme in Moduln. Das ermöglicht in vielen Fällen überhaupt erst eine transparente Programmstruktur und senkt den Testaufwand. Beim Test gilt, daß alle Programmzweige zu durchlaufen sind. Bei einem Programm mit nur 10 Bedingungsprüfungen ergeben sich $2^{10} = 1024$ Testläufe. Wird aus diesem Programm ein Modul mit 5 Bedingungsprüfungen ausgegliedert, so sind nur $2^5 + 2^5 = 64$ Testläufe erforderlich. Entweder werden wirklich weniger Testläufe durchgeführt oder der Grad der Testung erhöht sich.
b) Unterprogrammtechnik ermöglicht die Übernahme bereits früher programmierter Lösungen zu Teilproblemen in den eigenen Quelltext. Jede PASCAL-Implementation besitzt eine Vielzahl vordefinierter Unterprogramme. Zu ihnen gehören übrigens auch die bereits benutzten WRITE, WRITELN, READ, READLN, GOTOXY und UPCASE. Außerdem gibt es PASCAL-Anwender-Bibliotheken und auch PASCAL-Literatur. Viele wichtige Abläufe, wie zum Beispiel Sortieren, Suchen, statistische Berechnungen, lineare Optimierungen, sind bereits programmiert, müssen also keinesfalls immer neu erfunden werden.
c) Unterprogrammtechnik ermöglicht Arbeitsteilung zwischen mehreren Programmierern. Man legt die Schnittstellen zwischen den Programmeinheiten fest und kann danach die einzelnen Teilprobleme gesondert lösen.

Mit Hilfe der Unterprogrammtechnik ist es außerdem möglich, eine Überlagerungsstruktur des späteren Programms zu erzeugen, so daß sich nur solche Programmeinheiten im Hauptspeicher befinden, die wirklich aktiv sein müssen. Damit können auch Programme erzeugt werden, die größer sind als die Kapazität des gerade auf Mikrorechnern begrenzten Hauptspeichers. Schließlich ermöglicht die Unterprogrammtechnik die Einbindung von "sprachfremden" Moduln. Die Entbindung ist je nach PASCAL-Implementation möglich für verschieblichen Objektcode oder Maschinencode.

Ein wichtiges Problem der Arbeit mit Unterprogrammen entsteht dadurch, daß in den meisten Fällen ein Datenaustausch zwischen den einzelnen Programmeinheiten erforderlich ist. Dieser Datenaustausch ist auf drei Wegen möglich:

1. durch die Gültigkeit der in einer übergeordneten Programmeinheit definierten Daten, das sogenannte Blockkonzept;
2. durch Parameterlisten, die aus der Sicht des Unterprogramms formale Parameter sind und zum Zeitpunkt des Aufrufs durch aktuelle Parameter der rufenden Programmeinheit ersetzt werden;
3. über den Namen von Funktionen, die im Gegensatz zu Namen von Prozeduren selbst Träger von Information für die rufende Progammeinheit sein können.

Implementationsabhängig gibt es außerdem die Möglichkeit, über die Festlegung absoluter Adressen Speicherplatz zu vereinbaren, auf den von verschiedenen Programmeinheiten gleichermaßen Bezug genommen werden kann. Schließlich kann zum Datenaustausch mit fremden Maschinencodeprogrammen der Stack (Kellerspeicher) der CPU (Central Processing Unit) benutzt werden.

Blockkonzept

PASCAL ist eine blockorientierte Sprache. Blockorientiert bedeutet, daß alle definierten und deklarierten Objekte, also Konstanten, Typen, Variablen und Unterprogramme, in dem gesamten Block gültig sind, in dem sie vereinbart (eingeführt) wurden. Eine Aus-

nahme bilden lediglich Marken. In den eingelagerten Funktionen und Prozeduren können also alle Objekte ohne eigene Deklaration benutzt werden, die im übergeordneten Block enthalten sind. Solche Objekte nennt man deshalb global.
Eine Kollision würde entstehen, wenn im Vereinbarungsteil eines Unterprogramms ein Objekt unter einem Bezeichner deklariert wird, der in der übergeordneten Programmeinheit bereits benutzt wurde. Entsprechend dem Blockkonzept wäre er auch im Unterprogramm noch gültig. PASCAL legt fest, daß in diesem Falle die (lokale) Deklaration im Unterprogramm die globale Gültigkeit aufhebt, natürlich nur lokal für den Block dieses Unterprogramms und auch genau nur für dieses Objekt.
Man beachte, daß sich die "Gültigkeitstür" entsprechend dem Blockkonzept nur "nach innen", also vom Globalen zum Lokalen hin öffnet. Die in einem Unterprogramm definierten und deklarierten Objekte – bei Variablen spricht man von lokalen Variablen – sind für die übergeordnete Programmeinheit nicht gültig, und es kann auch nicht auf sie zugegriffen werden. Das Blockkonzept geht aus folgenden Programmausschnitten hervor:

```
PROGRAM Hauptprogramm;
.
TYPE Feld = ARRAY[1..20] OF CHAR;
VAR  x,y  : Feld;
     j,i,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 Gültigkeit der globalen Variablen ist als Kommentar eingefügt. Es wurde schon im Zusammenhang mit der FOR-Anweisung erwähnt, daß Laufvariablen lokale Variablen sein müssen.
Das Blockkonzept regelt nicht nur die Gültigkeit von Bezeichnern innerhalb des Programmtextes. Es legt auch fest, daß lokale Variablen beim Verlassen eines Unterprogramms ihren Wert verlieren. Ausnahmen davon müssen gesondert mitgeteilt werden (vgl. Abschnitt 5.3.).

Parameterlisten

Hinter dem Namen einer Prozedur oder Funktion kann eine in Klammern geschriebene Liste von Parametern angegeben werden, zum Beispiel

```
.
PROCEDURE Test(A : CHAR; B,C : INTEGER; VAR D : REAL);
.
```

Die exakte Syntax ist der Definition für "Prozedurkopf" im Syntaxdiagramm zu entnehmen. Die einzelnen Parameter der Liste werden durch Semikolon getrennt. Die Parameter "A : CHAR" UND "B,C : INTEGER" bestehen jeweils aus einem oder mehreren Variablenbezeichnern und einem Typbezeichner. Der Typbezeichner kann über TYPE vordefiniert sein. Der Parameter "VAR D : REAL" hat außerdem den Zusatz VAR. Alle Parameter im Prozedurkopf heißen formale Parameter, weil mit ihrer Hilfe im Unterprogramm die Verarbeitung übergebener aktueller Parameter festgelegt wird.
Im Aufruf der Prozedur TEST heißt es dann im Anweisungsteil der rufenden Programmeinheit zum Beispiel

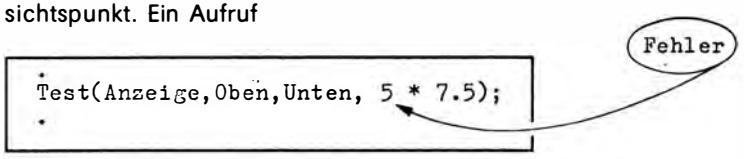
```
.
Test(Anzeige,Oben,Unten,Ergebnis);
.
```

Die aktuellen Parameter "Anzeige", "Oben", "Unten" und "Ergebnis" sind gültige Objekte der rufenden Programmeinheit. Sie müssen hinsichtlich Anzahl und Typ mit der formalen Definition übereinstimmen. Der nächste Aufruf des Unterprogramms kann mit ganz anderen aktuellen Parametern erfolgen. Aber es müssen vier sein, und sie müssen im Typ und in der Zuordnung von links nach rechts übereinstimmen. Man sieht, daß die aktuellen Parameter beim Aufruf des Unterprogramms an die Stelle der formalen Parameter treten. Mit den formalen Parametern werden also gar keine Operationen durchgeführt. Sie dienen nur der formalen Beschreibung der Operationen innerhalb des Unterprogramms, so, wie man in der Mathematik formal $y = f(x)$ definiert, die Operation selbst aber mit den jeweils aktuellen Werten durchführt.
Beim Ersetzen der formalen durch aktuelle Parameter zum Zeitpunkt des Aufrufs werden zwei verschiedene Techniken angewandt. Der Programmierer wählt zwischen ihnen, indem er das Schlüsselwort VAR vor dem jeweiligen formalen Parameter verwendet oder nicht.
Wird VAR nicht verwendet, spricht man bei formalen Parametern von Wertparametern. Im Beispiel TEST sind A, B, C Wertparameter. Die aktuellen Werte der rufenden Programmeinheit werden tatsächlich auf den für den formalen Parameter reservierten Speicherplatz "umgeschaufelt". Der Aufruf heißt dann Aufruf über Wert (Call by value). Der Vorteil eines solchen Aufrufs besteht darin, daß Operationen des Unterprogramms die Daten der rufenden Programmeinheit nicht verändern können. Ein Befehl

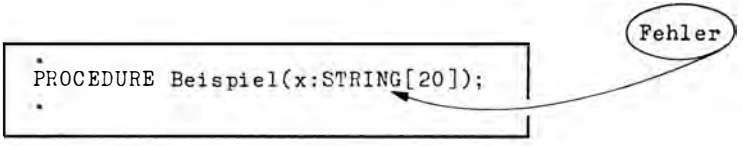
"B := B + 1;"

im Unterprogramm TEST verändert den Inhalt der Speicherzelle des entsprechenden aktuellen Parameters "Oben" nicht. Man schützt sich mit Wertparametern vor sogenannten Seiteneffekten, Veränderungen von Speicherinhalten. Der Nachteil allerdings ist erhöhter (doppelter) Speicherplatzbedarf und Zeitverzug durch Umspeichern. Moderne Implementationen speichern aber auch nur einfache Variablen wirklich um.
Wird VAR in der Parameterliste des Unterprogramms verwendet, so spricht man von Variablenparametern und dann vom Aufruf über Name (Call by reference). In diesem Falle wird eine Adresse an das Unterprogramm übergeben, und dieses arbeitet mit den realen Speicherplätzen des rufenden Programms. Im Beispiel TEST ist D ein Variablenparameter. Der Vorteil liegt auf der Hand: Speicherplatz und Zeit werden gespart. Der Nachteil

ist die erhöhte Gefahr unbeabsichtigter Seiteneffekte. Es gibt noch einen weiteren Gesichtspunkt. Ein Aufruf



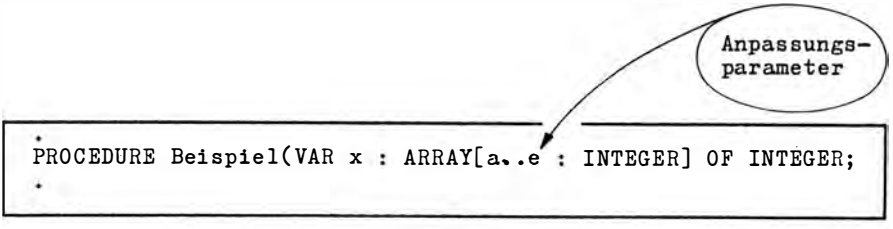
wäre offenbar falsch. Konstanten oder Ausdrücke haben nur bei Wertparametern einen programmtechnischen Sinn. Die Konstante oder der Wert des ausgerechneten Ausdrucks könnte dem formalen Wertparameter zugewiesen werden – und alles Folgende ist ohne Schwierigkeiten möglich. Konstanten und Ausdrücke sind deshalb als aktuelle Parameter für den Aufruf über Wert zugelassen. Anders bei Variablenparametern. Hier wird eine Adresse erwartet. Eine Konstante oder ein Ausdruck als aktueller Parameter ist mit Variablenparametern unverträglich. Die Compiler zeigen diesen Fehler an. In der praktischen Programmierung entsteht auch das Bedürfnis, als formale Parameter Felder zu benutzen. Die meisten Implementationen erlauben im Funktions- oder Prozedurkopf nur Typbezeichner. Also nicht das "... ARRAY OF ...". Feldvariablen für gepackte Felder können in keinem Falle als formale Parameter definiert werden. Der Prozedurkopf



ist deshalb in keiner Implementation erlaubt. Die Lösung ist einfach. Man definiert die Struktur mit

```
TYPE Kette = STRING[20];
PROCEDURE Beispiel(x:Kette);
```

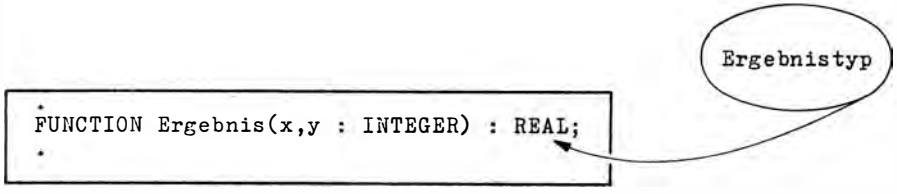
und das Problem ist gelöst. Allerdings kann das Unterprogramm nicht für einen aktuellen Parameter mit der Feldlänge 40 verwendet werden. Ein anderes Unterprogramm ist erforderlich. Beim Typ STRING kann das gegebenenfalls durch eine Compilerdirektive (V) überbrückt werden. Sicher wird es aber meistens möglich sein, auf die STRING-Unterschiede im Programm zu verzichten und dadurch trotzdem mit einem Unterprogramm auszukommen. Für Felder mit den Basistypen INTEGER und REAL halten viele PASCAL-Systeme die Möglichkeit bereit, als formale Parameter speziell definierte Felder zu verwenden, die Anpassungsfelder (Conformant array). Möglich wäre der Prozedurkopf



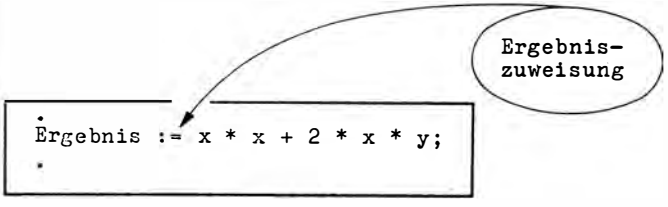
Mit "a" wird die untere Grenze, mit "e" die obere Grenze des späteren aktuellen Feldparameters in das Unterprogramm übernommen. "INTEGER" hinter dem Doppelpunkt bezeichnet den Typ der Indizes. Das können alle ordinalen Typen sein.

Funktionsbezeichner

Für den Datenaustausch zwischen rufender Programmeinheit und einem Funktionsunterprogramm gilt zunächst alles, was zu Prozedurunterprogrammen dargelegt wurde, also das Blockkonzept und der Austausch über Parameter. Zusätzlich, in der praktischen Anwendung am häufigsten genutzt, existiert aber ein weiterer Weg. Der Funktionsbezeichner selbst ist in der Lage, Daten vom Funktionsunterprogramm an die rufende Programmeinheit zu vermitteln, und zwar nur in dieser Richtung, also nicht von der rufenden Programmeinheit zum Unterprogramm. Die vermittelten Daten können unterschiedlichen Typs sein. Allerdings sind strukturierte Typen außer einem vordefinierten String nicht erlaubt. Um dem Rechner die Reservierung und Überwachung des Speicherplatzes für den Funktionsnamen zu ermöglichen, ist zusätzlich gegenüber Prozeduren eine Typdefinition erforderlich. Das geschieht hinter dem Funktionskopf:



Für den Anweisungsteil einer Funktion folgt daraus, daß dem Speicherplatz "Ergebnis" ein Wert, das Ergebnis der Operationen, zugewiesen werden muß, also



Einige Implementationen schränken den Ausdruck rechts vom Ergibtzeichen bei Zuweisungen zum Funktionsnamen ein. Dann muß zwischengespeichert werden.

Vereinbarung gemeinsamen Speicherplatzes

PASCAL ermöglicht dem Programmierer, die Adresse der Variablen im Hauptspeicher selbst festzulegen. Damit lassen sich gewisse Komponenten und Arbeitsweisen des Betriebssystems ausnutzen. Die wichtigste Anwendung ist aber die des Datenaustausches zwischen verschiedenen Einheiten eines Programms, aber auch von Programm zu Programm.

Ein Beispiel ist folgendes:

```
PROGRAM A;
VAR x : ARRAY[0..63] OF CHAR ABSOLUTE ¤8000;
.
PROCEDURE B;
VAR y : ARRAY[0..31] OF CHAR ABSOLUTE ¤8000;
    z : ARRAY[0..31] OF CHAR ABSOLUTE ¤8020;
.
END;
BEGIN
.
```

Wird y [0] in der Prozedur B verändert, so ist zugleich x [0] im Programm A betroffen, denn es wird auf denselben Speicherplatz zugegriffen. Ebenso belegen z [0] in B und x [32] in A den gleichen Speicherplatz. Die exakte Syntax ist implementationsabhängig. So kann es sein, daß Adressen in eckige Klammern eingeschlossen werden müssen.
In der praktischen Programmierung werden natürlich die verschiedenen Möglichkeiten des Datenaustausches entsprechend den Erfordernissen kombiniert.

5.2. Prozeduren und Funktionen

Unter dem Anwendungsgesichtspunkt kann man unterscheiden

a) die Nutzung vordefinierter Prozeduren und Funktionen. Der PASCAL-Compiler versteht sie direkt, und sie brauchen deshalb nicht vom Programmierer deklariert zu werden. Ihr Gültigkeitsbereich erstreckt sich auf alle Programmeinheiten. Der Programmierer muß Anzahl und Typ der aktuellen Parameter beachten. Er entnimmt sie der Systemunterlagen-Dokumentation. Vordefinierte Funktionen und Prozeduren sind im Anhang J aufgeführt. Sie sind bis auf wenige, wie WRITE, READ, SUCC, PRED, ODD und ORD, stark implementationsabhängig. Die jeweils gültigen Prozeduren sollten angekreuzt werden;
b) die Nutzung von Prozeduren und Funktionen aus Bibliotheken oder der PASCAL-Literatur. Der internationale Trend geht zunehmend zur Offenlegung von Quelltexten in PASCAL für die vielfältigsten Anwendungen;
c) die Schaffung eigener Prozeduren und Funktionen für das aktuelle Programm oder für die Bibliothek.

Programmtechnisch ist die Unterscheidung von Prozeduren und Funktionen von Bedeutung.

Prozeduren

Prozeduren werden benutzt, wenn mehr als ein Wert oder kein Wert an die rufende Programmeinheit zu übermitteln ist. Als Beispiel soll mit einem interaktiven Programm die Eingabe von Zeichenketten (zum Beispiel als Name, Vorname und Telefonnummer) in beliebiger Reihenfolge ermöglicht werden. Nach Beendigung der Eingabe soll eine alphabetisch geordnete Liste aller Eingaben (zum Beispiel das Telefonverzeichnis) gedruckt werden.

Als Bibliotheksprogramm steht die folgende Prozedur zur Verfügung:

```
PROCEDURE Ordnen(VAR Basis:Feld;Anzahl:INTEGER);
{Sortieren wie mit Blasen}
VAR Tausch  :BOOLEAN;
    Sichern :Zeile;      {je nach Typ der Sortierdaten}
    i       :INTEGER;
BEGIN
  REPEAT
    Tausch := FALSE;
    FOR i := 1 TO Anzahl - 1 DO
      IF Basis[i] > Basis[i + 1] THEN BEGIN
        Tausch       := TRUE;
        Sichern      := Basis[i];
        Basis[i]     := Basis[i + 1];
        Basis[i + 1] := Sichern
      END;
  UNTIL NOT Tausch;
END;
```

Man sieht, daß die alphabetisch zu ordnenden Eingaben als Variablenparameter zu übergeben sind. Es werden also die Variablen direkt (selbst) geordnet. Ihre tatsächliche Anzahl ist dagegen als Wertparameter vom Typ INTEGER einzusetzen. In der Prozedur ist außerdem vermerkt, daß die Variable Tausch vom Typ her Element des Feldes Basis sein muß. Bevor das Hauptprogramm betrachtet wird, soll noch festgelegt werden, daß die Prozedur ORDNEN sich als Quelltextfile unter der Bezeichnung ORDNEN.BIB auf demselben externen Datenträger, der Diskette, befindet wie der Quelltext des Hauptprogramms. Dann ist das Einfügen des Prozedurquelltextes als Include-File mit der Compilerdirektive

{¤I␣<Name>}

möglich. <Name> ist durch den konkreten Namen des Datenbestandes zu ersetzen. Die beiden Quelltexte werden bei dieser Vorgehensweise nicht physisch vereinigt. Der Text des Include-Files wird lediglich beim Kompilieren und Linken an der Stelle berücksichtigt, an der die I-Compilerdirektive steht. Der Name des Datenbestandes muß nicht mit dem Namen des Unterprogramms übereinstimmen, aber die entsprechende Programmeinheit enthalten. Diese Include-Technik wird im folgenden Hauptprogramm angewendet:

```
PROGRAM Verzeichnis;
{Alphabetische Sortierung von Text}
CONST Umfang = 300;
      Laenge =  30;
TYPE  Zeile = STRING[Laenge];
      Feld  = ARRAY[1..Umfang] OF Zeile;
VAR   Eintrag   :Feld;
      Ende      :BOOLEAN;
      Nummer,i  :INTEGER;
{¤I ORDNEN.BIB}
BEGIN
  writeln('Anlegen eines alphabetischen Verzeichnisses');
  Nummer := 1;
```

Compiler-direktive

```
  REPEAT
    write(Nummer:3,'.Eintragung(max.',Laenge,' Zeichen oder <ET>): ');
    readln(Eintrag[Nummer]);
    Ende := Eintrag[Nummer] = '';
    IF NOT Ende THEN Nummer := succ(Nummer)
    ELSE Nummer := pred(Nummer)
  UNTIL Ende;
  IF Nummer > 0 THEN BEGIN
    Ordnen(Eintrag,Nummer); writeln;
    writeln('Alphabetisches Verzeichnis'); writeln;
    FOR i := 1 TO Nummer DO
      writeln(i:3,'.',Eintrag[i]);
  END;
  write('Ende');
END.
```

Aufruf ORDNEN

Führt man dieses Programm aus und tastet dabei die Währungen einiger Länder ein, so ergibt sich das folgende Protokoll:

```
  Anlegen eines alphabetischen Verzeichnisses
    1.Eintrag(max.30 Zeichen oder <ET>): Peso<ET>
    2.Eintrag(max.30 Zeichen oder <ET>): Forint<ET>
    3.Eintrag(max.30 Zeichen oder <ET>): Rubel<ET>
    4.Eintrag(max.30 Zeichen oder <ET>): Mark<ET>
    5.Eintrag(max.30 Zeichen oder <ET>): Krone<ET>
    6.Eintrag(max.30 Zeichen oder <ET>): Zloty<ET>
    7.Eintrag(max.30 Zeichen oder <ET>): Tugrik<ET>
    8.Eintrag(max.30 Zeichen oder <ET>): Lewa<ET>
    9.Eintrag(max.30 Zeichen oder <ET>): <ET>

  Alphabetisches Verzeichnis

    1.Forint
    2.Krone
    3.Lewa
    4.Mark
    5.Peso
    6.Rubel
    7.Tugrik
    8.Zloty
  Ende
```

Man beachte, daß hier ganze Zeichenketten mit Vergleichsoperationen sortiert werden. Deshalb würde unterlassene Großschreibung am Anfang zu fatalen Folgen führen.
Inwieweit man sich beim Schreiben eines solchen Hauptprogramms auch mit dem Inhalt des Unterprogramms beschäftigen muß, ist verschieden. Wichtig ist natürlich die richtige Auswahl entsprechend dem Anwendungszweck. Hier wurde ein Sortieralgorithmus gewählt, der für wenige zu sortierende Objekte günstig ist. Er ist bekannt als "Blasensortierung", weil die sortierten Objekte im Verlaufe der Ausführung wie "Blasen" nach oben steigen.
Bei der Ausführung werden die Speicherinhalte der Komponenten i und i + 1 verglichen. Ist der Folgewert größer als der Vorgänger, werden die Inhalte vertauscht. Das geschieht, bis kein Tausch mehr notwendig ist. Den Ablauf zeigt die folgende Tabelle (die "Blasen" wurden gekennzeichnet):

Feld-index	Anfangs-belegung	Belegung nach REPEAT - Zyklus 1	2	3	4	5	6	End-belegung
1	Peso	(F)	F	F	F	F	F	Forint
2	Forint	P	(M)	(K)	K	K	K	Krone
3	Rubel	(M)	(K)	M	M	(L)	L	Lewa
4	Mark	(K)	P	P	(L)	M	M	Mark
5	Krone	R	R	(L)	P	P	P	Peso
6	Zloty	(T)	(L)	R	R	R	R	Rubel
7	Tugrik	(L)	T	T	T	T	T	Tugrik
8	Lewa	Z	Z	Z	Z	Z	Z	Zloty

Ein solcher Sortieralgorithmus benötigt im Mittel bei n Werten $n^2/2$ Operationen. Bei mehr als 100 Werten ist der dadurch entstehende Zeitverzug deutlich spürbar. Man benutzt dann einen Schnellsortieralgorithmus, der etwa $n * \log_2 n$ Operationen realisiert, allerdings für nur wenige Werte unzweckmäßig ist. Das Grundprinzip des Schnellsortierens besteht in der Teilung der Liste zu sortierender Objekte in einen Teil, der kleiner und einen Teil, der größer ist als ein Vergleichswert. Das geschieht im Wechsel, so daß mal nach oben und mal nach unten transportiert wird. Jeder der entstehenden beiden Teile wird als selbständige Liste betrachtet und erneut geteilt. Der Teilungsvorgang wird beendet, wenn die Liste genügend klein ist (meist kleiner zehn), und für diesen Teil wird mit dem einfachen Sortieralgorithmus die endgültige Reihenfolge hergestellt. Für die Effektivität der Schnellsortierung ist die Wahl des Vergleichswertes von Bedeutung. Dieser muß geschätzt werden. Einzelheiten sind in der Literatur nachzulesen.[7] Eine solche Prozedur SORTIEREN, für die es Hunderte von Varianten gibt, enthält Anlage I.

Natürlich kann es bei Bibliotheksprogrammen erforderlich sein, das Sortierkriterium oder den Typ des Sortierobjektes zu ändern. Dann müssen die veränderten Unterprogramme erneut getestet werden. Das geschieht in der Regel mit einem gesonderten Rahmenprogramm, das nur für die Testzwecke geschrieben und später nicht mehr benötigt wird.

Abschließend sei darauf verwiesen, daß das Programm VERZEICHNIS (einschließlich Bibliotheksprozedur) auch die vordefinierten Prozeduren WRITELN, READLN und WRITE nutzt.

Funktionen

Funktionen sind spezielle Prozeduren. Für eine Funktion entscheidet man sich, wenn ein Wert einfachen (nichtstrukturierten) Typs das Ergebnis bildet. Dieses Ergebnis wird dem Namen der Funktion zugewiesen. Der Vorteil besteht darin, daß im rufenden Programm der Funktionsname in Ausdrücken verwendet werden kann wie jede andere Variable; allerdings führt die Verwendung zusätzlich zum Aufruf des Unterprogramms.

7 Vgl. Bowles, K. L.: PASCAL für Mikrocomputer. Berlin – Heidelberg – New York: Springer-Verlag 1982, S. 527ff.

Zur Demonstration wird das Programm STATISTIK benutzt. Es fordert die Eingabe einer Zeitreihe und berechnet das einfache arithmetische Mittel sowie die Variationsbreite als Differenz zwischen dem kleinsten und größten Wert der Zeitreihe. Es ist kein Problem, die Berechnung auf weitere Kennziffern der Streuungsanalyse, wie Varianz, Standardabweichung, mittlere absolute Abweichung und Relationen zwischen diesen, zu erweitern. Zuerst wird das Hauptprogramm geschrieben, die Funktionen MITTEL sowie VARIATION als Include-File STREUUNG.STA werden ausgeklammert:

```
PROGRAM Statistik;
{Berechnung von Mittelwert und Variationsbreite}
CONST Anzahl    = 50;
TYPE  Richtung = (Maximum,Minimum);
VAR   Daten     : ARRAY[1..Anzahl] OF REAL;
      i,Fehler : INTEGER;
      Eingabe  : STRING[12];
{¤I STREUUNG.STA}
BEGIN {Hauptprogramm}
  writeln('Mittelwert und Variationsbreite einer Zeitreihe');
  i:= 1;
  REPEAT
    write('Eingabe ',i,'.Wert(Ende mit <ET>): ');
    readln(Eingabe);
    val(Eingabe,Daten[i],Fehler);
    IF (Fehler = 0) AND (Eingabe <> '') THEN i := succ(i)
    ELSE i := pred(i)
  UNTIL (Eingabe = '') OR (i = Anzahl);
  IF i > 0 THEN BEGIN
    writeln('Mittelwert = ',Mittel(i):14:2);
    writeln('Variationsbreite = ',
            abs(Variation(i,Maximum)-Variation(i,Minimum)):8:2);
  END;
  writeln('Ende');
END.
```

Aufruf MITTEL

Aufruf VARIATION

Das Hauptprogramm verwendet die vordefinierte Prozedur VAL. Dadurch ist es möglich, eine nutzerfreundliche Endebedingung zu programmieren. Alle Eingaben werden zunächst als Text angesehen. VAL liefert einen Fehlercode zurück, der verschieden von Null ist, wenn keine Konvertierung in eine Zahl des eingetragenen Datentyps möglich ist (hier "Daten [i]" also REAL). Bei Eingabe nur von <ET> ist der STRING "Eingabe" leer und "Fehler" größer als Null. Die Unterprogramme MITTEL und VARIATION werden aus der Parameterliste der WRITELN-Prozeduren gerufen. Dabei wechselt für VARIATION einer der aktuellen Parameter.
Das Quelltextfile STREUUNG.STA enthält die folgenden Unterprogramme:

```
FUNCTION Mittel(n:INTEGER) : REAL;
VAR j      : INTEGER;
    Summe : REAL;
BEGIN
  Summe := 0;
  FOR  j:= 1 TO n DO Summe := Summe + Daten[j];
  Mittel:= Summe / n;
END;
```

Wert-zuweisung

```
FUNCTION Variation(n:INTEGER; x:Richtung) : REAL;
VAR i     : INTEGER;
    Wert : REAL;
BEGIN
  Wert := Daten[1];
  IF  x = Maximum THEN
    FOR i:= 2 TO n DO
      IF Daten[i] > Wert THEN Wert := Daten[i] ELSE
  ELSE
    FOR i := 2 TO n DO
      IF Daten[i] < Wert THEN Wert := Daten[i];
  Variation:= Wert;
END;
```

Wert-
zuweisung

Der Start ergäbe zum Beispiel das folgende Protokoll:

```
Mittelwert und Variationsbreite einer Zeitreihe
Eingabe 1.Wert(Ende mit <ET>): 414.63<ET>
Eingabe 2.Wert(Ende mit <ET>): 934.56<ET>
Eingabe 3.Wert(Ende mit <ET>): 899.49<ET>
Eingabe 4.Wert(Ende mit <ET>): <ET>
Mittelwert =          749.56
Variationsbreite =    519.93
Ende
```

Man achte darauf, daß in beiden Funktionen dem Funktionsnamen, wie bereits in Abschnitt 5.1. erläutert, ein Wert zugewiesen wird. Geschieht das nicht, so ist die Funktion undefiniert.
Natürlich kann man mit einer Funktion auch andere Wirkungen erzeugen als nur die Ermittlung des Wertes der Funktion. Das entspricht aber nicht der PASCAL-Konzeption und wäre abzulehnen. Die Unterscheidung zwischen Prozedur und Funktion bereitet in der praktischen Programmierung keine Schwierigkeiten.

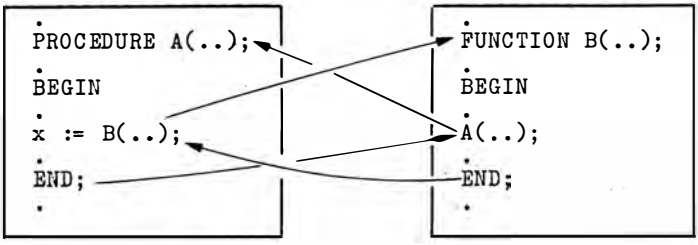
5.3. Rekursion

In Programmiersprachen mit Blockkonzept besteht auch die Möglichkeit, daß Prozeduren oder Funktionen aufgerufen werden, die zu diesem Zeitpunkt noch aktiv sind. Ein solcher Aufruf heißt rekursiv. Der einfachste Fall liegt vor, wenn sich die Prozedur oder Funktion selbst ruft. Das Grundschema für eine Prozedur wäre

```
PROCEDURE A(..);
.
BEGIN
.
A(...);
.
END;
```

rekursiver
Aufruf

Die Prozedur ist selbst noch aktiv, hat also die Steuerung noch nicht an die rufende Programmeinheit zurückgegeben; da erfolgt der erneute Aufruf. Diese Form des rekursiven Aufrufs nennt man direkte Rekursion. Es kann aber auch sein, A ruft eine Funktion (oder Prozedur) B und diese (vor ihrer Beendigung) erneut A:



Dieser Fall wird als indirekte Rekursion bezeichnet. Charakteristisch ist auch hier, daß A selbst noch nicht beendet ist und schon wieder selbst gerufen wird. Bei indirekter Rekursion entsteht das Problem der Reihenfolge in der Deklaration der Unterprogramme, weil in einem stets das andere benutzt wird, ohne vorher deklariert sein zu können. Die Lösung ist die im Abschnitt 5.1. erläuterte FORWARD-Direktive.
Der Vorgang der Rekursion ist vergleichbar mit einem Fernsehbild, das sich selbst abbildet und so als eine Folge von (immer kleiner werdenden) Bildern erscheint. In der Programmierung können durch Nutzung der Rekursion äußerst leistungsstarke Befehle notiert werden.
Zur Demonstration dieser Programmiertechnik wird nachstehend das Programm LAGERUNG notiert. Es fordert die Eingabe von Lagernummern bis zu einem Text "Ende". Danach werden die Lagernummern in umgekehrter Reihenfolge, also nach dem LAST-IN-FIRST-OUT-Prinzip (das Letzte ist das Erste) auf dem Bildschirm ausgegeben. Als Lagertechnik ist das natürlich nicht für verderbliche oder verschleißende Waren möglich.

