

Lehrbriefe für das Hochschulfernstudium

Herausgegeben

von der Zentralstelle für das Hochschulfernstudium
des Ministeriums für Hoch- und Fachschulwesen

Grundlagen der Informatik

3. Lehrbrief

Programmieren in PASCAL

Grundlagen der Informatik

3. Lehrbrief

Programmieren in PASCAL

Verfaßt von

Prof. Dr. sc. nat. Klaus M ä t z e l

Dipl.-Math. Christel M ö c k e l

Technische Universität Karl-Marx-Stadt

Sektion Informatik

Redaktionsschluß: März 1987

1. Ausgabe

1. Auflage

Bestell-Nr. 02 1019 03 0

Verfaßt für die Zentralstelle für das Hochschulfremstudium des Ministeriums für Hoch- und Fachschulwesen Dresden.

Herausgegeben im Auftrag des Ministeriums für Hoch- und Fachschulwesen der Deutschen Demokratischen Republik von der Zentralstelle für das Hochschulfremstudium Dresden.

Inhaltsverzeichnis

	Seite
Vorwort -----	4
4. Programmieren in PASCAL -----	6
4.1. Einfuehrung -----	6
4.2. Aufgaben des Zahlenrechnens -----	9
4.3. Einfache Dialogprogramme -----	18
4.4. Steuerung des Programmablaufs -----	25
4.5. Umgang mit Vektoren und Matrizen -----	35
4.6. Grundsatzliches zu Datentypen -----	41
Anhang: Syntaxdiagramme -----	54

Vorwort

Die Lehrbriefe 3 und 4 der Lehrbriefreihe "Grundlagen der Informatik" sind eine Einfuehrung in die Programmierung im allgemeinen und eine grundlegende Orientierung fuer die Arbeit mit der Programmiersprache PASCAL im besonderen.

Sie wenden sich hauptsaechlich an Personen, die ueber wenig Grundkenntnisse im Programmieren verfuegen, so dass der Text einen systematischen und strukturierten Zugang zur Programmierung vermittelt und fuer das Selbststudium geeignet ist.

Obwohl die Anfang der 70er Jahre entwickelte Programmiersprache PASCAL fuer die Ausbildung konzipiert war, hat sie sich heute international zur beliebtesten Sprache fuer professionelle Programmierung entwickelt und steht auf allen Rechnerklassen zur Verfuegung.

Fuer diese Entwicklung gibt es wohl folgende Ursachen:

- Die Sprache foerdert den strukturierten Entwurf von Programmen und softwaretechnische Aspekte.
- Die konsequente Nutzung des Datentypkonzeptes erlaubt, Datenstrukturen genauso zu konstruieren, wie man es von den Algorithmen fordert.
- Die Sprache ist "klein" und ueberschaubar fuer die Anwendung.

Der in zwouelf Abschnitten vorliegende Text ueberdeckt praktisch alle wesentlichen Eigenschaften der Sprache.

Die Abschnitte selbst charakterisieren die Stoffvermittlung in drei Hauptabschnitten, die auf die Lehrbriefe 3 und 4 aufgeteilt sind:

1. Die Abschnitte 4.1. bis 4.6., Lehrbrief 3, umfassen im wesentlichen den Stoff eines Einfuehrungskurses in die Programmierung; behandelt werden Konstanten, Variablen, Ausdruecke, einfache und strukturierte Anweisungen, elementare Datentypen und Felder.
2. Im Abschnitt 4.7., Lehrbrief 4, werden Unterprogramme als Mittel der Programmstrukturierung eingefuehrt. Der Abschnitt 4.8. vermittelt prinzipielles Wissen ueber den schrittweisen Entwurf wohlstrukturierter Programme; das methodische Vorgehen wird exemplarisch demonstriert.
3. Die Abschnitte 4.9. und 4.10. vermitteln Kenntnisse zu strukturierten Datentypen und dynamischen Datenstrukturen, wodurch weitere Moeglichkeiten der Programmiersprache in ihrer Handhabung vorgestellt werden. Der Abschnitt 4.11. fuehrt die Datenstruktur `set` (Menge) ein, und Abschnitt 4.12. behandelt den Gebrauch der `goto`-Anweisung und der Marken.

In allen Abschnitten werden an zahlreichen Beispielen die Prinzipien der Programmierung diskutiert. Dabei wird bei den Problemstellungen und ihrer Loesung dem Dialog zwischen Mensch und Maschine besondere Aufmerksamkeit gewidmet.

Mit dem zunehmenden Einsatz von Arbeitsplatzcomputern und dem Vordringen der dialogorientierten Arbeitsweise gewinnt der Entwurf von Dialogprogrammen an Bedeutung. Der aufmerksame Leser des Lehrtextes wird auch die Unterschiede zwischen einem Programm, das ohne Kommunikation mit dem Nutzer waehrend des Verarbeitungsprozesses ablaeuft, und einem Dialogprogramm erkennen.

Fuer die Entwicklung praktischer Fertigkeiten im Programmieren mit PASCAL ist der Umgang mit einem Computer unbedingt erforderlich; er sollte moeglichst parallel zur Arbeit in den einzelnen Abschnitten erfolgen.

Aufgrund verschiedener Typen von Computern ist es notwendig, sich ueber die Besonderheiten des zu benutzenden PASCAL-Systems zu informieren. Dabei koennen kleine Abweichungen von dem im Text verwendeten Standard-PASCAL auftreten. 1)

Ferner wird darauf hingewiesen, dass nicht alle Sprachelemente von PASCAL in den vorliegenden Lehrbriefen behandelt sind. Dies betrifft

- die varianten Records,
- die Feldschema-Parameter.

Diese Themen sollten vertiefenden Studien vorbehalten sein.

Karl-Marx-Stadt, Dezember 1986

K. Maetzel
C. Moeckel

1) Sprachbeschreibung PASCAL: Bell/Schiemangk,
Rechentechnik/Datenverarbeitung, Beiheft 3/81.

4. Programmieren in PASCAL

4.1. Einfuehrung

Der Computer, der als technisches Hilfsmittel zur Loesung informationeller Aufgabenstellungen verwendet wird, ist ein automatisch arbeitendes Gerat, das so konstruiert ist, dass es auf Zahlen und anderen Daten einfache Operationen der Informationsverarbeitung ausfuehren kann.

Um das Grundprinzip der Verarbeitung zu erlautern, wird zu-naechst die Ausfuehrung von Berechnungsprozessen behandelt, die auch zum Algorithmusbegriff fuehrt.

Ein elementares technisches Gerat zur Ausfuehrung von Grundope-rationen auf Zahlen ist der Taschenrechner (TR).

Ein einfaches Berechnungsbeispiel verdeutlicht seine Anwendung.

Die Ermittlung der Summe $176,23 + 64,224$ wird in folgenden Schritten ausgefuehrt:

- Schritt 1: Druecke C-Taste
{Akkumulator (AC) wird geloescht}.
- Schritt 2: Druecke Tastenfolge
1, 7, 6, ., 2, 3
{im AC steht 176.23}.
- Schritt 3: Druecke Taste +
- Schritt 4: Druecke Tastenfolge
6, 4, ., 2, 2, 4
{im AC steht 64.224}.
- Schritt 5: Druecke Taste =
{im AC steht 240.454 als Ergebnis}.

Folgerung:

Um eine Berechnungsaufgabe (mit Hilfe des TR) loesen zu koennen, ist eine wohldefinierte Folge elementarer Handlungen (Aktionen) auszufuehren, wobei jede dieser Handlungen wiederum eine elementare Aktion des TR veranlasst.

Die Anzahl der elementaren Aktionen ist endlich, wobei das Ergebnis ein im Akkumulator anzeigbarer endlicher Zahlenwert ist.

Die Gesamtheit derartiger Aktionen bezeichnen wir als Algorithmus.

Vom abstrakten Standpunkt aus kann eine Berechnung B durch die Abbildung $Y = B(X)$ beschrieben werden, wobei

X die Gesamtheit der Input-(Eingabe-)Informationen und
Y die Gesamtheit der Output-(Ausgabe-)Informationen

bezeichnet.

Wenn diese Berechnung B mit Hilfe eines TR oder Computers ausgeführt werden soll, muss eine Menge elementarer Aktionen (Handlungen) existieren, die festlegt, wie die geforderte Berechnung realisiert werden soll.

Wir präzisieren den Begriff des Algorithmus:

Fuer eine Berechnung $Y = B(X)$ existiert ein Algorithmus, wenn eine geordnete Folge von Handlungsvorschriften angebar ist, die automatisch auf einem technischen Gerat abgearbeitet werden koennen, so dass jede dieser Handlungen in einer endlichen Zeit ausfuehrbar ist und mit dem Gerat entweder

1. $Y = B(X)$ bei gegebenem X durch die Abarbeitung der Handlungsvorschriften in vorgeschriebener Reihenfolge erzeugt und angezeigt wird

oder

2. Anzeigbar ist, dass kein Y ermittelt werden kann, das den Berechnungsbedingungen genuegt.

Dabei werden stets nur endlich viele Berechnungsschritte ausgeführt.

Das folgende Beispiel stellt eine allgemeine Berechnungsvorschrift dem Algorithmus gegenueber.

Es ist zu berechnen

Schritt 1: $x = 1,5,$

Schritt 2: $y = 5x^2 + 3x + 2.$

Bezuglich der Ausfuehrung auf einem TR (mit vier Grundoperationen) ist die Schrittfolge kein Algorithmus, da nicht angegeben ist, wie die Berechnung realisiert werden soll.

Dennoch ist diese Berechnungsvorschrift fuer einen Menschen mit mathematischen Grundkenntnissen interpretierbar.