```
PROGRAM Lagerung;
{LIFO mit rekursiver Programmiertechnik}
{¤A-}
PROCEDURE Lifo;
VAR Lagernummer: STRING[30];
BEGIN
  write('Einlagerung Nummer(oder ''Ende''): ');
  readln(Lagernummer);
  IF Lagernummer <> 'Ende' THEN Lifo;
  writeln('Auslagerung Nummer: ',Lagernummer);
END;
BEGIN{Hauptprogramm}
  writeln('LAST-IN-FIRST-OUT-Lagertechnik');
  Lifo;
  write('Ende')
END.
```

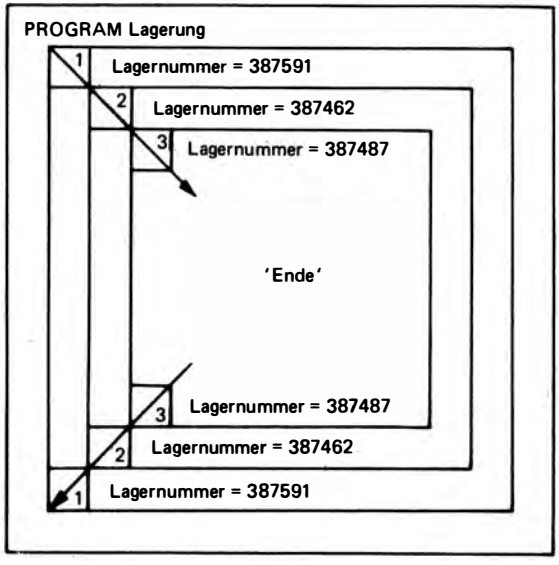
Compiler-schalter

rekursiver Aufruf

Die Arbeitsweise des Programms demonstriert das folgende Bildschirmprotokoll.

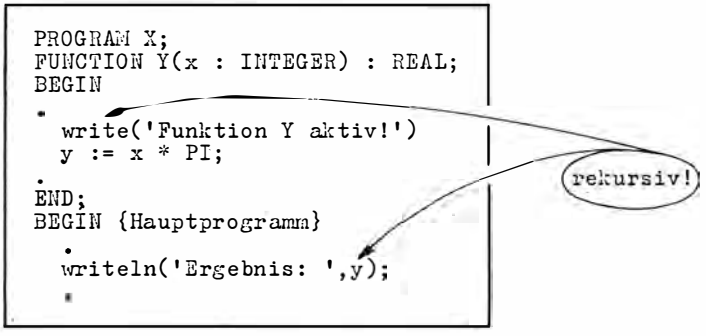
```
LAST-IN-FIRST-OUT-Lagertechnik
Einlagerung Nummer(oder 'Ende'): 387591 Kneifzange<ET>
Einlagerung Nummer(oder 'Ende'): 387462 Kombinationszange<ET>
Einlagerung Nummer(oder 'Ende'): 387487 Eckrohrzange<ET>
Einlagerung Nummer(oder 'Ende'): Ende<ET>
Auslagerung Nummer: 387487 Eckrohrzange
Auslagerung Nummer: 387462 Kombinationszange
Auslagerung Nummer: 387591 Kneifzange
Ende
```

Grundlage der rekursiven Technik ist, daß jeder Aufruf des Unterprogramms so erfolgt, als sei es der erste bzw. der Aufruf eines Unterprogramms mit anderem Namen. Jedesmal wird also Speicherplatz für lokale Variablen reserviert, und zwar bei jedem Aufruf anderer. Jedesmal werden auch Wertparameter kopiert. So hat jede lokale Variable oder jeder Wertparameter gewissermaßen einen (unsichtbaren) Aufrufindex. Die Parameter werden je Aufruf in einen "Kellerspeicher" geschoben und – wenn sich die Schachtelung durch Rückkehr auflöst – in umgekehrter Reihenfolge wieder daraus entnommen. Diesen Ablauf für das Anwendungsbeispiel zeigt das folgende Bild:



Die Schachtelung durch den wiederholten, aber doch eigenständigen Aufruf führt zur Speicherung mehrerer Datenobjekte, obwohl explizit nur für *ein* solches Objekt Speicherplatz vereinbart wurde. Es ist ersichtlich, daß das gleiche Problem durch Iteration gelöst werden kann. Die Lagernummern wären auf ein Feld zu speichern und in umgekehrter Reihenfolge wieder bereitzustellen. Jede Rekursion kann durch eine Iteration realisiert werden, wenn der Programmierer Speicherplatz und einen der Iterationsbefehle FOR, REPEAT oder WHILE benutzt. Diese Vorgehensweise bringt in den meisten Fällen sogar Zeitvorteile in der Ausführung. Rekursive Programmierung spart Quelltext, kann zu einem verringerten Editieraufwand beitragen und die Transparenz erhöhen.

Die Rekursionstechnik erfordert allerdings zusätzlichen Maschinencode, erhöht die Kompilierzeit und den Umfang des Programms. Dei meisten Compiler sehen deshalb die Rekursion nicht standardmäßig vor. Es muß eine Compilerdirektive gesetzt werden. Im Programm LAGERUNG ist das {¤A−}. Implementationsabhängig können die Direktiven auch anders sein.
Abschließend soll noch darauf hingewiesen werden, daß der rekursive Ruf selbst an eine Bedingung gebunden sein muß. Diese Bedingung muß sich im Rekursionszyklus auch ändern – sonst entsteht eine Schleife ohne Ende, und das Programm kann nur durch Kaltstart beendet werden. Um eine solche Situation zu vermeiden, sollte während des Tests eine Compilerdirektive gesetzt werden, die jederzeit den Abbruch ermöglicht. Das gilt generell, besonders aber bei der Rekursion.
Schließlich muß man beachten, daß in den meisten Implementationen die Rekursivität für vordefinierte Prozeduren und Funktionen untersagt ist. Das folgende Beispiel zeigt einen solchen Fehler:



Hier hilft auch keine Compilerdirektive. Man kann das umgehen, indem der UP-Ruf für das zweite Unterprogramm als gesonderter Befehl vorgezogen wird.

5.4. Externe Bezüge und modulare Kompilation

PASCAL-Compiler für Mikrorechner ermöglichen die Verwendung fremder Sprachelemente und ausführbarer Maschinencodes sowie die Einbindung von (externen) Maschinencodeprogrammen. Außerdem ist es möglich, Unterprogramme gesondert zu kompilieren (implementationsabhängig) und gegebenenfalls zu überlagern.

Interner Maschinencode (INLINE-Anweisung)
An beliebigen Stellen im Anweisungsteil kann zur Programmierung in Assembler, zumindest aber in Maschinencode übergegangen werden. Das geschieht mit der INLINE-Anweisung. Dazu das folgende Programm. Es ermöglicht, einen Laufwerksbezeichner A, B, C, D, E oder F einzugeben. Das Laufzeitsystem wählt dieses Laufwerk als aktuelles Laufwerk an.
Die Operation ist sichtbar, weil die Kontrolldiode eines angewählten Laufwerks aufleuchtet. Das geschieht allerdings nur, wenn das Diskettenverzeichnis vom Laufzeitsystem noch nicht gelesen wurde.

```
PROGRAM Selektion;
VAR Eingabe :STRING[4];
PROCEDURE Laufwerk(x:CHAR);
VAR Geraet  :INTEGER;
BEGIN
  Geraet := ord(x) - 65;
  INLINE( #e / #d    /       {LD   C ,13       }
          #cd/ >5    /       {CALL 5           }
          #2a/ Geraet/       {LD   HL,(Geraet)}
          #5d/               {LD   E ,L        }
          #e / #e    /       {LD   C ,14       }
          #cd/ >5    );      {CALL 5           }
END;
BEGIN{Hauptprogramm}
  REPEAT
    writeln('Laufwerksselektion und Directory lesen');
    writeln('Waehlen Sie ein vorhandenes Laufwerk (Ende mit <ET>): ');
    readln(Eingabe);
    IF Eingabe <> '' THEN BEGIN
      Eingabe[1] := upcase(Eingabe[1]);
      IF Eingabe[1] IN ['A'..'F'] THEN Laufwerk(Eingabe[1]);
    END;
  UNTIL Eingabe = '';
  write('Ende');
END.
```

CALL 5 ist der Ruf der BDOS-Komponente des Laufzeitsystems SCPX. Dafür stehen in PASCAL-Systemen gegebenenfalls auch vordefinierte Funktionen zur Verfügung (vgl. Anlage J).

Die INLINE-Anweisung sollte nur verwendet werden, wenn laufzeit- oder speicherplatzkritische Abläufe das unbedingt erfordern oder wenn keine PASCAL-Sprachelemente zur Verfügung stehen. Die Syntax der INLINE-Anweisung ist dem Syntaxdiagramm zu entnehmen. Der Schrägstrich trennt die einzelnen Elemente, auch Operationscode und Operand. Ob als Operationscode Assembler-Mnemonik (Sprachelemente der Assemblersprache) zugelassen ist oder nicht, ist implementationsabhängig. Die hexadezimale oder dezimale Darstellung der Befehle ist in jedem Falle erlaubt. Dabei gibt es auch keine Einschränkungen, außer daß der Befehl im Befehlsvorrat der CPU enthalten sein muß. Der Befehlsvorrat für den 8-Bit-Prozessor U880 ist in Anhang H enthalten. Die Benutzung ist nur möglich mit Kenntnissen der Maschinen- und Assemblerprogrammierung. Als Operanden sind Konstanten oder Variablen des PASCAL-Programms zugelassen. Bei Variablen wird der Adreßwert des angegebenen Bezeichners verwendet. Konstanten werden in einem Byte abgebildet, wenn sie im Bereich 0 bis 255 liegen. Sonst wird ein 16-Bit-Wort dafür bereitgestellt. Ist ein 16-Bit-Wort als Adresse festgelegt (im Beispiel nach CALL), so kann die sonst Ein-Byte-Abbildung der "5" durch ">5" in eine Zwei-Byte-Abbildung übergeführt werden. Entsprechend würde das Zeichen "<" vor der Konstanten eine Zwei-Byte- in eine Ein-Byte-Abbildung umwandeln. Der höherwertige Teil ginge verloren. Implementationsabhängig sind als Konstanten auch Zeichenketten zugelassen. Zur Steuerung des Ablaufs im Maschinencode stehen sehr eingeschränkte Möglichkeiten zur Verfügung. Sie beziehen sich

– auf den aktuellen Stand des Befehlszählers. Zum Beispiel definiert das Element

eine Adresse, die 5 Byte unterhalb des aktuellen Standes des Befehlszählers liegt. Soll ein Sprung zu einer höheren Adresse erfolgen, so ist ein "+" zu verwenden;

– auf die Adresse des Bezeichners einer Variablen oder eines Unterprogramms. Das Element

  Laufwerk + 1

  bewirkt eine Adressierung ein Byte oberhalb der Adresse von "Laufwerk". Auch Minuszeichen sind erlaubt.

Beide Formen können kombiniert werden. So bewirkt "Laufwerk −*−5" die Fortsetzung der Programmausführung bei der Adresse laut Adreßrechnung
Adresse Laufwerk – aktueller Befehlszähler – 5.
Eine absolute Adressierung ist nicht möglich, weil zum Zeitpunkt der Programmierung die wirkliche Lage des Maschinencodes im Hauptspeicher nicht bekannt ist. Über INLINE kann auf jedes Register zugegriffen werden. Geschieht das, so hat der Programmierer zu gewährleisten, daß alle Register beim Austritt aus INLINE denselben Wert hat wie beim Eintritt in den INLINE-Befehl.

Externer Maschinencode (EXTERNAL-Direktive)

In allen PASCAL-Systemen für Mikrorechner gibt es die Möglichkeit, Programme oder Unterprogramme, die in anderen Programmiersprachen (darunter FORTRAN, COBOL, ASSEMBLERSPRACHE) erstellt wurden, zu nutzen. Die Art, wie das geschieht, ist in hohem Maße implementationsabhängig. Die Möglichkeiten reichen von der einfachen Bezugnahme auf extern zum eigenen Programmcode geladene Maschinenprogramme bis zur Einbindung relokativer (verschieblicher) Codes beim Linken.
Eine Möglichkeit ist die EXTERNAL-Direktive. Die Anwendung ist denkbar einfach. Zur Deklaration des Unterprogramms wird ein Prozedur- oder Funktionskopf (mit oder ohne Parameter) in der üblichen PASCAL-Darstellung geschrieben. Statt des Unterprogrammblocks jedoch wird das reservierte Wort EXTERNAL und danach eine Adresse notiert:

```
PROCEDURE Maschinencode(x,y : INTEGER); EXTERNAL ¤9000;
```

Die Adresse gibt an, wo die Prozedur im Hauptspeicher steht. Die Parameter werden im CPU-Stack abgelegt. Das geschieht wie bei internen Unterprogrammen auch je Parameter mit wenigstens einem 16-Bit-Wort. Danach wird zur angegebenen Adresse gesprungen. Die Entnahme der Parameter aus dem Stack muß durch das Unterprogramm erfolgen. Auch der Rücksprung ist dort zu organisieren. Die Bereitstellung des Maschinencodes auf der Adresse (hier ¤9000) ist mit BLOCKREAD (vgl. Abschnitt 6.3.) auf absolut (hier mit ¤9000) adressierte Variablen möglich.

Start eines Programms durch ein Programm

Die PASCAL-Implementationen für Mikrorechner sehen vor, daß ein Programm von einem anderen Programm aus gestartet werden kann. Diesen Vorgang bezeichnet man als Programmverkettung. Programmverkettung ist von Bedeutung, wenn aus Gründen der begrenzten Hauptspeicherkapazität eine (sehr große) Aufgabe durch zwei Programme

realisiert werden muß. Das zu startende Programm muß unter SCPX als COM-File (bei 16-Bit-Computern auch als CMD-File) auf der Diskette vorliegen. Zu beachten ist, daß eine Rückkehr zum Erstprogramm nicht erfolgt, also programmiert werden muß. Diese Technik zeigen die folgenden beiden Programme:

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

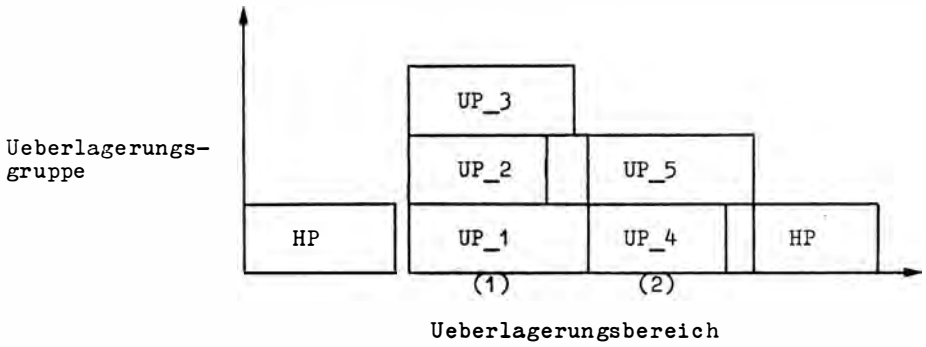
Programm-start

```
PROGRAM Zwei;
{Demonstrationsprogramm fuer programmierten Start}
BEGIN
  writeln('Programmverkettung');
  write('Ende')
END.
```

Eine Datenübergabe und der Rückstart wurden hier nicht vorgesehen. Die Datenübergabe ist mit ABSOLUTE oder über ein externes File möglich (vgl. Kapitel 6). Für den Rückstart wäre die entsprechende Befehlsfolge im Programm ZWEI zu notieren.
Es gibt auch Implementationen, in denen CHAIN oder EXECUTE verwendet werden kann. Dann steuert CHAIN eine Verkettung bei gemeinsamer Laufzeitbibliothek. Das Grundprinzip der Verkettung bleibt unverändert.

Überlagerungen

Bei Programmen, die die Kapazität des Hauptspeichers übersteigen, kann eine Überlagerungsstruktur für die Unterprogramme erzeugt werden. Die Hauptspeicherbelegung in einem solchen Falle zeigt das folgende Schema:



Die wirkliche Anordnung der Überlagerungsbereiche und des Hauptprogramms im Hauptspeicher ist unterschiedlich. Das Prinzip besteht darin, daß nichtaktive Unterpro-

gramme einer Überlagerungsgruppe auf den externen Datenspeicher ausgelagert werden. Erst im Falle ihres Aufrufs werden sie in den Hauptspeicher transportiert. Sie überschreiben dort ein eventuell vorher aktives Unterprogramm derselben Überlagerungsgruppe. Der Vorteil besteht darin, daß der Überlagerungsbereich für eine Überlagerungsgruppe nur so groß sein muß wie das größte Unterprogramm dieser Gruppe. Die mögliche Einsparung an Hauptspeicherkapazität kann beträchtlich sein. Auf diese Weise ist es möglich, große Programme auf Rechnern mit relativ kleiner Hauptspeicherkapazität auszuführen.
Die technische Realisierung einer Überlagerungsstruktur erfolgt bei geschlossener Technologie der Maschinencodeerzeugung während des einheitlichen Kompilier-/Linkvorgangs, sonst beim Linken. Dabei entsteht je Überlagerungsgruppe ein Überlagerungsfile, das sofort auf einen externen Datenträger ausgelagert wird. Es enthält den Maschinencode aller zur Gruppe gehörenden Unterprogramme. Implementationsabhängig können zum Beispiel bis zu 100 Überlagerungsfiles angelegt werden. Sie erhalten den Filenamen des Hauptprogramms und eine Nummer 000, 001 ... als Filenamenerweiterung.
Natürlich muß dem jeweiligen PASCAL-System mitgeteilt werden, daß eine Überlagerungsstruktur erforderlich ist und welche Unterprogramme eine Überlagerungsgruppe bilden sollen. Dabei ist zu beachten, daß sich Unterprogramme der gleichen Gruppe niemals untereinander rufen, aktivieren können. Für die Bildung der Gruppen sind noch die folgenden Hinweise zu beachten:

a) Annähernd gleich große Unterprogramme sollten in eine Gruppe aufgenommen werden. Die Einsparung an Hauptspeicherkapazität ist dann besonders groß.
b) Häufig aktive Programme sollten verschiedenen Gruppen zugeordnet werden. Dadurch wird der Zeitverzug, der durch das ständige Laden der aktivierten Unterprogramme entsteht, geringer.

Natürlich sind die Einschränkungen aus der Ruffolge zu beachten. Die Form der Mitteilung an das PASCAL-System ist implementationsabhängig. Besonders einfach ist der folgende Verfahrensweg. Ein Unterprogramm, das Bestandteil einer Überlagerungsstruktur werden soll, erhält vor dem Schlüsselwort PROCEDURE oder FUNCTION den Zusatz OVERLAY. Alle aufeinanderfolgenden Unterprogramme mit dem Schlüsselwort OVERLAY bilden eine Überlagerungsgruppe. Die Gruppe gilt als abgeschlossen, wenn ein folgendes Unterprogramm kein OVERLAY enthält. Folgt nach diesem Unterprogramm ohne OVERLAY wieder ein Unterprogramm mit OVERLAY, so wird eine neue Überlagerungsgruppe eröffnet. Da die Reihenfolge der Unterprogrammdeklaration, gegebenenfalls mit FORWARD, vom Programmierer frei gewählt werden kann und auch "leere" (Pseudo-) Unterprogramme deklariert werden können, ist auf diese Art eine einfache, aber vollständige Mitteilung an das PASCAL-System möglich. Das Programm DEVISEN im nächsten Kapitel ist mit OVERLAY-Technik programmiert und soll als Beisplel gelten.

Modulare Kompilation

Im gewissen Sinn kann die OVERLAY-Technik als modulare Kompilation bezeichnet werden. Sie ist einfach und leistungsfähig. Allerdings muß das Hauptprogramm zum Zeitpunkt der Erzeugung des Modulfiles anwesend sein. Im engeren Sinn bezeichnet man als modulare Kompilation die Übersetzung vollständig separater Unterprogramme in einen verschieblichen Objektcode. Dabei ist das Hauptprogramm nicht anwesend. Einige PASCAL-Implementationen ermöglichen eine solche modulare Kompilation. Dazu wird

das bzw. werden die Unterprogramme in spezielle Schlüsselworte eingebunden und als gesondertes File abgelegt. Die Schlüsselworte sind je nach PASCAL-System MODULE und MODEND oder UNIT und END. MODEND bzw. END folgt in diesem Falle ein Punkt. Der Modul muß die vollständige Information für die Syntaxprüfung enthalten. Deshalb sind globale Bezeichner für Konstanten, Typen oder Variablen nach MODULE bzw. UNIT und vor PROCEDURE bzw. FUNCTION vollständig aufzuführen. Um in diesem Falle die Schnittstelle zum Hauptprogramm zu vereinfachen, gibt es zusätzliche Schlüsselworte (EXTERNAL, INTERFACE).
Die gesondert kompilierten Moduln müssen dann durch den Linker mit dem Hauptprogramm zu einem ausführbaren Programm verbunden werden.

Übungsaufgaben

1. Zur Beschleunigung des Sortiervorgangs im Programm VERZEICHNIS bei einer größeren Zahl von Einträgen ist das Unterprogramm ORDNEN gegen das Unterprogramm SORTIEREN (File SORT.BIB) in Anhang I auszuwechseln. Führen Sie die Anpassung durch, und testen Sie das neue Programm!

2. Bekannt sind die Matrix $\underline{N}$ der Maschinenzeitnormen und der Vektor z der Maschinenzeiten für die Erzeugnisse der Gesamtproduktion $\underline{p}$. Es gelten

   $\underline{z} = \underline{N}\underline{p}$

   $\underline{p} = \underline{N}^{-1}Z$

   $\underline{N}^{-1}$ ist die inverse Matrix von $\underline{N}$. Entwickeln Sie ein interaktives Programm, das nach Eingabe der Matrix und des Vektors $\underline{z}$ die Gesamtproduktion $\underline{p}$ ermittelt und ausgibt! Verwenden Sie das Unterprogramm INVERSE aus Anhang I!

3. Entwickeln Sie ein interaktives Programm, das nach Eingabe einer Zeitreihe den Regressionskoeffizienten und den Wert für den T-Test ermittelt und ausgibt! Benutzen Sie das Unterprogramm REGRESSION (REG.BIB) aus Anlage I!

6. Externe Speicherung und Verwaltung von Daten

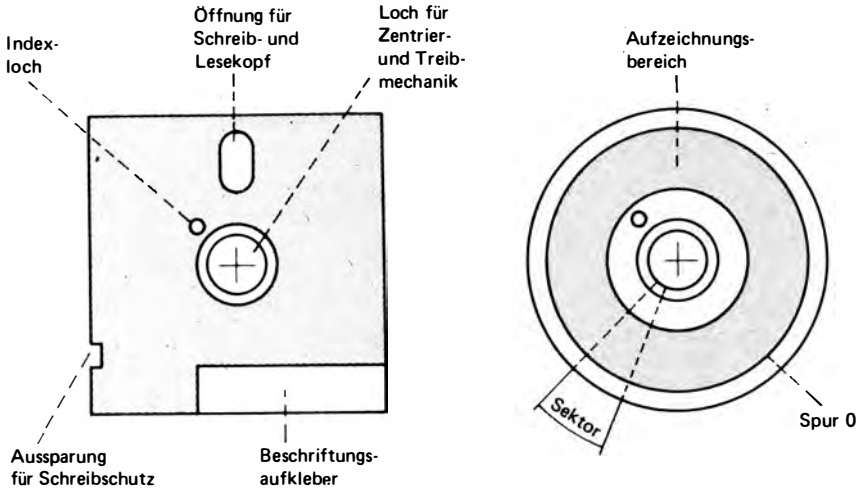
6.1. Speicherfiles und Filetyp

Es ist nicht möglich, die für die vielen Anwendungen des Computers erforderlichen Daten im Hauptspeicher aufzubewahren. Ein Computer arbeitet im zeitlichen Wechsel mit verschiedenen Programmen, und jedes Programm verarbeitet Daten. Für ökonomische Anwendungen sind das meistens sehr viele Daten. Die Kapazität des Hauptspeichers wäre schnell überschritten. Es kommt hinzu, daß die Hauptspeicher von Mikrorechnern in RAM-Technologie (Random-Access-Memory) ausgeführt sind und beim Ausschalten ihren Inhalt verlieren. Aus diesem Grunde ist die Arbeit des Computers ständig von der Auslagerung von Daten auf externe Speicher und dem Wiedereinlesen begleitet. Das beginnt beim Programmieren, wenn PASCAL-Quelltext- und Kommando-Files (COM/CMD-Files ausführbarer Programme) angelegt und bearbeitet werden, und spielt in fast allen Anwendungen bezüglich der Verarbeitungsdaten eine wichtige Rolle.
Der auf einen externen Datenträger ausgelagerte Datenbestand heißt Speicherfile. Kennzeichnend für Speicherfiles ist im Gegensatz zu internen Datenstrukturen, daß die Anzahl der Filekomponenten nicht festgelegt wird. Die Filekomponenten, die Elemente eines Speicherfiles, werden einfach aneinandergefügt. Das Ende wird durch spezielle Zeichen, die End-of-File-Zeichen (meist□1A) gekennzeichnet. Nach dem Datenträger kann man Diskettenfile oder Bandfile (bei Mikrorechnern Kassettenmagnetband) unterscheiden. Hier werden Diskettenfiles behandelt.

Disketten
Disketten sind flexible, magnetbeschichtete Platten (in Normtext: Flexible disk cardridges), auch Floppy disk genannt. Technische Einzelheiten können der Fachliteratur[8] entnommen werden. Die Diskettenoberfläche ist in konzentrische Spuren aufgeteilt. Diese Spuren (Tracks) sind in Sektoren unterteilt. Die physische Diskettensteuerung (als Teil der Speicherperipherie) ist in der Lage, die einzelnen Spuren und Sektoren einzeln zu adressieren. Der Orientierung dient dabei das Indexloch. Die für die Anwendung erforderlichen Angaben zur Diskette sind in der Abbildung auf Seite 96 enthalten.

Das Laufzeitsystem SCPX adressiert jedoch nicht spur- und sektorweise, sondern in grö-

8 Vgl. Krauß, M.; Kutschbach, E.; Woschni, E.-G.: Handbuch der Datenerfassung. Berlin: VEB Verlag Technik 1984, S. 363ff.



ßeren Einheiten, die Aufzeichnungsblöcke genannt werden. Bei Disketten mit einfacher Aufzeichnungsdichte ist die Größe des Aufzeichnungsblocks 1024 Byte (1KByte) oder 2048 Byte (2 KByte), bei Disketten mit doppelter Schreibdichte 2, 4, 8 oder 16 KByte. Ein Datenbestand wird also bei einer Diskette mit einfacher Schreibdichte mit mindestens 1024 Byte geschrieben, auch wenn die Anzahl der Nutzbytes geringer ist. Einige PASCAL-Systeme füllen den Rest des Aufzeichnungsblocks mit Steuerzeichen. Im allgemeinen ist natürlich mehr als ein Aufzeichnungsblock erforderlich. Wieviel Aufzeichnungsblöcke bzw. KByte eine Diskette aufnehmen kann, richtet sich nach

a) der Sektorlänge, die beim Initialisieren (dem Prüfen und Einteilen) vom Anwender festgelegt wird. Möglich sind z. B. 16 Sektoren je Spur zu je 256 Byte (entspricht 160 KByte je Diskettenoberfläche) und 5 Sektoren zu je 1024 Byte (200 KByte);
b) der Anzahl der Spuren (meist 40 bei einfacher, 80 bei doppelter Schreibdichte) und dem Spuroffset. Für den Anwender sind meist, in einem Laufwerk unbedingt, je nach Sektorierung 2 oder 3 Spuren nicht zugänglich. Dort befinden sich das Laufzeitsystem und der Systemlader. Sie belegen bei SCPX normalerweise 77 Sektoren zu je 128 Byte. Die Kapazität der Diskette reduziert sich für den Nutzer je nach Sektorunterteilung um etwa 10 KByte;
c) der Anzahl beschreibbarer Diskettenoberflächen. Es wird zwischen einseitig und doppelseitig unterschieden.

Natürlich muß der auf die Diskette ausgelagerte Datenbestand wiedergefunden werden. Der Programmierer hat deshalb den externen Datenbestand, das File, zu bezeichnen, mit einem Namen zu belegen. Das geschieht in PASCAL über die vordefinierte Prozedur ASSIGN.

Filenamen

Die Bildung des Filenamens unterliegt den Regeln des Laufzeitsystems. Unter SCPX besteht ein Filename aus

– dem Basisnamen mit bis zu acht Zeichen. Es können Buchstaben, alle Ziffern und ei-

nige Sonderzeichenfolgen verwendet werden. Groß- und Kleinschreibung für Buchstaben sind nicht signifikant. SCPX setzt intern alle Buchstaben in Großbuchstaben um;
- der Filenamenserweiterung mit bis zu drei Zeichen. Einige Erweiterungen sind reserviert und sollten nicht benutzt werden, darunter COM oder CMD für ausführbare Programme, SRC und PAS für PASCAL-Quelltexte, ERL und REL für Objektcodedateien (relokativer Code).

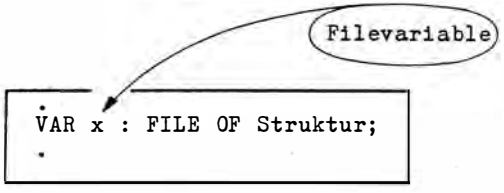
Die Erweiterung kann auch fehlen. Ist sie vorhanden, müssen Basisname und Erweiterung durch einen Punkt getrennt werden (zum Beispiel STAMM.DAT, BETRIEBS.DOK).
Unter Benutzung des vom Programmierer festgelegten Filenamens wird auf der Diskette ein Inhaltsverzeichnis, die Directory, angelegt.

Directory
Die Directory kann mindestens 64 (0...63) Verzeichniseinträge aufnehmen. Ein Verzeichniseintrag, ein sogenannter File-Control-Block (FCB), besteht aber nicht nur aus dem Namen des Datenbestandes. Ein FCB umfaßt 32 (0...31) Byte. Die wichtigsten Belegungen der Byte sind folgende:

Byte-Nummer	Inhalt
0	Benutzernummer
1 - 8	Basisname, bei weniger als acht Zeichen mit Leerzeichen aufgefuellt
9 - 11	Namenserweiterung oder Leerzeichen
12	Abschnittsnummer (Extent), wenn das File mehr als einen FCB benoetigt. Die Abschnittsnummer dient dann zur Verbindung zusammengehoerender FCB.Sonst ist dieses Byte null
15	Anzahl der 128 - Byte - Saetze des Files
16 - 31	Nummern der zum File gehoerenden Aufzeichnungsbloecke

Die Directory einer Diskette belegt mindestens 2 KByte ($64 \times 32 = 2048$) und verringert die für den Nutzer verfügbare Diskettenkapazität.

Filevariablen
Soll ein neues File angelegt oder auf ein bestehendes File zugegriffen werden, so ist zunächst eine Filevariable zu deklarieren. Das geschieht in PASCAL zum Beispiel mit

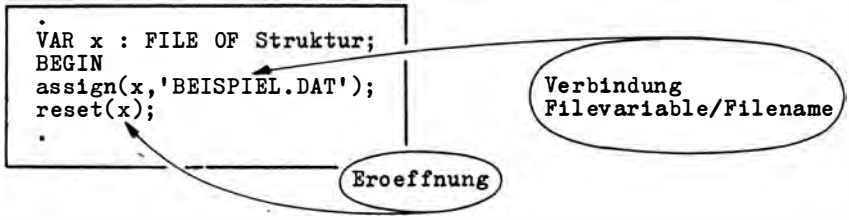


"Struktur" ist irgendein mit TYPE vordefinierter Typbezeichner. "x" heißt Filevariable. Mit einer solchen Deklaration wird Speicherplatz reserviert für
a) die Zwischenspeicherung von Daten bei der Übertragung zur Diskette (der sogenannte Filepuffer);

b) einen Zeiger auf die nächste zu lesende oder zu schreibende Filekomponente (Filefenster);
c) die Charakteristik des Zustandes eines Files wie Lesezustand, Schreibzustand, Ende des Files erreicht und anderes.

Fileeröffnung

Unter Benutzung der deklarierten Filevariablen muß das File nunmehr für die Datenübertragung vorbereitet, eröffnet werden. Das erfolgt im Anweisungsteil eines Programms mit den vordefinierten Prozeduren REWRITE (wenn ein File neu angelegt werden soll) oder RESET (wenn es bereits existiert). Die Eröffnung eines auf der Diskette vorhandenen Files mit der Bezeichnung "BEISPIEL.DAT" zeigt der folgende Ausschnitt:



Wird vor dem Filenamen kein Laufwerksbezeichner angegeben, so wird das File im aktuellen Laufwerk bei RESET gesucht und bei REWRITE angelegt. Bei "B:BEISPIEL.DAT", also mit dem Laufwerksbezeichner "B:", geschieht das nur im Laufwerk b.
Beim Eröffnen eines Files mit RESET werden vom Programm folgende Operationen ausgeführt:

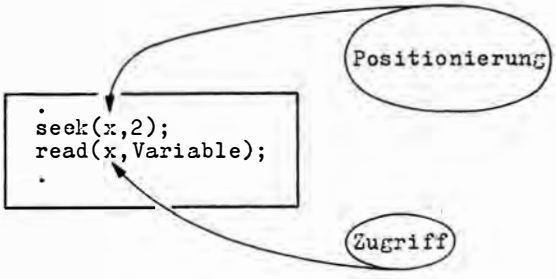
a) Der FCB wird von der Diskette in den Hauptspeicher kopiert. Dabei werden einige Byte verändert (zum Beispiel aus Benutzernummer wird die Laufwerksnummer) und einige angehängt. Aus diesem Grunde wird auch zwischen Disketten-FCB und Speicher-FCB unterschieden.
b) Das Filefenster wird auf 0 (die erste Komponente des Files) gesetzt.
c) End-of-File, zugleich als EOF eine vordefinierte Funktion, wird mit FALSE belegt.

Ob bei RESET bereits die erste Filekomponente in den Hauptspeicher geholt wird, ist implementationsabhängig.
Beim Eröffnen eines neuen Files mit REWRITE wird ein neuer FCB gebildet und der Filezeiger ebenfalls auf 0 gesetzt. Die Funktion EOF erhält allerdings den Wert TRUE. Wird REWRITE auf ein bestehendes File angewandt, so wird wie bei RESET der FCB in den Hauptspeicher kopiert, der Filezeiger auf 0 und EOF auf TRUE gesetzt. Dadurch sind alle in diesem File enthaltenen Daten nicht mehr zugänglich und in diesem Sinne gelöscht. Mit REWRITE auf bestehende Files ist also sehr vorsichtig umzugehen.

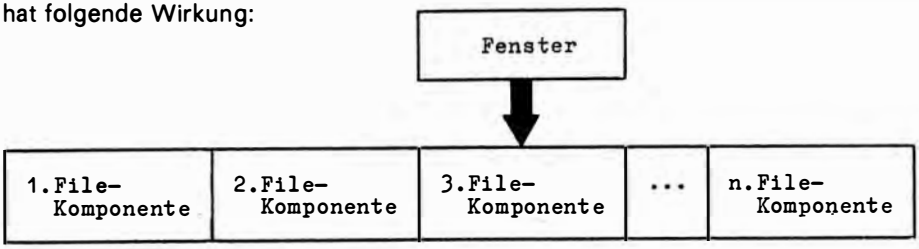
Zugriff

Ist das File eröffnet, kann auf seine Komponenten zugegriffen werden. Unter Zugriff versteht man, daß eine Filekomponente gelesen oder geschrieben wird. Dafür gibt es eine Reihe von vordefinierten Prozeduren. Sie unterscheiden sich nach sequentiellem Zugriff (mit der 1. Filekomponente beginnend und ohne Auslassungen in der logischen Reihenfolge ihrer Speicherung) und wahlfreiem Zugriff (eine beliebige Komponente unter Auslassung aller anderen Komponenten des Files). Für den sequentiellen Zugriff sind das

zum Beispiel die Prozeduren GET (Lesen), PUT (Schreiben) oder wie bei Gerätefiles READ und WRITE (READLN, WRITELN hätten hier keine besondere Wirkung). Für den wahlfreien Zugriff gibt es gesonderte Prozeduren, wie SEEKREAD/SEEKWRITE, GETD/PUTD, READDIRECT/WRITEDIRECT, oder er ist ebenfalls mit READ/WRITE möglich.
Als sehr leistungsfähig haben sich Techniken erwiesen, bei denen "sequentiell" und "wahlfrei" kombiniert werden. Dabei wird das File in jedem Falle für den wahlfreien Zugriff organisiert und zugleich eine sequentielle Verarbeitung unterstützt. Dazu wird ein interner Filezeiger geführt. Er zeigt jeweils auf eine bestimmte Filekomponente, am Anfang auf die Komponente 0. Der Programmierer erhält die Möglichkeit, die Position des Filezeigers zu beeinflussen. Er kann die aktuelle Position abfragen (zum Beispiel mit der vordefinierten Funktion FILEPOS) und selbst positionieren. Die Positionierung des Zeigers wird mit SEEK durchgeführt. Dabei ist zu beachten, daß die Zählung der Filekomponenten mit Null beginnt.
Die Positionierung für das File "BEISPIEL.DAT" mit



hat folgende Wirkung:



Durch SEEK wird gewissermaßen ein "Fenster" über der jeweiligen Komponente des Files positioniert. Der Zugriff führt dann zum Lesen oder Schreiben der Filekomponente im Fenster.
Eine Besonderheit und damit eine Unterstützung der sequentiellen Zugriffstechnik besteht darin, daß bei jeder Lese- oder Schreiboperation der Filezeiger um 1 erhöht wird. Da er durch RESET oder REWRITE am Anfang auf Null gesetzt und anschließend durch das System nach jeder Operation um 1 erhöht wird, kann SEEK entfallen. Es erfolgt dann sequentielles Lesen oder Schreiben.

File schließen

Nicht jeder Lese- oder Schreibbefehl führt auch zum physischen Lesen und Schreiben der Diskette. Deshalb sind logischer und physischer Zugriff zu unterscheiden. Zwischen

ihnen vermittelt ein interner Sektorpuffer. Dadurch verringern sich die Anzahl der physischen Zugriffe und die Zeit für die Datenübertragung. Implementationsabhängig kann der Programmierer mit der vordefinierten Funktion FLUSH den Pufferinhalt auf die Diskette schreiben, allerdings nur, wenn der letzte Filezugriff eine Schreiboperation war.

Auch wenn physisch auf die Diskette geschrieben wurde, erfolgt nicht sofort eine Aktualisierung des Disketten-FCB. Zunächst wird nur der Inhalt des Speicher-FCB verändert. Deshalb kann es passieren, daß physisch geschriebene Aufzeichnungsblöcke später nicht auffindbar sind, und zwar dann, wenn der Disketten-FCB nicht aktualisiert wurde. Das Aktualisieren des Disketten-FCB durch Kopieren der entsprechenden Bytes des Speicher-FCB muß vom Programmierer mit der vordefinierten Prozedur CLOSE veranlaßt werden. Erfolgt das nicht, tritt ein Datenverlust auf. Aus Sicherheitsgründen ist auch ein CLOSE gebräuchlich, wenn nur gelesen wurde. Es gibt Laufzeitsysteme mit einer FCB-Technik, die CLOSE nach dem Lesen sogar erforderlich macht, weil nur wenige Files gleichzeitig geöffnet sein können.

Datensicherheit

Auf Mikrorechnern im Ein-Nutzer-Betrieb bestehen Möglichkeiten, Files vor unbefugter Einsichtnahme oder irrtümlichem Überschreiben zu schützen. Sie sind weitgehend abhängig vom Laufzeitsystem. Unter SCPX sind möglich

a) die Benutzung einer von Null verschiedenen USER-Identifikation. Sie kann mit dem SCPX-Kommando USER<Nummer> eingestellt werden. <Nummer> ist eine Zahl zwischen 0 und 15. Danach sind alle Files bei der Verzeichnisanzeige (Kommando DIR) nur sichtbar, wenn sich der Nutzer vorher mit dieser Nummer anmeldet;
b) die Unterdrückung der Verzeichnisanzeige für das File durch Setzen des SYS-Attributs;
c) der Schutz vor dem Überschreiben durch Setzen des R/O(read only)-Attributs. Soll das File später verändert werden, muß dieses Attribut rückgesetzt werden;
d) der physische Schreibschutz für die gesamte Diskette, auf der sich das File befindet. Das geschieht im allgemeinen durch Verkleben der Aussparung für den Schreibschutz an der Diskettenhülle.

Abschließend soll darauf hingewiesen werden, daß auswechselbare Disketten (also nicht Festplatten) beim Schreiben zusätzlich vom Laufzeitsystem überwacht werden. Dazu wird beim Anwählen des jeweiligen Laufwerks, in dem sich eine Diskette befindet, eine interne Blockzuordnungstabelle einschließlich Prüfinformation aufgebaut. In einer solchen Tabelle wird jeder Aufzeichnungsblock durch ein Bit dargestellt. Ist der Aufzeichnungsblock belegt, ist dieses Bit 1, sonst 0. In dieser Tabelle ermittelt das System, auf welche Aufzeichnungsblöcke Daten abgelegt werden können, und zwar vor jeder Schreiboperation. Würde die Diskette im Laufwerk gewechselt, so daß die interne Zuordnungstabelle nicht mehr mit der wirklichen Belegung übereinstimmt, wird auf Fehler erkannt. Deshalb ist nach jedem Diskettenwechsel die Eingabe des Steuerzeichens CTRL C (Tastenkombination des CTRL und C) oder auf andere Weise ein Rücksetzen der Laufwerke in die Grundstellung erforderlich.

Die Schutzmöglichkeiten für Files (Setzen der USER-Nummer, Setzen oder Rücksetzen des SYS- bzw. R/O-Attributs) werden von PASCAL im allgemeinen nicht direkt unterstützt. Natürlich ist die Programmierung mit INLINE, gegebenenfalls auch mit BDOS-/BIOS-Rufen möglich.

6.2. Records und getypte Binärfiles

Im vorigen Abschnitt wurde die konkrete Struktur der einzelnen Komponenten des Files offengelassen. Es wurde auf die Typvereinbarung für den Typbezeichner "Struktur" verwiesen. PASCAL läßt jeden Typ zu, einfache und auch strukturierte Typen. Charakteristisch für Filearbeit ist jedoch der RECORD-Typ. Records (Sätze) sind Zusammenfassungen von Daten beliebigen Typs, außer vom Typ FILE, unter einem einheitlichen Namen. Natürlich gibt es einen Zusammenhang zwischen den Bestandteilen eines Records. Aber der wird vom Problem und nicht von PASCAL diktiert.

Records

Ein Beispiel für die Vereinbarung des strukturierten Datentyps RECORD enthält das folgende Programm DEVISEN:

```
PROGRAM Devisen;
{Verwalten/Nutzen einer Waehrungstabelle fuer Devisenumrechnungen}
CONST Laenge = 30; Filename = 'TABELLE.DAT';
TYPE  Zeichen = STRING[14];
      Waehrung  = RECORD
        Land : Zeichen;
        Geld : STRING[7];
        Kurs : REAL;
      END;
VAR   Extern  : FILE OF Waehrung;
      Tabelle : ARRAY[0..Laenge] OF Waehrung;
      Umfang,Basis,
      Ziel    : INTEGER;
      Eingabe : Zeichen;
{¤I MENUE.DEV    }
{¤I SUCHEN.DEV   }
{¤I ANLEGEN.DEV  }
{¤I VERWALTE.DEV }
{¤I NUTZEN.DEV   }
BEGIN {Hauptprogramm}
  writeln('A) nlegen K) orrigieren  der Waehrungstabelle oder');
  writeln('N) utzen  E) nde');
  write('A/K/N/E: ');
  readln(Eingabe);
  CASE Eingabe[1] OF
    'A','a': BEGIN Anlegen; Verwalten END;
    'K','k': Verwalten;
    'N','n': Nutzen;
    'E','e': BEGIN write('Ende'); halt END;
  ELSE writeln('Fehlerhafte Eingabe')
  END;
  write('Ende')
END.
```

Recordtyp

FILE mit Recordkomponenten

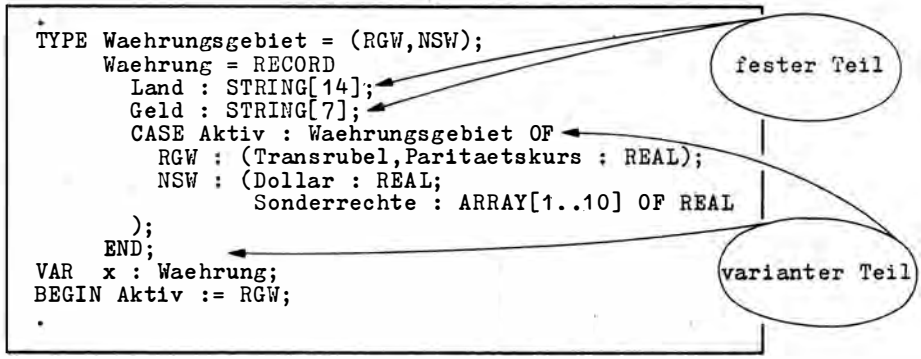
Feld von Recordvariablen

Die Syntax der RECORD-Vereinbarung ist dem Syntaxdiagramm zu entnehmen.
Der für den Record vereinbarte Typbezeichner heißt WAEHRUNG. Was in Feldern nicht möglich war, nämlich die Zusammenfassung von Daten verschiedenen Typs, hier STRING [14] für LAND und STRING [7] für GELD sowie REAL für KURS, ist für Records charakteristisch. Gerade ökonomische Anwendungen verlangen oft vom Problem her zusammenhängende Daten verschiedenen Typs. Man denke nur an Personaldaten mit den

Typen STRING (für den Namen), INTEGER (Einstellungsdatum) und REAL (Einkommen). Materialdaten enthalten ebenfalls zumindest die Datentypen STRING (Bezeichnung, Nomenklaturnummer) und INTEGER (Lieferdatum, Bestand in Stück).

Records mit variantem Teil

PASCAL läßt auch eine variable Gestaltung von Records zu. Im Beispiel DEVISEN kann es erforderlich sein, die Besonderheiten verschiedener Währungsgebiete zu berücksichtigen. Die Bezeichnung des Landes und der Währungseinheiten ist für alle Länder erforderlich. In den Ländern des Rates für Gegenseitige Wirtschaftshilfe (RGW) haben der transferable Rubel und der Paritätskurs eine große Bedeutung. Für Länder des nichtsozialistischen Wirtschaftsgebietes (NSW) sind der Dollarkurs und die Sonderziehungsrechte des Internationalen Währungsfonds wichtig. Diesem Sachverhalt würde folgende Datenstruktur entsprechen:



Der Record besteht aus einem festen und einem varianten Teil. Der variante Teil sieht zwei Möglichkeiten vor. Welche der beiden Möglichkeiten gültig ist, wird durch den Wert der Variablen "Aktiv" entschieden, die vom Typ Waehrungsgebiet ist. Ist "Aktiv" gleich

RGW, so besteht der Record aus "Land", "Geld" (fester Teil) und "Transrubel", "Paritaetskurs" (varianter Teil).

NSW, so besteht der Record aus "Land", "Geld" (fester Teil) und "Dollar", "Sonderrechte" (varianter Teil).

"Aktiv" heißt Etikettenvariable. Ihre in CASE zur Variantenunterscheidung benutzten Konstanten nennt man Etikettenkonstanten. Records können aus nur einem festen Teil, aus einem festen und varianten Teil oder nur aus einem varianten Teil bestehen. Feldlisten hinter Etikettenkonstanten können erneut feste und variante Teile enthalten. Folgende Regeln sind noch wichtig:

a) Alle Elemente eines Records müssen eindeutig und nur einer Variante zugeordnet sein.
b) Der variante Teil ist stets nach dem festen Teil zu notieren.
c) Die Etikettenvariable muß vom ordinalen Typ sein.
d) Vor einer Record-Variante können mehrere Etikettenkonstanten notiert sein.

Es ist auch möglich, auf die Etikettenvariable zu verzichten. Man spricht in diesem Falle

von freien Varianten. Das PASCAL-System kann dann allerdings nicht überwachen, welche der Varianten gerade aktiv ist. Das muß der Programmierer tun. Eine Anwendung zeigt der folgende Ausschnitt:

```
.
TYPE Ueberlagerung = RECORD
       CASE INTEGER OF
         1 : (r : REAL);
         2 : (b : ARRAY[1..6] OF BYTE
       );
     END;
VAR Zahl : Ueberlagerung;
      i : INTEGER;
BEGIN
  read(Zahl.r);
  FOR i := 1 TO 6 DO write(Zahl.b[i]:4);
.
```

Hier wird eine Realzahl eingelesen und byteweise wieder ausgegeben. Die Zugriffe, hier "Zahl.r" und "Zahl.b", werden noch erläutert.
Nun zurück zum Beispiel DEVISEN. Der Record besteht dort nur aus einem festen Teil. Der Typbezeichner WAEHRUNG wird in der Variablenvereinbarung mehrfach genutzt. Zunächst wird mit "FILE OF Waehrung" das File getypt, das heißt, die Struktur seiner Komponenten festgelegt (nicht ihre Anzahl!). Danach ist ein Feld "Tabelle" vereinbart, dessen einzelne Elemente einen Record der definierten Struktur bilden. Also Tabelle [1] enthält zwei Stringvariablen der Länge 20 und eine Realvariable, ebenso Tabelle [2], Tabelle [3] usw. Wie auf solche Komponenten zugegriffen werden kann, zeigen die Unterprogramme. Das wird noch erläutert. Zunächst zum Problem.

Problem

Das Programm DEVISEN ermöglicht die Umrechnung von Währungen verschiedener Länder im Dialog unter Nutzung einer als File verwalteten Währungstabelle. Beim praktischen Einsatz am Bankschalter wären weitere Maßnahmen zur Kontrolle der Eingabedaten und zur Protokollierung der Geldwechselvorgänge erforderlich. Außerdem müßte man die direkten Kurse der Währungen zueinander verwenden. MENUE, SUCHEN, VERWALTEN, NUTZEN sind Unterprogramme. Den Entwurf des Gesamtprogramms bildete die folgende Strukturtabelle:

```
PROGRAM Devisen;
BEGIN
{1. Eroeffnungsbild                                              }
{2. Nutzerentscheidung fuer die Programmfunktion                 }
{3. Programmausfuehrung in Abhaengigkeit von der                 }
{   Nutzerentscheidung                                           }
{3.1 Verwalten der Waehrungstabelle                              }
{3.1.1 Anlegen, wenn kein File vorhanden                         }
{3.1.2 Eroeffnung des Files und lesen in den Hauptspeicher       }
{3.1.3 Wiederholte Korrektur oder Erweiterung des Files          }
{3.1.4 Schliessen des Files                                      }
{3.2 Devisenumrechnung an Hand der Waehrungstabelle              }
{3.2.1 Eroeffnen des Files und lesen in den Hauptspeicher        }
{3.2.2 Wiederholte Devisenumrechnung nach Eingabe der Ziel-      }
       und Basiswaehrung sowie des Umtauschbetrages              }
```

Fortsetzung S. 104

```
{3.2.3 Schliessen des Files                                    }
{3.3 Beenden der Programmausfuehrung                            }
{3.3.1 Endemitteilung                                          }
{3.3.2 Rueckkehr zum Laufzeitsystem                            }
{4. Beendigung der Programmausfuehrung und Endemitteilung      }
END.
```

Anlegen der Währungstabelle

Wählt der Nutzer des Programms entsprechend dem Angebot ein "A", so existiert entweder kein File TABELLE.DAT, oder die gesamte Tabelle soll neu angelegt werden. Aus der CASE-Anweisung des Hauptprogramms wird dann die Prozedur ANLEGEN gerufen. Sie hat den folgenden Quelltext:

```
PROCEDURE Anlegen;
BEGIN
  assign(Extern,Filename);
  rewrite(Extern);
  close(Extern);
  writeln('Waehrungstabelle angelegt');
END;
```

Diese Vorgehensweise ist typisch für das Anlegen eines neuen Files in PASCAL. Es wird zunächst nur ein leeres File angelegt. Nach CLOSE gibt es auf der Diskette einen FCB, aber keinen belegten Aufzeichnungsblock.

Für diesen konkreten Anwendungsfall wäre es gegebenenfalls sinnvoll, die Daten vor unbeabsichtigtem Verlust zu schützen. Gibt der Anwender versehentlich "A", so sind die Daten verloren. Eine Möglichkeit, das zu verhindern, ist die in mehreren Implementationen verfügbare vordefinierte Funktion IORESULT. Die Funktion gestattet dem Programmierer den Zugriff auf ein spezielles Kontrollbyte des Laufzeitsystems. Sie liefert den ganzzahligen Wert Null, wenn Ein- und Ausgabeoperationen fehlerlos verlaufen sind, sonst einen von Null verschiedenen Wert. Um allerdings diesen Fehlercode selbst auswerten zu können, muß der Programmierer vorher die Standardreaktionen des Laufzeitsystems (meist Programmabbruch) ausschalten. Das geschieht mit der Compilerdirektive {¤I-}. Die Wiedereinschaltung erfolgt mit {¤I+}. Die Nutzung dieser Technik im Beispiel würde folgenden Programmausschnitt ermöglichen.

```
.
VAR e : CHAR;
BEGIN
  assign(Extern,'Tabelle.DAT');
  {¤I-} reset(Extern) {¤I+};
  IF ioresult = 0 THEN BEGIN
    write('Loeschen der vorhandenen Daten (J/N): ');
    readln(e);
    IF (e <> 'j') AND (e <> 'J') THEN exit
  END;
  rewrite(Extern);
  close(Extern)
.
```

Die Verwaltung der Währungstabelle

Unter Verwalten wird hier das Ändern einschließlich des Erweiterns und des Löschens von Eintragungen in der Tabelle bzw. dem File verstanden. Es erfolgt mit der Prozedur VERWALTEN.

```
OVERLAY PROCEDURE Verwalten;
VAR Index      : INTEGER;
    Eintragung : Waehrung;
    Gefunden   : BOOLEAN;
BEGIN
  assign(Extern,Filename);
  REPEAT
    reset(Extern);
    Umfang := 0;
    WHILE NOT eof(Extern) DO BEGIN
      read(Extern,Tabelle[Umfang]);
      Umfang := succ(Umfang)
    END;
    writeln;
    writeln('Korrigieren oder Erweitern der Waehrungstabelle');
    write('Waehrungseinheit(Suchbegriff/Ende mit <ET>): ');
    readln(Eingabe);
    IF Eingabe <> '' THEN BEGIN
      Index := Suchen(Eingabe);
      IF Index >= 0 THEN BEGIN
        Gefunden := TRUE;
        writeln('Tabelleneintragung: ',Tabelle[Index].Land,'/',
                Tabelle[Index].Kurs:7:4)
      END ELSE BEGIN
        Gefunden := FALSE;
        writeln('Neue Tabelleneintragung')
      END;
      WITH Eintragung DO BEGIN
        write('Land: '); readln(Land);
        write('Waehrungseinheit: '); readln(Geld);
        write('Kurs zu einer MARK: '); read(Kurs); writeln;
      END;
      IF Gefunden THEN seek(Extern,Index)
      ELSE seek(Extern,filesize(Extern));
      write(Extern,Eintragung);
    END;
    close(Extern)
  UNTIL Eingabe = ''
END;
```

RECORD-variable

Lesen des Files

WITH-Anweisung

Das File wird zunächst eröffnet. Es existiert unbedingt, unter Umständen als leeres File. Deshalb wird das RESET nicht gesichert. Ist RESET nicht erfolgreich, wird ein Programmabbruch in Kauf genommen.

Nach der Eingabe der Währungseinheit stellt das Unterprogramm SUCHEN fest, ob dafür bereits eine Eintragung in der Währungstabelle vorliegt. In diesem Falle wird die Nummer der Filekomponente zurückgegeben, sonst eine – 1:

```
FUNCTION Suchen(x : Zeichen) : INTEGER;
VAR i : INTEGER;
BEGIN
  Suchen := -1;
  FOR  i := 0 TO Umfang DO
    IF x  = Tabelle[i].Geld THEN Suchen := i
END;
```

Ist das Unterprogramm VERWALTEN aktiv und in der Währungstabelle bisher nur die Eintragung Land = DDR, Währungseinheit = Mark und Kurs zur Mark = 1 eingetragen, so ergibt sich das folgende Protokoll (Eingaben unterstrichen):

```
A) nlegen K) orrigieren der Waehrungstabelle oder
N) utzen  E) nde
A/K/N/E: K<ET>

Korrigieren oder Erweitern der Waehrungstabelle
Waehrungseinheit(Suchbegriff/Ende mit <ET>): Rubel<ET>
Land: UdSSR<ET>
Waehrungseinheit: Rubel<ET>
Kurs zu einer Mark: 0.3125<ET>
.
```

Das Einlesen des gesamten Files, wie es hier geschieht, ist eigentlich nicht typisch für ökonomische Anwendungen. Es kann aber auch nicht auf der Diskette verbleiben, weil jeder Suchvorgang mit dem Transport der einzelnen Komponenten zwischen Diskette und Hauptspeicher verbunden und deshalb zu langsam wäre. Günstiger ist es deshalb, ein Indexfile im Hauptspeicher aufzubauen, das die für den Zugriff auf das eigentliche File erforderliche Information verwaltet. Diese Information enthält den Schlüsselbegriff für den Nutzer und die Nummer der Filekomponente. Für das Beispiel wäre festzulegen, daß die Identifikation anhand der Währungsbezeichnungen, also zum Beispiel "Mark" oder "Rubel", erfolgt. Das Indexfile würde dann nur aus den Währungsbezeichnungen und der Nummer der Filekomponente bestehen, die alle weiteren Daten zu dieser Währungsbezeichnung enthält. Hier sind das LAND und KURS. Allerdings bleibt aus Gründen der Datensicherheit auch der Schlüsselbegriff in der Filekomponente. Dadurch entsteht ein Vierschrittzugriff, nämlich

1. Suchen des Schlüsselbegriffes im hauptspeicherresidenten Indexfile,
2. Ermittlung der Filekomponente und Positionierung des Filefensters,
3. Lesen der Filekomponente des Filefensters in den Hauptspeicher,
4. Zugriff auf den Inhalt der gelesenen Filekomponente.

Da die Währungstabelle nur wenig Speicherplatz belegt, wurde im Beispiel sofort das gesamte File in den Hauptspeicher geholt. Das vereinfacht die Organisation. Umständlich ist, daß jedesmal das File geschlossen, eröffnet und erneut gelesen werden muß. Darauf wird in den Übungsaufgaben zurückgekommen.

Wichtig für die Arbeit mit Records ist das richtige Identifizieren der Teile des Datensatzes. Wie in VERWALTEN zu sehen ist, erfolgt das unter Verwendung des Bezeichners der Recordvariablen und des Teilbezeichners in der Recordstruktur. Diese beiden sind durch einen Punkt getrennt. Dadurch entsteht eine zweistufige Bezeichnung. Sie könnte natürlich mehrstufig sein, wenn der Record selbst wieder einen Record enthält. Um diese komplizierte Schreibweise für den Programmierer zu verkürzen, gibt es in PASCAL eine spezielle Anweisung, die WITH-Anweisung. Sie klammert innerhalb ihres Bereiches (eine Anweisung, gegebenenfalls eine Verbundanweisung) den Bezeichner einer oder mehrerer Recordvariablen aus. Es können also nur die Teilbezeichner verwendet werden. Das verkürzt den PASCAL-Text zum Teil erheblich. Die Anwendung der WITH-Anweisung ist in VERWALTEN gezeigt. Die WITH-Anweisung klammert "Eintragung" aus. Im Bereich

der WITH-Anweisung heißt es also vollständig "Eintragung.Land", "Eintragung.Geld" und "Eintragung.Kurs".

Nutzen des Files

Die Nutzung des Files besteht im Beispiel darin, daß die Filekomponenten bei der Berechnung von Umtauschbeträgen von einer Währung in die andere benutzt werden:

```
OVERLAY PROCEDURE Nutzen;
VAR Betrag  : REAL;
    Richtig : BOOLEAN;
BEGIN
  assign(Extern,Filename);
  reset(Extern);
  Umfang := 0;
  WHILE NOT eof(Extern) DO BEGIN
    read(Extern,Tabelle[Umfang]);
    Umfang := succ(Umfang)
  END;
  REPEAT
    writeln; writeln;
    REPEAT
      writeln('Devisenumrechnung von Basis- in Zielwaehrung fuer');
      Menue;
      write('Basiswaehrung(Ende mit <ET>): ');
      readln(Eingabe);
      Eingabe[1] := upcase(Eingabe[1]);
      Basis := 0; Ziel := 0;
      IF Eingabe <> '' THEN BEGIN
        Basis := Suchen(Eingabe);
        write('Zielwaehrung(Ende mit  <ET>): ');
        readln(Eingabe);
        IF Eingabe <> '' THEN Ziel := Suchen(Eingabe)
      END;
      Richtig := Basis <> Ziel;
    UNTIL (Eingabe = '') OR Richtig;
    IF Eingabe <> '' THEN BEGIN
      WITH Tabelle[Basis] DO BEGIN
        write('Umtauschbetrag in ',Geld,': ');
        read(Betrag); writeln;
        IF Betrag > 0 THEN
          writeln('Fuer',Betrag:9:2,' ',Geld,' erhalten Sie',
                  Betrag * Tabelle[Ziel].Kurs/Kurs:9:2,' ',
                  Tabelle[Ziel].Geld,' der ',Tabelle[Ziel].Land);
      END;
    END;
  UNTIL Eingabe = '';
  close(Extern)
END;
```

Die Prozedur NUTZEN ruft die Prozedur MENUE, um die in der Währungstabelle vorhandenen Währungen anzuzeigen:

```
PROCEDURE Menue;
VAR i : INTEGER;
BEGIN
  FOR i := 0 TO Umfang -1 DO BEGIN
    IF odd(i + 1) THEN writeln;
    write(Tabelle[i].Geld:20)
  END;
  writeln;
END;
```

Die vordefinierte Prozedur ODD liefert TRUE, wenn das Argument eine ungerade Zahl ist, sonst FALSE. Es wird hier genutzt, um eine zweispaltige Ausschrift der Währungseinheiten zu erzeugen.
Nach dem Start von DEVISEN und bei Aktivierung des Unterprogramms NUTZEN kann das folgende Protokoll entstehen (Eingaben unterstrichen):

```
A) nlegen K) orrigieren der Waehrungstabelle oder
N) utzen  E) nde
A/K/N/E: N<ET>

Devisenumrechnung von Basis- in Zielwaehrung fuer
                Mark                Rubel
                Lewa                Zloty
              Kronen               Forint
                 Lei              Tugriki
Basiswaehrung(Ende mit <ET>): Mark<ET>
Zielwaehrung (Ende mit <ET>): Rubel<ET>
Umtauschbetrag in Mark: 320<ET>
Fuer   320.00 Mark erhalten Sie   100.00 Rubel der UdSSR
.
```

Die Lösung enthält gegenüber der Realität eine Vereinfachung. Erfaßt werden lediglich die Umrechnungskurse zur Mark der DDR (in VERWALTEN). Diese Kurse werden zur Umrechnung anderer Währungen untereinander (in NUTZEN) verwendet. Jeder Kurs wäre gegebenenfalls als Vektor (Feld) in Relation zu jeder anderen Währung zu erfassen. Alle PASCAL-Befehle sind bereits erläutert.

Interne Darstellung als Binärfile

Mit dem Programm DEVISEN soll bei Aktivierung des Unterprogramms VERWALTEN die folgende Währungstabelle angelegt worden sein:

Zyklus	Land	Waehrungseinheit	Kurs
1	DDR	Mark	1.0000
2	UdSSR	Rubel	0.3125
3	VR Bulgarien	Lewa	0.3125
4	VR Polen	Zloty	26.2467
5	CSSR	Kronen	3.0157
6	VR Ungarn	Forint	6.1013
7	VR Rumaenien	Lei	2.5940
8	Mongolische VR	Tugriki	1.3062

Dieses File, es ist mit TABELLE.DAT bezeichnet, soll etwas näher untersucht werden. Zunächst wird das File selbst in seiner internen Darstellung betrachtet. Dabei werden die Inhalte der einzelnen Byte hexadezimal angezeigt. Jedes Byte hat 256 mögliche Zustände, die sich in den Grenzen 0 bis 255 oder hexadezimal □0 bis □FF bewegen.
Sofern sich dieses Byte in den Grenzen von □20 bis □7F befindet, kann es durch ein Zeichen des Zeichensatzes dargestellt werden. Deshalb wird für das File zunächst die hexadezimale und danach die Zeichendarstellung angegeben. Liegt ein Byte außerhalb des Zeichensatzes, erfolgt seine Darstellung durch den Punkt. Man beachte, daß nicht alle

Bytes, die eine Zeichensatzdarstellung ermöglichen, auch ihrer Bedeutung nach Zeichen sind. Der auf diese Art geschriebene Inhalt des Files TABELLE.DAT ist folgender:

```
08 00 1D 00  03 44 44 52  00 00 00 00  00 00 00 00   .....DDR........
00 00 00 04  4D 61 72 6B  00 00 00 81  00 00 00 00   ....Mark........
00 05 55 64  53 53 52 00  00 00 00 00  00 00 00 00   ..UdSSR.........
05 52 75 62  65 6C 00 00  7F 00 00 00  00 20 0C 56   .Rubel....... .V
52 20 42 75  6C 67 61 72  69 65 6E 00  00 04 4C 65   R Bulgarien...Le
77 61 00 00  00 7F 00 00  00 00 20 08  56 52 20 50   wa........ .VR P
6F 6C 65 6E  00.00 00 00  00 00 05 5A  6C 6F 74 79   olen.......Zloty
00 00 85 7F  D9 3D F9 51  04 43 53 53  52 00 00 00   .....=.Q.CSSR...
00 00 00 00  00 00 00 06  4B 72 6F 6E  65 6E 00 82   ........Kronen..
A3 92 3A 01  41 09 56 52  20 55 6E 67  61 72 6E 00   ..:.A.VR Ungarn.
00 00 00 00  06 46 6F 72  69 6E 74 00  83 62 7F D9   .....Forint..b..
3D 43 0C 56  52 20 52 75  6D 61 65 6E  69 65 6E 00   =C.VR Rumaenien.
00 03 4C 65  69 00 00 00  00 82 74 93  18 04 26 0E   ..Lei.....t...&.
4D 6F 6E 67  6F 6C 69 73  63 68 65 20  56 52 07 54   Mongolische VR.T
75 67 72 69  6B 69 81 04  C5 8F 31 27  6C 6F 74 79   ugriki....1'loty
00 00 85 7F  D9 3D F9 51  04 43 53 53  52 00 00 00   .....=.Q.CSSR...
```

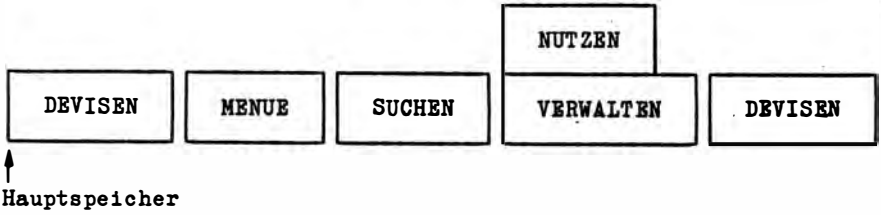
Die genaue Struktur eines solchen Files ist implementationsabhängig. Hier wurden an den Anfang des Files noch Informationen über den Umfang der Filekomponenten und deren Anzahl eingetragen. In diesem Falle fehlt ein EOF-Zeichen. Wichtig ist, und das ist besonders an den Zahlen zu erkennen, daß der Inhalt der Speicherworte unverändert in das File übernommen wurde. Wäre das anders, müßte zum Beispiel für den Umrechnungskurs des Rubel die Bytefolge 30 2E 33 31 32 35 im File stehen. Das entspricht nämlich den Zeichen des Zeichensatzes für "0.3125". Es wurde also nicht in die Zeichendarstellung konvertiert, sondern die binäre Darstellung aus dem Hauptspeicher übernommen. Deshalb nennt man solche Files Binärfiles. Der Verzicht auf Konvertierung der Daten, die dem Anwender ohnehin nicht direkt zugänglich sind, vermeidet einen Zeitverlust.
Abschließend soll noch die Eintragung in die Directory betrachtet werden. Sie ist natürlich für ein und dasselbe File abhängig vom konkreten Inhalt der Diskette, weil sie auf irgendeinen gerade freien Aufzeichnungsblock untergebracht wird. Eine Möglichkeit ist folgende:

```
00 54 41 42  45 4C 4C 45  20 44 41 54  00 00 00 02   .TABELLE DAT....
0A 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
```

Man sieht, daß das File mit 2 Sätzen zu je 128 Byte in dem Aufzeichnungsblock ¤0A untergebracht ist. Alle anderen Elemente der Directory wurden in Abschnitt 6.1. erläutert.

OVERLAY-Struktur

Zu beachten ist noch die Überlagerungsstruktur des Gesamtprogramms DEVISEN, denn die Unterprogramme VERWALTEN und NUTZEN tragen das Attribut OVERLAY. Die grundsätzliche Wirkung von OVERLAY wurde im Abschnitt 5.4. beschrieben. Hier ergibt sich die auf Seite 110 dargestellte Struktur.
VERWALTEN und NUTZEN bilden eine Überlagerungsgruppe. Der für sie reservierte Platz im Hauptspeicher ist nur so groß wie das größte der beiden Unterprogramme. Hier wird der gesamte Speicherplatz für das kleinere NUTZEN gespart. VERWALTEN und



NUTZEN können sich nicht gegenseitig rufen. Außerdem entsteht Zeitverzug, weil das aktivierte OVERLAY-File erst von der Diskette geholt werden muß. Da sich die beiden Unterprogramme vom Problem her ausschließen, ist dieser Nachteil kaum spürbar. Allerdings würde der Hauptspeicher auch noch für NUTZEN ausreichen. Die Überlagerungsstruktur als Ganzes dient also nur der Demonstration.

6.3. Ungetypte Files

Es gibt in PASCAL auch die Möglichkeit, Files zu schreiben und zu lesen, ohne sich hinsichtlich der Struktur festzulegen. Ein solches Erfordernis liegt zum Beispiel vor, wenn von einem File eine Sicherheitskopie anzufertigen ist. Der Typ der Filekomponenten ist dabei völlig nebensächlich. Das folgende Programm zeigt den Verfahrensweg:

```
PROGRAM Kopieren;
{Kopieren von Files beliebigen Typs}
VAR Quellfile,
    Zielfile : FILE;
    Filename : STRING[12];
    Laufwerk : CHAR;
    Puffer   : ARRAY[1..1024] OF BYTE;
BEGIN
  writeln('Kopieren eines Files');
  write('Filename: ');
  readln(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.
```

ungetyptes FILE

Bei der Vereinbarung der Filevariablen ist also nur das Schlüsselwort FILE zu benutzen. Der Zugriff über BLOCKREAD und BLOCKWRITE erfolgt mit dem Vielfachen von Sätzen zu 128 Byte. Die Syntax für diese beiden vordefinierten Prozeduren ist nicht einheitlich. Hier enthalten sie neben dem Filenamen die als Puffer benutzte Variable und die Anzahl der mit jeder Operation zu transportierenden 128-Byte-Sätze. Natürlich muß mit der Varia-

blen so viel Speicherplatz bereitgestellt werden, daß die mit einer Operation gelesenen Bytes untergebracht werden können. Im Beispiel KOPIEREN werden jeweils 8 Sätze zu je 128 Byte (gleich 1024 Byte) mit einer Operation gelesen oder geschrieben. Deshalb wurde das Feld PUFFER in dieser Größe angelegt. Wichtig ist noch, daß ungetypte Files keinen Sektorpuffer benötigen. Bei allen Operationen, die nicht den Inhalt des Files berühren, wird deshalb durch eine solche Vereinbarung Speicherplatz gespart. Das trifft zum Beispiel zu, wenn ein File umzubenennen ist. Dafür steht die vordefinierte Prozedur RENAME zur Verfügung. Die Vorgehensweise zeigt der folgende Programmausschnitt:

```
.
VAR x : FILE;
BEGIN
assign(x,'ALT.DAT');
rename(x,'NEU.DAT');
reset(x);
.
```

Man beachte, daß RENAME vor dem Eröffnen des Files durch RESET aufgerufen wird. Nach der Eröffnung ist das Umbenennen nicht erlaubt. Es ist zu sichern, daß der neue Filename auf der Diskette noch nicht existiert. Das kann geprüft werden, wenn die Ein- und Ausgabeüberwachung des Systems mit {¤I−} ausgeschaltet und mit dem neuen Namen eine Eröffnung durch RESET versucht wird. Ist danach IORESULT gleich Null, so existiert das File bereits.
Ein weiterer Vorgang, für den sich ungetypte Files als günstig erweisen, ist das Löschen von Files. Dafür steht die vordefinierte Prozedur ERASE zur Verfügung. Ihre Anwendung zeigt der folgende Ausschnitt:

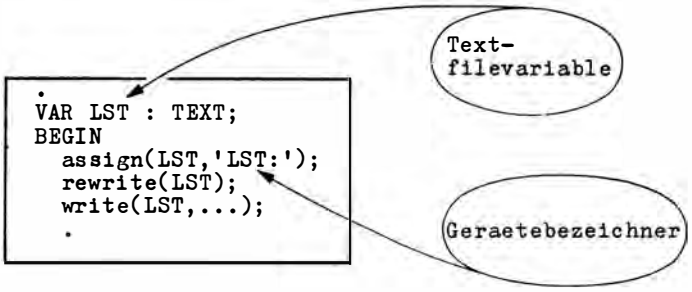
```
.
VAR x : FILE;
BEGIN
  assign(x,'BEISPIEL.DAT');
  erase(x);
.
```

Sollte das File mit RESET oder REWRITE bereits eröffnet sein, muß es vor ERASE mit CLOSE geschlossen werden.
Sowohl RENAME als auch ERASE werden direkt am Disketten-FCB vorgenommen. Deshalb darf zu diesem Zeitpunkt kein Speicher-FCB für dieses File existieren. RENAME und ERASE sind also unbedingt an geschlossenen Files auszuführen.

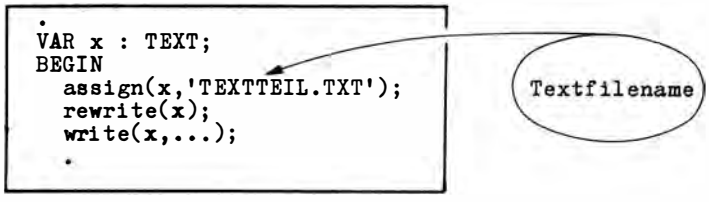
6.4. Textfiles

Textfiles sind Files aus Elementen des gültigen Zeichensatzes. Ist der Zeichensatz (einschließlich Steuerzeichen) von ¤0 bis ¤7F definiert, so liegen die Werte aller Bytes in diesem Bereich. Nicht alle Bytes repräsentieren dabei druckbare Zeichen. Textfiles, soweit es Gerätefiles sind, wurden bereits im Abschnitt 2.3. behandelt. Dort wurde zum Beispiel der Bezeichner LST benutzt. LST ist eine vordefinierte Filevariable vom Typ TEXT.

Man kann das prüfen, denn die folgenden Befehle erzielen die gleiche Wirkung:



Benutzt man hier den Bezeichner DRUCKER, so kann man in allen folgenden WRITE-Prozeduren "DRUCKER" statt wie vordefiniert "LST" verwenden. Als Gerätebezeichner stehen auch 'KBD:', 'CON:', 'TRM:' und weitere zur Vefügung (Großschreibung beachten). Wenn in der jeweiligen PASCAL-Implementation keine Textfilevariablen vordefiniert sind, kann das so ohne Schwierigkeiten nachgeholt werden. Die Wirkung ist dann genauso, wie in Abschnitt 2.4. beschrieben.
Nicht behandelt wurde im Abschnitt 2.4. das Schreiben und Lesen von Textfiles auf bzw. von der Diskette, denn vordefinierte Bezeichner waren immer fest mit den Gerätefiles der Ein- und Ausgabe verbunden. Das Speichern von Files in Textform hat aber durchaus praktische Bedeutung. Sollte der Filetyp mit CHAR oder STRING vereinbart sein, dann ist das File seinem gesamten Inhalt nach vom Typ TEXT. Es kann aber auch möglich sein, daß Ausgaben in Textform auf den externen Speicher gebracht werden sollen, um sie zu einer anderen Zeit durch ein einfaches Druckprogramm auszuschreiben. Bei druckintensiven Programmen kann dadurch CPU-Zeit gewonnen werden. Textfiles verschiedener PASCAL-Systeme sind kompatibel, so daß sie für den Filetransfer zwischen den Systemen genutzt werden können.
Sind Textfiles ihrem Inhalt nach nicht bereits ausschließlich Zeichen des Zeichensatzes, so erfolgt bei der Ausgabe als Textfile eine Konvertierung. Natürlich kostet das Zeit. Die Vorgehensweise beim Schreiben und Lesen von Diskettentextfiles ist einfach:



Für den Gerätebezeichner bei Gerätefiles steht hier der Name des Datenbestandes auf der Diskette. Existiert das File bereits, ist mit RESET zu eröffnen.
Für Textfiles gelten einige Besonderheiten. Die wichtigste ist, daß sie keine Filekomponenten gleicher Länge besitzen wie getypte Binärfiles. Die Komponenten eines Textfiles sind Zeilen verschiedener Länge. Die Zeilen werden durch Steuerzeichen (□D □A) getrennt. Bei der Ausgabe auf ein Gerät werden diese Zeichen wirksam und sorgen für den Zeilenvorschub. Weil keine gleich großen Filekomponenten existieren, lassen sich vorde-

finierte Prozeduren wie SEEK, FILEPOS, FILESIZE nicht ausführen. Textfiles sind deshalb immer sequentiell organisiert und können auch nur so gelesen und geschrieben werden.

Übungsaufgaben

1. Trennen Sie im Programm DEVISEN das Korrigieren der Währungstabelle vom Erweitern, so daß das Erweitern nutzerfreundlicher wird! Dabei darf die Hauptspeicherbelegung nicht vergrößert werden.

2. Ändern Sie das Programm VERZEICHNIS aus Kapitel 5 so, daß die sortierten Daten als Textfile abgelegt werden!

3. Entwickeln Sie ein Programm, das Ihnen den Inhalt beliebiger Files sektorweise auf dem Bildschirm anzeigt! Nichtdarstellbare Zeichen sind als Punkt auszugeben!

7. Anwendung spezieller Datenstrukturen

7.1. Mengen und Operationen mit Mengen

In vielen Anwendungen sind zusammen mit Zahlen und Zeichen Attribute zu verarbeiten und zu speichern, zum Beispiel für die Disposition von Textilwaren das Material (Polyester, Baumwolle, Wolle, Leinwand), die Musterung (Glencheck, Fischgrat, Karo), die Farbe (Grau, Braun, Blau, Beige) und ein Saisonmerkmal (Sommer, Winter). Natürlich wäre dieses Problem mit den bisher behandelten programmtechnischen Mitteln lösbar. Jedes Attribut könnte als Zeichenkette erfaßt oder ihm eine Zahl zugeordnet werden. Sucht man dann Artikel mit bestimmten Attributen, so benutzt man mehrere geschachtelte IF-Anweisungen.
Nimmt die Zahl der Attribute zu (für die praktische Anwendung bei Textilwaren sind mehr als dreißig erforderlich), wird diese Verfahrensweise aufwendig, fehleranfällig und auch speicherplatzintensiv. PASCAL stellt deshalb einen speziellen Datentyp, den SET- oder Mengentyp, bereit.

Mengen
Mengen in PASCAL sind wie in der Mathematik Zusammenfassungen von Objekten. Sie werden als Einheit betrachtet, mit der genau definierte Operationen möglich sind. Die Objekte selbst heißen Elemente. Werden Mengen explizit (wie eine Konstante dargestellt), so fordert PASCAL, daß sie in eckige Klammern eingeschlossen werden. Besitzt ein Artikel die Attribute Wolle, Karo, Braun – so ist die Mengenschreibweise

[Wolle, Karo, Braun].

Natürlich müssen solche Konstanten vorher vereinbart sein, so daß die möglichen Attribute durch eine TYPE-Definition einzuführen wären. Sind die Elemente der Menge vordefiniert, so ist das nicht erforderlich. Beispiele sind:

['a'..'z']	Menge der Kleinbuchstaben
['A'..'Z']	Menge der Großbuchstaben
['a','e','i','o','u']	Menge der Vokale (als Kleinbuchstaben)
[2, 4, 8, 16, 32]	Menge der Zweipotenzen bis 2^5.

Man beachte die Möglichkeit, aufeinanderfolgende Elemente bereichsweise mit ".." anzugeben. Da mit Hilfe solcher Mengenkonstanten die späteren Mengenvariablen zu konstruieren sind, werden Mengenkonstanten auch als Mengenkonstruktoren bezeichnet.

Mengen sind ungeordnete Zusammenfassungen, deshalb ist die Reihenfolge der Elemente ohne Einfluß auf den Wert des Ausdrucks. Der Mengenkonstruktor

['i','a','u','e','o']

beschreibt ebenso die Menge der Vokale wie der Mengenkonstruktor

['a','e','i','o','u'].

Die Wirkung der beiden Ausdrücke ist in jeder Beziehung gleich.
Auch die Häufigkeit, mit der ein Element in der Menge aufgezählt wird, ist für den Wert ohne Belang. Der Mengenkonstruktor

['a','e','i','o','u']

ist völlig gleichwertig mit dem Mengenkonstruktor

['a','e','i','a','i','u','a','o'].

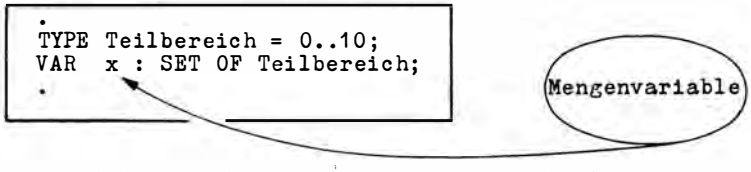
Natürlich muß PASCAL auch Variablen bereitstellen, die solche Mengen als Ganzes aufnehmen können. Dabei gibt es gegenüber der Mathematik die Einschränkung, daß alle Elemente einer Menge vom gleichen Typ sein müssen. Dieser Typ wird als Grund- oder Basistyp der Menge bezeichnet. Der Basistyp unterliegt Einschränkungen. Das sind folgende:

a) Strukturierte Typen sind nicht erlaubt, also werden Strings, Felder, Records, Files oder natürlich Mengen selbst nicht als Elemente zugelassen.
b) Der Datentyp REAL ist nicht erlaubt, da nur endliche Mengen behandelt werden können.

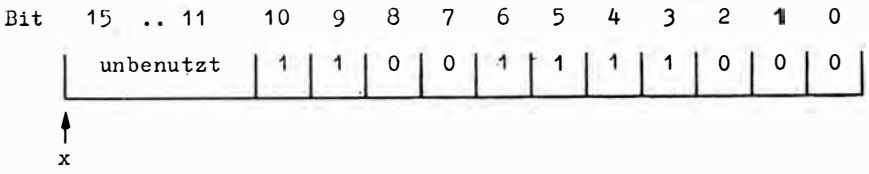
Also bleiben als mögliche Basistypen einfache, ordinale Datentypen. Für alle gegenwärtigen PASCAL-Systeme auf Mikrorechnern ist außerdem die Anzahl der Elemente mit 256 begrenzt.

Mengenvariablen

Die Vereinbarung einer Variablen, die eine Menge aufnehmen kann, geschieht mit den Schlüsselworten "SET OF":



Natürlich hätte auch direkt "SET OF 1..10" geschrieben werden können. Für die praktische Anwendung sind vor allem Fälle von Bedeutung, in denen der Basistyp ein Aufzählungstyp ist. Dazu folgt später ein Beispiel. Hier ist zunächst eine Mengenvariable definiert, die als Elemente Zahlen im Bereich 0 bis 10 aufnehmen und als Gesamtheit verarbeiten kann.
Typisch für Mikrorechner ist, daß je Element 1 Bit – gerundet auf volle Byte – bereitgestellt wird. Jedes Bit repräsentiert das Enthaltensein des zugeordneten Elements (Bit gesetzt) oder dessen Nichtenthaltensein (Bit nicht gesetzt). Im Beispiel wird folgender Speicherplatz reserviert:



Im Hauptspeicher steht wie bei INTEGER das niederwertige Byte zuerst. Die dargestellte Menge enthält die Elemente 3, 4, 5, 6, 9 und 10. Bei einer Begrenzung auf 256 Elemente belegt eine Mengenvariable maximal 32 Byte.
Eine Mengenvariable ist zu Beginn der Programmausführung bezüglich ihres Inhalts – wie es jede andere Variable auch sein kann – undefiniert. Ihr kann zugewiesen werden

a) eine Mengenkonstante in Form eines Mengenkonstruktors, zum Beispiel

   x := [3..6, 9, 10]

   Hier steht Mengenvariable gegen Mengenkonstante, also Menge gegen Menge. Die Variable x wird mit den Elementen 3, 4, 5, 6, 9 und 10 geladen. Intern erhält die Mengenvariable die bereits dargestellte Belegung. Man beachte, daß die Zuweisung

   x := 2

   nicht typverträglich ist, denn links steht eine Mengenvariable und rechts eine Zahl. Typgerecht wäre x := [2];

b) eine leere Menge mit dem Symbol "[]". Da jede Menge leer sein, also kein Element enthalten kann, ist "[]" verträglich mit jeder Mengenvariablen. Die Wirkung ist, daß alle Bits, die die Elemente repräsentieren, Null gesetzt werden. Im Beispiel belegt x := []

   die Mengenvariable mit 0000 0000 0000 0000;

c) der Inhalt einer anderen Mengenvariablen, zum Beispiel

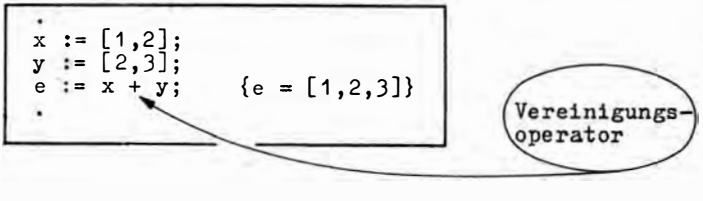
   x := y;

   y muß eine Mengenvariable vom gleichen Typ sein wie x. In diesem Falle wird die Bitbelegung der Variablen y auf den für x reservierten Speicherplatz übertragen.

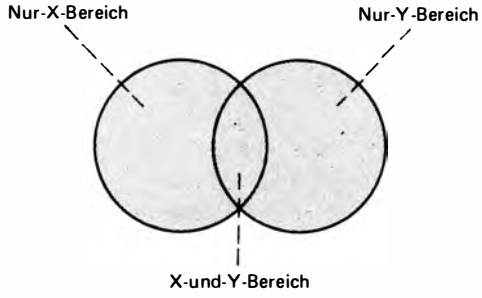
Mengenkonstanten (Mengenkonstruktoren) und Mengenvariablen können auf verschiedene Weise verbunden werden. Dazu stehen die Operatoren +, − und * mit mengenspezifischer Wirkung (Mengenoperatoren) zur Verfügung. Voraussetzung ist natürlich stets die Typverträglichkeit der Operanden.

Mengenoperatoren

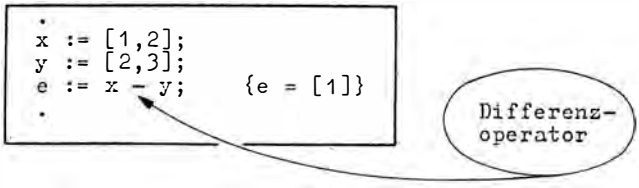
Der Mengenoperator "+" bewirkt die Vereinigung zweier Mengen. Die Programmausführung



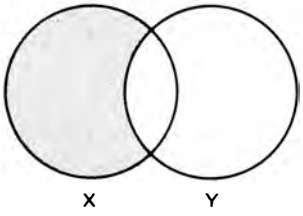
ergibt für "e" den Wert [1, 2, 3]. Eine solche Vereinigungsmenge enthält die Elemente, die in "x" oder in "y", also in der einen oder anderen Menge vorkommen. Grafisch kann das für zwei Mengen in drei Bereichen dargestellt werden, das sind hier der Nur-X-Bereich, der Nur-Y-Bereich und der X-und-Y-Bereich:



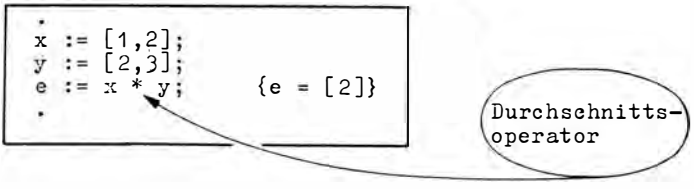
Die Vereinigungsmenge "e" umfaßt alle drei Bereiche. Die Vereinigung der Attributsmengen [Polyester, Wolle] und [Wolle, Leinwand] liefert die Attributsmenge [Polyester, Wolle, Leinwand]
Der Operator "−" bewirkt eine Mengendifferenz. Die Ausführung der Anweisungen



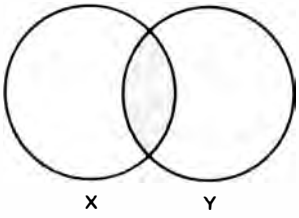
ergibt für "e" den Wert [1]. Die Mengendifferenz enthält nur die Elemente, die in "x", nicht aber in "y" enthalten sind. Grafisch wird das so dargestellt:



Die Mengendifferenz "e" wurde hervorgehoben. Sie besteht aus dem Nur-X-Bereich. Angewendet auf das Beispiel der Artikelattribute mit den Mengen [Polyester, Wolle] und [Wolle, Leinwand] ergibt sich als Mengendifferenz [Polyester].
Schließlich bewirkt der Mengenoperator "*" die Bildung einer Durchschnitts- oder Produktmenge. Nach Ausführung der Anweisungen



ist der Wert der Variablen "e" gleich der Durchschnittsmenge [2]. Die Durchschnittsmenge enthält alle Elemente, die in "x" und in "y" vorkommen. Grafisch bedeutet das



Die Durchschnittsmenge "e" wurde wieder hervorgehoben. Sie enthält den X- und Y-Bereich. Der Durchschnitt der Artikelattribute [Polyester, Wolle] und [Wolle, Leinwand] wäre also [Wolle].

Vergleiche von Mengen

Mengen können miteinander verglichen werden. Dabei existieren die Vergleichsrelationen =, < >, >=, <= (< und > sind für Mengen nicht definiert).
Mengen sind gleich, wenn sie genau dieselben Elemente enthalten – in jedem anderen Falle sind sie ungleich. Dabei wird vernachlässigt, wie oft dasselbe Element in der Menge enthalten ist und in welcher Reihenfolge die Elemente aufgezählt werden. Demnach sind die Mengen

[1, 2, 3]
[2, 3, 1]
[3, 2, 1]
[1, 1, 2, 2, 3, 3]

alle gleich. Ebenso sind es die Mengen

[Polyester, Baumwolle]
[Baumwolle, Polyester]
[Baumwolle, Polyester, Baumwolle].

Eine Menge kann Elemente enthalten, die sämtlich auch in einer anderen Menge enthalten sind. Die erste Menge ist in diesem Falle Teilmenge der zweiten Menge. Das wird in PASCAL durch die Teilmengenrelationen < = und > = ausgedrückt, wobei die Spitze des Ungleichheitszeichens auf die Teilmenge zeigt. Im Grenzfall ist eine Teilmenge auch gleich der Menge (die Mathematik unterscheidet deshalb echte und unechte Teilmengen). Für die folgenden Beispiele ist der Mengenvergleich jeweils TRUE:

[1, 2, 1] <= [1, 2, 3]
[2, 1] <= [1, 2, 3]
[] <= [1, 2, 3]
[1, 3, 2] >= [2]
[1, 2] <= [1, 2].

Für eine leere Menge [] ist die "<="-Relation immer erfüllt. Ebenso gilt TRUE für

[Wolle, Polyester] <= [Polyester, Wolle]
[Polyester] <= [Polyester, Baumwolle, Wolle].

Ist die Menge, auf die die Spitze des Ungleichheitszeichens zeigt, nicht vollständig in der Vergleichsmenge enthalten, wird das Ergebnis des Vergleichs FALSE. Das trifft für die folgenden Beispiele zu

```
[1, 2, 3]  >= [1, 4]
[1]        <= [2, 3].
```

Beim letztgenannten Beispiel beachte man, daß sich zwei Mengen gegenüberstehen. Für die Prüfung, ob ein einzelnes Element in einer Menge enthalten ist oder nicht, wäre dieser Vergleich umständlich. Das Element müßte jedesmal einer Menge zugewiesen und dann die Prüfung durchgeführt werden. Zur Vereinfachung dieses Vorgangs stellt PASCAL den Enthaltenseins-Operator IN zur Verfügung. Seine Anwendung zeigt der folgende Programmausschnitt:

```
.
VAR Alphabet : SET OF 'A' .. 'z';
Zeichen : CHAR;
BEGIN
  Alphabet := ['A' .. 'Z'];
  read(Zeichen);
  IF Zeichen IN Alphabet THEN write('Grossbuchstabe');
.
```

Enthaltenseins-operator

Die Bedingungsprüfung der IF-Anweisung liefert TRUE, wenn ein Großbuchstabe eingegeben wurde. Der IN-Operator kann auch dann eine vorteilhafte Schreibweise ermöglichen, wenn nicht mit Mengen gearbeitet wird. Das zeigt das folgende Beispiel:

```
.
VAR x : CHAR;
BEGIN
  REPEAT
    writeln('Bestaetigen Sie (J/N): ');
    read(KBD,x);
  UNTIL x IN ['J','j','N','n'];
.
```

Der Nutzer wird zur Antwort "Ja" oder "Nein" gezwungen. Übrigens ist durch die Verwendung von CHAR für x und KBD in READ keine Endetaste erforderlich.
Vor einer Anwendung der eleganten IN-Operation bei zeitkritischen Abläufen muß gewarnt werden. Wie man an der internen Darstellung von Mengen erkennt, erfolgen Mengenoperationen auf Bitebene. Der Zugriff verzögert sich dadurch.

7.2. Die Verarbeitung von Mengen

Die praktische Anwendung des Mengentyps von PASCAL wird nun am Beispiel gezeigt. Das Programm DISPOSITION mit den Unterprogrammen AUSWAHL, ANZEIGE und VERKAUF unterstützt einen Absatz- oder Verkaufsdisponenten für Herrenoberbekleidung (HOB). Anhand von Kundenwünschen hinsichtlich Material, Musterung, Farbe und Sai-

sonmerkmal wird das gesamte Sortiment nach Artikeln durchsucht, die die vorgegebenen Eigenschaften je nach Fragestellung entweder besitzen oder nicht besitzen. Sie werden mit Bestellnummer, Bezeichnung, Bestand und Einzelpreis angezeigt und zum Verkauf angeboten. Beim Verkauf wird eine Liste geschrieben und der Bestand im File geändert. Es wurde der folgende Entwurf benutzt:

```
PROGRAM Disposition;

BEGIN
{ 1. Eroeffnung der Datei                          }
{ 2. Wiederholung bis zur Ablehnung                }
{ 2.1 Eroeffnungsbild                              }
{ 2.2 Angebot und Eingabe der Auswahlkriterien     }
{ 2.3 Durchmustern bis zum Fileende                }
{ 2.3.1 Lesen einer Filekomponente                 }
{ 2.3.2 Pruefen auf Auswahl                        }
{ 2.3.2.J Verkaufsdisposition                      }
{ 2.3.2.J.1 Anzeige der Artikeldaten               }
{ 2.3.2.J.2 Verkaufsangebot                        }
{ 2.3.2.J.2.J Verkauf und Filekorrektur            }
{ 2.3.2.N Lesen der naechsten Filekomponente       }
{ 2.3.3 Fehlmeldung - wenn erforderlich            }
{ 2.4 Fortsetzungsangebot und Eingabe              }
{ 3. Schliessen des Files und Endemitteilung       }
END.
```

Dieser Entwurfstext wurde nicht in den Programmtext übernommen.
Das Programm wird zunächst nur hinsichtlich der Vereinbarung der Mengenvariablen und der Bildung (dem Laden) der Menge betrachtet:

```
PROGRAM Disposition;
{Unterstuetzung der Verkaufsdisposition}
TYPE Attribut = (Polyester,Baumwolle,Wolle,Leinwand,Glencheck,
                 Fischgrat,Karo,Ohne_Muster,Grau,
                 Braun,Blau,Beige,Sommer,Winter);
     Charakteristik = SET OF Attribut;
     Artikeldaten   = RECORD
       BNR          : ARRAY[1..6] OF CHAR;
       Bezeichnung  : STRING[30];
       Gebrauchswert: Charakteristik;
       Bestand      : INTEGER;
       Preis        : REAL
     END;
     Kette          = STRING[40];
VAR  Merkmale          : Charakteristik;
     Artikel           : Artikeldaten;
     Daten             : FILE OF Artikeldaten;
     Eingabe           : STRING[4];
     Gefunden,Zustimmung: BOOLEAN;
{¤I ANTWORT.BIB}
{¤I AUSWAHL.DIS}
{¤I ANZEIGE.DIS}
{¤I VERKAUF.DIS} {oder ¤I SIMULAT.DIS}
BEGIN
  assign (Daten,'HOB.ART');
  REPEAT
    reset(Daten);
    writeln('Verkaufsdisposition Herrenoberbekleidung');
    writeln('nach Material,Musterung und Farbe');
```

Mengentyp

Mengenvariable

```
    Zustimmung := Antwort('Auswahl mit/ohne Merkmale ','M','O');
    Auswahl; writeln;
    Gefunden := FALSE;
    WHILE NOT eof(Daten) DO BEGIN
      read(Daten,Artikel);
      WITH Artikel DO BEGIN
        IF ((Zustimmung) AND (Merkmale <= Gebrauchswert)) OR
        ((NOT Zustimmung) AND (Merkmale * Gebrauchswert = []))
        THEN BEGIN
          writeln(BNR,' ',Bezeichnung,' Bestand ',Bestand,
                  ' Stueck',' a'' ',Preis:6:2,' M');
          Anzeige; writeln;
          Verkauf; {oder Verkaufsimulation}
          Gefunden := TRUE;
        END;
      END;
    END;
    IF NOT Gefunden THEN writeln('Artikel nicht vorraetig');
    write('Fortsetzung mit <ET>');
    readln(Eingabe);
  UNTIL Eingabe <> '';
  close(Daten);
  write('Ende');
END.
```

Um festzustellen, ob der Nutzer eine Auswahl von Artikeln mit noch zu definierenden Eigenschaften oder ohne diese Eigenschaften wünscht, wird das Unterprogramm ANTWORT benutzt:

```
FUNCTION Antwort(Frage:Kette;j,n:CHAR):BOOLEAN;
VAR x       :CHAR;
    Richtig:BOOLEAN;
BEGIN
  REPEAT
    write(Frage,' (',j,'/',n,'): ');
    read(KBD,x);
    x := upcase(x);
    Richtig:= (x = j) OR (x = n);
    IF NOT Richtig THEN write(chr(7));
    writeln
  UNTIL Richtig;
  Antwort := x = j;
END;
```

ANTWORT sichert, daß keine anderen als die vorgegebenen Antworten gegeben werden können, und liefert TRUE, wenn die erste der Antworten benutzt wurde. Fehler werden mit akustischem Signal quittiert. Nun zum eigentlichen Gegenstand der Betrachtung.

Vereinbarung der Mengenvariablen

DISPOSITION verwendet als Basistyp einen Aufzählungstyp, der verschiedene Eigenschaften der Ware hinsichtlich Material, Musterung, Farbe und Saison beschreibt. Für die praktische Anwendung wären sicher weitere Eigenschaften zu betrachten. Die Erweiterung bereitet keine Probleme. Dieser Aufzählungstyp wird zum Basistyp der Mengenvariablen. Der Mengentyp "Charakteristik" wird im Vereinbarungsteil noch benutzt, um den RECORD-Teilbezeichner "Gebrauchswert" zu definieren. Das geschieht, weil ein File mit Artikeldaten zu verwalten ist, das Angaben über die Eigenschaften der Ware enthält.

Außerdem ist "Merkmale" eine Mengenvariable. Sie wird der Arbeit im Hauptspeicher dienen und erst kurz vor dem Schreiben eines Records in das File "HOB.DAT" (HOB-Herrenoberbekleidung) der Mengenvariablen "Artikel.Gebrauchswert" zugewiesen. DISPOSITION geht davon aus, daß das File HOB.DAT bereits existiert.

Belegung der Mengenvariablen

Die Belegung der Mengenvariablen "Merkmale" erfolgt im Unterprogramm AUSWAHL:

```
PROCEDURE Auswahl;
VAR Material,Musterung : CHAR;
    Parameter          : STRING[14];
    i                  : INTEGER;
    m                  : Charakteristik;{verkuerzt Schreibweise
                                         fuer "Merkmale"}
BEGIN
  writeln;
  writeln('A= Polyester B= Baumwolle C= Wolle  D= Leinwand');
  writeln('E= Glencheck F= Fischgrat G= Karo   H= Ohne Muster');
  writeln('I= Grau      J= Braun     K= Blau   L= Beige');
  writeln('S= Sommer    W= Winter');
  write('A/AB/BC...: ');
  readln(Parameter);
  m := [];
  FOR i := 1 TO length(Parameter) DO BEGIN
    Parameter[i] := upcase(Parameter[i]);
    CASE Parameter[i] OF
      'A': m := m + [Polyester];    'H': m := m + [Ohne_Muster];
      'B': m := m + [Baumwolle];    'I': m := m + [Grau];
      'C': m := m + [Wolle];        'J': m := m + [Braun];
      'D': m := m + [Leinwand];     'K': m := m + [Blau];
      'E': m := m + [Glencheck];    'L': m := m + [Beige];
      'F': m := m + [Fischgrat];    'S': m := m + [Sommer];
      'G': m := m + [Karo];         'W': m := m + [Winter]
    END;
  END;
  Merkmale := m
END;
```

Belegung der Mengenvariablen

Da Mengen nicht direkt eingegeben werden können, wird in AUSWAHL der String "Parameter" zur Aufnahme des Eingabetextes bereitgestellt. Eigentlich wäre hier noch Aufwand erforderlich, um fehlerhafte Eingaben zu vermeiden. Während die Materialarten Polyester, Baumwolle, Wolle, Leinwand und die Farben kombinierbar sind, schließen sich andere Eigenschaften aus (zum Beispiel Fischgrat – Glencheck). Das müßte geprüft und Fehler müßten angezeigt werden. Sicher sollte für die gesamte Eingabe auch eine Kontrollanzeige mit Bestätigung vorgesehen werden. Man beachte das Zuweisen der leeren Menge für "m" (als Zwischenspeicher für "Merkmale") und das zeichenweise Umsetzen des Strings "Parameter" durch die FOR-Anweisung, die für ein wiederholtes Benutzen der CASE-Anweisung sorgt. Bei jedem Durchlauf der CASE-Anweisung wird ein Element in die Mengenvariable "m" und später "Merkmale" aufgenommen, wenn einer der gültigen Buchstaben eingegeben wurde.

Der prüfende Vergleich

Die eigentliche Prüfung, ob der Artikel ausgewählt wird oder nicht, erfolgt im Hauptprogramm DISPOSITION. Dabei sind die Kundenwünsche Elemente der Menge "Merkmale"

und die Eigenschaften der vorrätigen Artikel Elemente der Menge "Artikel.Gebrauchswert". Es sind dann zwei verschiedene, sich ausschließende Vergleiche programmiert. Welcher dieser Vergleiche für die Auswahl wirksam wird, richtet sich nach der Antwort auf die Frage "Auswahl mit/ohne Merkmale (MIO):"
Der eine Vergleich wird programmiert für den Fall, daß der Nutzer Artikel wünscht, die die definierten Merkmale enthalten (Antwort = 'M', Zustimmung = TRUE), also wenn zum Beispiel

[Polyester, Wolle, Glencheck, Grau]

eingegeben wurde und Artikel mit diesen Eigenschaften gesucht sind. Mengentheoretisch ausgedrückt, sind jene Artikel auszuwählen, in deren Menge "Artikel.Gebrauchswert" die Menge "Merkmale" enthalten ist. Das ist die Prüfung

"Merkmale <= Artikel.Gebrauchswert".

Wie in Abschnitt 7.1. dargelegt, werden nicht nur Artikel ausgewählt, die genau diese Eigenschaften besitzen. Das entspräche der Prüfung

Merkmale = Artikel.Gebrauchswert.

Die mit <= ausgewählten Artikel können also weitere Eigenschaften besitzen.

Der andere Vergleich soll ausgewählte Merkmale nicht enthalten (Antwort <> 'M', Zustimmung = FALSE).
Wird

[Sommer, Winter]

gewählt, dann sollen Artikel ohne diese Eigenschaften (saisonunabhängig) ausgewählt werden. Bei der Auswahl

[Polyester]

werden alle Artikel gesucht, die dieses Material (Polyesterseide und Polyesterfaser) nicht enthalten. Mengentheoretisch ist zu fordern, daß die Durchschnittsmenge zwischen "Merkmale" und "Artikel.Gebrauchswert" (der X-und-Y-Bereich) eine leere Menge sein muß. Die Prüfung lautet deshalb

"Merkmale * Artikel.Gebrauchswert = []".

Welche Eigenschaften die ausgewählten Artikel insgesamt besitzen, ist nach der Auswahl nicht bekannt, da in beiden Fällen nur Teilforderungen erhoben werden. Deshalb ist mit dem Unterprogramm ANZEIGE eine Ausgabe vorgesehen (vgl. S. 124, oben).

Die Schwierigkeiten der Ausgabe von Werten des Aufzählungstyps sind bereits aus dem Abschnitt 3.3. bekannt. ANZEIGE nutzt in der CONST-Definition eine spezielle Form der Anfangswertzuweisung. Der Bezeichner "Settext" repräsentiert nach der Definition eine strukturierte Variable, also keine Konstante, obwohl die Vereinbarung im CONST-Teil erfolgt. Diese Variable erhält beim Compilieren den nach dem Gleichheitszeichen aufgeführten Wert. Da es sich um eine strukturierte Variable handelt, geschieht das komponentenweise.