Indem wir nun den Schritt 2 zur Vorschrift

$y = (5 * x + 3) * x + 2$

umformen, koennen wir den folgenden Algorithmus auf dem TR ausfuehren:

Schritt 1: Druecke Taste C

Schritt 2: Druecke Tastenfolge 5, *, 1, ., 5, +, 3, =
{als Zwischenergebnis erscheint 10.5 im AC}

Schritt 3: Druecke Tastenfolge *, 1, ., 5, +, 2, =
{im AC steht das Ergebnis 17.25}

Bemerkung:

Bei der TR-Anwendung haben wir die Algorithmenschritte in der natuerlichen Sprache beschrieben. Ein Mensch als "Interpreter" hat keine Schwierigkeiten im Verstaendnis der einzelnen Handlungen und fuehrt diese auf dem TR aus. Jedoch ist zu beachten,

dass beim Aufstellen der Einzelschritte die Faehigkeit des Menschen und die Arbeitsweise des TR zu beruecksichtigen sind.

Soll im Berechnungsprozess der Mensch ersetzt werden, so muss der Algorithmus so formuliert werden, dass er von der Maschine (eben dem Computer) sowohl interpretiert (also "verstanden") als auch ausgefuehrt werden kann. Deshalb ist es notwendig, ein formales, den Erfordernissen der Maschine angepasstes Beschreibungsmittel fuer Algorithmen zu benutzen; ohne dieses ist eine Kommunikation mit der Maschine nicht moeglich.

Zusammenfassend stellen wir fest:

1. Um einen Berechnungsprozess automatisch ausfuehren zu koennen, ist es notwendig, diesen in der Form eines Algorithmus zu beschreiben.
2. Bei der Verarbeitung auf einem Computer ist der Algorithmus in einer formalen, der Maschine verstaendlichen Form zu schreiben.

Wir bezeichnen einen von einem Computer ausfuehrbaren Algorithmus als Programm und die ausfuehrbaren Aktionen als Befehle oder Anweisungen.

Als Mittel zur formalen Beschreibung eines maschinell ausfuehrbaren Algorithmus benutzen wir eine Programmiersprache (PS).

Folglich ist eine PS sowohl ein Beschreibungsmittel fuer Algorithmen als auch ein Kommunikationsmittel zwischen Mensch und Maschine. Zusaetzlich soll eine gute PS auch den Entwurf von Algorithmen unterstuetzen helfen.

Wir unterscheiden bei den PS maschinenorientierte und hoehere PS.

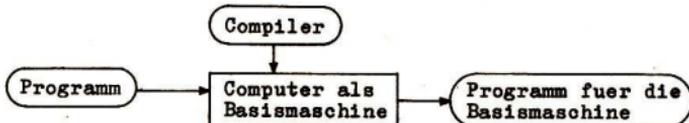
Bei der Nutzung maschinenorientierter PS zur Entwicklung eines Programms wird die genaue Kenntnis eines Computers verlangt, und die Programme beruecksichtigen seine speziellen Eigenheiten. Beim Einsatz hoeherer PS, die sich an den Denkgewohnheiten und Faehigkeiten des Menschen orientieren, wird vom allgemeinen Computerprinzip abstrahiert.

Die Bruecke zwischen einem in einer hoeheren PS beschriebenen Programm und seiner Ausfuehrung auf dem Computer wird durch die Software realisiert.

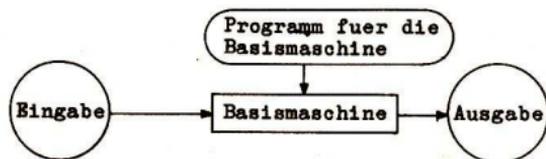
Ein Softwaresystem, das die Uebersetzung eines Programms (geschrieben in einer hoeheren PS) in ein Programm, das der Computer direkt interpretieren kann, realisiert, heisst Compiler.

Folglich haben wir zwei Phasen der Verarbeitung von Programmen zu beachten: die Uebersetzungs- und die Ausfuehrungsphase.

1. Uebersetzungsphase:



2. Ausfuehrungsphase:



In den folgenden Abschnitten wird zur Entwicklung von Programmen die hoehere Programmiersprache PASCAL benutzt. Ihre Anwendung setzt also einen PASCAL-Compiler auf einem Computer voraus. Zudem ist es notwendig, sich ueber die Benutzung und die Besonderheiten eines Compilers auf einem konkreten Computersystem und das Prinzip der Verarbeitung des von ihm erzeugten Programms zu informieren.

4.2. Aufgaben des Zahlenrechnens

Im Abschnitt 4.2. werden einfache Aufgaben des Zahlenrechnens behandelt, die so zu programmieren sind, dass eine Abarbeitung auf einem Computer moeglich ist.

Beispiele:

$$\frac{11,29^2}{4}, \sqrt{3}, \frac{3,14}{2}, (1,257 + 0,385) (1,257 - 0,385).$$

Zur Formulierung und Entwicklung von Programmen, die uns die Aufgabenloesung gestatten, bedienen wir uns der Programmiersprache PASCAL. Dazu werden die sprachlichen Mittel schrittweise beschrieben; es werden immer nur die Konstrukte eingefuehrt, die zur Loesung der jeweiligen Aufgabe noetig sind.

Ein Programm, das in einer solchen Programmiersprache geschrieben ist, besteht aus Folgen von Zeichen, die Zeichenreihen genannt werden. Um mit einer konkreten Programmiersprache zu arbeiten, ist zunaechst die Menge verwendbarer Zeichen zu definieren.

PASCAL verwendet als Zeichen Buchstaben, Ziffern und Sonderzeichen:

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

Ziffern: 0 1 2 3 4 5 6 7 8 9

Sonderzeichen: + - * / = † < > ≤ ≥ () { } []
:= . , ; : ' † ..b

Bemerkung:

1. Das Symbol \square (manchmal auch \square) bezeichnet das Leerzeichen.
2. Obwohl in PASCAL sowohl die Klein- als auch die Grossbuchstaben zugelassen sind, werden in den meisten Systemen diese nicht unterschieden. Deshalb werden im folgenden die Programmtexte mit Kleinbuchstaben geschrieben.

Ausserdem werden in PASCAL bestimmte vordefinierte Zeichenreihen verwendet, die als einziges Symbol mit fester Bedeutung aufzufassen sind. Diese werden als Wortsymbole (Grundsymbole) bezeichnet. In alphabetischer Reihenfolge sind dies die folgenden Symbole:

**and array begin case const div do downto else end file for
function goto if in label mod nil not of or packed procedure
program record repeat set then to type until var while with**

Bemerkung:

Die Grundsymbole werden hier durch Fettdruck hervorgehoben; andere Darstellungen benutzen die Unterstreichung (z. B. begin).

Fuer eine Programmiersprache muss ferner angegeben werden, welche Zeichenfolge eine exakte sprachliche Notation ist und welche Bedeutung dieser zuzuweisen ist. Damit muessen wir Syntax und Semantik der Programmiersprache beschreiben.

Zur Beschreibung der Syntax bedienen wir uns formaler Regeln, die gestatten, zur Sprache gehoerige Zeichenfolgen zu erzeugen. Diese Regeln werden in Form syntaktischer Diagramme beschrieben, die eine Kombination von Knoten (die Texte enthalten) und Verbindungskanten darstellen.

Es gibt zwei Arten von Knoten:

- Rechteckknoten, z. B.

anweisung

- elliptische Knoten, z. B.

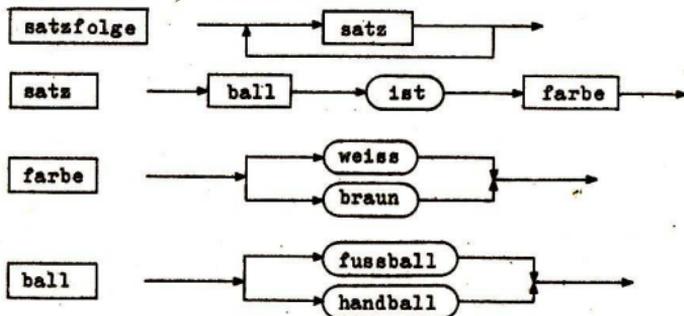
begin

Rechteckknoten repraesentieren Nichtterminalsymbole, und elliptische Knoten stehen fuer Terminalsymbole.

Dabei sind die Terminalsymbole definiert durch Buchstaben, Ziffern, Sonderzeichen und Grundsymbole, die keinen weiteren syntaktischen Aufbau haben.

Nichtterminalsymbole sind Symbole (syntaktische Objekte), die durch syntaktische Regeln beschrieben werden. Fuer solche syntaktischen Objekte existieren beschreibende syntaktische Diagramme. Die Verbindungspfeile geben an, wie die Symbole miteinander verknuepft werden. Alle moeglichen Wege durch ein solches Syntaxdiagramm legen syntaktisch richtige Konstruktionen fest.

Als Beispiel betrachten wir den Aufbau einer einfachen Syntax, die aus Sätzen besteht:



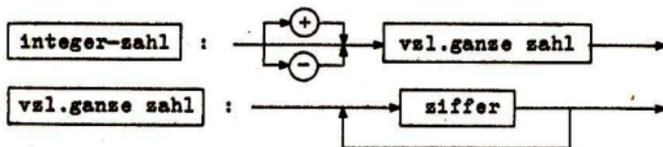
Offenbar ist folgende Satzfolge erzeugbar:

fussball ist weiss
 fussball ist braun
 handball ist weiss
 handball ist braun

Fuer die Programmiersprache PASCAL sind alle Syntaxdiagramme im Anhang angegeben.

Bei der Ausfuehrung der Zahlenrechnung auf einem Computer sind die Zahlenobjekte Gegenstand der Berechnung. Fuer die Beschreibung dieser Objekte gelten in der Programmiersprache folgende Festlegungen:

1. Integer-Zahlen (ganze Zahlen) bestehen aus Ziffernfolgen, die zusätzlich mit Vorzeichen versehen sein koennen. Nach der oben eingefuehrten Beschreibungsmethode ist dann eine Integer-Zahl:



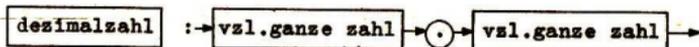
Beispiele:

100 +25 -1773 syntaktisch richtig,
 1.u0 +u25 +-0 syntaktisch falsch.

2. Real-Zahlen entsprechen reellen Zahlen mit folgenden Darstellungen:

a) Decimalzahl.

Als Trennzeichen zwischen ganzem und gebrochenem Anteil wird der Punkt verwendet:



Beispiele:

3.14 73.87 syntaktisch richtig,
 .31 92. syntaktisch falsch.

b) halblogarithmische Form.

Die reelle Zahl entspricht einem Zahlentupel (Mantisse, Exponent) mit der Bedeutung:

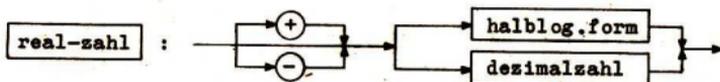
$$\text{mantisse} * 10^{\text{exponent}}$$



Beispiele:

3.21E7 0.5E+8 }
 111.11E-1 3E-11 } syntaktisch richtig,
 E10 15E 23+5 } syntaktisch falsch.

Eine Real-Zahl hat demnach die folgende Form:



Die Grösse der zulaessigen Zahlenbereiche fuer integer- und fuer real-Zahlen haengt im allgemeinen von den Grundeigenschaften des Rechners ab.

Bei der Arbeit mit einem konkreten System informiere man sich ueber die zulaessigen Zahlenbereiche.

Die so definierten Zahlenobjekte werden auch als arithmetische Konstanten vom real- oder integer-Typ bezeichnet.

Beispiele:

3.141592 -6.3 }
 0.12E-2 0.5 } real-Konstanten,
 87E7 -13.2E-5 }
 100 +81 -201 } integer-Konstanten.

Rechnen mit Konstanten:

Konstanten koennen mit Operatoren verknuepft werden, und mittels Klammern "(" und ")" kann die Reihenfolge der Abarbeitung festgelegt werden. Eine so gebildete Zeichenreihe heisst Ausdruck.

Werden die arithmetischen Operatoren +, -, *, /, mod, und div verwendet, so sprechen wir von arithmetischen Ausdruecken.

Die arithmetischen Operatoren mod und div haben folgende Wirkung: Seien a und b integer-Konstanten, so liefert

a div b den ganzen Anteil der Division von a und b,
a mod b den Rest der ganzzahligen Division von a und b.

Beispiele fuer arithmetische Ausdruecke:

3.14/3*(1.257+0.385)*(1.257-0.385)
2.29*2.29*3.14/4
0.123E-5+3.75*0.323E6
(53mod 17+81div 6)*23

Ausdruecke repraesentieren stets eine Wert. Die Wertbestimmung folgt den Gesetzen der Zahlenarithmetik.

Die Reihenfolge der Auswertung eines arithmetischen Ausdrucks, der mehr als einen Operator enthaelt, wird durch zwei Mechanismen gesteuert:

1. durch explizite Klammerung der Operanden,
2. durch Auswertung von links nach rechts entsprechend vordefinierten Operatorprioritaeten (*, /, mod, div haben eine hoehere Prioritaet als + und -). Bei gleichrangigen Operatoren erfolgt die Abarbeitung von links nach rechts.

Der Wert eines arithmetischen Ausdrucks gehoert in Abhaengigkeit von den verwendeten Operatoren entweder dem real- oder dem integer-Typ an. Dabei gelten folgende Regeln:

<u>Operator</u>	<u>Operandentyp</u>	<u>Ergebnistyp</u>
+ Addition	} integer, integer	integer
- Subtraktion		real
* Multiplikation		real
/ Division		real
div ganzz. Division	integer, integer	integer
mod Rest der ganzz. Division	integer, integer	integer

Ein einfaches Programmbeispiel verdeutlicht die Ausfuehrung von Aufgaben des Zahlenrechnens.

Beispiel 1:

```

program bsp1(output);
{berechnungen mit integer-konstanten}
begin
  write(output,16+125);
  write(output,1234*8);
  write(output,(17+4)*21,5+6*7);
  write(output,9mod 5+1,-40div 9*5+31)
end.

```

Anhand des Beispiels 1 wird die Programmstruktur diskutiert.
Ein PASCAL-Programm hat die allgemeine Form

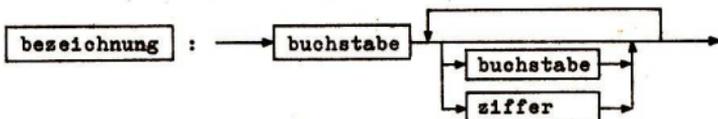
```
program prname ( parameter );  
    vereinbarungsteil  
    anweisungsteil
```

Bedeutung der syntaktischen Objekte im einzelnen:

program ist ein Wortsymbol, das den Anfang eines PASCAL-Programms charakterisiert. Das dasugehoerige Programmende wird durch "." dargestellt.

prname charakterisiert den Namen des Programms. Er ist eine vom Programmierer frei wahlbare Bezeichnung fuer das Objekt Programm.

Eine Bezeichnung im Sinne der Programmiersprache dient zur Benennung verschiedener Objekte. Eine Bezeichnung besteht aus Buchstaben und Ziffern, beginnend mit einem Buchstaben.



Beispiele:

OTTO, B1, X1Y, C syntaktisch richtig,
BSP., 1X syntaktisch falsch.

Bezeichnungen koennen beliebige Laenge haben.
(Hinweis: Die einzelnen Compiler koennen die Laenge von Bezeichnungen einschraenken!)

parameter sind die Programmparameter input und output, die die Ein- bzw. Ausgabedatenfiles beschreiben. Auf die Moeglichkeiten und die Bedeutung dieser Parameter wird im Abschnitt 4.9., Lehrbrief 4, naeher eingegangen. In Beispiel 1 wird der Parameter "output" benoetigt, der angibt, dass das Programm Werte "ausgeben", d. h. nach aussen sichtbar machen soll.

```
program prname ( parameter );
```

wird auch als Programmkopf bezeichnet.

vereinbarungsteil wird in den ersten Beispielen noch nicht benoetigt (siehe Abschnitt 4.3.).

anweisungsteil Im Anweisungsteil werden die einzelnen Anweisungen (Instruktionen) beschrieben, die vom Computer ausgefuehrt werden sollen. Der Anweisungsteil besteht aus einer durch die Wortsymbole begin und end eingeschlossenen Anweisungsfolge. Die Anweisungen werden voneinander durch Semikolon getrennt.

In unserem Beispiel 1 wird noch eine in {,} eingeschlossene Zeichenfolge verwendet. Diese Konstruktion heisst Kommentar und kann an beliebiger Stelle des Programms stehen. Kommentare haben fuer die Funktion eines Programms keinerlei Bedeutung, sie werden bei der Abarbeitung ignoriert. Sie dienen der Erlaeuterung von Programmtellen und erhoehen die Lesbarkeit und Verstaendlichkeit von Programmen.

Die Anweisung, die wir benoetigen, um die Ergebnisse unserer Berechnung auf dem Bildschirm sichtbar zu machen, ist die Anweisung zur Ausgabe (kurz: write-Anweisung). Sie hat in ihrer einfachsten Form folgendes Aussehen:

```
write(output,w1,w2,...,wn)
```

Dabei sind die Parameter w_i in unserem Fall arithmetische Ausdruecke, deren Werte ausgegeben werden sollen.

Neben der write-Anweisung gibt es eine Anweisung

```
writeln(output)
```

die den Uebergang zu einer neuen Ausgabezeile bewirkt.

writeln(output,w1,w2,...,wn) ist dann gleichzusetzen mit

```
write(output,w1,w2,...,wn);  
writeln(output);
```

Damit besteht die Moeglichkeit einer zeilenweisen Ausgabe. write und output sind keine Wortsymbole, sondern vordefinierte Bezeichnungen, die der Programmierer nicht fuer selbstdefinierte Namen verwenden sollte.

Nach diesen Erlaeuterungen bewirkt die Abarbeitung des Programms bsp1 die folgende Ausgabezeile:

```
141 9872 441 47 5 11
```

Bemerkung:

Das Format der Ausgabe von Zahlen ist wiederum vom Compiler abhaengig.

Beispiel 2:

```
program bsp2(output);
{berechnungen mit real-konstanten}
begin
  writeln(output, 'berechnungen mit real-konstanten');
  writeln(output);
  writeln(output, 'berechnung der kreisringflaeche');
  writeln(output, '3.14/4*(1.257+0.385)*(1.257-0.385)=',
    3.14/4*(1.257+0.385)*(1.257-0.385));
  writeln(output, 'radius=', 1.5, ' kreisumfang=',
    2*1.5*3.141592);
  writeln(output, 'halblogarithmische darstellung: ',
    0.123E-5+3.15*0.323E8)
end.
```

Ergebnisse:

berechnungen mit real-konstanten

berechnung der kreisringflaeche
3.14/4*(1.257+0.385)*(1.257-0.385)= 1.1239818400E+00
radius= 1.5000000000E+00 kreisumfang= 9.4247760000E+00
halblogarithmische darstellung: 1.0174500000E+08

Erlaeuterung:

Um neben der Ausgabe reiner Zahlenwerte eine Moeglichkeit zu haben, diese zu dokumentieren, verwenden wir die Zeichenkettenkonstante. Diese ist eine beliebige Folge von Zeichen des PASCAL Zeichensatzes und wird in Apostrophe eingeschlossen. Zeichenkettenkonstanten koennen in der write-Anweisung anstelle der Parameter stehen. Ihr Wert, d. h. die Zeichenfolge innerhalb der Apostrophe, wird ausgedruckt.