```
PROCEDURE Anzeige;
CONST Settext : ARRAY[0..13] OF STRING[11]
              = ('Polyester','Baumwolle','Wolle','Leinwand',
                 'Glencheck','Fischgrat','Karo','Ohne Muster',
                 'Grau','Braun','Blau','Beige',
                 'Sommer','Winter');
VAR i : Attribut;
    z : INTEGER;
BEGIN
  z := 0;
  FOR i := Polyester TO Winter DO BEGIN
    IF i IN Artikel.Gebrauchswert THEN BEGIN
      z := succ(z);
      write(Settext[ord(i)],' ');
    IF z MOD 5 = 0 THEN writeln;
    END
  END;
END;
```

Die aufgezählten Werte werden in Klammern gesetzt und durch Komma getrennt. Variablen dieser Art werden als typisierte Konstanten bezeichnet. Ist die Variable nichtstrukturiert, entfallen die Klammern, und es wird nur ein Wert geschrieben, zum Beispiel

```
.
CONST x : INTEGER = 5;
      z : REAL    = 0.5;
.
```

Typisierte Konstanten können im Programm wie Variablen belegt und verwendet werden. Es ist aber zu beachten, daß der Compiler eine einmalige Anfangswertzuweisung vornimmt, die dann im COM/CMD-File oder in einer OVERLAY-Struktur auf der Diskette gespeichert ist. Die Veränderung typisierter Konstanten wird also erst beim Neustart des

```
PROCEDURE Verkauf;
VAR Eingabe : BOOLEAN;
    Umsatz  : INTEGER;
BEGIN
  Eingabe := Antwort('Verkauf ','J','N');
  IF Eingabe THEN BEGIN
    write('Verkaufsmenge: ');
    read(Umsatz); writeln;
    IF Artikel.Bestand >= Umsatz THEN
      Artikel.Bestand := Artikel.Bestand - Umsatz
    ELSE Umsatz := Artikel.Bestand;
    write(LST,'Verkauf: ',Artikel.Bezeichnung:31,' ',Umsatz,' a'' ',
          Artikel.Preis:5:2,' = ',Umsatz * Artikel.Preis:8:2);
    seek(Daten,filepos(Daten)-1);
    write(Daten,Artikel);
    writeln(Umsatz,' verkauft');
  END;
END;
```

Programms zurückgesetzt. Leider stehen typisierte Konstanten in den meisten PASCAL-Systemen nicht zur Verfügung. Das Unterprogramm VERKAUF bringt wenig Neues (vgl. S. 124, unten).
Man achte auf das Rücksetzen des Filefensters mit FILEPOS, bevor die geänderte Filekomponente in das File geschrieben wird. Das ist erforderlich, um dieselbe Filekomponente zu überschreiben, die vorher gelesen wurde. Das Filefenster wurde, wie bereits dargestellt, vom System beim Lesen weiterpositioniert.
Das Unterprogramm ANTWORT wird diesmal zum Erzwingen der Antworten J oder N benutzt.
Um die Arbeitsweise des Programms zu demonstrieren, wird angenommen, es existiere ein File HOB.DAT mit folgendem Inhalt:

Bestell-nummer	Bezeichnung	Be-stand	Preis M	Merkmale A	B	C	D	E	F	G	H	I	J	K	L	S	W
791006	Mantel	104	151	+	+						+	+				+	
496588	Anzug	63	274	+						+			+		+		
798391	Sakko	58	162	+							+				+		
494844	Anzug	71	222	+		+							+				
802791	Hose	30	96	+						+				+			
494625	Kombination	42	244	+		+		+					+				
798254	Blouson	21	112	+		+						+		+			+

Dann kann sich das folgende Protokoll ergeben (Eingaben unterstrichen):

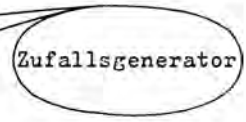
```
Verkaufsdisposition Herrenoberbekleidung
nach Material, Musterung und Farbe
Auswahl mit/ohne Merkmale (M/O): <M>

A= Polyester B= Baumwolle C= Wolle D= Leinwand
E= Glencheck F= Fischgrat G= Karo  H= Ohne Muster
I= Grau      J= Braun     K= Blau  L= Beige
S= Sommer    W= Winter
A/AB/BC...: ACE<ET>

494625 Kombination Bestand 42 Stueck a' 244.00 M
Polyester Wolle Fischgrat Braun
Verkauf (J/N): <N>
```

Eine interessante Möglichkeit besteht darin, die Eingaben des Käufers im Unterprogramm VERKAUF durch einen Zufallsgenerator zu realisieren. In diesem Beispiel beschleunigt man das Testen des Gesamtprogramms. Sonst ermöglichen Zufallsgeneratoren die Simulation an ökonomischen Modellen mit stochastischen Eingangsgrößen oder in der Ausbildung die Verhaltenssimulation in Planspielen.
Das folgende Unterprogramm übernimmt die Simulation des Verkaufsprozesses:

```
PROCEDURE Verkaufssimulation;
VAR Eingabe : BOOLEAN;
    Umsatz  : INTEGER;
BEGIN
  Eingabe := random(2) = 1;
  writeln;
  IF Eingabe THEN BEGIN
    Umsatz := random(Artikel.Bestand) + 1;
    writeln(LST,'Verkauf: ',Artikel.Bezeichnung:31,' ',Umsatz,' a'' '
          Artikel.Preis:5:2,' = ',Umsatz * Artikel.Preis:8:2);
    seek(Daten,filepos(Daten)-1);
    write(Daten,Artikel);
    writeln(Umsatz,' verkauft');
  END
  ELSE writeln('Kauf abgelehnt');
END;
```



Ist die Funktion RANDOM nicht vordefiniert, müßte sie selbst programmiert werden. Dazu gibt es einfache Beispiele in der Literatur und für höhere Anforderungen auch spezielle Darlegungen.[9]
Die vordefinierte Funktion RANDOM liefert in VERKAUFSSIMULATION mit dem Aufruf "random (2)" einen Zufallswert Null oder Eins, allgemein zwischen Null und dem Aufrufparameter minus 1. Es wird dadurch zwischen Kauf (1) und Ablehnung (0) entschieden.
Bei einer Kaufentscheidung wird der Zufallsgenerator noch einmal gerufen, um die Kaufmenge zu simulieren. Das geschieht mit "random (Artikel.Bestand)+1".
Als obere Grenze gilt der Bestand für diesen Artikel. Die Addition der Eins ist die Folge der Parameterdefinition (obere Grenze ist gleich Parameter minus Eins). Dadurch wird gleichzeitig die Verkaufsmenge Null vermieden.
Die Funktion RANDOM kann auch ohne Parameter gerufen werden. Dann liefert sie eine gleich verteilte Zufallszahl im Realformat zwischen Null und Eins.
Die Benutzung des Unterprogramms VERKAUFSSIMULATION für VERKAUF ist im Programm DISPOSITION durch Kommentarklammern angedeutet.
Für die praktische Anwendung von DISPOSITION ist ein weiteres Programm erforderlich, das das Anlegen der Datenbasis und deren Aktualisierung übernimmt. Bei den Übungsaufgaben für Kapitel 7 wird darauf zurückgekommen.

7.3. Zeiger

Es gibt Anwendungsfälle, in denen ein Feld gleichartiger Datentypen benötigt wird, aber die Anzahl der Feldkomponenten vorher nicht bekannt ist. Man hilft sich mit der Auslagerung der Feldgrenze als Konstante:

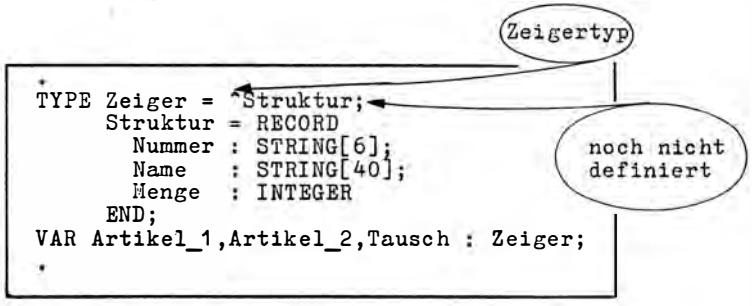
9 Vgl. Stuchlik, F.: Eine Unterstützung für den PASCAL-Nutzer. In: EDV-Aspekte 4/1985, S. 55;
Thesen, A.: An efficient generator of uniformly distributed random variates between zero and one. In: Simulation (USA), 1985 (44/1), S. 17f

```
.
CONST Grenze = 500;
TYPE Struktur = RECORD
       Nummer : STRING[6];
       Name   : STRING[40];
       Menge  : INTEGER;
     END;
VAR  Artikel : ARRAY[1..Grenze] OF Struktur;
.
```

Der Speicherplatz für die Feldvariable "Artikel" wird mit dem Eintritt in den Block, zu dem dieser Ausschnitt gehört, reserviert und nach Verlassen des Blockes wieder freigegeben. Die Größe des Feldes bleibt in dieser Zeit gleich, und es existiert in der gesamten Zeit, in der der Block existiert. Man sagt, der Speicherplatz wird statistisch verwaltet, und "Artikel" ist eine statistische Variable.
Ein Programm soll das Anlegen und Verwalten von Angebotskatalogen (für den Absatz eines Kombinates oder als Bestellkatalog für den Einzelhandel) unterstützen. Der Umfang des Katalogs hängt ab von der Anzahl der im Angebot befindlichen Artikel. Im Programmausschnitt wurde auf maximal 500 geschätzt. Aber die Zahl kann schwanken. Übersteigt sie 500, ist sogar eine Programmänderung erforderlich. Es können aber auch weniger als 500 Artikel sein. Dann wird zuviel Speicherplatz reserviert, und das kann sich ebenfalls nachteilig auswirken. Probleme ergeben sich, wenn im Angebotskatalog zu streichen oder einzufügen ist. In der Datenstruktur "Artikel" des Programmausschnitts entstehen beim Streichen "Löcher". Einfügungen sind nur am Ende möglich – oder man programmiert die Verschiebung der gesamten Belegung. Wird lediglich "angehängt", so ist jedesmal eine Sortierung erforderlich.

Zeiger und namenlose Variablen

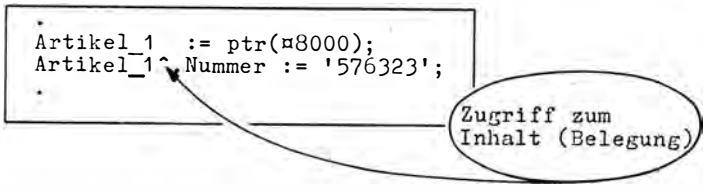
PASCAL ermöglicht auch eine dynamische Verwaltung von Datenstrukturen und stellt dazu einen speziellen Datentyp, den Pointer- oder Zeigertyp zur Verfügung. Mit ihm ist es möglich, Datenstrukturen während der Programmausführung je nach Bedarf entstehen und wieder beseitigen zu lassen. Das soll zunächst nur mit zwei, später mit beliebig vielen Artikeln gezeigt werden. Die Zeigertechnik wird hier und im folgenden ausschließlich für 8-Bit-Computer dargestellt.




```
.
TYPE Zeiger = ^Struktur;
     Struktur = RECORD
       Nummer : STRING[6];
       Name   : STRING[40];
       Menge  : INTEGER
     END;
VAR Artikel_1,Artikel_2,Tausch : Zeiger;
.
```

Der durch diese Vereinbarungen reservierte Speicherplatz beträgt nur 6 Byte, entsprechend dem Umfang der 3 Variablen "Artikel_1", "Artikel_2" und "Tausch" mit je 2 Byte.

Man achte darauf, daß bei der Definition von "Zeiger" der Bezeichner "Struktur" verwendet wurde, obwohl "Struktur" selbst zu diesem Zeitpunkt noch nicht definiert ist. Zeiger sind in PASCAL die einzige Ausnahme, bei denen das erlaubt ist. Die praktischen Erfordernisse dieser Lockerung werden noch dargelegt.
Die Zeigervariable unterscheidet sich wesentlich von den bisher behandelten Variablen. Sie enthält eine Speicheradresse. Es ist die Adresse einer einfachen oder strukturierten Variablen. Im Beispiel formuliert man "Zeiger" auf "Struktur", wobei mit "Zeiger" der Typbezeichner und dann mit zum Beispiel "Artikel_1" die Zeigervariable bereitgestellt wird.
Da nun "Artikel_1" eine Variable ist, läßt sich der Platz der Struktur im Hauptspeicher durch verschiedene Belegungen der Variablen "Artikel_1" beeinflussen. Hat "Artikel_1" den Wert ¤8000, so stehen die 50 Byte der Beispielstruktur ab Adresse ¤8000, hat "Artikel_1" den Wert ¤8100, so stehen die 50 Byte ab dieser Adresse. Wenn man die VAR-Deklaration im Programmausschnitt betrachtet, stellt man fest, daß dort mit "Artikel_1", "Artikel_2" und "Tausch" Namen für die Zeigervariablen existieren, nicht aber für die eigentliche Variable des Typs "Struktur". Das hat diesen Variablen die Bezeichnung "die Namenlosen" oder "die Anonymen" eingebracht. Der Zugriff zu den "Namenlosen" erfolgt, indem der Zeigervariablen das Zeichen "^" nachgestellt wird. Mit



wird der Speicherplatz ab Adresse ¤8000 mit den angegebenen Werten belegt. Die Wertzuweisung für die Zeigervariable erfolgt hier mit der Pseudofunktion PTR. Implementationsabhängig kann auch WRD verwendet werden. Die direkte Zuweisung einer Adresse an die Zeigervariable ist nur eine Form der Arbeit mit Zeigern. Bei praktischen Anwendungen müßte der Programmierer in diesem Falle den Speicherplatz selbst verwalten, also eine Übersicht über freien Speicherplatz führen und diese ständig aktuell halten. PASCAL hält dafür spezielle Funktionen und Prozeduren bereit und unterstützt den Programmierer. Ehe das erläutert wird, sollen aber einige nützliche Anwendungen der direkten Zuweisung gezeigt werden.

Nicht unterstützte Wertzuweisungen
So ist es zum Beispiel möglich, einen Speicherbereich mit bestimmten Zeichen aufzufüllen. Eine Möglichkeit ist:

```
.
VAR Beginn : ^BYTE;
BEGIN
  Beginn := ptr(¤8000);
  fillchar(Beginn^,1024,chr(0));
.
```

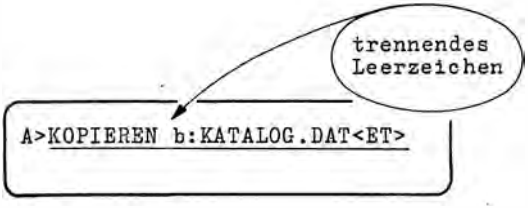
Der Zeiger "Beginn" wird auf ¤8000 gesetzt, und unter Nutzung der vordefinierten Funktion FILLCHAR werden bei ¤8000 beginnend 1024 Byte gleich Null gesetzt. Mit ^ BYTE soll gezeigt werden, daß auch Zeiger auf vordefinierte einfache Datentypen vereinbart werden können. Die Anwendung von FILLCHAR ist eine schnelle Möglichkeit, zu einer nützlichen Vorbelegung von Speicherbereichen zu gelangen.
Zur Anfangsbelegung von Variablen ist eine Verbindung mit der Funktion SIZEOF zweckmäßig, die als INTEGER-Wert die Länge in Byte der jeweiligen Variablen zurückgibt. Soll im Unterprogramm VERWALTEN des Programms DEVISEN in Kapitel 6 der Datensatz "Eintragung" eine Anfangsbelegung Null für alle Byte erhalten, so ist zu programmieren:

```
.
fillchar(Eintragung,sizeof(Eintragung),0);
.
```

Beim Anlegen des Files TABELLE.DAT wurde das genutzt. So ist der genau definierte Inhalt zu erklären, denn sonst besitzen nicht belegte Byte in Strings rein zufällige Werte. Unter Nutzung der Zeigertechnik mit direkter Wertzuweisung ist es auch möglich, Speicherbereiche zu verschieben:

```
.
VAR Von,Nach : ^BYTE;
BEGIN
  Von  := ptr(¤8100);
  Nach := ptr(¤8000);
  move(Von^,Nach^,50);
.
```

Der Zeiger "Von" wird auf die Adresse ¤8100, der Zeiger "Nach" auf die Adresse ¤8000 gesetzt und durch die Prozedur MOVE der Speicherbereich ¤8100 bis ¤8132, also 50 Byte, in den Bereich ¤8000 bis ¤8032 kopiert.
Schließlich soll noch gezeigt werden, wie mit Hilfe dieser Zeigertechnik die Arbeitsweise des Betriebssystems genutzt werden kann. Um das praktische Anliegen der folgenden Beschreibung zu erkennen, ist noch einmal das Programm KOPIEREN im Abschnitt 6.3. zu betrachten. Für dieses Programm läßt sich die Laufzeit verkürzen, wenn der Nutzer den Namen des zu kopierenden Files beim Startkommando sofort mit angeben kann, also zum Beispiel:



Getrennt durch ein Leerzeichen folgt hier noch ein weiterer Parameter, ein sogenannter Kommandozeilenparameter (Command line parameter). Es können auch mehrere sein. Die Trennung wird bei Leerzeichen erkannt. SCPX trägt den Parameterteil (Command

tail) einschließlich des führenden Leerzeichens in den Systempuffer ab Adresse ¤80 ein. Dabei steht auf ¤80 die dynamische Länge des Textes nach dem eigentlichen Kommando (im Beispiel 10 bzw. ¤A) und ab ¤81, beginnend mit dem Leerzeichen, der Kommandozeilenparameter. Folglich kann das Programm KOPIEREN wie folgt ergänzt werden:

```
.
TYPE Kette = STRING[14];
VAR  Parameter = ^Kette;
BEGIN
  Parameter := ptr(¤80);
  Filename  := Parameter^;
.
```

Uebernahme Kommando-zeilenparameter

Mit "Parameter^" wird auf den Kommandozeilenparameter zugegriffen und sein Wert (der Text des Kommandozeilenparameters) der Variablen "Filename" zugewiesen. Die Eingabeaufforderung "Filename:" und die Eingabe des Filenamens im Programm könnten entfallen oder – was nutzerfreundlicher wäre – nur für den Fall erfolgen, daß kein Parameter in der Kommandozeile angegeben wurde. Die PASCAL-Systeme unterstützen die Parameterübergabe auch durch Funktionen. So übergibt @CMD die Adresse des 1. Kommandozeilenparameters an eine Zeigervariable. PARAMCOUNT liefert die Anzahl der Parameter auf der Kommandozeile als INTEGER-Wert und PARAMSTR (x) mit x >= 1 den x-ten Kommandozeilenparameter als STRING.@CMD und PARAMCOUNT/PARAMSTR schließen sich in Abhängigkeit vom PASCAL-System aus. Das Programm KOPIEREN müßte diesmal nur im Anweisungsteil verändert werden, und zwar wie folgt:

```
.
IF paramcount > 0 THEN Filename := paramstr(1)
ELSE BEGIN
  write('Filename: ');
  readln(Filename)
END;
.
```

Soweit zur Möglichkeit, mit direkter Wertzuweisung für eine Zeigervariable zu operieren. Die Festlegungen des Programmierers werden dabei nicht geprüft, so daß besonders sorgfältig zu verfahren ist.

Unterstützte Wertzuweisung

Typisch für die Anwendung der Zeigertechnik ist, daß man sich bei der Zuweisung freier Adressen an eine Zeigervariable und deren Verwaltung unterstützen läßt. Zur Demonstration wird noch einmal das Beispiel am Anfang dieses Abschnittes benutzt, indem ein Zeigertyp auf "Struktur" und danach die Zeigervariablen "Artikel_1", "Artikel_2" und "Tausch" deklariert wurden. Wie bereits erwähnt, werden bisher nur 6 Byte als Speicherplatz benötigt. Dieser Speicherplatz nimmt Adressen auf. Sollen die "namenlosen" Variablen der Struktur nun wirklich mit STRING- und INTEGER-Werten belegt werden, so sind natürlich 50 Byte erforderlich. Von diesen 50 Byte muß man fordern, daß sie bisher nicht belegt sind. Die Reservierung des erforderlichen freien Speicherplatzes wird in PASCAL von der Prozedur NEW übernommen.

```
.
new(Artikel_1);
readln(Artikel_1^.Nummer);
readln(Artikel_1^.Name);
.
```

NEW weist der Zeigervariablen eine Adresse zu, der so viel freier Speicherplatz folgt, wie für "Struktur" mit 50 Byte erforderlich ist. Über diesen Speicherplatz kann dann verfügt werden. Hier werden mit READLN Daten auf diesen Speicherplatz gelesen.
Die von NEW vergebene Adresse ist für den Programmierer nicht von Interesse. Sie kann durch Anwendung der Funktion ORD ermittelt werden. Das ist aber nicht erforderlich, weil PASCAL die Überwachung von sich aus organisiert. Die Gesamtheit des belegten Speicherbereiches, der nicht zusammenhängend sein muß, wird in der Fachsprache als Halde (Heap) bezeichnet. Seine Überwachung erfolgt durch einen Heap-Verwalter, den PASCAL in das Programm einfügt.
Wichtig ist zu wissen, daß jeder Aufruf von NEW zu neuer Speicherplatzreservierung führt. Der Programmausschnitt

```
.
new(Artikel_1);
readln(Artikel_1^.Nummer);
new(Artikel_1);
write(Artikel_1^.Nummer);
.
```

führt also nicht zur Kontrollausgabe der gerade eingegebenen Artikelnummer. Der Ausgabeparameter Artikel_1^.Nummer ist undefiniert, denn NEW hat die Adresse seit der letzten Eingabe weitergestellt. Es ist ersichtlich, daß die Verwendung von NEW nur praktikabel ist, wenn mehrere Zeigervariablen verwendet werden. Denn nach jedem NEW wird die Adresse geändert, und die früheren Daten sind nicht mehr zugänglich.
Eine Lösung für zwei Artikel wäre

```
.
new(Artikel_1);
readln(Artikel_1^.Nummer);
.
new(Artikel_2);
readln(Artikel_2^.Nummer);
.
```

Übrigens ließe sich das Gesamtproblem auch mit einem Zeigerfeld lösen. Es wäre dann das Feld

```
.
VAR Artikel : ARRAY[1..Grenze] OF Zeiger;
.
```

zu deklarieren und NEW zyklisch auf "Artikel [i]^" anzuwenden. Aber es gibt noch eine günstigere Lösung für das aufgeworfene Problem des Anlegens und Verwaltens eines Angebotskatalogs.

Freigabe und weitere Prozeduren

Zunächst sollen die Freigabe nicht mehr benötigten Speicherplatzes und weitere PASCAL-Unterstützungen gezeigt werden. Das geschieht mit der Prozedur DISPOSE:

```
.
dispose(Artikel_1);
.
```

Ab sofort wird der mit "Artikel_1" adressierte Speicherplatz für "Struktur" nicht mehr als belegt betrachtet. Die Halde wird verkleinert. Der Speicherplatz wird dann für NEW wieder verfügbar. Da im allgemeinen der Speicherbereich dynamischer Variablen (Halde) und der Speicherbereich statischer Variablen (Keller) aufeinander zuwachsen, wird die Differenz zwischen Keller und Halde durch DISPOSE vergrößert.
Der Wert der Zeigervariablen nach DISPOSE ist undefiniert. Die Anwendung von DISPOSE auf eine Zeigervariable mit dem Wert NIL führt zu einem Laufzeitfehler.
Implementationsabhängig gibt es weitere Prozeduren zur Unterstützung der Arbeit mit dynamischen Variablen. Das sind (Prozedurparameter ist jeweils eine Zeigervariable):

MARK Mit MARK(x) kann man auf x (einer Zeigervariablen) den aktuellen Stand der Halde festhalten. Das erfolgt mit dem Ziel der späteren Freigabe ab dieser Position.

RELEASE Mit RELEASE(x) kann man eine Halde ab der Position freigeben, die vorher mit MARK(x) fixiert wurde; x darf zwischen den MARK- und RELEASE-Rufen nicht verändert werden. Mit RELEASE(x) wird der Zustand wiederhergestellt, der zum Zeitpunkt des vorhergehenden Prozedurrufes MARK(x) existierte. MARK/RELEASE und DISPOSE dürfen nicht gleichzeitig verwendet werden.

MAXAVAIL Mit dieser Funktion, die einen Wert vom Typ INTEGER liefert, kann der größte noch freie Speicherblock ermittelt werden. Gegebenenfalls erfolgt eine Reorganisation der Halde.

MEMAVAIL Diese Funktion ermittelt den freien Bereich zwischen Keller und Halde in Byte.

Abschließend sollen noch einige Regeln für die Arbeit mit Zeigern zusammengestellt werden.

1. Zeigervariablen sind nach ihrer Deklaration undefiniert. Vor dem Zugriff auf die "Namenlosen" muß ihnen ein Wert zugewiesen sein.
2. Zeigervariablen kann direkt eine Konstante, implementationsabhängig auch ein Ausdruck zugewiesen werden. Dieser Wert bezeichnet eine Adresse. Die Zuweisung wird durch eine Konvertierungsfunktion (PTR oder WRD) vermittelt.
3. Zeigervariablen desselben Typs sind zuweisungsverträglich. Im Beispiel führt die Befehlsfolge

```
.
Tausch := Artikel_1;
Artikel_1 := Artikel_2;
Artikel_2 := Tausch;
.
```

zu einer Vertauschung der Zeigerinhalte ohne Umspeicherung der Variablen.

4. Zeigervariablen kann auch die vordefinierte Konstante NIL zugewiesen werden. NIL-gesetzte Variablen sind definiert, zeigen aber auf keine Variable.
5. Zeiger können immer nur auf eine Datenstruktur zeigen. Sie sind an diesen Typ gebunden ("Tausch" könnte im Beispiel nicht auch noch auf eine andere Datenstruktur zeigen).
6. Zeigervariablen können auf Gleichheit und Ungleichheit verglichen werden.

Die praktische Anwendung dieser Regeln wird im folgenden Abschnitt sichtbar.

7.4. Dynamische Variablen

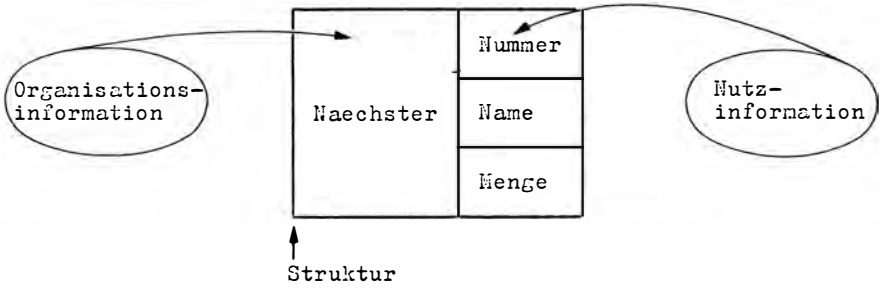
Das am Anfang des Abschnittes 7.3. aufgeworfene Problem des Anlegens und Verwaltens eines Angebotskatalogs wird nun mit dynamischer Speicherplatzverwaltung für beliebig viele Einträge gelöst. Das entsprechende Hauptprogramm ist folgendes:


```
PROGRAM Angebot;
{Anlegen und Verwalten eines Angebots-/Bestellkatalogs}
TYPE Schluessel = STRING[6];
     Zeiger     = ^Struktur;
     Struktur   = RECORD
       Naechster: Zeiger;
       Nummer   : Schluessel;
       Name     : STRING[40];
       Menge    : INTEGER
     END;
VAR Anker,Artikel: Zeiger;
    Bestellnummer: Schluessel;
    Bezeichnung  : STRING[40];
    i            : INTEGER;
{¤I FINDEN.ANG  }
{¤I BILDEN.ANG}
{¤I LOESCHEN.ANG}
{¤I AUSGABE.ANG }
BEGIN
  Anker:= NIL;
  writeln;
  writeln('Anlegen,Erweitern,Korrigieren eines Angebotskatalogs');
  REPEAT
    write('Bestellnummer oder ''Ende'':');
    readln(Bestellnummer);
    IF Bestellnummer <> 'Ende' THEN BEGIN
      Artikel:= Finden(Bestellnummer);
      IF Artikel = NIL THEN Bilden(Artikel)
      ELSE BEGIN
        writeln(Artikel^.Nummer,' ',Artikel^.Name);
        write('Neue Bezeichnung oder ''Loeschen'':');
        readln(Bezeichnung);
        IF Bezeichnung = 'Loeschen' THEN Loeschen(Artikel)
        ELSE Artikel^.Name:=Bezeichnung;
      END
    END
    ELSE Ausgabe;
  UNTIL Bestellnummer='Ende';
  writeln('Ende');
END.
```

Natürlich ist die Datenstruktur für die praktische Anwendung zu erweitern. Zumindest der Preis wäre noch erforderlich.

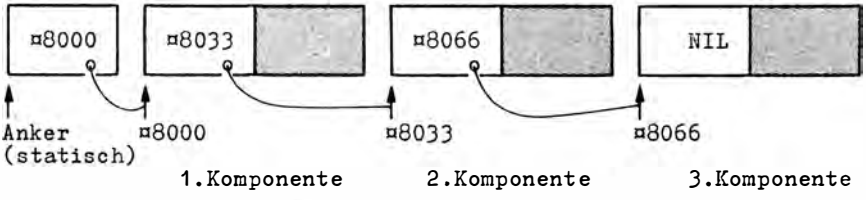
Gekettete Listen

Zu beachten ist, daß diesmal eine Zeigervariable in die Struktur eingelagert wurde. Diese Einlagerung ist typisch für die Arbeit mit dynamischen Variablen. Sie soll deshalb noch grafisch dargestellt werden:



Der eigentlichen Nutzinformation wird der Zeiger "Naechster" eingelagert und so mit Speicherkomponenten, den Objekten selbst, verbunden. Über diesen eingelagerten Zeiger werden die verschiedenen Objekte untereinander verbunden. Programmtechnisch wird deutlich, warum PASCAL die Verwendung nicht definierter Bezeichner bei Zeigertypen gestatten muß.

Für die Speicherstruktur werden mit "Anker" und "Artikel" statisch nur 4 Byte reserviert. "Anker" wird als Zeiger auf die erste Speicherkomponente verwendet, damit der Anfang nicht verlorengeht. "Artikel" dient der Arbeit mit der aktuellen Struktur. Der Aufbau einer zusammenhängenden dynamischen Speicherstruktur aus den Komponenten ab Adresse ¤8000 geht aus der folgenden Darstellung hervor:



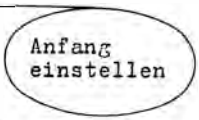
Eine solche Struktur nennt man vorwärtsgekettete Liste.

Zu beachten ist, daß die Adresse der ersten Speicherkomponente als Inhalt einer statischen Variablen des Zeigertyps aufzubewahren ist. Jede Speicherkomponente enthält dann den Zeiger auf die folgende Speicherkomponente. Existiert keine Folgekomponente, so ist der Inhalt der eingelagerten Zeigervariablen NIL. Mit NIL kann man also zugleich auch die letzte Speicherkomponente feststellen.

Suchen einer Komponente

Wie eine dynamische Speicherstruktur durchlaufen wird, hier zum Zwecke des Suchens eines Strings, der später die Bestellnummer enthält, zeigt die Funktion FINDEN:

```
FUNCTION Finden(b:Schluessel):Zeiger;
VAR Suchen:Zeiger;
BEGIN
  Suchen:= Anker;
  Finden:= NIL;
  WHILE Suchen <> NIL DO
  IF Suchen^.Nummer = b THEN BEGIN
    Finden:= Suchen;
    Suchen:= NIL
  END
  ELSE Suchen:= Suchen^.Naechster;
END;
```



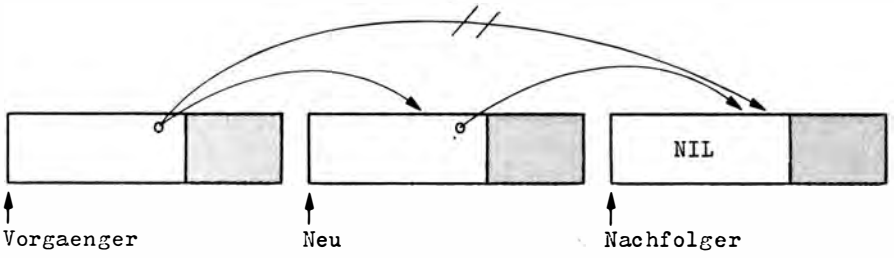
Mit "Suchen" wird ein Laufzeiger definiert. Am Anfang wird ihm der Wert von "Anker" zugewiesen. Danach zeigt er auf die erste Speicherkomponente der dynamischen Struktur.
Es wird geprüft, ob der Teil "Nummer" der aktuellen Speicherkomponente mit dem gesuchten String "Schluessel" übereinstimmt. Ist das der Fall, wird die Adresse der aktuellen Speicherkomponente auf dem Funktionsnamen FINDEN bereitgestellt und durch Zuweisung von NIL an den Laufzeigern die Beendigung der Suche vorbereitet.
Ist das nicht der Fall, wird dem Laufzeiger die Adresse der nächsten Filekomponente zugewiesen.
Ist die letzte Komponente erreicht, wird "Suchen" NIL und der Durchlauf beendet. Der Funktionsname FINDEN übermittelt NIL, wenn die Suche ergebnislos war.

Bilden einer Komponente

Die Bildung einer neuen Speicherkomponente zeigt die Prozedur BILDEN. Ihr Aufruf vom Hauptprogramm aus erfolgt, wenn die Funktion FINDEN den Wert NIL zurückgibt, also bisher noch keine Komponente mit dem Inhalt von "Schluessel" in der dynamischen Struktur existiert.