PASCAL ist eine formatfreie Sprache, d. h., die Leerzeichen koennen an beliebigen Stellen des Programms stehen. Es gibt eine Einschraenkung dieser Regel der Formatfreiheit: Eine Zeichenkettenkonstante darf nicht das Zeilenende ueberschreiten.

Beispiel:

Nicht erlaubt ist:

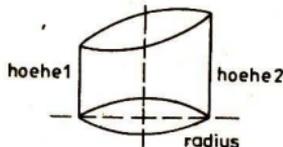
```
write(output, 'kreisflaechen
             berechnung')
```

Statt dessen sollte folgendes geschrieben werden:

```
write(output, 'kreisflaechen',
           'berechnung')
```

Beispiel 3:

Man schreibe ein Programm zur Berechnung der Oberflaeche eines schraeg abgeschnittenen geraden Kreiszyklinders.



```

program oberflaeche(output);
{berechnung der oberflaeche eines schraeg abgeschnittenen
geraden kreiszyllinders}
begin
  writeln(output,'berechnung der oberflaeche eines schraeg',
           ' abgeschnittenen ');
  writeln(output,'geraden kreiszyllinders');
  writeln(output);
  writeln(output,'hoehe1 =',3.81,'hoehe2 =',1.74);
  writeln(output,'radius =',0.85);
  writeln(output,'oberflaeche =',3.14*0.85*(3.81+1.74+
           0.85+sqrt(sqr(0.85)+sqr(3.81-1.74)/4)))
end.

```

Erlaeuterung:

`sqrt(...)` und `sqr(...)` sind die Funktionsaufrufe von sogenannten Standardfunktionen, die die Quadratwurzel bzw. das Quadrat des Arguments berechnen.

Diese Funktionen sind vordefinierte Operationen und koennen in Ausdruecken gleichwertig mit Operanden verwendet werden. Der Aufruf einer Funktion geschieht ueber ihren Namen. In Klammern werden die fuer die Auswertung des Funktionsaufrufs notwendigen Parameter eingeschlossen. Die Parameter werden durch Kommata voneinander getrennt, sie koennen, wie in unserem Beispiel, auch Ausdruecke sein.

Hinweis:

Die Namen der Standardfunktionen sollten vom Programmierer nicht fuer eigene Bezeichnungen vergeben werden. PASCAL benutzt folgende arithmetische Standardfunktionen:

<code>abs(x)</code>	fuer	$ x $,
<code>sqr(x)</code>	fuer	x^2 ,
<code>sin(x)</code>	fuer	$\sin x$ (x im Bogenmass),
<code>cos(x)</code>	fuer	$\cos x$ (x im Bogenmass),
<code>exp(x)</code>	fuer	e^x ,
<code>ln(x)</code>	fuer	$\ln x$ fuer $x > 0$,
<code>sqrt(x)</code>	fuer	\sqrt{x} fuer $x \geq 0$,
<code>arctan(x)</code>	fuer	$\arctan x$.

Damit kann der Begriff des Ausdrucks erweitert werden:

Ein Ausdruck kann Konstanten, Funktionsaufrufe, Operatoren und die Klammern "(" und ")" enthalten.

4.3. Einfache Dialogprogramme

Nach der Berechnung von arithmetischen Ausdruecken mit Zahlenkonstanten soll nun die Aufgabe behandelt werden, den allgemeinen mathematischen Ausdruck

$$25,7 x^2 - 2,5 x + 14,75$$

fuer einen beliebigen vorgegebenen Wert x zu berechnen.

Schreiben wir den gegebenen mathematischen Ausdruck in der uns bekannten programmspezifischen Notation

$$27.5*x*x - 2.5*x + 14.75$$

oder

$$27.5*sqr(x) - 2.5*x + 14.75 ,$$

dann bezeichnen wir x als eine Variable in dem so dargestellten arithmetischen Ausdruck, der wir folgende Bedeutung zuordnen muessen:

Variable sind benannte Objekte, die einen Wert annehmen koennen, der sich im Verlauf des Berechnungsprozesses aendern kann. Der Wertebereich, dem der Wert entstammt, wird als Typ der Variablen bezeichnet.

An dieser Stelle kann die Beschreibung der arithmetischen Ausdruecke wiederum erweitert werden:

Ein arithmetischer Ausdruck kann ausser den Konstanten auch die Variablen als Operanden enthalten.

Die Voraussetzung fuer die Auswertung des Ausdrucks ist in diesem Fall jedoch, dass die enthaltenen Variablen einen Wert besitzen.

Es gibt in PASCAL fuer Variable, die arithmetische Werte annehmen koennen, zwei vordefinierten Wertebereiche, real und integer.

Zum Objekt Variable gehoeren also der Name, der Typ und der Wert. Die Festlegung des Namens und des Typs der Variablen geschieht im Vereinbarungsteil des Programms (siehe Programmaufbau im Abschnitt 4.2.), die Wertzuweisung im Anweisungsteil.

Der grundsaeztliche Aufbau der Variablenvereinbarung fuer Objekte vom Typ integer und/oder real ist der folgende:

```
var   [varname1] ,..., [varnamek] : integer;
      [varnamei] ,..., [varnamej] : real;
```

Bemerkung:

Die detaillierte syntaktische Beschreibung enthaelt der Anhang.

Wenn es im Anweisungsteil des Programms keine Variablen eines der Typen gibt, wird die entsprechende Vereinbarung weggelassen.

Beispiele:

```
- var a,b:real;  
  1,j,k:integer;  
  
- var alpha,beta1:real;  
  
- var k2:integer; k3:integer;  
  
- var i:integer;  
  var x:real; in einem Vereinbarungsteil ist  
  syntaktisch falsch.
```

Alle im Anweisungsteil eines Programms vorkommenden Variablenobjekte muessen im Vereinbarungsteil deklariert werden. Mit dieser Deklaration erfolgt fuer den Nutzer sichtbar die Festlegung von Namen und Typ der Variablen, fuer den Nutzer unsichtbar jedoch die Zuordnung eines geeigneten Speicherplatzes zum Namen der Variablen.

var x,y,z:integer; mit der Bedeutung

x: ==>	<input type="text"/>	Speicherplaetze
y: ==>	<input type="text"/>	fuer
z: ==>	<input type="text"/>	integer-Objekte

Die Zuweisung des Wertes zu einer Variablen, d. h. eine Belegung des fuer diese Variable reservierten Speicherplatzes, geschieht im Anweisungsteil. Es gibt zu diesem Zweck zwei Anweisungen:

- die Anweisung zur Eingabe (kurz: read-Anweisung),
- die Ergibtanweisung.

1. read-Anweisung

Die read-Anweisung hat die Aufgabe, Werte aus der Computerumgebung "einzulesen" und dem Programm zur Verarbeitung zur Verfuegung zu stellen. Sie ist das Gegenstueck zur write-Anweisung. Erfolgt bei der Abarbeitung eines Programms der wechselseitige Austausch von Information zwischen Computer und Nutzer, so bezeichnen wir dieses Arbeitsprinzip als Dialog.

Der Nutzer bedient sich dazu eines Terminals, das aus Tastatur und Bildschirm besteht, wobei der Bildschirm zur visuellen Darstellung sowohl der einzugebenden als auch der ausgegebenen Zeichen dient (siehe write-Anweisung) und ueber die Tastatur das "Eintasten" von Zeichenfolgen vorgenommen wird.

Die Tatsache, dass ein Programm Werte einlesen soll, muss auch im Programmkopf deutlich gemacht werden. Ausser dem Parameter output wird der Parameter input angegeben.

program prname (input,output);

Die Parameter input bzw. output bezeichnen vordefinierte Ein- bzw. Ausgabebereiche (Ein- bzw. Ausgabefiles) fuer das Programm und sind bei der Dialogarbeit dem Terminal zugewiesen.

Die read-Anweisung selbst hat in ihrer einfachsten Form folgendes Aussehen:

```
read(input,w1,w2,...,wn)
```

Dabei sind die Parameter w_i Variablen, denen ein Wert zugewiesen werden soll.

Die Wirkung dieser Anweisung ist die folgende:

Eine Zeichenreihe in Form einer arithmetischen Konstanten, die ueber die Tastatur eingegeben wird, wird mittels der read-Anweisung eingelesen, und der Wert der Konstanten wird der Variablen w_i zugewiesen.

Diese Zuweisung erfolgt in der Reihenfolge, in der die durch getrennten Zahlenkonstanten auf dem Terminal und die Variablen w_i in der read-Anweisung aufgefuehrt sind.

Beispiele:

read-Anweisung	Bildschirm	Wirkung
read(input,a,b,c)	1.3┆2.4┆3.5	a←1.3 b←2.4 c←3.5
read(input,x,i,y)	135.1┆15┆11.2	x←135.1 i←15 y←11.2

Zu beachten ist, dass nicht nur die Reihenfolge und Anzahl der Variablen in der read-Anweisung und der Zahlenkonstanten auf dem Bildschirm uebereinstimmen muessen, sondern auch deren Typ. Der Typ der arithmetischen Konstanten muss mit dem fuer die Variable im Vereinbarungsteil festgelegten Typ uebereinstimmen.

Ausnahme:

Ist die arithmetische Konstante vom integer-Typ und die korrespondierende Variable vom real-Typ, erfolgt eine automatische Typumwandlung.

Neben der read-Anweisung gibt es eine Anweisung

```
readln(input,w1,w2,...,wn);
```

Diese hat die gleiche Wirkung wie die read-Anweisung. Ausserdem wird nach dem Einlesen der Werte fuer die Variablen w_i zur naechsten Zeile des Eingabebereichs (im Dialog der Bildschirm) uebergangen.

Ein Programm zur Berechnung des eingangs gegebenen Polynomausdrucks fuer einen beliebigen Eingabewert x ist Beispiel 4.