```
PROCEDURE Bilden(a:Zeiger);
VAR Suchen,Sichern : Zeiger;
BEGIN
  new(a);
  a^.Nummer := Bestellnummer;
  write('Artikelbezeichnung: ');
  readln(a^.Name);
  write('Bestellmenge: ');
  read(a^.Menge); writeln;
  a^.Naechster := NIL;
  Suchen  := Anker;
  Sichern := NIL;
  WHILE Suchen <> a^.Naechster DO
    IF a^.Nummer > Suchen^.Nummer THEN BEGIN
      Sichern := Suchen;
      Suchen  := Suchen^.Naechster
    END
    ELSE a^.Naechster := Suchen;
  IF Sichern = NIL THEN Anker := a
  ELSE Sichern^.Naechster := a;
END;
```

In dieser Prozedur erhält die formale Zeigervariable a und damit die aktuelle Zeigervariable "Artikel" die Adresse des nächsten freien Speicherplatzes zugewiesen. Die bereits vorher eingegebene "Bestellnummer", die auch der Suche zugrunde lag, wird mit "a^. Nummer:= Bestellnummer" direkt in die Speicherkomponente übernommen. Die anderen Angaben für den Katalog werden angefordert.
Ist die Speicherkomponente belegt, wird die Speicherstruktur durchgemustert, um die neue Komponente an der richtigen Stelle einzuordnen. Die Durchmusterung wird abgebrochen, wenn die neue Bestellnummer größer ist als die Bestellnummer der gerade geprüften Speicherkomponente. Das nun folgende Einfügen der neuen Speicherkomponente ist denkbar einfach. Es ist nämlich kein Umspeichern erforderlich, sondern nur das Ändern der eingelagerten Zeiger. Die allgemeine Verfahrensweise des Einfügens in vorwärtsgekettete Listen kann wie folgt dargestellt werden (die frühere Verbindung ist gestrichen):



"Vorgaenger" kann natürlich im Ausnahmefall "Anker" sein, wenn die neue Komponente am Anfang eingefügt wird.
Um zwischen den Speicherkomponenten "Vorgaenger" und "Nachfolger" (symbolisch für Adressen) eine Speicherkomponente "Neu" einzufügen, genügt es,

a) dem Vorwärtszeiger der Speicherkomponente "Neu" mit dem Zeigerwert "Vorgaenger" zu belegen (der auf "Nachfolger" weist);
b) den Vorwärtszeiger der Speicherkomponente "Vorgaenger" mit der Adresse der Speicherkomponente "Neu" zu laden.

Die Speicherobjekte "Vorgaenger^", "Nachfolger^" und "Neu^" bleiben an dem Platz, der ihnen mit NEW zugewiesen wurde. Nur die Organisationsinformation wird geändert.
Im Beispiel wird die Einfügestelle durch Vergleich der Bestellnummern gesucht. Das geschieht mit der Bedingung "a^.Nummer>Suchen^.Nummer". Ist die Bedingung erfüllt, wird weitergesucht. Die neue Speicherkomponente wird erst eingefügt, wenn sie kleiner ist (gleich wird im Aufruf ausgeschlossen) als die Bestellnummer der gemusterten aktuellen Komponente. Dann zeigt "Suchen" auf den Nachfolger und "Sichern^.Naechster" auf den Vorgänger. "Suchen" wird also für die neue Komponente der Verweis auf den Nachfolger. Die Zuweisung erfolgt im ELSE-Zweig mit "a^.Naechster := Suchen". Das wurde zugleich als Endebedingung des WHILE-Zyklus programmiert. Das Laden des Vorwärtszeigers der Speicherkomponente "Vorgaenger" mit der Adresse der neu eingefügten Speicherkomponente erfolgt mit "Sichern^.Naechster :=a". Dabei ist zu berücksichtigen, daß die Adresse der ersten Speicherkomponente der statischen Zeigervariablen "Anker" zugewiesen werden muß. Dem dient die IF-Anweisung. Diese Vorgehensweise ist typisch für Bildungs- und Einfügeoperationen in dynamischen vorwärtsgeketteten Strukturen.

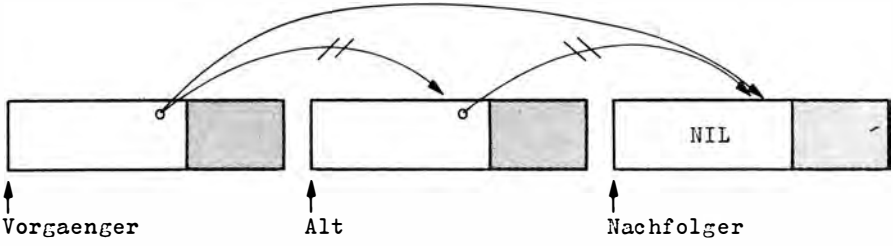
Löschen einer Komponente

Die Freigabe von dynamisch verwaltetem Speicherplatz zeigt die Prozedur LOESCHEN. Die Prozedur wird vom Hauptprogramm gerufen, wenn die eingegebene Bestellnummer bereits vorhanden ist und der Text "Loeschen" statt einer Korrekturbezeichnung eingegeben wurde. In der praktischen Anwendung wäre hier auch die Korrekturmöglichkeit für "Nummer" und "Menge" im Hauptprogramm vorzusehen. Die Prozedur LOESCHEN wird also gerufen, wenn eine Speicherkomponente aus der dynamischen Struktur herausgenommen werden soll. Die Vorgehensweise zeigt der folgende Programmtext:

```
PROCEDURE Loeschen(a:Zeiger);
VAR Suchen,Sichern : Zeiger;
BEGIN
  Suchen  := Anker;
  Sichern := NIL;
  WHILE Suchen <> a DO BEGIN
    Sichern := Suchen;
    Suchen  := Suchen^.Naechster
  END;
  IF Sichern = NIL THEN Anker := a^.Naechster
  ELSE Sichern^.Naechster := a^.Naechster;
  dispose(a)
END;
```

Freigabe

Auch hier ist es erforderlich, die Herausnahme einer Speicherkomponente zunächst grafisch zu erläutern:



Die eingelagerte Zeigervariable der Vorgängerkomponente (im Ausnahmefall "Anker") muß die Adresse des Nachfolgers der auszusondernden Speicherkomponente erhalten. Wie man sieht, ist der Zeigervariablen des Vorgängers der eingelagerte Wert der Zeigervariablen in der zu löschenden Komponente zuzuweisen. Im Beispiel wird das durch "Sichern^.Naechster := a^.Naechster" bzw. durch "Anker := a^.Naechster" erreicht. Die Freigabe des von der zu löschenden Komponente bisher belegten Speicherplatzes erfolgt durch die Prozedur LOESCHEN und ist typisch für das Streichen von dynamischen Speicherkomponenten und die Speicherplatzfreigabe. Für die Speicherplatzfreigabe gibt es neben dem DISPOSE-Verfahren nach Wirth das bereits erwähnte MARK/RELEASE-Verfahren nach Bowles. MARK (Anker) und danach RELEASE (Anker) würden sofort den gesamten Speicherplatz der dynamischen Variablen freigeben. Die beiden Freigabeverfahren dürfen in einigen Systemen nicht gleichzeitig verwendet werden.

Die abschließende Prozedur AUSGABE sorgt dafür, daß der Inhalt der gesamten dynamischen Struktur ausgegeben wird:

```
PROCEDURE Ausgabe;
VAR Katalogfile : TEXT;
BEGIN
  assign(Katalogfile,'KATALOG.DAT');
  rewrite(Katalogfile);
  Artikel := Anker;
  WHILE Artikel <> NIL DO BEGIN
    writeln(Katalogfile,
            Artikel^.Nummer,' ',Artikel^.Name,Artikel^.Menge);
    Artikel := Artikel^.Naechster;
  END;
  close(Katalogfile);
END;
```

Die Prozedur wird gerufen, wenn das Wort "Ende" statt einer Bestellnummer eingegeben wird. Die Ausgabe erfolgt als Textfile auf die Diskette. Es ist sicher kein Problem, die Prozedur so zu verändern, daß eine Ausgabe über Drucker oder Bildschirm erfolgt. Dann müßte natürlich noch eine Überschrift ausgegeben werden. Darauf wird in den Übungsaufgaben zurückgekommen.
Einen kurzen Einblick in die Arbeitsweise des Programms ANGEBOT vermittelt das folgende Protokoll (Eingaben unterstrichen):

```
Anlegen,Erweitern und Korrigieren eines Angebotskataloges
Bestellnummer oder 'Ende': 791006<ET>
Artikelbezeichnung: Mantel<ET>
Bestand: 104<ET>
Bestellnummer oder 'Ende': 496588<ET>
.
```

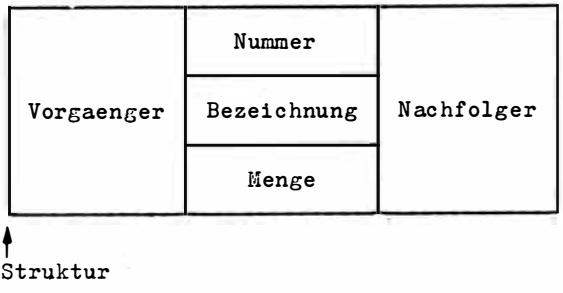
Nach der Eingabe der Bestellnummer und der Artikelbezeichnungen aus Abschnitt 7.2. entsteht auf der Diskette ein Textfile mit folgendem Inhalt (byteweise, hexadezimal mit versuchter Zeichendarstellung):