Beispiel 4:

```
program polynom(input,output);
{berechnung eines polynoms in x}
  var x:real;
begin
  writeln(output,'berechnung eines polynoms in x');
  writeln(output);
  writeln(output,'geben sie den wert von x ein');
  readln(input,x);
  writeln(output,'der wert des polynoms 27.5*sqr(x)',
    '-2.5*x+14.75');
  writeln(output,' fuer x= ',x,' ist');
  writeln(output,27.5*sqr(x)-2.5*x+14.75)
end.
```

2. Ergibtanweisung

Sie hat die syntaktische Form:

variable	:=	ausdruck
----------	----	----------

In unserem Falle muss das syntaktische Objekt variable eine Variable vom integer- oder real-Typ sein und das syntaktische Objekt ausdruck ein arithmetischer Ausdruck.

Die Abarbeitung der Ergibtanweisung erfolgt so, dass der Ausdruck nach den im Abschnitt 4.2. angegebenen Regeln ausgewertet und der so entstandene Wert der Variablen links vom Ergibtzeichen " := " zugewiesen wird.

Zu beachten ist in diesem Zusammenhang, dass der Typ der Variablen mit dem Typ des Wertes, der sich bei der Auswertung des Ausdrucks ergibt, uebereinstimmen muss (Ausnahme: real \leftarrow integer).

Beispiele fuer Zuweisungen:

```
x:=1;           {zuweisung der konstanten 1 an x}
read(input,y); {einlesen von y}
z:=x+y;        {zuweisung des wertes x+y an z}
x+y:=z;        {syntaktisch falsche anweisung}
```

Bei der Wertzuweisung an eine Variable, sei es durch read- oder Ergibtanweisung, geht der bisherige Wert der Variablen verloren, d. h., die Variable enthaelt stets den zuletzt zugewiesenen Wert. Ferner ist zu beachten, dass eine vollstaendige Wertbestimmung eines Ausdrucks erfordert, dass Variablen vorher ein Wert zugewiesen wird.

Beispiel:

Programmstueck

```
var x:integer;
begin
  read(input,x);
  x:=1;
  x:=x+1;
```

Belegung der Speicherzellen

x: ==>	
x: ==>	3
x: ==>	1
x: ==>	2

Die letzte Ergibtanweisung zeigt deutlich, dass die Wertzuweisung mittels Ergibtanweisung nichts mit der mathematischen Gleichheit von Ausdruecken zu tun hat.

`x:=x+1` bedeutet die Auswertung des Ausdrucks `x+1`; wenn `x` den Wert 1 hat, ergibt sich dabei der Wert 2; dieser wird nun der Variablen `x` zugewiesen.

Jetzt laesst sich auch Beispiel 3 allgemeiner formulieren (Beispiel 5).

Beispiel 5:

```
program oberfl(input,output);
{berechnung der oberflaeche eines schraeg abgeschnittenen
geraden kreiszylinders}
  var h1,h2,r:real;
      w,ofl:real;
begin
  writeln(output,'berechnung der oberflaeche eines schraeg ',
           'abgeschnittenen');
  writeln(output,'geraden kreiszylinders');
  writeln(output);
  writeln(output,'geben sie hoehen und radius als ',
           'reelle zahlen');
  writeln(output,'in der reihenfolge h1 h2 r ein');
  readln(input,h1,h2,r);
  w:=(h2-h1)/2.0;           {anw1}
  w:=sqrt(sqr(r)+sqr(w));  {anw2}
  ofl:=3.1415*r*(h1+h2+r+w);
  writeln(output,'oberflaeche= ',ofl)
end.
```

Erlaeuterungen zum Programm:

Die Variable `w`, die in der Anweisung `anw1` einen Wert zugewiesen bekam, wird in der Anweisung `anw2` rechts und links des Ergibtzeichens `:=` verwendet.

Mit dem in der Anweisung `anw1` berechneten Wert wird der Ausdruck auf der rechten Seite berechnet, dann wird der 'alte' Wert von `w` nicht mehr benoetigt, und `w` erhaelt den Wert des Ausdrucks als neuen Wert zugewiesen.

Die Konstante 3.1415 ist eine bezueglich des betrachteten Problembereiches feste Konstante.

In PASCAL existiert die Moeglichkeit, Konstanten mit einem Namen zu versehen und diesen im Programm in Ausdruecken zu verwenden. Solche Konstantennamen muessen vereinbart werden.

Die Vereinbarung einer Konstanten hat folgende Form:

```
const konstname = konstante ;
```

Die Konstantenvereinbarung steht, wenn sie existiert, im Vereinbarungsteil des Programms vor der Variablenvereinbarung.

Mit der Moeglichkeit der Konstantendefinition ist wiederum eine Erweiterung des Begriffs des arithmetischen Ausdrucks moeglich:

Ausser den Konstanten und Variablen kann ein arithmetischer Ausdruck auch Konstantennamen enthalten.

Diese muessen, wie die Variablen, einen Wert besitzen, sind jedoch als "Read-only-Variable" zu betrachten, d. h., sie duerfen nur gelesen werden und nicht auf der linken Seite einer Ergibtanweisung stehen.

Beispiele:

```
const pi=3.141592;  
o='oeberflaeche=' ;
```

Mit diesen Konstantenvereinbarungen veraendern sich in Beispiel 5 die letzten beiden Anweisungen:

```
ofl:=pi*r*(h1+h2+r+w);  
write(output,o,ofl)
```

Zu einigen Problemen des Zahlenrechnens

Der Typ integer ist, bezogen auf eine konkrete Computeranwendung, stets implementationsabhaengig und definiert eine Teilmenge der Menge der ganzen Zahlen. Aufgrund der (endlichen) Anzahl von Bits (oder Stellen), die fuer die Zahlendarstellung zur Veruegung stehen, wird die Teilmenge z durch $|z| \leq \text{MAX}$ festgelegt, wobei MAX die groesste darstellbare Zahl ist.

Beispiel:

Angenommen, es stehen 6 Stellen zur Darstellung von ganzen Zahlen im Dezimalsystem zur Veruegung, dann sind alle ganzen Zahlen z mit

$$|z| \leq 10^6 - 1$$

darstellbar.

Die Menge der Operationen (+, -, *, mod, div) definiert Abbildungen der Form

integer x integer \longrightarrow integer.

Dabei ist offensichtlich, dass, bezogen auf die integer-Teilmenge, das Resultat der Operation nicht notwendig definiert sein muss.

Beispiel:

Wenn die integer-Menge durch $|z| < 10^3 - 1$ definiert ist, dann gilt

$$\begin{aligned} 600 + (600 + (-500)) &\longrightarrow 600 + 100 \longrightarrow 700, \\ (600 + 600) + (-500) &\longrightarrow \text{Ueberlauf} + (-500) \\ &\quad (\text{Resultat nicht definiert}). \end{aligned}$$

Es gilt das Assoziativgesetz nicht allgemein!

Wir stellen fest, dass entweder die Operationen korrekt ausgeführt werden, wenn alle Operationen Resultate liefern, die im Teilmengebereich integer liegen, oder aber nicht definiert sind (Ueberlauf!).

Wie der Typ integer so ist auch der Typ real durch eine implementationsabhaengige Teilmenge der real-Zahlen definiert. Diese real-Zahlen werden im Computer im allgemeinen durch sogenannte Gleitpunktdarstellungen repraesentiert; mit den integer-Zahlen $|e| < E$ und $|m| < M$ wird die reelle Zahl x durch

$$x = m * B^e$$

dargestellt, wobei B die Basis, e den Exponenten und m die Mantisse der Darstellung bezeichnet. B , M und E sind implementationsabhaengig. Bei fester Basis B kann x durch mehrere Paare (m, e) dargestellt werden; zur Erzielung einer normalisierten Darstellung wird haeufig gefordert:

$$1/B \leq |m| < 1.$$

Aufgrund der endlichen Teilmenge von real-Zahlen ist eine beliebige reelle Zahl durch eine real-Zahl zu approximieren. Ebenso bedeutet die Ausfuehrung der Operationen (+, -, *, /, sin, cos, sqrt, ln, exp, arctan) die Approximation des Resultats.

Beispiel:

Man pruefe das Resultat des arithmetischen Ausdrucks
 $a := \text{sqrt}(\sin(1.7) * \sin(1.7) + \cos(1.7) * \cos(1.7))$.
Obwohl theoretisch $a = 1$ sein muss, wird durch die Approximation ein Wert $1 + \epsilon$ erzeugt.

Die Verletzung der Assoziativ- und Distributivgesetze zeigen folgende Beispiele:

Betrachtet man die Werte

$$\begin{aligned} a &= 0.990 * 10^1, \quad b = 0.100 * 10^1, \quad c = -0.999 * 10^0, \\ x &= 0.1100 * 10^4, \quad y = -0.5000 * 10^1, \quad z = 0.5001 * 10^1, \end{aligned}$$

so lassen sich folgende Ergebnisse bei drei- bzw. vierstelliger Mantissenlaenge ermitteln:

1. $(a + b) + c = 0.109 \cdot 10^2 + (-0.999 \cdot 10^0) = 0.100 \cdot 10^2$,
2. $a + (b + c) = 0.990 \cdot 10^1 + (0.001 \cdot 10^1) = 0.991 \cdot 10^1$,
3. $(x \cdot y) + (x \cdot z) = -0.5000 \cdot 10^4 + 0.55001 \cdot 10^4 = 0.1000 \cdot 10^1$,
4. $x \cdot (y + z) = 0.1100 \cdot 10^4 \cdot 0.1000 \cdot 10^{-2} = 0.1100 \cdot 10^1$.

Bei der Ausführung der Rechnungen ist festzustellen, dass die Addition/Subtraktion dann ungunstig ist, wenn fast gleichgrosse Zahlen subtrahiert werden; in diesem Falle tritt ein Verlust führender Ziffern ein (Genauigkeitsverlust!).

Die Approximationsfehler, die bei der Gleitpunktarithmetik auftreten, koennen in ungunstigen Faellen zu voellig verfaelschten Ergebnisresultaten fuehren.