```
34 39 34 36  32 35 20 4B  6F 6D 62 69  6E 61 74 69  494625 Kombinati
6F 6E 34 32  0D 0A 34 39  34 38 34 34  20 41 6E 7A  on42.. 494844 Anz
75 67 37 31  0D 0A 34 39  36 35 38 38  20 41 6E 7A  ug71.. 496588 Anz
75 67 36 33  0D 0A 37 39  31 30 30 36  20 4D 61 6E  ug63.. 791006 Man
74 65 6C 31  30 34 0D 0A  37 39 38 32  35 34 20 42  tel104.. 798254 B
6C 6F 75 73  6F 6E 32 31  0D 0A 37 39  38 33 39 31  louson21.. 798391
20 53 61 6B  6B 6F 35 38  0D 0A 38 30  32 37 39 31   Sakko58.. 802791
20 48 6F 73  65 33 30 0D  0A 1A 9A C8  9A C8 74 21   Hose30.....t!
```

Das Programm ANGEBOT ist für viele Zwecke direkt anwendbar, wenn eine Anpassung an die erforderliche Datenstruktur erfolgt.

Zweifach gekettete Listen

Eine weitere Form der Arbeit mit dynamischen Variablen und verketteten Listen entsteht, wenn der Struktur ein Zeiger auf die folgende und ein Zeiger auf die vorhergehende Speicherkomponente eingelagert wird. Die Komponenten hätten dann die folgende Struktur:



"Vorgaenger" und "Nachfolger" nehmen Adressen auf. Mit der Zeigervariablen "Anfang" könnte die Adresse der ersten und mit "Ende" die Adresse der letzten Komponente fixiert werden. Der Bestellkatalog ließe sich dann je nach Erfordernis vorwärts oder rückwärts ausgeben. Rückwärts würde der Ausgabezyklus mit "Ende" beginnen, und dem Laufzeiger wäre immer die Adresse des Vorgängers zuzuweisen, bis NIL erkannt wird. Besondere Gebilde erhält man, wenn die Variable "Vorgaenger" der ersten Speicherkomponente nicht den Wert NIL, sondern die Adresse der letzten Speicherkomponente enthält. Auch der letzten Speicherkomponente wäre dann als "Nachfolger" nicht NIL, sondern die Adresse der ersten Speicherkomponente zuzuweisen. Die Speicherkomponenten sind jetzt zu einem Ring verbunden. Da auch viele reale Vorgänge sich ständig wiederholen (darunter Zeitrhythmen, zum Beispiel Wochentage, Regalbelegungen in der Lagerhaltung), ist eine leistungsstarke Widerspiegelung möglich.

Baumstrukturen

Eine weitere Möglichkeit für die Arbeit mit dynamischen Variablen entsteht, wenn zwei Zeiger mit klassifizierender Wirkung eingelagert werden. Die Struktur im Beispiel wäre dann folgende:

"Links" und "Rechts" sind Zeigervariablen. Eine der beiden Variablen ist NIL, die andere mit einer Adresse belegt. Welche dieser Möglichkeiten realisiert wird, richtet sich nach einem festgelegten Kriterium. Im Beispiel könnte die Bestellnummer das Kriterium sein. Ist die Nummer größer als die des aktuellen Vergleichswertes, wird "Links" belegt und "Rechts" NIL gesetzt. Andernfalls wird "Rechts" mit der Adresse belegt und "Links" NIL gesetzt. Wird diese dynamische Struktur belegt, so kann man die Anordnung der Spei-

cherkomponenten als Baum darstellen. Die Wurzel dieses Baumes ist allerdings oben. Jeder Ast kann nach links oder rechts verzweigen.
Bei der Bildung eines Baumes hat man sich zunächst für das Strukturkriterium zu entscheiden. Im Beispiel war das mit "Bestellnummer" größer als Vorgänger geschehen. Die Speicherkomponenten mit den Bestellnummern

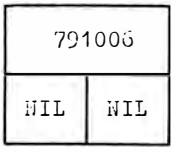
791006
496588
798391
494844
802791
496625

aus dem Beispiel in Abschnitt 7.2. sollen zu einem solchen Baum formiert werden. Das geschieht wie folgt:

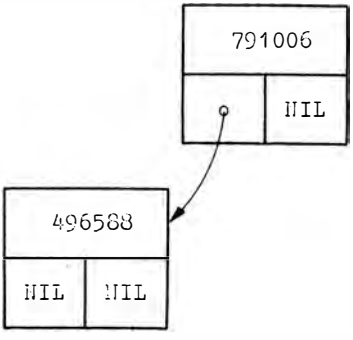
1. Zuweisung der Adresse zur Zeigervariablen, gegebenenfalls auch Belegung mit Nutzinformation. Die erste Adresse ist wie im Programmbeispiel als "Anker" zu sichern;
2. Eintragung der Verkettungsinformation.
   a) Ist es die erste Speicherkomponente, werden die Zeiger "Links" und "Rechts" auf NIL gesetzt. Das ist die Wurzel des Baumes.
   b) Ist bereits eine Speicherkomponente belegt, so vergleicht man die einzufügende Speicherkomponente mit der Wurzel. In Abhängigkeit vom Prüfkriterium wird der Zeiger "Links" oder "Rechts" verfolgt. Ist der dort gefundene Wert NIL, so ist der Platz für die neue Speicherkomponente gefunden, und sie wird eingefügt. Ist der Wert verschieden von NIL, wird der Verzweigung gefolgt und erneut geprüft.

Im Beispiel entsteht schrittweise der folgende Baum (es werden nur die Nutzinformation "Nummer" und der eingelagerte Zeiger dargestellt):

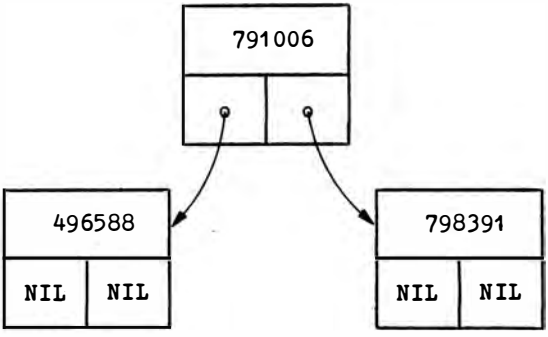
Einfügen der ersten Komponente



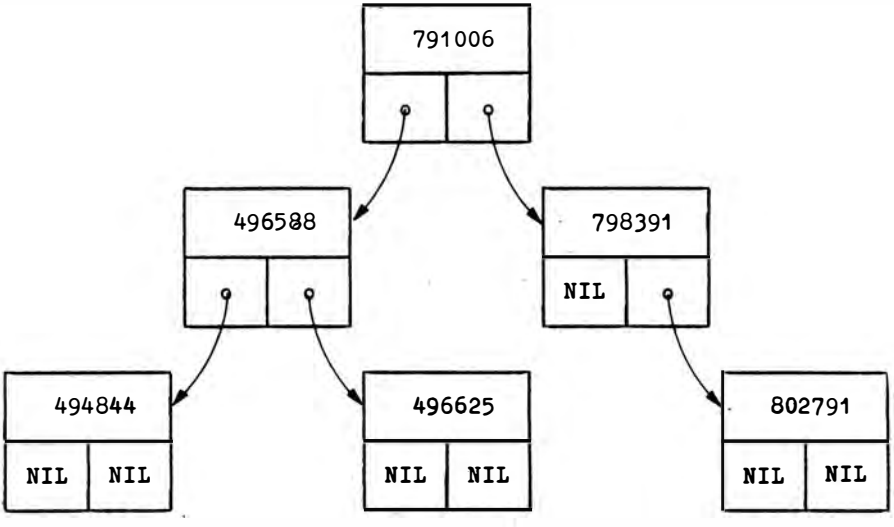
Einfügen der zweiten Komponente



Einfügen der dritten Komponente



Einfügen der vierten bis siebenten Komponente



Der Vorteil dieser Struktur besteht darin, daß bedeutend weniger (abhängig von der Baumstruktur) Vergleichsoperationen erforderlich sind, um neue Komponenten einzufügen oder um zu prüfen, ob gesuchte Komponenten vorhanden sind. Sogenannte "Suchbäume" sind eine sehr gute algorithmische Lösung für schnelles Suchen, das für viele ökonomische Anwendungen typisch ist. Solche Strukturen sind äußerst leistungsfähig. Sie sind in der Literatur gut verständlich beschrieben.[10] Im Interesse der Qualität der Programme sollten sie auch genutzt werden.

10 Vgl. Wirth, N.: Algorithmen und Datenstrukturen. Stuttgart: Teubner-Verlagsgesellschaft 1975.

Übungsaufgaben

1. Entwickeln Sie ein Programm, das das Anlegen, Erweitern und Korrigieren der Datenbasis HOB.ART ermöglicht! Nutzen Sie dafür das Unterprogramm AUSWAHL!

2. Schreiben Sie ein Unterprogramm zur Ausgabe des Bildschirminhalts über Drucker! Der Bildwiederholspeicher beginnt für 1024 Zeichen (16 * 64) bei Adresse ¤FC00 und für 1920 Zeichen (24 * 80) bei Adresse ¤F800. Sorgen Sie für die Zeilenstruktur! Das Unterprogramm ist mit einem Rahmenprogramm zu testen.

3. Erweitern Sie das Programm ANGEBOT durch ein Unterprogramm für die Ausgabe der dynamischen Datenstruktur über Drucker! Fügen Sie es so in das vorhandene Programm ANGEBOT ein, daß der Nutzer die Druckmöglichkeit wählen kann!

Anhang

Strukturübersicht

<table>
<tr><td colspan="15">PASCAL-Programm</td></tr>
<tr><td rowspan="4">Pro-
gramm-
kopf</td><td colspan="14">Block</td></tr>
<tr><td colspan="5">Vereinbarungsteil</td><td colspan="9">(BEGIN..) Anweisungsteil (..END)</td></tr>
<tr><td rowspan="2">LABEL</td><td rowspan="2">CONST</td><td rowspan="2">TYPE</td><td rowspan="2">VAR</td><td colspan="2">UP-
Deklaration</td><td colspan="3" rowspan="2">einfache
Anweisung</td><td colspan="6">strukturierte Anweisung</td></tr>
<tr><td colspan="2">Sequenz</td><td colspan="2">Selektion</td><td colspan="2">Iteration</td></tr>
<tr><td>UP-
Kopf</td><td>Block</td><td>Er-
gibt</td><td>UP-
RUF</td><td>GOTO</td><td>Ver-
bund</td><td>WITH</td><td>IF</td><td>CASE</td><td>FOR</td><td>WHILE</td><td>RE-
PEAT</td></tr>
</table>

Datentypübersicht

<table>
<tr><td colspan="9">PASCAL-Datentyp</td></tr>
<tr><td colspan="5">Einfacher Datentyp</td><td colspan="4" rowspan="2">Strukturierter Datentyp</td><td rowspan="3">Zeiger-
Typ</td></tr>
<tr><td colspan="4">ordinaler Typ</td><td rowspan="2">REAL</td></tr>
<tr><td>INTEGER
/BYTE</td><td>CHAR</td><td>Teil-
bereich</td><td>Aufzäh-
lung</td><td>BOOLEAN</td><td>STRING</td><td>ARRAY</td><td>RECORD</td><td>FILE</td><td>SET</td></tr>
</table>

PASCAL - Morpheme (implementationsabhaengige Teile in geschweiften Klammern)

1.Wortsymbole

{ABSOLUTE} AND ARRAY BEGIN CASE CONST
DIV DO DOWNTO ELSE END {EXTERNAL} FILE
FOR FORWARD FUNCTION GOTO IF IN {INLINE}
LABEL MOD {MODULE} {MODEND} NIL NOT OF OR
{OVERLAY} PACKED PROCEDURE PROGRAM RECORD REPEAT
SET {SHL} {SHR} THEN TO TYPE
{UNIT} UNTIL VAR WHILE WITH {XOR}

2.Vordefinierte Typbezeichner

BOOLEAN {BYTE} CHAR INTEGER REAL {STRING} TEXT {WORD}

3.Vordefinierte Konstanten und Variablen

{AUX} {CON} {'CON:'} FALSE INPUT {KBD} {'KBD:'} {LST} {'LST:'}
MAXINT {MEM} NIL OUTPUT {PI} {TRM} {'TRM:'} TRUE {USR}

4.Vordefinierte Funktionen und Prozeduren

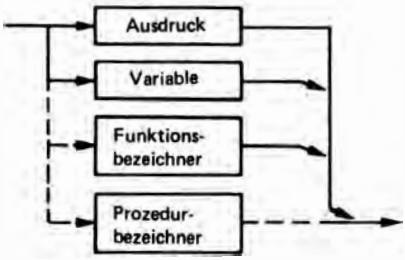
ABS {ADDR} {APPEND} {ARCTAN} {ASSIGN}
{BDOS} {BDOSHL} {BIOS} {BIOSHL} {BLOCKREAD} {BLOCKWRITE}
{CARD} {CHAIN} CHR {CLOCK} CLOSE {CLRBIT} {CLREOL}
{CLRSCR} {CONCAT} {COPY} COS
{DATE} {DELETE} {DELAY} {DELLINE} DISPOSE
EOF EOLN {ERASE} {EXECUTE} EXP {EXIT}
{FILEPOS} {FILESIZE} {FILLCHAR} {FRAC} {FREEMEM}
{GET} {GETMEM} {GOTOXY}
{HALT} {HI}
{INSERT} {INSLINE} {INT} {IORESULT}
{KEYPRESSED}
{LENGTH} {LN} {LO}
{MARK} {MAXAVAIL} {MEMAVAIL} {MOVE}
NEW
ODD {OPEN} ORD {OVRDRIVE}
{PAGE} {PACK} {PARAMCOUNT} {PARAMSTR} {POS} PRED {PTR} {PUT}
{RANDOM} {RANDOMIZE} READ {READHEX} READLN
{RELEASE} {RENAME} RESET REWRITE ROUND
{SEEK} {SEEKREAD} {SEEKWRITE} {SETBIT} SIN {SIZEOF} SQR SQRT {STR}
SUCC {SWAP}
{TSTBIT} TRUNC
{UNPACK} {UPCASE}
{VAL}
WRITE {WRITEHEX} WRITELN

5. Spezialsymbole

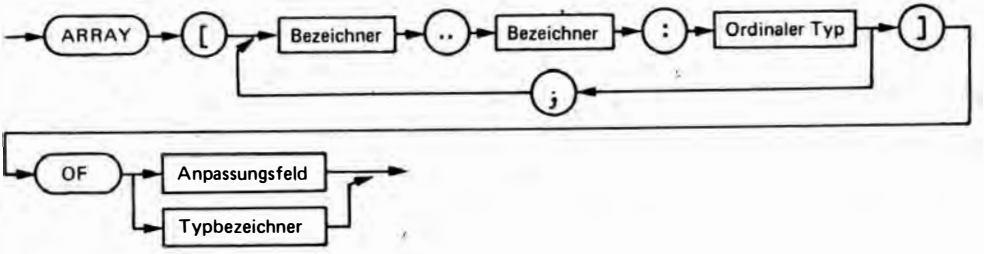
{!} {#} {¤} {&} ' ( ) * + , . '.. / : ; < = >
{?} [ ] ^ { } {|}
Transkriptionen : (. und .) fuer [ ]
(* und *) fuer { }

Adresse: Vorzeichenlose ganze Zahl

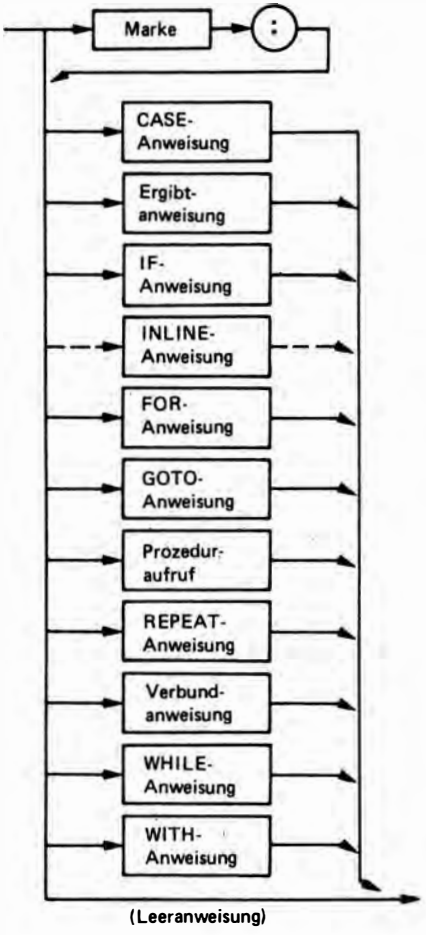
Aktueller Parameter



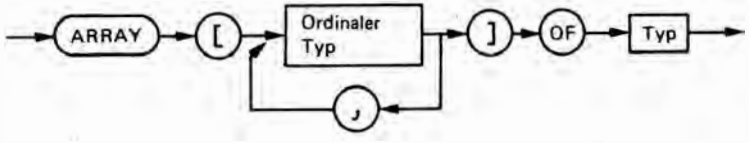
Anpassungsfeld



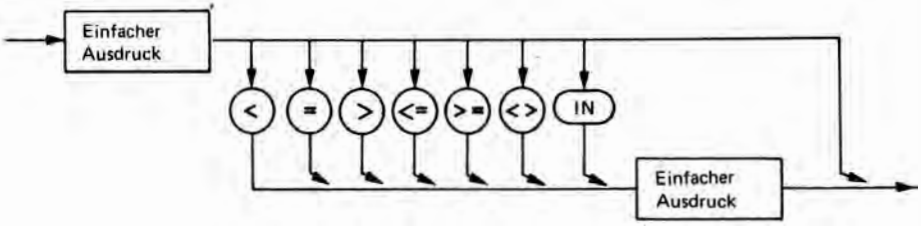
Anweisung



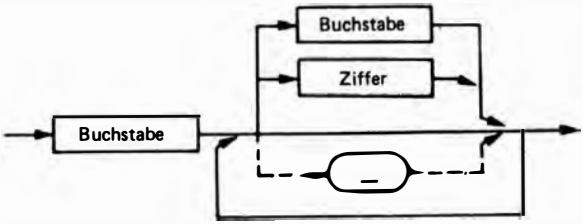
ARRAY-Typ



Ausdruck



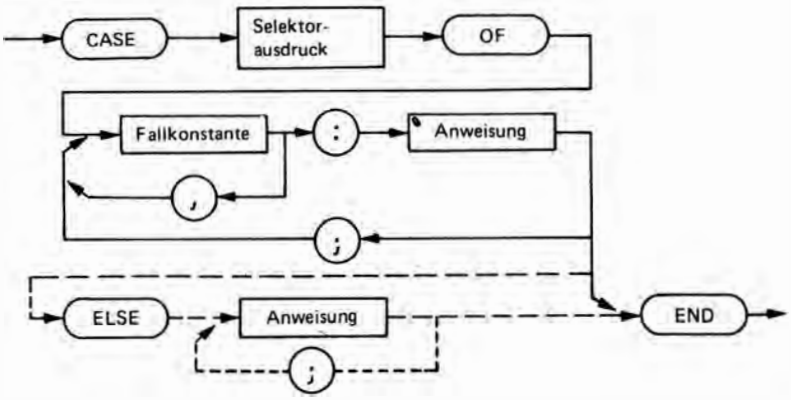
Bezeichner



Block

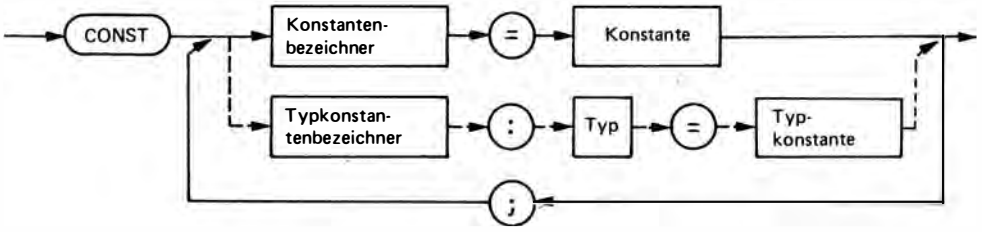
Buchstabe: A...Z, a...z

CASE-Anweisung



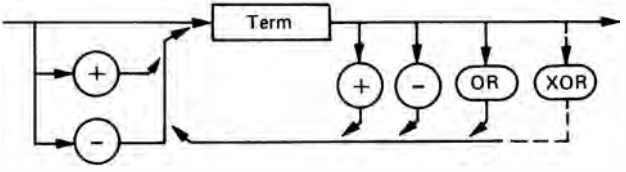
Fallkonstante: Konstante (wie Typ des Selektors)
Selektorausdruck: Ausdruck (ordinaler Typ)

CONST-Definition

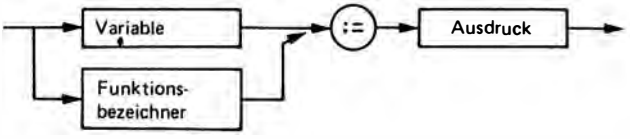


Typkonstantenbezeichner: Bezeichner

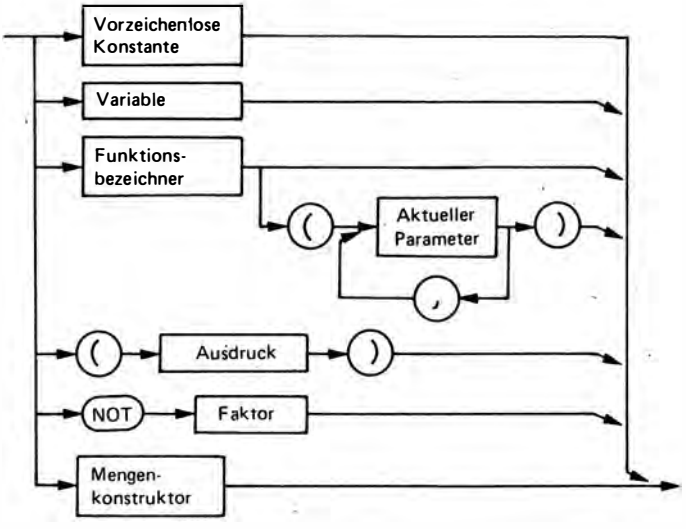
Einfacher Ausdruck



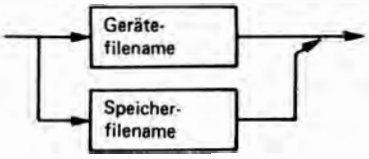
Ergibtanweisung



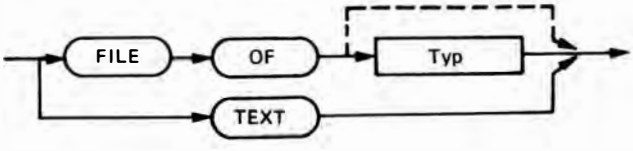
Faktor



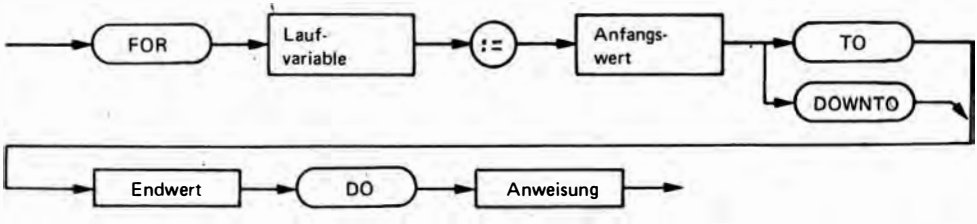
Filename



Filetyp

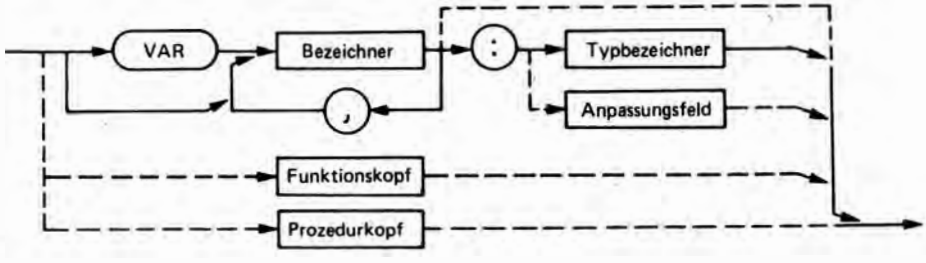


FOR-Anweisung

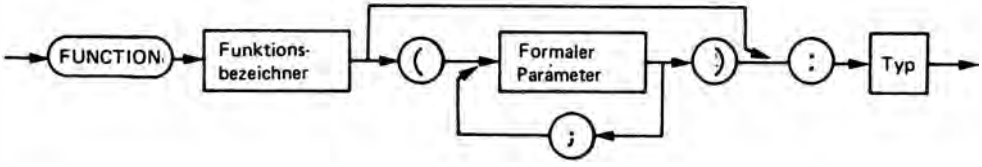


Laufvariable: einfache Variable (ordinaler Typ, lokal)
Anfangs-/Endwert: Ausdruck (ordinaler Typ)

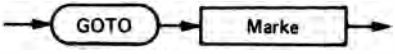
Formaler Parameter



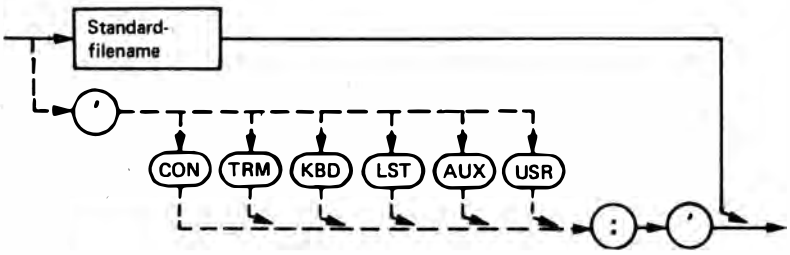
Funktionsbezeichner: Funktionskopf
Funktionskopf



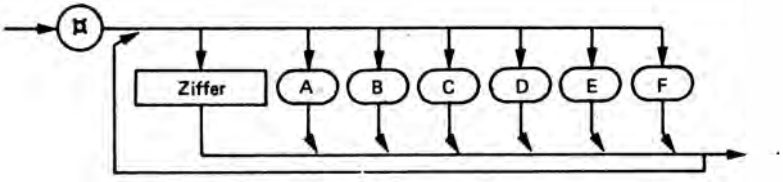
GOTO-Anweisung



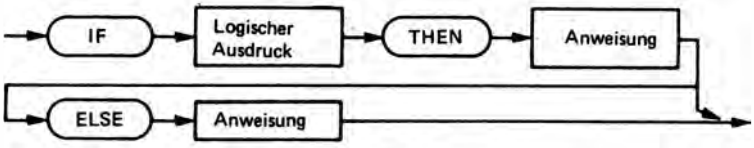
Gerätefilename



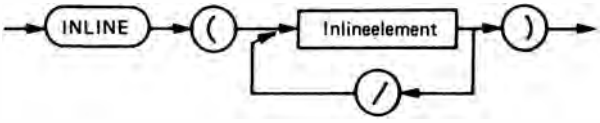
Hexadezimalzahl



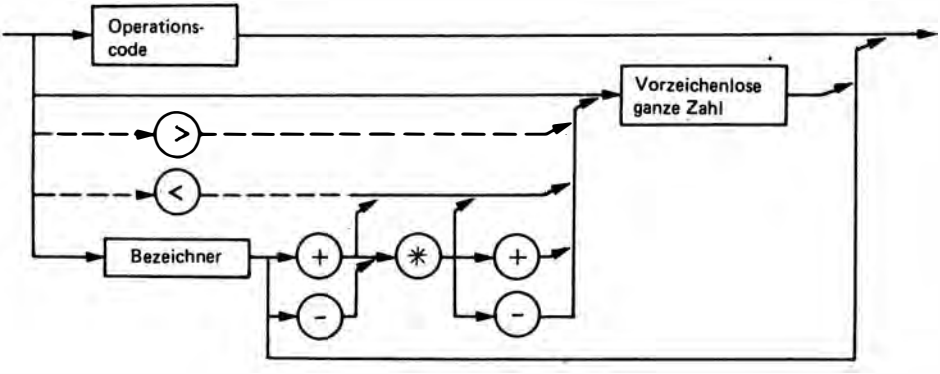
IF-Anweisung



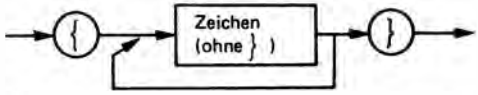
INLINE-Anweisung



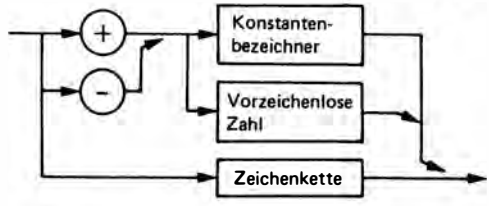
Inlineelement



Kommentar

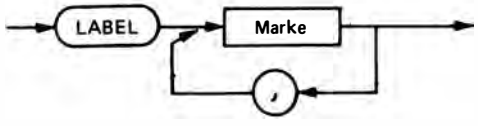


Konstanten



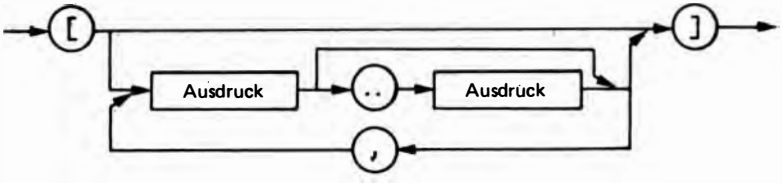
Konstantenbezeichner: Bezeichner

LABEL-Deklaration

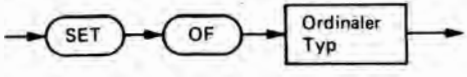


Logischer Ausdruck: Ausdruck
Marke

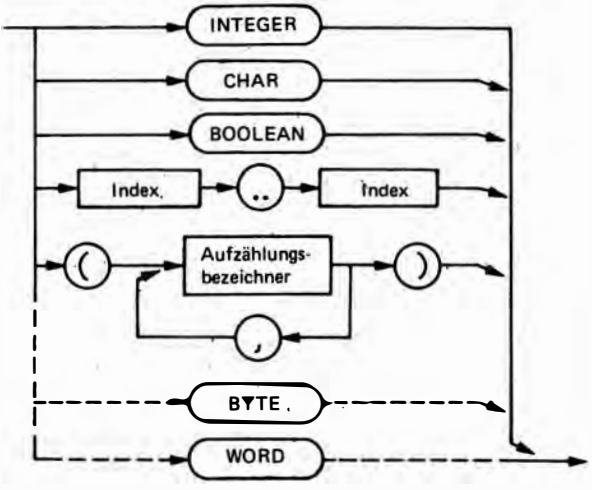
Mengenkonstruktor



Mengentyp

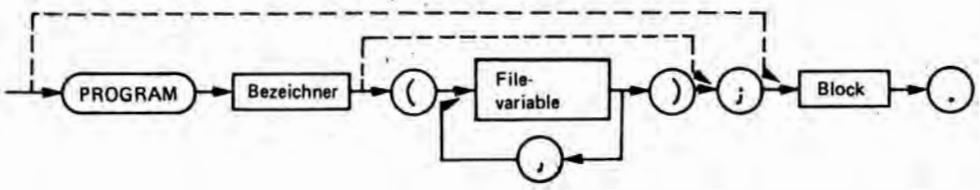


Ordinaler Typ



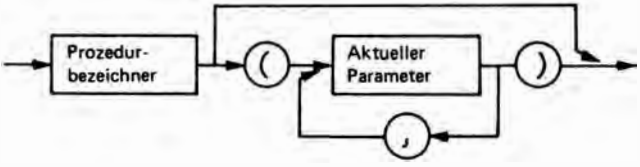
Aufzählungsbezeichner: Bezeichner
Index: Konstante (ordinaler Typ)

Programm

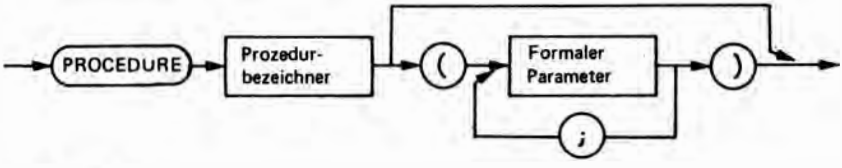


Filevariable: Variablenbezeichner (Filetyp)

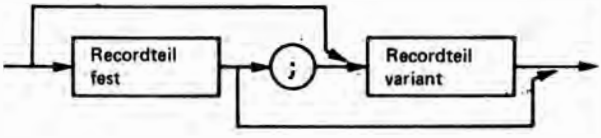
Prozedurbezeichner: Bezeichner
Prozeduraufruf



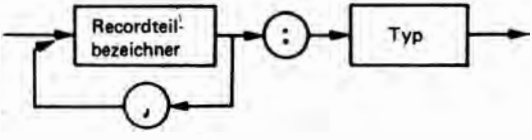
Prozedurkopf



Recordliste

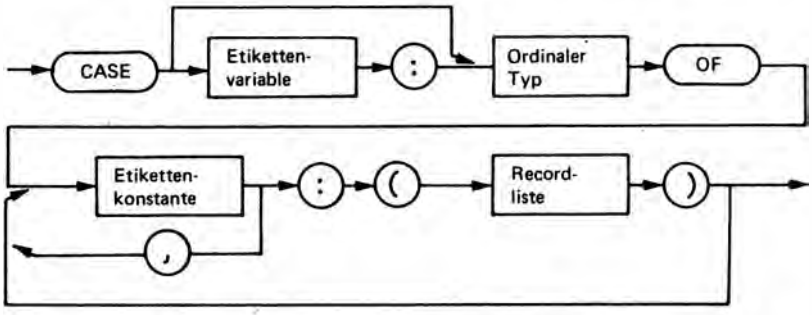


Recordteil fest



Recordteilbezeichner: Bezeichner

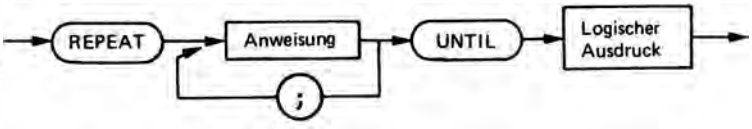
Recordteil variant



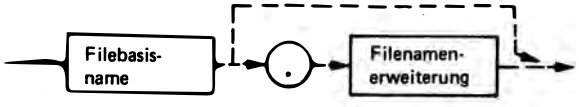
Etikettenvariable: Variable (ordinaler Typ)
Etikettenkonstante: Konstante (ordinaler Typ)

Recordtyp

REPEAT-Anweisung

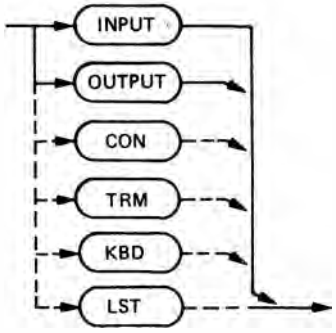


Speicherfilename (SCPX)

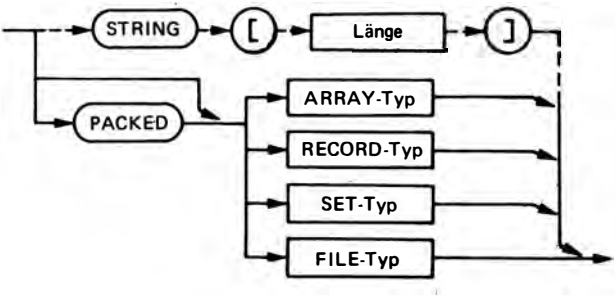


Filebasisname: max. 8 zulässige Zeichen
Filenamenerweiterung: max. 3 zulässige Zeichen
zulässige Zeichen: Zeichen ohne ⟨⟩.,:=;*?[]und Leerzeichen

Standardfilenamen

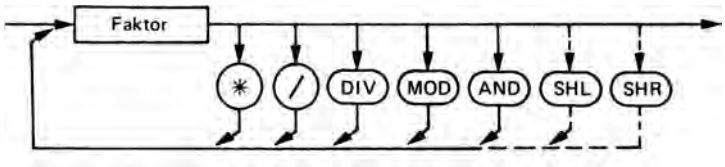


Strukturierter Typ

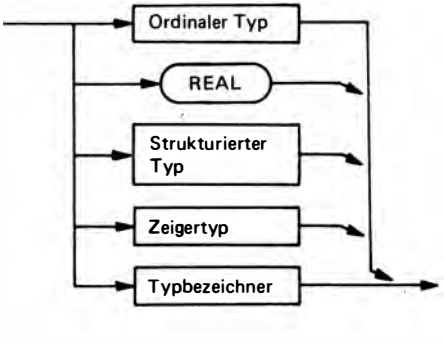


Länge: 1..255 oder Konstantenbezeichner mit diesem Wert

Term

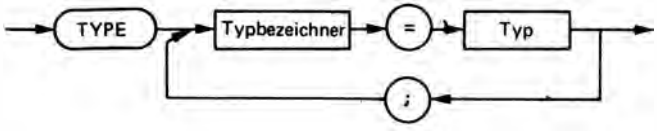


Typ

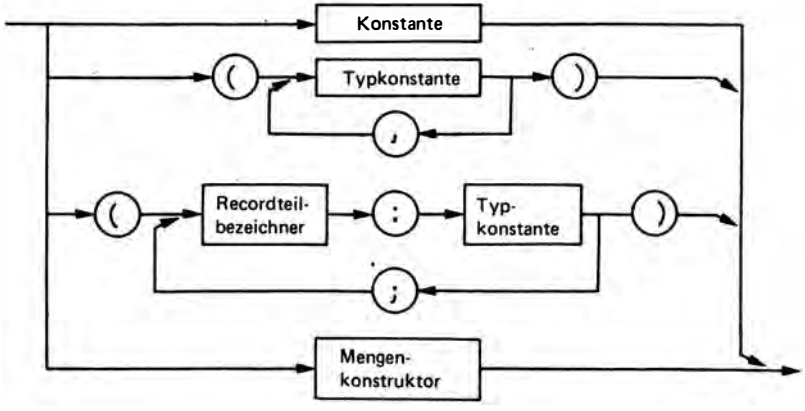


Typbezeichner: Bezeichner

TYPE-Definition

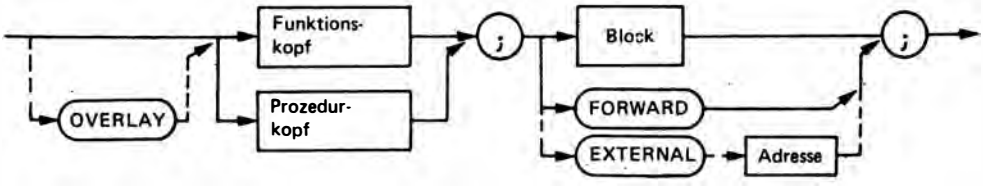


Typkonstante

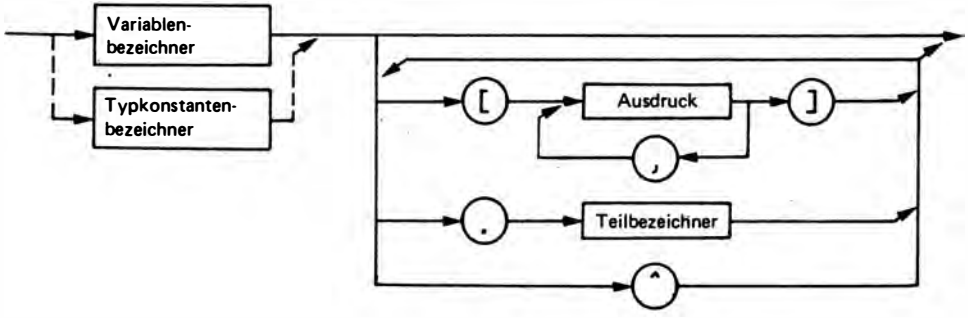


Typkonstantenbezeichner: Bezeichner

Unterprogrammdeklaration

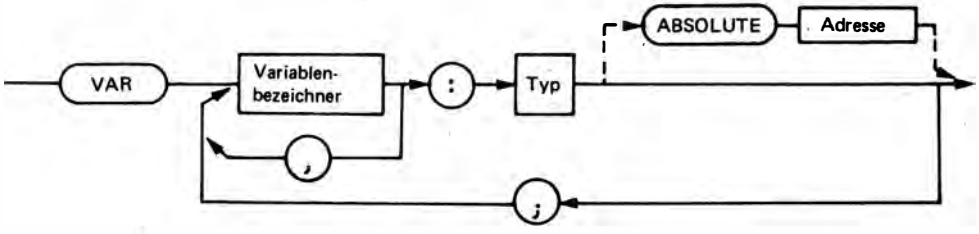


Variablen

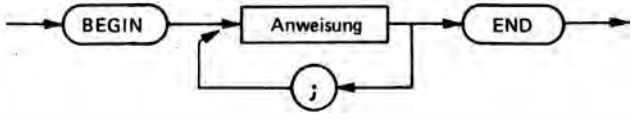


Variablenbezeichner: Bezeichner

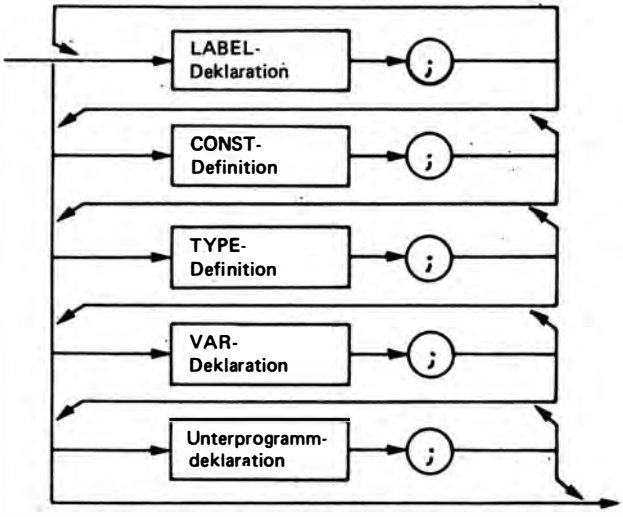
VAR(iablen)-Deklaration



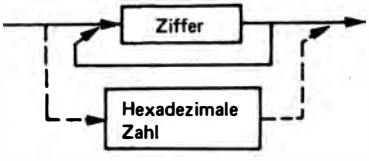
Verbundanweisung



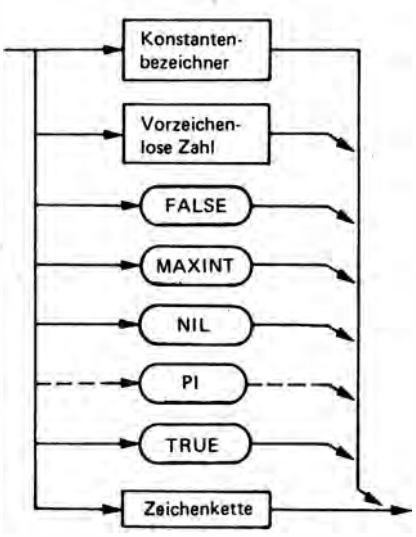
Vereinbarungsteil



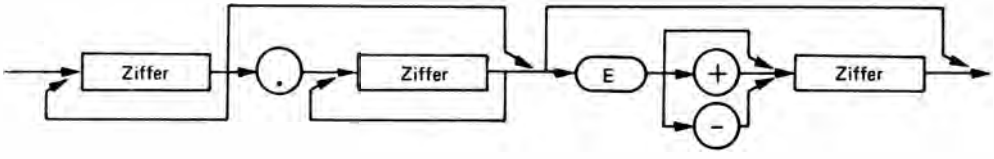
Vorzeichenlose ganze Zahl



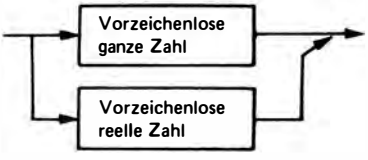
Vorzeichenlose Konstante



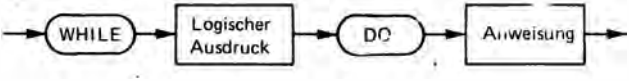
Vorzeichenlose reelle Zahl



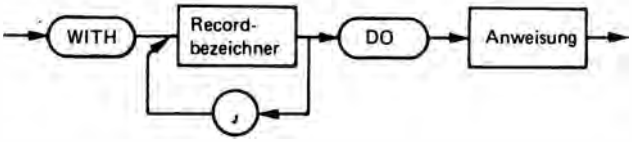
Vorzeichenlose Zahl



WHILE-Anweisung



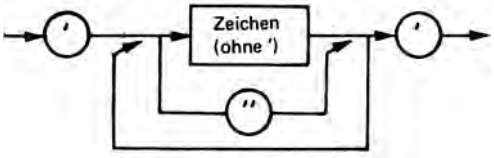
WITH-Anweisung



Recordbezeichner: Variablenbezeichner

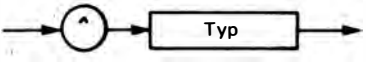
Zeichen (vgl. Anhang G)

Zeichenkette

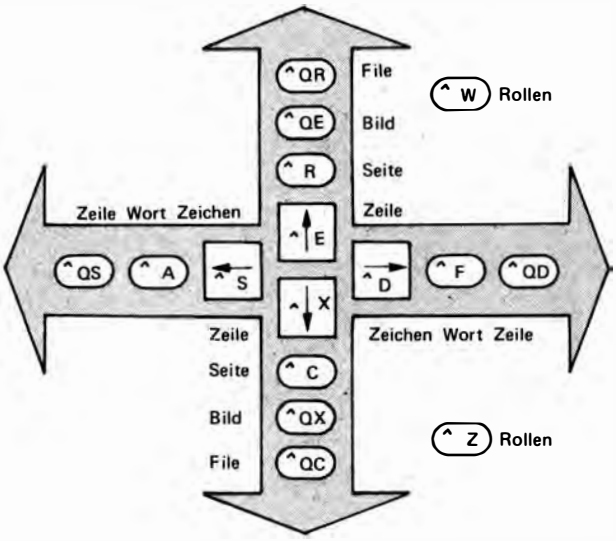


Ziffer: 0...9

Zeigertyp



Cursorbewegung



Löschen

^G	Zeichen über Cursor
^T	Wort rechts
^Y	Zeile
^DEL	Zeichen links
^QY	bis Zeilenende
^QL	Änderung zurück

Suchen/Ersetzen

^QF	Suchen
^QA	Suchen und Ersetzen
^L	Vorgang wiederholen
^QP	Cursor zurück

Block markieren

^KB	Blockanfang
^KK	Blockende
^KT	Wort (links)
^KH	Marken löschen

File lesen/schreiben (CP)

^KR	Lesen eines Files
^KW	Schreiben als Files

^ entspricht CTRL

Einfügen

^V	Einfügen/Ersetzen ein/aus
^N	Leerzeile

Sonstiges

^I	Tabulator
^QI	Einrücken ein/aus
^U	Abbruch
^KD	Ende

Optionen

U	Groß-/Kleinbuchstaben
W	ganze Worte
G	global (überall)
N	ohne Rückfrage
B	rückwärts
n	n-mal (n ist eine Zahl)

Block bewegen (an CP)/Löschen

^KC	Kopieren
^KV	Verschieben
^KY	Löschen

Cursorposition

^QB	Blockanfang
^QK	Blockende

CP Cursorposition

Konvertierung dezimal/binaer/hexadezimal im Bereich 0..65535(¤0..¤FFFF)

DEZ	BIN	HEX	¤0	¤1	¤2	¤3	¤4	¤5	¤6	¤7	¤8	¤9	¤A	¤B	¤C	¤D	¤E	¤F	00	000
0	0000	¤0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	0001	¤1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	0010	¤2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	0011	¤3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	0100	¤4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	0101	¤5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	0110	¤6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	0111	¤7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	1000	¤8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	1001	¤9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
10	1010	¤A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
11	1011	¤B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
12	1100	¤C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
13	1101	¤D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
14	1110	¤E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
15	1111	¤F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

Compilerdirektiven (Auswahl)

Compilerdirektiven werden mit {¤<Direktive>} an den Beginn einer Quelltextzeile geschrieben (implementationsabhaengige Alternativen in Klammer).

Direktive (Standard erstgenannt)	Wirkung
A+ (S-) A- (S+)	Keine Rekursion (absoluter Code) Rekursion zugelassen
B+ B-	Standardfile INPUT gleich CON Standardfile INPUT gleich TRM
C+ C-	Eingabeinterpretation ^C Programmabbruch ^S Unterbrechung Bildschirmausgabe CTRL-Zeichen werden nicht interpretiert (beschleunigter Ablauf)
I+ I-	E/A-Fehlerbehandlung durch das Laufzeit-/PASCAL-System E/A-Fehlerbehandlung ueber IORESULT durch den Programmierer
R- R+	Ohne Index- und Bereichsueberwachung waehrend der Laufzeit Index- und Bereichsueberwachung (langsamerer Ablauf)
U- U+	Keine Programmunterbrechung durch den Benutzer waehrend der Laufzeit Unterbrechung waehrend der Laufzeit mit ^C moeglich(langsamerer Ablauf)
V+ V-	Stringlaenge bei Parameteruebergabe wird geprueft Stringlaenge kann verschieden sein
Wn	Schachtelungstiefe von WITH-Anweisungen (n = 0..9;Standard 2)
X- X+	Zugriffgeschwindigkeit normal Zugriff auf Felder beschleunigt (hoeherer Speicherplatzbedarf)
I <Name>	File <Name> wird an dieser Stelle kopiert

X+ / X- steuert implementationsabhaengig auch Ueberlaufkontrollen fuer Integer und Real sowie bei Division durch Null.

Mikrorechnertypischer Zeichensatz (ASCII)

1. Steuerzeichen

DEZ	HEX	CTRL-Zeichen	Bezeichnung	Wirkungen(Auswahl) fuer Cursor oder Drucker fuer SCPX(versionsabhaengig)
0	00	^@	NUL	
1	01	^A	SOH	Cursor HOME(Position 1,1)
2	02	^B	STX	(+ ¤80) Cursor einschalten
3	03	^C	ETX	(+ ¤80) Cursor ausschalten
4	04	^D	EOT	(+ ¤80) Zeichendarstellung normal
5	05	^E	ENQ	(+ ¤80) Zeichendarstellung invers
6	06	^F	ACK	(+ ¤80) Zeichendarstellung intensiv
7	07	^G	BEL	akustisches u. optisches Signal
8	08	^H	BS	ein Zeichen zurueck
9	09	^I	HT	Tabulatorsprung
10	0A	^J	LF	Zeile nach unten
11	0B	^K	VT	
12	0C	^L	FF	Bildschirm loeschen und ¤01 oder Blattvorschub
13	0D	^M	CR	Anfang der laufenden Zeile
14	0E	^N	SO	
15	0F	^O	SI	
16	10	^P	DLE	
17	11	^Q	DC1	
18	12	^R	DC2	
19	13	^S	DC3	
20	14	^T	DC4	Loeschen des Bildschirms ab Cursorposition
21	15	^U	NAK	Zeichen nach rechts
22	16	^V	SYN	Loeschen der Zeile ab Cursorposition
23	17	^W	ETB	
24	18	^X	CAN	Loeschen Cursorzeile und ¤0D
25	19	^Y	EM	
26	1A	^Z	SUB	Zeile nach oben
27	1B	^[	ESC	Erstzeichen Cursorpositionierung
28	1C	^\	FS	
29	1D	^]	GS	
30	1E	^^	RS	
31	1F	^_	US	

2.Druckbare Zeichen

Ziffer	¤0	¤1	¤2	¤3	¤4	¤5	¤6	¤7	¤8	¤9	¤A	¤B	¤C	¤D	¤E	¤F
¤2		!	"	#	¤	%	&	'	(	)	*	+	,	-	.	/
¤3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
¤4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
¤5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
¤6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
¤7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Uebergangstabelle
von Assemblercode (Z80 - Mnemonik) zu INLINE-Operationscode
fuer 8 - Bit - Computer

Es bedeuten
n 8 - Bit - Parameter
nn 16 - Bit - Parameter
d 8 - Bit - Distanz(mit Vorzeichen)

Die Erlaeuterung des mnemonischen Assemblercodes und die Wirkungen der Befehle auf die Flagregister sind der Literatur zu entnehmen (zum Beispiel Classen,L.;Oefler,U.:Wissensspeicher Mikrorechnerprogrammierung.Berlin : VEB Verlag Technik 1986,S.19ff)

Mnemonischer Code	Operationscode
ADC A,(HL)	¤8E
ADC A,(IX+d)	¤DD 8Ed
ADC A,(IY+d)	¤FD 8Ed
ADC A,A	¤8F
ADC A,B	¤88
ADC A,C	¤89
ADC A,D	¤8A
ADC A,E	¤8B
ADC A,H	¤8C
ADC A,L	¤8D
ADC A,n	¤CE n
ADC HL,BC	¤ED 4A
ADC HL,DE	¤ED 5A
ADC HL,HL	¤ED 6A
ADC HL,SP	¤ED 7A
ADD A,(HL)	¤86
ADD A,(IX+d)	¤DD 86d
ADD A,(IY+d)	¤FD 86d
ADD A,A	¤87
ADD A,B	¤80
ADD A,C	¤81
ADD A,D	¤82
ADD A,E	¤83
ADD A,H	¤84
ADD A,L	¤85
ADD A,n	¤C6 n
ADD HL,BC	¤09
ADD HL,DE	¤19
ADD HL,HL	¤29
ADD HL,SP	¤39
ADD IX,BC	¤DD 09
ADD IX,DE	¤DD 19
ADD IX,IX	¤DD 29
ADD IX,SP	¤DD 39
ADD IY,BC	¤FD 09
ADD IY,DE	¤FD 19
ADD IY,IY	¤FD 29
ADD IY,SP	¤FD 39
AND (HL)	¤A6
AND (IX+d)	¤DD A6d
AND (IY+d)	¤FD A6d
AND A	¤A7

Mnemonischer Code	Operationscode
AND B	¤A0
AND C	¤A1
AND D	¤A2
AND E	¤A3
AND H	¤A4
AND L	¤A5
AND n	¤E6 n
BIT 0,(HL)	¤CB 46
BIT 0,(IX+d)	¤DD CBd46
BIT 0,(IY+d)	¤FD CBd46
BIT 0,A	¤CB 47
BIT 0,B	¤CB 40
BIT 0,C	¤CB 41
BIT 0,D	¤CB 42
BIT 0,E	¤CB 43
BIT 0,H	¤CB 44
BIT 0,L	¤CB 45
BIT 1,(HL)	¤CB 4E
BIT 1,(IX+d)	¤DD CBd4E
BIT 1,(IY+d)	¤FD CBd4E
BIT 1,A	¤CB 4F
BIT 1,B	¤CB 48
BIT 1,C	¤CB 49
BIT 1,D	¤CB 4A
BIT 1,E	¤CB 4B
BIT 1,H	¤CB 4C
BIT 1,L	¤CB 4D
BIT 2,(HL)	¤CB 56
BIT 2,(IX+d)	¤DD CBd56
BIT 2,(IY+d)	¤FD CBd56
BIT 2,A	¤CB 57
BIT 2,B	¤CB 50
BIT 2,C	¤CB 51
BIT 2,D	¤CB 52
BIT 2,E	¤CB 53
BIT 2,H	¤CB 54
BIT 2,L	¤CB 55
BIT 3,(HL)	¤CB 5E
BIT 3,(IX+d)	¤DD CBd5E
BIT 3,(IY+d)	¤FD CBd5E
BIT 3,A	¤CB 5F
BIT 3,B	¤CB 58

Mnemonischer Code	Operationscode
BIT 3,C	¤CB 59
BIT 3,D	¤CB 5A
BIT 3,E	¤CB 5B
BIT 3,H	¤CB 5C
BIT 3,L	¤CB 5D
BIT 4,(HL)	¤CB 66
BIT 4,(IX+d)	¤DD CBd66
BIT 4,(IY+d)	¤FD CBd66
BIT 4,A	¤CB 67
BIT 4,B	¤CB 60
BIT 4,C	¤CB 61
BIT 4,D	¤CB 62
BIT 4,E	¤CB 63
BIT 4,H	¤CB 64
BIT 4,L	¤CB 65
BIT 5,(HL)	¤CB 6E
BIT 5,(IX+d)	¤DD CBd6E
BIT 5,(IY+d)	¤FD CBd6E
BIT 5,A	¤CB 6F
BIT 5,B	¤CB 68
BIT 5,C	¤CB 69
BIT 5,D	¤CB 6A
BIT 5,E	¤CB 6B
BIT 5,H	¤CB 6C
BIT 5,L	¤CB 6D
BIT 6,(HL)	¤CB 76
BIT 6,(IX+d)	¤DD CBd76
BIT 6,(IY+d)	¤FD CBd76
BIT 6,A	¤CB 77
BIT 6,B	¤CB 70
BIT 6,C	¤CB 71
BIT 6,D	¤CB 72
BIT 6,E	¤CB 73
BIT 6,H	¤CB 74
BIT 6,L	¤CB 75
BIT 7,(HL)	¤CB 7E
BIT 7,(IX+d)	¤DD CBd7E
BIT 7,(IY+d)	¤FD CBd7E
BIT 7,A	¤CB 7F
BIT 7,B	¤CB 78
BIT 7,C	¤CB 79
BIT 7,D	¤CB 7A
BIT 7,E	¤CB 7B
BIT 7,H	¤CB 7C
BIT 7,L	¤CB 7D
CALL C,nn	¤DC nn
CALL M,nn	¤FC nn
CALL NC,nn	¤D4 nn
CALL nn	¤CD nn
CALL NZ,nn	¤C4 nn
CALL P,nn	¤F4 nn
CALL PE,nn	¤EC nn

Mnemonischer Code	Operationscode
CALL PO,nn	¤E4 nn
CALL Z,nn	¤CC nn
CCF	¤3F
CP (HL)	¤BE
CP (IX+d)	¤DD BEd
CP (IY+d)	¤FD BEd
CP A	¤BF
CP B	¤B8
CP C	¤B9
CP D	¤BA
CP E	¤BB
CP H	¤BC
CP L	¤BD
CP n	¤FE n
CPD	¤ED A9
CPDR	¤ED B9
CPI	¤ED A1
CPIR	¤ED B1
CPL	¤2F
DAA	¤27
DEC (HL)	¤35
DEC (IX+d)	¤DD 35d
DEC (IY+d)	¤FD 35d
DEC A	¤3D
DEC B	¤05
DEC BC	¤0B
DEC C	¤0D
DEC D	¤15
DEC DE	¤1B
DEC E	¤1D
DEC H	¤25
DEC HL	¤2B
DEC IX	¤DD 2B
DEC IY	¤FD 2B
DEC L	¤2D
DEC SP	¤3B
DI	¤F3
DJNZ d	¤10 d
EI	¤FB
EX (SP),HL	¤E3
EX (SP),IX	¤DD E3
EX (SP),IY	¤FD E3
EX AF,AF'	¤08
EX DE,HL	¤EB
EXX	¤D9
HALT	¤76
IM 0	¤ED 46
IM 1	¤ED 56
IM 2	¤ED 5E
IN A,(C)	¤ED 78
IN A,(n)	¤DB n
IN B,(C)	¤ED 40

Mnemonischer Code		Operations-code
IN	C,(C)	¤ED 48
IN	D,(C)	¤ED 50
IN	E,(C)	¤ED 58
IN	H,(C)	¤ED 60
IN	L,(C)	¤ED 68
INC	(HL)	¤34
INC	(IX+d)	¤DD 34d
INC	(IY+d)	¤FD 34d
INC	A	¤3C
INC	B	¤04
INC	BC	¤03
INC	C	¤0C
INC	D	¤14
INC	DE	¤13
INC	E	¤1C
INC	H	¤24
INC	HL	¤23
INC	IX	¤DD 23
INC	IY	¤FD 23
INC	L	¤2C
INC	SP	¤33
IND		¤ED AA
INDR		¤ED BA
INI		¤ED A2
INIR		¤ED B2
JP	(HL)	¤E9
JP	(IX)	¤DD E9
JP	(IY)	¤FD E9
JP	C,nn	¤DA nn
JP	M,nn	¤FA nn
JP	NC,nn	¤D2 nn
JP	nn	¤C3 nn
JP	NZ,nn	¤C2 nn
JP	P,nn	¤F2 nn
JP	PE,nn	¤EA nn
JP	PO,nn	¤E2 nn
JP	Z,nn	¤CA nn
JR	C,d	¤38 d
JR	d	¤18 d
JR	NC,d	¤30 d
JR	NZ,d	¤20 d
JR	Z,d	¤28 d
LD	(BC),A	¤02
LD	(DE),A	¤12
LD	(HL),A	¤77
LD	(HL),B	¤70
LD	(HL),C	¤71
LD	(HL),D	¤72
LD	(HL),E	¤73
LD	(HL),H	¤74
LD	(HL),L	¤75
LD	(HL),n	¤36 n

Mnemonischer Code		Operations-code
LD	(IX+d),A	¤DD 77d
LD	(IX+d),B	¤DD 70d
LD	(IX+d),C	¤DD 71d
LD	(IX+d),D	¤DD 72d
LD	(IX+d),E	¤DD 73d
LD	(IX+d),H	¤DD 74d
LD	(IX+d),L	¤DD 75d
LD	(IX+d),n	¤DD 36dn
LD	(IY+d),A	¤FD 77d
LD	(IY+d),B	¤FD 70d
LD	(IY+d),C	¤FD 71d
LD	(IY+d),D	¤FD 72d
LD	(IY+d),E	¤FD 73d
LD	(IY+d),H	¤FD 74d
LD	(IY+d),L	¤FD 75d
LD	(IY+d),n	¤FD 36dn
LD	(nn),A	¤32 nn
LD	(nn),BC	¤ED 43nn
LD	(nn),DE	¤ED 53nn
LD	(nn),HL	¤22 nn
LD	(nn),IX	¤DD 22nn
LD	(nn),IY	¤FD 22nn
LD	(nn),SP	¤ED 73nn
LD	A,(BC)	¤0A
LD	A,(DE)	¤1A
LD	A,(HL)	¤7E
LD	A,(IX+d)	¤DD 7Ed
LD	A,(IY+d)	¤FD 7Ed
LD	A,(nn)	¤3A nn
LD	A,A	¤7F
LD	A,B	¤78
LD	A,C	¤79
LD	A,D	¤7A
LD	A,E	¤7B
LD	A,H	¤7C
LD	A,I	¤ED 57
LD	A,L	¤7D
LD	A,n	¤3E n
LD	A,R	¤ED 5F
LD	B,(HL)	¤46
LD	B,(IX+d)	¤DD 46d
LD	B,(IY+d)	¤FD 46d
LD	B,A	¤47
LD	B,B	¤40
LD	B,C	¤41
LD	B,D	¤42
LD	B,E	¤43
LD	B,H	¤44
LD	B,L	¤45
LD	B,n	¤06 n
LD	BC,(nn)	¤ED 4Bnn
LD	BC,nn	¤01 nn

Mnemonischer Code		Operations-code
LD	C,(HL)	¤4E
LD	C,(IX+d)	¤DD 4Ed
LD	C,(IY+d)	¤FD 4Ed
LD	C,A	¤4F
LD	C,B	¤48
LD	C,C	¤49
LD	C,D	¤4A
LD	C,E	¤4B
LD	C,H	¤4C
LD	C,L	¤4D
LD	C,n	¤0E n
LD	D,(HL)	¤56
LD	D,(IX+d)	¤DD 56d
LD	D,(IY+d)	¤FD 56d
LD	D,A	¤57
LD	D,B	¤50
LD	D,C	¤51
LD	D,D	¤52
LD	D,E	¤53
LD	D,H	¤54
LD	D,L	¤55
LD	D,n	¤16 n
LD	DE,(nn)	¤ED 5Bnn
LD	DE,nn	¤11 nn
LD	E,(HL)	¤5E
LD	E,(IX+d)	¤DD 5Ed
LD	E,(IY+d)	¤FD 5Ed
LD	E,A	¤5F
LD	E,B	¤58
LD	E,C	¤59
LD	E,D	¤5A
LD	E,E	¤5B
LD	E,H	¤5C
LD	E,L	¤5D
LD	E,n	¤1E n
LD	H,(HL)	¤66
LD	H,(IX+d)	¤DD 66d
LD	H,(IY+d)	¤FD 66d
LD	H,A	¤67
LD	H,B	¤60
LD	H,C	¤61
LD	H,D	¤62
LD	H,E	¤63
LD	H,H	¤64
LD	H,L	¤65
LD	H,n	¤26 n
LD	HL,(nn)	¤2A nn
LD	HL,nn	¤21 nn
LD	I,A	¤ED 47
LD	IX,(nn)	¤DD 2Ann
LD	IX,nn	¤DD 21nn
LD	IY,(nn)	¤FD 2Ann

Mnemonischer Code		Operations-code
LD	IY,nn	¤FD 21nn
LD	L,(HL)	¤6E
LD	L,(IX+d)	¤DD 6Ed
LD	L,(IY+d)	¤FD 6Ed
LD	L,A	¤6F
LD	L,B	¤68
LD	L,C	¤69
LD	L,D	¤6A
LD	L,E	¤6B
LD	L,H	¤6C
LD	L,L	¤6D
LD	L,n	¤2E n
LD	R,A	¤ED 4F
LD	SP,(nn)	¤ED 7Bnn
LD	SP,HL	¤F9
LD	SP,IX	¤DD F9
LD	SP,IY	¤FD F9
LD	SP,nn	¤31 nn
LDD		¤ED A8
LDDR		¤ED B8
LDI		¤ED A0
LDIR		¤ED B0
NEG		¤ED 44
NOP		¤00
OR	(HL)	¤B6
OR	(IX+d)	¤DD B6d
OR	(IY+d)	¤FD B6d
OR	A	¤B7
OR	B	¤B0
OR	C	¤B1
OR	D	¤B2
OR	E	¤B3
OR	H	¤B4
OR	L	¤B5
OR	n	¤F6 n
OTDR		¤ED BB
OTIR		¤ED B3
OUT	(C),A	¤ED 79
OUT	(C),B	¤ED 41
OUT	(C),C	¤ED 49
OUT	(C),D	¤ED 51
OUT	(C),E	¤ED 59
OUT	(C),H	¤ED 61
OUT	(C),L	¤ED 69
OUT	(n),A	¤D3 n
OUTD		¤ED AB
OUTI		¤ED A3
POP	AF	¤F1
POP	BC	¤C1
POP	DE	¤D1
POP	HL	¤E1
POP	IX	¤DD E1

Mnemonischer Code	Operations-code
POP IY	¤FD E1
PUSH AF	¤F5
PUSH BC	¤C5
PUSH DE	¤D5
PUSH HL	¤E5
PUSH IX	¤DD E5
PUSH IY	¤FD E5
RES 0,(HL)	¤CB 86
RES 0,(IX+d)	¤DD CBd86
RES 0,(IY+d)	¤FD CBd86
RES 0,A	¤CB 87
RES 0,B	¤CB 80
RES 0,C	¤CB 81
RES 0,D	¤CB 82
RES 0,E	¤CB 83
RES 0,H	¤CB 84
RES 0,L	¤CB 85
RES 1,(HL)	¤CB 8E
RES 1,(IX+d)	¤DD CBd8E
RES 1,(IY+d)	¤FD CBd8E
RES 1,A	¤CB 8F
RES 1,B	¤CB 88
RES 1,C	¤CB 89
RES 1,D	¤CB 8A
RES 1,E	¤CB 8B
RES 1,H	¤CB 8C
RES 1,L	¤CB 8D
RES 2,(HL)	¤CB 96
RES 2,(IX+d)	¤DD CBd96
RES 2,(IY+d)	¤FD CBd96
RES 2,A	¤CB 97
RES 2,B	¤CB 90
RES 2,C	¤CB 91
RES 2,D	¤CB 92
RES 2,E	¤CB 93
RES 2,H	¤CB 94
RES 2,L	¤CB 95
RES 3,(HL)	¤CB 9E
RES 3,(IX+d)	¤DD CBd9E
RES 3,(IY+d)	¤FD CBd9E
RES 3,A	¤CB 9F
RES 3,B	¤CB 98
RES 3,C	¤CB 99
RES 3,D	¤CB 9A
RES 3,E	¤CB 9B
RES 3,H	¤CB 9C
RES 3,L	¤CB 9D
RES 4,(HL)	¤CB A6
RES 4,(IX+d)	¤DD CBdA6
RES 4,(IY+d)	¤FD CBdA6
RES 4,A	¤CB A7
RES 4,B	¤CB A0

Mnemonischer Code	Operations-code
RES 4,C	¤CB A1
RES 4,D	¤CB A2
RES 4,E	¤CB A3
RES 4,H	¤CB A4
RES 4,L	¤CB A5
RES 5,(HL)	¤CB AE
RES 5,(IX+d)	¤DD CBdAE
RES 5,(IY+d)	¤FD CBdAE
RES 5,A	¤CB AF
RES 5,B	¤CB A8
RES 5,C	¤CB A9
RES 5,D	¤CB AA
RES 5,E	¤CB AB
RES 5,H	¤CB AC
RES 5,L	¤CB AD
RES 6,(HL)	¤CB B6
RES 6,(IX+d)	¤DD CBdB6
RES 6,(IY+d)	¤FD CBdB6
RES 6,A	¤CB B7
RES 6,B	¤CB B0
RES 6,C	¤CB B1
RES 6,D	¤CB B2
RES 6,E	¤CB B3
RES 6,H	¤CB B4
RES 6,L	¤CB B5
RES 7,(HL)	¤CB BE
RES 7,(IX+d)	¤DD CBdBE
RES 7,(IY+d)	¤FD CBdBE
RES 7,A	¤CB BF
RES 7,B	¤CB B8
RES 7,C	¤CB B9
RES 7,D	¤CB BA
RES 7,E	¤CB BB
RES 7,H	¤CB BC
RES 7,L	¤CB BD
RET	¤C9
RET C	¤D8
RET M	¤F8
RET NC	¤D0
RET NZ	¤C0
RET P	¤F0
RET PE	¤E8
RET PO	¤E0
RET Z	¤C8
RETI	¤ED 4D
RETN	¤ED 45
RL (HL)	¤CB 16
RL (IX+d)	¤DD CBd16
RL (IY+d)	¤FD CBd16
RL A	¤CB 17
RL B	¤CB 10
RL C	¤CB 11

Mnemonischer Code	Operations-code
RL D	¤CB 12
RL E	¤CB 13
RL H	¤CB 14
RL L	¤CB 15
RLA	¤17
RLC (HL)	¤CB 06
RLC (IX+d)	¤DD CBd06
RLC (IY+d)	¤FD CBd06
RLC A	¤CB 07
RLC B	¤CB 00
RLC C	¤CB 01
RLC D	¤CB 02
RLC E	¤CB 03
RLC H	¤CB 04
RLC L	¤CB 05
RLCA	¤07
RLD	¤ED 6F
RR (HL)	¤CB 1E
RR (IX+d)	¤DD CBd1E
RR (IY+d)	¤FD CBd1E
RR A	¤CB 1F
RR B	¤CB 18
RR C	¤CB 19
RR D	¤CB 1A
RR E	¤CB 1B
RR H	¤CB 1C
RR L	¤CB 1D
RRA	¤1F
RRC (HL)	¤CB 0E
RRC (IX+d)	¤DD CBd0E
RRC (IY+d)	¤FD CBd0E
RRC A	¤CB 0F
RRC B	¤CB 08
RRC C	¤CB 09
RRC D	¤CB 0A
RRC E	¤CB 0B
RRC H	¤CB 0C
RRC L	¤CB 0D
RRCA	¤0F
RRD	¤ED 67
RST 0	¤C7
RST 8	¤CF
RST 10H	¤D7
RST 18H	¤DF
RST 20H	¤E7
RST 28H	¤EF
RST 30H	¤F7
RST 38H	¤FF
SBC A,(HL)	¤9F
SBC A,(IX+d)	¤DD 9Ed
SBC A,(IY+d)	¤FD 9Ed
SBC A,A	¤9F

Mnemonischer Code	Operations-code
SBC A,B	¤98
SBC A,C	¤99
SBC A,D	¤9A
SBC A,E	¤9B
SBC A,H	¤9C
SBC A,L	¤9D
SBC A,n	¤DE n
SBC HL,BC	¤ED 42
SBC HL,DE	¤ED 52
SBC HL,HL	¤ED 62
SBC HL,SP	¤ED 72
SCF	¤37
SET 0,(HL)	¤CB C6
SET 0,(IX+d)	¤DD CBdC6
SET 0,(IY+d)	¤FD CBdC6
SET 0,A	¤CB C7
SET 0,B	¤CB C0
SET 0,C	¤CB C1
SET 0,D	¤CB C2
SET 0,E	¤CB C3
SET 0,H	¤CB C4
SET 0,L	¤CB C5
SET 1,(HL)	¤CB CE
SET 1,(IX+d)	¤DD CBdCE
SET 1,(IY+d)	¤FD CBdCE
SET 1,A	¤CB CF
SET 1,B	¤CB C8
SET 1,C	¤CB C9
SET 1,D	¤CB CA
SET 1,E	¤CB CB
SET 1,H	¤CB CC
SET 1,L	¤CB CD
SET 2,(HL)	¤CB D6
SET 2,(IX+d)	¤DD CBdD6
SET 2,(IY+d)	¤FD CBdD6
SET 2,A	¤CB D7
SET 2,B	¤CB D0
SET 2,C	¤CB D1
SET 2,D	¤CB D2
SET 2,E	¤CB D3
SET 2,H	¤CB D4
SET 2,L	¤CB D5
SET 3,(HL)	¤CB DE
SET 3,(IX+d)	¤DD CBdDE
SET 3,(IY+d)	¤FD CBdDE
SET 3,A	¤CB DF
SET 3,B	¤CB D8
SET 3,C	¤CB D9
SET 3,D	¤CB DA
SET 3,E	¤CB DB
SET 3,H	¤CB DC
SET 3,L	¤CB DD

Mnemonischer Code	Operationscode
SET 4,(HL)	¤CB E6
SET 4,(IX+d)	¤DD CBdE6
SET 4,(IY+d)	¤FD CBdE6
SET 4,A	¤CB E7
SET 4,B	¤CB E0
SET 4,C	¤CB E1
SET 4,D	¤CB E2
SET 4,E	¤CB E3
SET 4,H	¤CB E4
SET 4,L	¤CB E5
SET 5,(HL)	¤CB EE
SET 5,(IX+d)	¤DD CBdEE
SET 5,(IY+d)	¤FD CBdEE
SET 5,A	¤CB EF
SET 5,B	¤CB E8
SET 5,C	¤CB E9
SET 5,D	¤CB EA
SET 5,E	¤CB EB
SET 5,H	¤CB EC
SET 5,L	¤CB ED
SET 6,(HL)	¤CB F6
SET 6,(IX+d)	¤DD CBdF6
SET 6,(IY+d)	¤FD CBdF6
SET 6,A	¤CB F7
SET 6,B	¤CB F0
SET 6,C	¤CB F1
SET 6,D	¤CB F2
SET 6,E	¤CB F3
SET 6,H	¤CB F4
SET 6,L	¤CB F5
SET 7,(HL)	¤CB FE
SET 7,(IX+d)	¤DD CBdFE
SET 7,(IY+d)	¤FD CBdFE
SET 7,A	¤CB FF
SET 7,B	¤CB F8
SET 7,C	¤CB F9
SET 7,D	¤CB FA
SET 7,E	¤CB FB
SET 7,H	¤CB FC
SET 7,L	¤CB FD
SLA (HL)	¤CB 26
SLA (IX+d)	¤DD CBd26
SLA (IY+d)	¤FD CBd26
SLA A	¤CB 27
SLA B	¤CB 20
SLA C	¤CB 21

Mnemonischer Code	Operationscode
SLA D	¤CB 22
SLA E	¤CB 23
SLA H	¤CB 24
SLA L	¤CB 25
SRA (HL)	¤CB 2E
SRA (IX+d)	¤DD CBd2E
SRA (IY+d)	¤FD CBd2E
SRA A	¤CB 2F
SRA B	¤CB 28
SRA C	¤CB 29
SRA D	¤CB 2A
SRA E	¤CB 2B
SRA H	¤CB 2C
SRA L	¤CB 2D
SRL (HL)	¤CB 3E
SRL (IX+d)	¤DD CBd3E
SRL (IY+d)	¤FD CBd3E
SRL A	¤CB 3F
SRL B	¤CB 38
SRL C	¤CB 39
SRL D	¤CB 3A
SRL E	¤CB 3B
SRL H	¤CB 3C
SRL L	¤CB 3D
SUB (HL)	¤96
SUB (IX+d)	¤DD 96d
SUB (IY+d)	¤FD 96d
SUB A	¤97
SUB B	¤90
SUB C	¤91
SUB D	¤92
SUB E	¤93
SUB H	¤94
SUB L	¤95
SUB n	¤D6 n
XOR (HL)	¤AE
XOR (IX+d)	¤DD AEd
XOR (IY+d)	¤FD AEd
XOR A	¤AF
XOR B	¤A8
XOR C	¤A9
XOR D	¤AA
XOR E	¤AB
XOR H	¤AC
XOR L	¤AD
XOR n	¤E n