Zur Vertiefung der Kenntnisse zum Problem des Zahlenrechnens wird das Studium weiterführender Literatur empfohlen.

4.4. Steuerung des Programmablaufs

In diesem Abschnitt sollen die Elemente der Programmiersprache behandelt werden, die eine explizite Steuerung des Programmablaufs ermoeglichen, die sogenannten strukturierten Anweisungen.

Die Anweisungen des Anweisungsteils werden, mit der ersten beginnend, sequentiell verarbeitet. Diese bisher betrachteten Anweisungen waren sogenannte einfache Anweisungen:

- Ergibtanweisung,
- write-Anweisung,
- read-Anweisung.

Zu den strukturierten Anweisungen gehoeren:

- Verbundanweisung,
- bedingte Anweisung,
- repetierte Anweisung.

1. Verbundanweisung

Die Verbundanweisung hat die folgende syntaktische Form:

begin

anweisung1 ;

anweisung2 ;

.

.

;

anweisungn

end

Sie legt fest, dass die in `begin` und `end` eingeschlossenen Anweisungen in der Reihenfolge verarbeitet werden, in der sie aufgeschrieben wurden. Die Wortsymbole `begin` und `end` haben die Aufgabe, die zwischen ihnen stehenden Anweisungen zu "klammern", sie zu einer Anweisung zusammenzufassen.

Beispiel:

```
begin
  read(input,x,y);
  h:=y;
  y:=x;
  x:=h;
  writeln(output,x,y)
end
```

Dieses Beispiel realisiert ein Programmstueck, in dem zwei Variablen `x` und `y` eingelesen, mittels einer Hilfsvariablen `h` umgespeichert und wieder ausgegeben werden.

```
read(input,x,y)    x: ==> [ 2 ]  y: ==> [ 3 ]  h: ==> [ - ]
h:=y;              x: ==> [ 2 ]  y: ==> [ 3 ]  h: ==> [ 3 ]
y:=x;              x: ==> [ 2 ]  y: ==> [ 2 ]  h: ==> [ 3 ]
x:=h;              x: ==> [ 3 ]  y: ==> [ 2 ]  h: ==> [ 3 ]
writeln(output,x,y)  -----> 3 2  auf dem Bildschirm
```

2. Bedingte Anweisung (Alternative)

Die bedingte Anweisung hat die folgende syntaktische Form:

```
if [ bedingung ] then [ anweisung1 ] else [ anweisung2 ]
```

Diese Anweisung steuert in Abhaengigkeit von dem syntaktischen Objekt bedingung die alternative Ausfuehrung der syntaktischen Objekte anweisung1 oder anweisung2.

Ist bedingung "erfuellt", wird anweisung1 abgearbeitet, ist sie "nicht erfuehrt", anweisung2.

Die Bedingung, von der die Ausfuehrung der Anweisungen abhaengt, kann im einfachsten Fall ein Vergleichsausdruck sein.

Er hat die Form:

```
[ arith.ausdruck ] -> [ vergl.operator ] -> [ arith.ausdruck ]
```

Der Aufbau arithmetischer Ausdruecke ist bekannt.

Vergleichsoperatoren sind die folgenden:

```
= Gleichheit,          † Ungleichheit,
< kleiner oder gleich, < kleiner als,
> groesser oder gleich, > groesser als.
```


Wuerde also die Bedingung geaendert, z. B. in $x \leq y$, dann muessten entsprechend die Anweisungen nach **then** und **else** vertauscht werden:

```
if x<=y then writeln(output,y) else writeln(output,x)
```

Das Ergebnis des Programms selbst bleibt das gleiche.

Wuenscht man den kleineren der beiden Werte auszugeben, muss man also entweder die Bedingung aendern oder die Anweisungen vertauschen:

```
if x<=y then writeln(output,x) else writeln(output,y)
oder
if x>y then writeln(output,y) else writeln(output,x)
```

Das Programm laesst sich auch auf andere Weise realisieren:

```
program maximum1(input,output);
{berechnung des groesseren von zwei gegebenen werten}
  var x,y,max:integer;
begin
  writeln(output,'berechnung des groesseren von zwei ',
    'gegebenen werten');
  writeln(output);
  writeln(output,'geben sie zwei ganzzahlige werte ein');
  readln(input,x,y);
  if x>y then max:=x
    else max:=y;
  write(output,'die groessere zahl ist: ',max)
end.
```

Hier wird der groessere der beiden Werte einer Variablen **max** zugewiesen, die dann ausgedruckt wird.

Die Syntax der Programmiersprache laesst auch eine unvollstaendige Form der Alternative zu:

```
if bedingung then anweisung
```

Diese wird immer dann Verwendung finden, wenn nur in einem der beiden Faelle, die durch **bedingung** definiert werden, eine Aktion erfolgen soll.

```
program maximum2(input,output);
{berechnung des groesseren von zwei gegebenen werten}
  var x,y,max:integer;
begin
  writeln(output,'berechnung des groesseren von zwei ',
    'gegebenen werten');
  writeln(output);
  writeln(output,'geben sie zwei ganzzahlige werte ein');
  readln(input,x,y);
  max:=y;
  if max<x then max:=x;
  writeln(output,'die groessere zahl ist: ',max)
end.
```

Erläuterung:

Die Variable max erhielt hier den Wert y zugewiesen und wurde dann mit x verglichen. Nur im Falle, dass x die grössere der beiden Zahlen ist, muss max seinen Wert ändern, im Falle $\text{max} > x$ erfolgt keine Aktion, und es wird zur nächsten Anweisung in der Sequenz (hier der write-Anweisung) uebergangen.

Das obige Beispiel koennte prinzipiell auch in folgender Weise geschrieben werden:

```
if max > x then else max := x;
```

Die zwischen then und else stehende Anweisung ist dann die sogenannte Leeranweisung, die keine Aktion des Rechners bewirkt.

```
leeranweisung : ----->
```

Die Leeranweisung ist eine einfache Anweisung.

Die nach then und else stehenden Anweisungen koennen sowohl einfache als auch strukturierte Anweisungen sein.

Tritt bei einer verschachtelten Alternative der Fall ein, dass die innere Alternative eine unvollstaendige ist, gibt es zwei Moeglichkeiten der Realisierung:

```
a) if B1 then if B2 then A1
      else
      else A2
```

oder

```
b) if B1 then begin if B2 then A1 end
      else A2
```

Hinweise:

1. Bei Implementationen, die keine Leeranweisung zulassen, kann nur Variante b verwendet werden.
2. Bei der Notation des Programmtextes unter Verwendung einfacher und strukturierter Anweisungen benutzt man zur Erhoehung der Uebersichtlichkeit eine entsprechende textliche Strukturierung. Diese ist weder in der Syntax noch in der Semantik der Programmiersprache festgelegt; vielmehr wird sie durch den Programmierstil gepraeagt, zu dem das Schreiben gut lesbarer Programme gehoert. Ein guter Programmierstil zeichnet sich aus durch Namenswahl, Anwendung von Kommentaren, Darstellung der Daten- und Programmstrukturen (siehe auch Abschnitt 4.8., Lehrbrief 4).

Beispiel 7:

Betrachten wir nun folgende Aufgabe:

Es werden die reellen Loesungen der quadratischen Gleichung

$a x^2 + b x + c = 0$ gesucht.

Gegeben: a, b, c reellwertige Koeffizienten.

$$\text{Gesucht: } x_{1,2} = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q} \quad \text{mit } p = \frac{b}{a}, q = \frac{c}{a}.$$

Dabei sind folgende Sonderfaelle zu behandeln:

1. $a \neq 0$, sonst keine quadratische Gleichung.
2. Es existieren keine reellen Loesungen.

Das fuehrt zu folgendem Programm:

```
program quadgl(input,output);
{loesung einer quadratischen gleichung}
  var a,b,c:integer;
      dis:real;
begin
  writeln(output,'loesung einer quadratischen gleichung');
  writeln(output);
  writeln(output,'geben sie die koeffizienten ',
    'a,b,c als integer-werte ein');
  readln(input,a,b,c);
  if a=0
  then writeln(output,'keine quadratische gleichung')
  else begin
        dis:=b*b-4.0*a*c;
        if dis<0 then
          writeln(output,'keine reelle loesung')
        else
          writeln(output,'die loesungen sind: ',
            'x1= ',(-b+sqrt(dis)/a/2,
            ' x2= ',(-b-sqrt(dis)/a/2))
          );
        end
  end.
end.
```

In diesem Programm tritt der Fall ein, dass fuer $a \neq 0$ eine Folge von Anweisungen ausgefuehrt werden sollen. Diese muessen durch **begin** und **end** zu einer strukturierten Anweisung (Verbundanweisung) zusammengefasst werden.

3. Repetierte Anweisung (Schleife)

Es gibt mehrere Schleifenformen, von denen wir hier zunaechst die while-Schleife betrachten.

Sie hat die folgende syntaktische Form:

```
while bedingung do anweisung
```

und besitzt folgende Wirkung:

Solange das syntaktische Objekt bedingung erfuehlt ist (den Wert true hat), wird die Anweisung ausgefuehrt.

Hinweis:

Wird die Anweisung in der **while**-Schleife mindestens einmal ausgeführt, muss sie Einfluss darauf nehmen, dass die Bedingung ihren Wert ändert; andernfalls würde die Schleife endlos sein! Die Bedingung kann wie bei der Alternative zunächst ein Vergleich sein.