```
PROCEDURE Inverse(VAR a:Matrix;n:INTEGER;VAR Richtig:BOOLEAN);
{Iversion der Matrix a(i,j),i=j=1..n, mit Pivotierung
 Der Erfolg wird mit Richtig=TRUE uebermittelt}
VAR ix,iy,iq       : ARRAY[1..Grenze] OF INTEGER;
    max,d,t        : REAL;
    i,j,k,l,is,iz : INTEGER;
BEGIN
  Richtig := TRUE;
  IF n > grenze THEN Richtig := FALSE
  ELSE BEGIN
    FOR j := 1 TO n DO iq[j] := 0;
{Ermittlung Index groesstes Element}
    FOR j := 1 TO n DO BEGIN
      max := 0;
      FOR i := 1 TO n DO BEGIN
        IF iq[i] <> 1 THEN BEGIN
          FOR k  := 1 TO n DO
          IF(iq[k] <> 1) AND (max <= abs(a[i,k])) THEN BEGIN
            is  := k;  iz := i;
            max := abs(a[i,k]);
          END;
        END;
      END;
      iq[is] := iq[is]+1;
{Pivotierung}
      IF iz <> is THEN FOR i := 1 TO n DO BEGIN
        t       := a[iz,i];
        a[iz,i] := a[is,i];
        a[is,i] := t;
      END;
      ix[j] := iz; iy[j] := is;
{Kontrolle auf Boesartigkeit}
      IF abs(a[is,is]) <= 1E-6 THEN Richtig := FALSE
      ELSE BEGIN
        d       := a[is,is];
        a[is,is] := 1;
        FOR i := 1 TO n DO a[is,i] := a[is,i] / d;
{Reduktion aller anderen Zeilen}
        FOR l := 1 TO n DO IF l <> is THEN BEGIN
          t       := a[l,is];
          a[l,is] := 0;
          FOR i := 1 TO n DO a[l,i] := a[l,i] - a[is,i] * t;
        END;
      END;
    END;
    IF Richtig THEN BEGIN
{Ruecktausch}
      FOR j := 1 TO n DO BEGIN
        i := n + 1 - j;
        IF ix[i] <> iy[i] THEN BEGIN
          iz := ix[i]; is := iy[i];
          FOR l := 1 TO n DO BEGIN
            t       := a[l,iz];
            a[l,iz] := a[l,is];
            a[l,is] := t;
          END;
        END;
      END;
    END;
  END;
END;
```

```
PROGRAM Real_Test;
CONST Maximum = 25;
VAR Kommastellen,i :INTEGER;
    Realzahl       :REAL;
BEGIN
  writeln('Test der Genauigkeit interner Real-Darstellungen');
  writeln('Fuer Anzeigegenauigkeit = 5 erhalten Sie:');
  Kommastellen := 5;
  REPEAT
    Realzahl:= -0.5;
    writeln('Anzeige fuer ',Kommastellen,' Stellen nach dem Komma');
    writeln;
    FOR i := 1 TO 11 DO BEGIN
      writeln(Realzahl:Maximum:Kommastellen);
      Realzahl := Realzahl + 0.1
    END;
    write('Anzeigegenauigkeit(groesser 5,kleiner ',Maximum,'): ');
    read(Kommastellen);
    writeln;
  UNTIL (Kommastellen <= 5) OR (Kommastellen > Maximum);
  write('Ende');
END.
```

```
PROCEDURE Regression(x,y:Feld;n:INTEGER;VAR a,b,r,t:REAL);
{Berechnung von a und b in Regressionsgerade y = a + bx sowie
 Korrelationskoeffizient  r und Wert t fuer T - Test
 aus den Zeitreihen x[i];y[i],i = 1(1)n}
VAR s               : ARRAY[1..5] OF REAL;
    i               : INTEGER;
    xq,yq,xx,yy,xy : REAL;
BEGIN
  FOR i := 1 TO 5 DO s[i] := 0;
  FOR i := 1 TO n DO BEGIN
    s[1] := s[1] + x[i];
    s[2] := s[2] + y[i];
    s[3] := s[3] + sqr(x[i]);
    s[4] := s[4] + sqr(y[i]);
    s[5] := s[5] + x[i] * y[i];
  END;
  xq := s[1] / n;
  yq := s[2] / n;
  xx := s[3] - s[1] * xq;
  yy := s[4] - s[2] * yq;
  xy := s[5] - s[1] * yq;
{Berechnung der Parameter der Regressionsgeraden}
  b  := xy / xx;
  a  := yq - b * xq;
{Berechnung des Korrelationskoeffizenten}
  r  := xy / (sqrt(xx * yy));
{Berechnung Wert fuer T-Test}
  t  := b / sqrt((yy - b * xy) / (n - 2) * xx);
END;
```

```
PROCEDURE Sortieren(Von,Bis:INTEGER);
{Schnellsortierung(rekursiv) fuer "VAR Feld: Struktur" des
 Basistyps "Element"(INTEGER,REAL,STRING,..)}
{¤A-}
VAR v,b,m     : INTEGER;
    Ab_Auf    : BOOLEAN;

PROCEDURE Tauschen(VAR Eins,Zwei : INTEGER);
{Vertauschen der Belegungen "Eins" und "Zwei"}
VAR Sichern : Element;
BEGIN {Tauschen}
  Sichern := Eins;
  Eins    := Zwei;
  Zwei    := Sichern
END; {Tauschen}

PROCEDURE Zentrieren(a,e : INTEGER);
{Der mittlere von drei Werten wird Inhalt der Position "a"}
VAR m : INTEGER;
BEGIN {Zentrieren}
  m := (a + e) DIV 2;
  IF Feld[m] > Feld[a] THEN Tauschen(Feld[m],Feld[a]);
  IF Feld[e] < Feld[a] THEN Tauschen(Feld[e] ,Feld[a]);
  IF Feld[m] > Feld[a] THEN Tauschen(Feld[m],Feld[a]);
END;  {Zentrieren}

BEGIN {Sortieren}
  IF Von < Bis THEN BEGIN
    Zentrieren(Von,Bis);
    Ab_Auf := TRUE;
    v := Von;
    b := Bis;
    m := Von;
    WHILE b > v DO BEGIN
      IF Ab_Auf THEN BEGIN
        WHILE NOT(Feld[b] < Feld[m]) AND (b > m) DO b := b - 1;
        IF b > m THEN BEGIN
          Tauschen(Feld[b],Feld[m]);
          m := b
        END;
        Ab_Auf := FALSE
      END ELSE BEGIN
        WHILE NOT(Feld[v] > Feld[m]) AND (v < m) DO v := v + 1;
        IF v < m THEN BEGIN
          Tauschen(Feld[v],Feld[m]);
          m := v
        END;
        Ab_Auf := TRUE
      END;
    END;
    Sortieren(Von, m - 1);
    Sortieren(m + 1, Bis)
  END;
END;{Sortieren}
```

Vordefinierte Funktionen und Prozeduren in alphabetischer Reihenfolge (Auswahl)

Es bedeuten
- c Konstante oder Variable vom Typ CHAR
- f Filevariable
- i Ausdruck des Typs INTEGER
- m Mengenvariable
- n Konstante des Typs INTEGER
- o Ausdruck ordinalen Typs
- r Ausdruck des Typs REAL
- s Stringkonstante oder -variable
- v Variable ohne Erlaeuterung einfach oder indiziert
- z Variable des Zeigertyps

BYTE ist Teilbereich von INTEGER. Die Notation "ir","irz" drueckt aus, dass jeder dieser Typen erlaubt ist.

Fuer 16-Bit-Systeme gelten in Funktionen und Prozeduren, die mit Speicherumfang und Adressen arbeiten, folgende Besonderheiten
1. Angaben zum Speicherumfang erfolgen nicht in Byte, sondern in Abschnitten (Paragraphen) zu 16 Byte.
2. Adressen und Zeigervariablen belegen 32 Bit fuer Segment- und Offsetadressteil. Es gibt die Funktionen CSEG, DSEG, OFS, SEG und SSEG zu ihrer Behandlung.
3. CHAIN und EXECUTE beduerfen einer Speichervoreinstellung.

Fuer Funktionen beginnt der Erlaeuterungstext mit "Ergebnistyp".

ABS (ir)	Ergebnistyp wie Argument. Absolutbetrag von ir.
ADDR (v)	Ergebnistyp Zeiger. Adresse des ersten Bytes der einfachen oder strukturierten Variablen v.
APPEND (f)	Eroeffnung des Files f und Positionierung des Filefensters auf das Fileende. EOF (f) wird TRUE.
ARCTAN (ir)	Ergebnistyp wie Argument. Ergebnis im Bogenmass(90 Grad = PI/2).
ASSIGN (f,s)	Verbindung der Filevariablen f mit dem Speicher- oder Geraetefilenamen s.
BDOS (n) BDOS (n,p) BDOSHL (n)	Aufruf als Funktion oder Prozedur zur Nutzung des BDOS von SCPX. n ist die Funktionsnummer, p der Aufrufparameter (vgl. Anhang K). Rueckgabewerte werden durch Aufruf als Funktion verfuegbar. Ist Rueckgabe im Register HL vorgesehen, hat der Aufruf mit BDOSHL zu erfolgen.

BIOS (n) BIOS (n,p) BIOSHL(n)	Aufruf als Funktion oder Prozedur zur direkten Nutzung der Treiberroutinen des Laufzeitsystems SCPX. Die Erlaeuterungen zu BDOS gelten sinngemaess. Die Aufrufparameter enthaelt Anhang K.
BLOCKREAD (f,v,i)	Lesen von i Saetzen zu 128 Byte des Files f beliebigen Typs(ungetypt) auf den Speicherplatz der (strukturierten) Variablen v.
BLOCKWRITE (f,v,i)	Schreiben von i Saetzen zu 128 Byte als Komponente des ungetypten Files f mit dem Inhalt ab Adresse der (strukturierten) Variablen v.
CARD (m)	Ergebnistyp INTEGER. Kardinalzahl (Anzahl der Elemente) von m.
CHAIN (f)	Start des Chainfiles f. Implementationsabhaengig auch wie EXECUTE.
CHR (i)	Ergebnistyp CHAR. Pseudofunktion fuer Typkonvertierung.
CLOCK	Ergebnistyp ARRAY[1..8] OF CHAR. Systemzeit in der Form hh.mm.ss.
CLOSE (f)	Schliessen des Files f und Aktualisierung des Diskettenverzeichnisses (Directory).
CLRBIT (v,i)	Das i-te Bit in v wird geloescht (die Zaehlung beginnt mit 0). v ist vom ordinalen Typ.
CLREOL	Loeschen der Bildschirmzeile ab Cursorposition (wie Steuerzeichen ¤16).
CLRSCR	Loeschen des Bildschirms und Cursor HOME (Position 1,1; wie Steuerzeichen ¤0C).
CONCAT (s1,s2,..)	Ergebnistyp STRING. Zusammenfuegen von Strings (implementationsabhaengig auch mit +).
COPY (s,i1,i2)	Ergebnistyp STRING. Kopieren des Strings ab Position i1 mit i2 Zeichen.

COS (ir)	Ergebnistyp wie Argument. Cosinus von ir. Argument im Bogenmass (90 Grad = PI/2).
DATE	Ergebnistyp ARRAY[1..8] OF CHAR. Systemdatum in der Form tt.mm.jj.
DELETE (s,i1,i2)	Ab Position i1 werden in s genau i2 Zeichen geloescht.
DELAY (i)	Verzoegerung um etwa i Millisekunden.
DELLINE	Loeschen der Zeile, in der sich der Cursor befindet (wie Steuerzeichen ¤18).
DISPOSE (z)	Freigabe des Speicherplatzes fuer die mit z adressierte dynamische Variable. Unvertraeglich mit RELEASE.
EOF (f)	Ergebnistyp BOOLEAN. Die Funktion liefert TRUE, wenn das Ende von f erreicht ist, sonst FALSE.
EOLN (f)	Ergebnistyp BOOLEAN. Der Wert ist TRUE, wenn im Textfile f die Steuerzeichen ¤0D ¤0A erkannt werden (bei Geraetefiles Endetaste <ET>).
ERASE (f)	Loeschen des geschlossenen, mit ASSIGN spezifizierten Files f. Implementationsabhaengig auch PURGE.
EXECUTE (f)	Starten eines COM-Files. Implementationsabhaengig auch CHAIN.
EXP (ir)	Ergebnistyp wie Argument. e hoch ir.
EXIT	Verlassen des aktuellen Blocks. In Hauptprogrammen Rueckkehr zum Laufzeitsystem.
FILEPOS (f)	Ergebnistyp INTEGER. Aktuelle Position des Filefensters im Binaerfile f.
FILESIZE (f)	Ergebnistyp INTEGER. Anzahl der Komponenten des Binaerfiles f.

FILLCHAR (v,i,c)	Ab Adresse der Variablen v werden i Zeichen c geschrieben. Fuer c ist auch Typ BYTE zulaessig.
FRAC (r)	Ergebnistyp REAL. Gebrochener Teil von r.
FREEMEM (z,i)	Freigabe von i Byte des mit z adressierten und GETMEM reservierten Speicherplatzes.
GET (f)	Lesen einer Komponente des Speicherfiles f (Binaerfile) in den Puffer f^.
GETMEM (z,i)	Ab Adresse z werden i Byte Speicherplatz reserviert.
GOTOXY (i1,i2)	Der Cursor wird auf Spalte i1, Zeile i2 positioniert (die Zaehlung beginnt mit 1).
HALT	Abbruch der Programmausfuehrung und Rueckkehr in das Laufzeitsystem.
HI (iz)	Ergebnistyp INTEGER. Hoeherwertiges Byte von iz.
INSERT (s1,s2,i)	s1 wird ab Position i in s2 eingefuegt.
INSLINE	Ab Cursorposition wird eine Leerzeile eingefuegt.
INT (ir)	Ergebnistyp wie Argument. Ganzer Teil von ir.
IORESULT	Ergebnistyp INTEGER. Fehlercode bei Ein- und Ausgabeoperationen (0: ohne Fehler)
KEYPRESSED	Ergebnistyp BOOLEAN. Die Funktion liefert TRUE, wenn die Tastatur betaetigt wurde, sonst FALSE.
LENGTH (s)	Ergebnistyp INTEGER. Aktuelle Laenge von s.
LN (ir)	Ergebnistyp REAL. Natuerlicher Logarithmus.
LO (iz)	Ergebnistyp INTEGER. Niederwertiger Teil von iz.

MARK (z)	Der Variablen z wird der aktuelle Wert des Haldenzeigers fuer ein spaeteres RELEASE zugewiesen. z darf sich zwischen MARK und RELEASE nicht aendern.
MAXAVAIL	Ergebnistyp INTEGER. Groesster zusammenhaengender freier Speicherblock in Byte.
MEMAVAIL	Ergebnistyp INTEGER. Freier Bereich zwischen Halde und Keller in Byte.
MOVE (v1,v2,i)	Der Inhalt von v1 wird mit i Byte nach v2 kopiert.
NEW (z)	Der Variablen z wird die Adresse des Speicherplatzes fuer die dynamische Variable z^ zugewiesen und der Speicherplatz reserviert.
ODD (i)	Ergebnistyp BOOLEAN. Die Funktion liefert TRUE, wenn i ungerade ist, sonst FALSE.
OPEN (f,s)	Zusammenfassung der Folge ASSIGN, RESET fuer File f mit dem Namen s. Implementationsabhaengig auch mit Fehlercoderueckgabe ueber IORESULT.
ORD (o)	Ergebnistyp INTEGER. Ordnungsnummer des Elementes o im ordinalen Typ.
OVRDRIVE (i)	Laufwerkszuweisung fuer OVERLAY-Files (i=0: aktuelles Laufwerk, i=1: Laufwerk A, i=2: Laufwerk B) usw.
PAGE (f)	f kann entfallen. Bildschirmloeschen/Cursor HOME oder Blattvorschub fuer Drucker (wie Steuerzeichen #0C).
PACK (u,i,g)	Uebertragung vom ungepackten Feld u, beginnend mit u[i], in das gepackte Feld g.
PARAMCOUNT	Ergebnistyp INTEGER. Anzahl der Kommandozeilenparameter.

PARAMSTR (i)	Ergebnistyp STRING. Der i-te Kommandozeilenparameter wird bereitgestellt.
POS (s1,s2)	Ergebnistyp INTEGER. Das Muster s1 wird in s2 gesucht. Es wird die Position ermittelt, ab der s1 in s2 erstmalig gefunden wurde. Ist s1 nicht in s2 enthalten, liefert die Funktion den Wert 0.
PRED (o)	Ergebnistyp wie Argument. Vorgaenger von o. i := pred (i) ist schneller als i := i - 1.
PTR (i)	Ergebnistyp Zeiger. Pseudofunktion fuer Typvertraeglichkeit. Implementationsabhaengig auch WRD.
PUT (f)	Schreiben einer Filekomponente aus dem Puffer f^ in das binaere Speicherfile f.
RANDOM	Ergebnistyp REAL. Gleichverteilte Zufallszahl zwischen 0 und 1.
RANDOM (i)	Ergebnistyp INTEGER. Erzeugung einer Zufallszahl zwischen 0 und i-1.
RANDOMSIZE	Der Zufallsgenerator wird in einen definierten Anfangszustand versetzt.
READ (f,...) READLN (f,...)	f kann entfallen. Lesen von File f oder vom Standardfile. Die Prozeduren sind in Abschnitt 2.3. ausfuehrlich beschrieben.
READHEX (f,v,n)	Vom Textfile f werden Daten in hexadezimaler Form gelesen und in der Breite von 1 (n=1) oder 2 (n=2) Byte in v eingetragen. v ist vom ordinalen Typ.
RELEASE (z)	Die Halde dynamischer Variablen wird ab der mit MARK (z) fixierten Adresse freigegeben.
RENAME (f,s)	Das (geschlossene) mit ASSIGN spezifizierte File erhaelt den Namen s.

RESET (f)	Eroeffnen eines existierenden Files f und Positionierung des Filefensters auf die erste Komponente. Implementationsabhaengig wird die erste Komponente auch gelesen.
REWRITE (f)	Eroeffnung eines nichtexistierenden oder zu ueberschreibenden Files f.
ROUND (r)	Ergebnistyp INTEGER. Addition von 0.5 zu r und Abtrennen des gebrochenen Teils.
SEEK (f,i)	Positionierung des Filefensters auf die i-1-te Komponente von f (die Zaehlung der Filekomponenten beginnt mit 0).
SEEKREAD (f,i)	Positionierung des Filefensters auf die Komponente i von f und Lesen in den Puffer f^ (implementationsabhaengig fuer die Folge SEEK,READ).
SEEKWRITE (f,i)	Positionierung des Filefensters von f und Schreiben des Puffers f^ als Komponente i (implementationsabhaengig fuer die Folge SEEK,WRITE).
SETBIT (v,i)	Das i-te Bit der Variablen v wird gesetzt (die Zaehlung beginnt mit 0). v ist vom ordinalen Typ.
SIN (ir)	Ergebnistyp REAL. Sinus von ir (ir im Bogenmass, 90 Grad = PI/2).
SIZEOF (v)	Ergebnistyp INTEGER. Speicherbelegung fuer v in Byte. v kann strukturiert sein.
SQR (i,r)	Ergebnistyp wie Argument. Quadrat von ir.
SQRT (ir)	Ergebnistyp wie Argument. Quadratwurzel von ir (ir > 0).
STR (ir,s)	Der Wert ir wird in die Zeichenkette s konvertiert. ir kann mit ir:Breitenformat: Dezimalstellenformat formatiert werden.

SUCC (o)	Ergebnistyp wie Argument. Nachfolger von o. i := succ(i) ist schneller als i := i + 1.
SWAP (iz)	Hoeherwertiges und niederwertiges Byte der 16 - Bit - Variablen werden vertauscht.
TSTBIT (v,i)	Ergebnistyp BOOLEAN. Die Funktion liefert TRUE, wenn das i-te Bit in v gesetzt ist, sonst FALSE (die Zaehlung beginnt mit 0).
TRUNC (r)	Ergebnistyp INTEGER. Der gebrochene Teil von r wird abgeschnitten.
UNPACK (g,i,u)	Uebertragung vom gepackten Feld g in das unpepackte Feld u, beginnend bei u[i].
UPCASE (c)	Ergebnistyp CHAR. Umwandlung eines Klein- in einen Groszbuchstaben. Bei c <> Kleinbuchstabe wirkungslos.
VAL (s,v1,v2)	Die Zeichenkette s wird in eine Zahl konvertiert und auf v1 gespeichert. v1 ist vom Typ INTEGER oder REAL. v2 ist vom Typ INTEGER und gibt einen Fehlercode zurueck (> 0 zeigt fehlerhaftes Element).
WRITE (f,...) WRITELN (f,...)	f kann fehlen. Schreiben des Files f oder eines Standardfiles. Die Prozedur ist in Abschnitt 2.3. ausfuehrlich beschrieben.
WRITEHEX((f,o,n)	Der Wert von o wird mit 1 Byte (n=1) oder 2 Byte (n=2) als hexadezimale Form in f geschrieben.

BDOS-/ BIOS - Funktionen zur Nutzung von Komponenten des Laufzeitsystems SCPX (Erlaeuterungen sind den SCP - Handbuechern zu entnehmen)

1. BDOS - Funktionen
   (n Funktionsnummer, zugleich Inhalt Register C; Aufrufparameter und Rueckgabewerte werden Inhalt der Register A,E,DE und HL; FCB, DMA, Vektor(Bitvektor,1=ja,0=nein) und Block stehen im Text fuer die entsprechenden Adressen; Aufbau des FCB wie in Abschnitt 6.1. und 3 Byte zusaetzlich; RA ist der Rueckkehrcode in A - Fehlercode erstgenannt)

n	Wirkung	Aufruf-parameter p	Rueckgabewert
0	Warmstart		
1	Eingabe (Konsole)		Zeichen (A)
2	Ausgabe (Konsole)	Zeichen (E)	
3	Eingabe (Leser)		Zeichen (A)
4	Ausgabe (Stanzer)	Zeichen (E)	
5	Ausgabe (Drucker)	Zeichen (E)	
6	Direkte Eingabe -->	E = ¤FF	Zeichen (A)
	oder Ausgabe (Konsole) -->	Zeichen (E)	0 (A)
7	I/O-Byte abfragen		I/O-Byte (A)
8	I/O-Byte belegen	I/O-Byte (E)	
9	Ausgabe Zeichenkette bis ¤	Adresse (DE)	
10	Eingabe (Konsolpuffer)	Adresse (DE)	
11	Konsolstatus abfragen		¤01/¤02 (A)
13	DISK-System ruecksetzen		
14	Laufwerk selektieren	Nummer 0..(E)	
15	Vorhandenes File eroeffnen	FCB (DE)	RA: ¤FF/ ¤00
16	File schliesen	FCB (DE)	RA: ¤FF/ ¤00
17	1.Directoryeintrag suchen		
	mit Byte 0..11 des FCB	FCB (DE)	RA: ¤FF/<¤FF
18	Suchen weitere Eintraege (17)	FCB (DE)	RA: ¤FF/<¤FF
19	File loeschen	FCB (DE)	RA: ¤FF/<¤FF
20	Sequentielles Lesen	FCB (DE)	RA: ¤FF/<¤FF
21	Sequentielles Schreiben	FCB (DE)	RA: ¤FF/<¤FF
22	Neues File anlegen	FCB (DE)	RA: ¤FF/<¤FF
23	File umbenennen (neuer Name)		
	Byte 16 - 32 des FCB	FCB (DE)	RA: ¤FF/<¤FF
24	Online-Laufwerke bestimmen		Vektor
25	Aktuelles Laufwerk bestimmen		Nummer 0..(A)
26	DMA-Adresse festlegen	DMA (DE)	
27	DISK-Belegungstabelle lesen		Vektor (HL)
28	Schreibschutz setzen		
29	Schreibgeschuetzte Laufwerke		Vektor (HL)
30	Fileattribute R/O und SYS in		
	Byte 10 und 11 des FCB	FCB (DE)	RA: ¤FF/<¤FF
31	Laufwerksparameter bestimmen		Block (HL)
32	Benutzercode festlegen -->	Nummer (E)	
	abfragen -->	E=¤FF	Nummer (A)
33	Lesen m. direktem Zugriff	FCB (DE)	RA: ¤FF/<¤FF
34	Schreiben m. direktem Zugriff	FCB (DE)	RA: >¤00/¤00
35	Freie Satznummer ermitteln	FCB (DE)	Bytes (FCB)
36	Aktuelle Satznummer ermitteln	FCB (DE)	Bytes (FCB)
37	Laufwerk ruecksetzen	Vektor (DE)	
40	Blockinitialisierung und n=34	FCB (DE)	RA: >¤00/¤00

2. BIOS - Funktionen
(n Funktionsnummer; autonomer Aufruf mit CALL xxxx, wobei xxxx = BIOS-Adresse + n * 3; Uebergabe in den Registern C,BC,DE und Rueckgabe in A oder HL; RA ist Rueckkehrcode in A - Fehlercode erstgenannt)

n	Wirkung	Aufruf-parameter p	Rueckgabewert
0	Warmstart		
1	Konsolstatus abfragen		¤FF/¤00 (A)
2	Eingabe (Konsole)		Zeichen (A)
3	Ausgabe (Konsole)	Zeichen (C)	
4	Ausgabe (Drucker)	Zeichen (C)	
5	Ausgabe (Stanzer)	Zeichen (C)	
6	Eingabe (Leser)		Zeichen (A)
7	Spur 0 einstellen		
8	Laufwerk selektieren	Nummer 0..(C)	Vektor (HL)
9	Spur auswaehlen	Spur (BC)	
10	Sektor auswaehlen	Sektor (BC)	
11	Pufferadresse setzen	Adresse (BC)	
12	Selektierten Sektor lesen		RA: ¤01/¤00
13	Selektierten Sektor schreiben		RA: ¤01/¤00
14	Druckerstatus abfragen		RA: ¤FF/¤00
15	Logische in physische Sektornummer umrechnen	Sektor (BC) Tabelle (DE)	Nummer (HL)

Sachwortregister