Beispiel 8:

Man berechne den Mittelwert von n vorgegebenen reellen Werten.

```
program miwe(input,output);
{berechnung des mittelwertes von n reellen werten}
  var n:integer;
      zaehler:integer;
      wert,summe:real;
begin
  writeln(output,'berechnung des mittelwertes von n ',
           'reellen werten');
  writeln(output);
  write(output,'eingabe der anzahl der werte ');
  readln(input,n);
  summe:=0.0; {setzen des anfangswertes fuer die summe}
  zaehler:=0; {setzen des anfangswertes fuer zaehler}
  while zaehler<n do
    begin
      write(output,'eingabe des ',zaehler+1,
             ' ten reellen wertes: ');
      readln(input,wert);
      summe:=summe+wert;
      zaehler:=zaehler+1
    end;
  write(output,'mittelwert= ',summe/n)
end.
```

Innerhalb der Schleife werden mehrere Anweisungen (durch **begin** und **end** zu einer Verbundanweisung zusammengefasst) ausgeführt:

- Einlesen des naechsten Wertes,
- Aktualisieren der Summe (Gesamtsumme wird schrittweise durch Addition des naechsten Wertes zur vorhergehenden Teilsumme gebildet),
- Weiterzaehlen des Zaehlers, der die Aufgabe hat, die Voraussetzung fuer die Ueberpruefung der Bedingung und damit fuer einen Abbruch bei $zaehler=n$ zu schaffen.

Sowohl das Aktualisieren der Summe als auch das Zaehlen verlangen das Setzen eines Anfangswertes vor Eintritt in die Schleife, da im anderen Fall beim ersten Schleifendurchlauf $summe$ und $zaehler$ keinen Wert haetten.

Beispiel 9:

Tabellieren einer Funktion $y = \sin x - 3b \cos x$ fuer x aus dem Intervall (a,b) mit der Schrittweite Δx .

```
program tabelle(input,output);
{tabellieren sie die funktion  $y=\sin(x)-3b\cos(x)$  im intervall
(a,b) mit der schrittweite deltax}
var x,y,a,b,deltax:real;
begin
  writeln(output,'tabelle der funktion  $y=\sin(x)-3b\cos(x)$  ',
    'fuer die werte x ');
  writeln(output,'aus dem intervall (a,b) mit der ',
    'schrittweite deltax');
  writeln(output);
  writeln(output,'geben sie a, deltax und b ',
    'als reelle zahlen ein');
  readln(input,a,deltax,b);
  writeln(output,' x-wert    y-wert');
  x:=a;
  while x<=b do
    begin
      write(output,x);
      y:=sin(x)-3*b*cos(x);
      writeln(output,y);
      x:=x+deltax
    end
end.
```

In diesem Beispiel wird nicht mit Hilfe einer Schleife eine Groesse ermittelt (wie bei Beispiel 8), sondern in jedem Schleifendurchlauf wird ein Wert erzeugt und ausgedruckt. Trotzdem die beiden write-Anweisungen an unterschiedlichen Stellen des Programms vorkommen, beziehen sie sich auf jeweils eine Ausgabezeile pro Schleifendurchlauf.

Beispiel 10:

Es soll ein Programm geschrieben werden, das fuer gegebene Werte von z die Quadratwurzel berechnet. Das Programm soll abbrechen, wenn $z < 0$ ist.

Zur Berechnung der Quadratwurzel werde ein Iterationsverfahren

$$x_{i+1} = \frac{1}{2} \left(x_i + \frac{z}{x_i} \right)$$

mit $x_0 = z$ und Abbruch, wenn die Differenz zweier benachbarter

Iterationswerte einen vorgegebenen Wert Epsilon unterschreitet, verwendet.

```

program wurzel(input,output);
{berechnung der quadratwurzel einer gegebenen zahl
 mittels eines iterationsverfahrens}
var xneu,xalt,eps,z:real;
begin
writeln(output,'berechnung der quadratwurzel einer ',
'gegebenen zahl');
writeln(output,'mittels eines iterationsverfahrens');
writeln(output);
writeln(output,'geben sie den wert ein, ',
'von dem die wurzel berechnet ');
writeln(output,'werden soll');
readln(input,z);
while z>=0 do
begin
writeln(output,'geben sie die genauigkeit ',
'epsilon als reelle zahl ein');
readln(input,eps);
xalt:=z;
xneu:=(xalt+z/xalt)/2;
while abs(xalt-xneu)>=eps do
begin
xalt:=xneu;
xneu:=(xalt+z/xalt)/2
end;
writeln(output,'die wurzel aus ',z,' ist ',xalt);
writeln(output);
writeln(output,'wollen sie weitere wurzeln ',
'berechnen, ');
writeln(output,'so geben sie diese zahl ein');
writeln(output,'sonst eine zahl <0');
readln(input,z)
end
end.

```

Betrachtet man den Teil des Programms, der die eigentliche Iteration ausmacht, also die Anweisungen 1.anw bis 7.anw, so sieht man, dass die Aktion der Wertzuweisung an x und y sowohl vor als auch innerhalb der Schleife vorgenommen wird. Das bedeutet aber, dass die Anweisungen der Schleife mindestens einmal abgearbeitet werden.

Um Schleifen solcher Strukturen beschreiben zu koennen, kennt PASCAL ausser der while-Schleife zusaetzlich die repeat-Schleife.

Ihre syntaktische Form ist:

```

repeat [anweisung1] ;
      .
      .
      [anweisungn]
until [bedingung]

```

Dabei werden die Anweisungen anweisung¹ bis anweisungn solange abgearbeitet, bis das syntaktische Objekt bedingung den Wert true liefert. Damit wuerde das Programmstueck von 1.anw bis 7.anw folgendes Aussehen haben:

```
xneu:=z;  
repeat xalt:=xneu;  
      xneu:=(xalt+z/xalt)/2  
until abs(xneu-xalt)<eps;
```

Mit dieser Darstellung wird zugleich auch die algorithmische Struktur der Aufgabenstellung besser widergespiegelt.

Es gelten folgende Beziehungen zwischen den beiden Schleifen:

Die repeat-Schleife

```
repeat  
      A  
until B
```

laesst sich mit Hilfe der while-Schleife folgendermassen beschreiben:

```
A;  
while not B do A .
```

Bemerkung:

Mit not B wird gekennzeichnet, dass die Operation not auf die Bedingung angewendet wird und folgende Bedeutung hat:

<u>B</u>	<u>not B</u>
true	false
false	true

Wenn fuer B $\text{abs}(x_{\text{neu}} - x_{\text{alt}}) >= \text{eps}$ steht, so kann not B durch $\text{abs}(x_{\text{neu}} - x_{\text{alt}}) < \text{eps}$ ausgedrueckt werden.

Aufgrund der Stellung der Bedingungspruefung in den Schleifen heisst

- die while-Schleife auch Abweisschleife und
- die repeat-Schleife auch Nichtabweisschleife.

4.5. Umgang mit Vektoren und Matrizen

Dieser Abschnitt behandelt Elemente der Programmierung, die es gestatten, Aufgaben zu bearbeiten, die den Umgang mit Vektoren und Matrizen ermöglichen, die aus der Mathematik bekannt sind.

Vektoren und Matrizen charakterisieren Datenobjekte mit folgenden Eigenschaften:

1. Datenobjekte werden durch einen Namen gekennzeichnet und bestehen aus Komponenten gleichen Typs (z. B. Matrix A, Vektor X).
2. Jede Komponente ist direkt ueber Indizes identifizierbar (z. B. x_1, x_2, x_3 als Komponenten des Vektors X).
3. Die Anzahl der Komponenten ist beliebig, aber fest vorgegeben.

Beispiele:

- Die Menge von reellen Messwerten m_1, \dots, m_n entspricht einem Vektor m , der die oben genannten Eigenschaften hat.
- Die Entfernungen in Kilometern zwischen k Stahlwerken und l Verarbeitungsbetrieben koennen in einer Matrix mit k Zeilen und l Spalten angeordnet werden, deren Elemente als ganze Zahlen festgelegt werden koennen.

Es gibt in PASCAL einen strukturierten Datentyp, der Datenmengen mit diesen Eigenschaften beschreibt: den Feldtyp oder array-Typ.

Sollen Objekte eines solchen Typs bearbeitet werden, sind sie durch eine Vereinbarung zu definieren:

```
var name1 , ... , namek :array [ug1 .. og1 , ug2 .. og2] of ktyp
```

Die verwendeten syntaktischen Objekte haben folgende Bedeutung:

name1, ..., namek werden nach den fuer Namen ueblichen Regeln gebildet und benennen die zu definierenden Datenobjekte.

array ist ein Grundsymbol und bezeichnet die Strukturart Feld.

ug und og stehen fuer Konstanten vom integer-Typ mit der Eigenschaft $ug \leq og$, sie bezeichnen die Indexgrenzen. Handelt es sich um ein zweidimensionales Feld, also die Abbildung einer Matrix auf den Feldtyp, so muessen zwei Indexgrenzenpaare angegeben werden, von denen sich das erste auf den Zeilenindex, das zweite auf den Spaltenindex bezieht.

Soll ein Vektor, d. h. ein eindimensionales Feld betrachtet werden, so faellt das zweite Indexgrenzenpaar weg.

Durch jedes Indexgrenzenpaar wird jeweils eine Indexmenge festgelegt:

$$I_1 = \{ug1, ug1+1, ug1+2, \dots, og1\},$$

$$I_2 = \{ug2, ug2+1, ug2+2, \dots, og2\}.$$

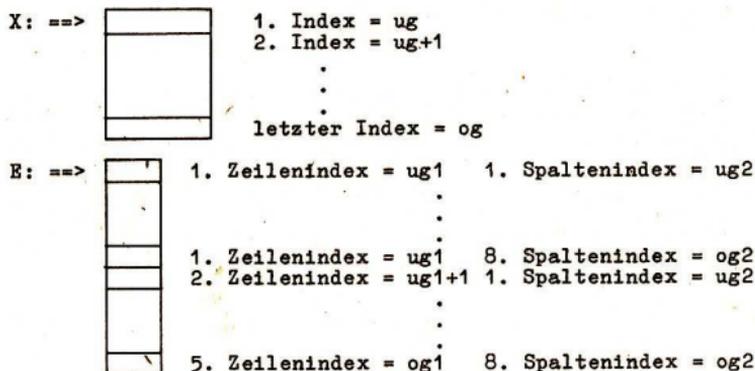
ktyp ist der Typ der Komponenten des Feldes. Es kann dies sowohl ein nichtstrukturierter als auch ein strukturierter Datentyp sein.

Beispiel:

```
var X:array [1..10] of real;
    E:array [1..5,1..8] of integer;
```

Erlaeuterung:

Diese Definitionen vereinbaren die strukturierten Datenobjekte X und E. Damit wird fuer jedes Objekt die Zuordnung von entsprechendem Speicherplatzes vorgenommen:



Da zum Zeitpunkt der Definition die Speicherplatzanforderungen festgelegt werden, ist es notwendig, dass ug und og konstante Werte haben.

Fuer die Festlegung der Konstanten gibt es zwei Moeglichkeiten:

a) Vor der Variablendefinition wird eine Konstantendefinition eingefuegt, die eine Wertzuweisung zu n, k und l vornimmt.

```
const n=10;
      k=5;
      l=8;
var x:array[1..n] of real;
    e:array[1..k,1..l] of integer;
```

b) Anstelle von n, k und l werden feste Konstantenwerte eingesetzt (siehe Beispiel).

```
var x:array[1..10] of real;  
    e:array[1..5,1..8] of integer;
```

Bemerkung:

Die definierten Feldstrukturen legen sogenannte eindimensionale bzw. zweidimensionale Felder fest; durch Hinzufügen weiterer Indexmengen können auch n-dimensionale Felder (n>=3) definiert werden (siehe auch Abschnitt 4.6.).

Der Zugriff zu den einzelnen Komponenten des Feldes erfolgt ueber die Angabe des Namens und eines Indexausdrucks:

```
[name] [ [indexausdruck] ]
```

Eine solche Konstruktion heisst indizierte Variable.

Beispiele:

```
x[1], x[2], ..., x[i+1]  
e[1,2], e[i,j], e[k,l]
```

Der Wert des Indexausdrucks muss innerhalb der durch das Indexgrenzenpaar definierten Indexmenge liegen!

So ist z. B. x[0] als indizierte Variable nach der o. g. Definition von x nicht definiert.

In ihrer Verwendung unterscheiden sich indizierte Variablen nicht von einfachen Variablen. Damit koennen in arithmetischen Ausdruecken ausser einfachen Variablen auch indizierte Variablen als Operanden auftreten.

Das folgende Beispiel zeigt die Anwendung von eindimensionalen Feldern.

Beispiel 11:

Gesucht ist das Skalarprodukt zweier Vektoren

$a = (a_1, \dots, a_n)$ und $b = (b_1, \dots, b_n)$,

deren Komponenten ganzzahlig sind.

Eingabewerte: n, $a_1, \dots, a_n, b_1, \dots, b_n$; $n < 20$.

Berechnungsvorschrift fuer das Skalarprodukt:

$$s = \sum_{i=1}^n a_i * b_i.$$

Ausgabewerte: s.

PASCAL-Programm:

```
program skalar(input,output);
{berechnung des skalarprodukts zweier vektoren a und b}
  var {definition zweier felder a und b, die den
      vektoren der aufgabe entsprechen}
      {beide felder haben maximal 20 komponenten}
      a,b:array[1..20] of integer;
      n,i,s:integer;
begin
  writeln(output,'berechnung des skalarprodukts zweier ',
            'vektoren a und b');
  writeln(output);
  writeln(output,'geben sie die ganze zahl n < 20 ein ');
  writeln(output,'die die anzahl der elemente der ',
            'vektoren angibt');
  readln(input,n);
  writeln(output,'geben sie ',n,' ganzzahlige ',
            'werte fuer den vektor a ein');
  i:=1;
  while i<=n do {einleseschleife fuer feld a}
    begin read(input,a[i]);
           i:=i+1
    end;
  writeln(output);
  writeln(output,'geben sie ',n,' ganzzahlige ',
            'werte fuer den vektor b ein');
  i:=1;
  while i<=n do {einleseschleife fuer feld b}
    begin read(input,b[i]);
           i:=i+1
    end;
  writeln(output);
  s:=0;
  i:=1;
  while i<=n do {berechnung des skalarprodukts}
    begin
      s:=s+a[i]*b[i];
      i:=i+1
    end;
  writeln(output,'das skalarprodukt ist ',s)
end.
```

Falls $n < 20$ ist, werden nur die ersten n Speicherplaetze auf den definierten Feldern belegt.

Bei der Arbeit mit Feldern ist es haeufig so, dass eine oder mehrere Operationen mit allen Komponenten des Feldes ausgefuehrt werden. Dazu sind i. allg. bekannt

- die Anzahl der Komponenten und damit die Anzahl der Wiederholungen der entsprechenden Operation (entspricht der Anzahl der Schleifendurchlaeufer)
- sowie
- die Tatsache, dass der Indexwert nach jeder Operation um den Wert "eins" erhoeht wird.

Um die Beschreibung solcher Schleifen zu vereinfachen, wird eine weitere strukturierte Anweisung eingefuehrt, die Laufanweisung (auch for-Anweisung oder Zaehlschleife).

Sie hat folgende syntaktische Form:

```
for [zaehlvar] := [anwert] to [endwert] do [anweisung]
```

Die syntaktischen Objekte haben folgende Bedeutung:

zaehlvar ist eine Variable vom Typ integer;

anwert, } sind arithmetische Ausdruecke, die zu einem integer-
endwert } Wert ausgewertet werden;

anweisung ist die von der Schleife gesteuerte Anweisung.

Die Abarbeitung der Laufanweisung laesst sich durch folgende while-Schleife beschreiben:

```
zaehlvar:=anwert;  
hilfsvar:=endwert; {var hilfsvar:integer;}  
while zaehlvar<=hilfsvar do  
  begin anweisung; zaehlvar:=zaehlvar+1 end;
```

Beachten Sie:

Die zaehlvar darf innerhalb der Anweisung anweisung nicht geaendert werden!

Mit dieser Struktur laesst sich Beispiel 11 wesentlich einfacher schreiben (Beispiel 12).

Beispiel 12:

```
program skal1(input,output);  
{berechnung des skalarprodukts zweier vektoren}  
  var a,b:array[1..20] of integer;  
      n,i,s:integer;  
begin  
  writeln(output,'berechnung des skalarprodukts ',  
            'zweier vektoren');  
  writeln(output);  
  writeln(output,'geben sie n ein');  
  readln(input,n);  
  writeln(output,'geben sie ',n,  
            ' werte fuer den vektor a ein');  
  for i:=1 to n do read(input,a[i]);  
  writeln(output);  
  writeln(output,'geben sie ',n,  
            ' werte fuer den vektor b ein');  
  for i:=1 to n do read(input,b[i]);  
  writeln(output);  
  s:=0;  
  for i:=1 to n do s:=s+a[i]*b[i];  
  writeln(output,'das skalarprodukt ist ',s)  
end.
```

Beispiel 13 behandelt zweidimensionale Felder.

Beispiel 13:

Gegeben seien zwei Matrizen $A_{m \times n}$ und $B_{p \times q}$, deren Elemente ganze Zahlen sind.

Gesucht ist das Produkt $C_{m \times q} = A_{m \times n} \times B_{p \times q}$.

Die Voraussetzung dafuer, dass die Multiplikation moeglich ist, soll vom Programm selbst geprueft werden, so dass es fuer beliebige Matrizen A und B anwendbar ist.

Die Elemente der Matrix C werden durch Skalarproduktbildung zwischen den Zeilen der Matrix A und den Spalten der Matrix B ermittelt.

$$c_{ij} = \sum_{k=1}^n a_{ik} * b_{kj} \quad \text{fuer } 1 \leq i \leq m, \quad 1 \leq j \leq q,$$

so dass notwendig $n=p$ sein muss.

Eingabewerte: m, n, p, q, A, B (zeilenweise),

Ausgabewerte: C.

```
program mamult(input,output);
{berechnung des produkts zweier matrizen a und b;
 a ist eine m*n-matrix, b ist eine p*q-matrix}
  var a,b,c:array[1..20,1..20] of integer;
    {es koennen beliebige matrizen mit maximal 20
     zeilen und spalten verarbeitet werden}
    i,j,k,m,n,p,q:integer;
begin
  writeln(output,'berechnung des produkts zweier ',
    'matrizen a und b');
  writeln(output,'a ist eine m*n-matrix ',
    'b ist eine p*q-matrix');
  writeln(output);
  writeln(output,'geben sie m,n,p,q ein, ',
    'alle werte muessen <=20 sein');
  readln(input,m,n,p,q);
  while n<>p do
    begin
      writeln(output,'geben sie m,n,p,q mit n=p ein,');
      writeln(output,'sonst koennen sie das programm ',
        'nicht beenden');
      readln(input,m,n,p,q)
    end;
  writeln(output,'geben sie die elemente der ',
    'matrix a zeilenweise ein');
  for i:=1 to m do
    for j:=1 to n do read(input,a[i,j]);
  writeln(output);
  writeln(output,'geben sie die elemente der ',
    'matrix b zeilenweise ein');
  for i:=1 to p do
    for j:=1 to q do read(input,b[i,j]);
  writeln(output);
```

```

for i:=1 to m do {berechnung der matrix c}
  for j:=1 to q do
    begin
      c[i,j]:=0;
      for k:=1 to n do
        c[i,j]:=c[i,j]+a[i,k]*b[k,j]
      end;
    end;
  end;
for i:=1 to m do {ausgabe von c}
  begin
    for j:=1 to q do write(output,c[i,j]);
    writeln(output)
  end
{die ausgabe der matrix ist so programmiert worden,
dass ein druckbild in matrixform entsteht}
end.

```

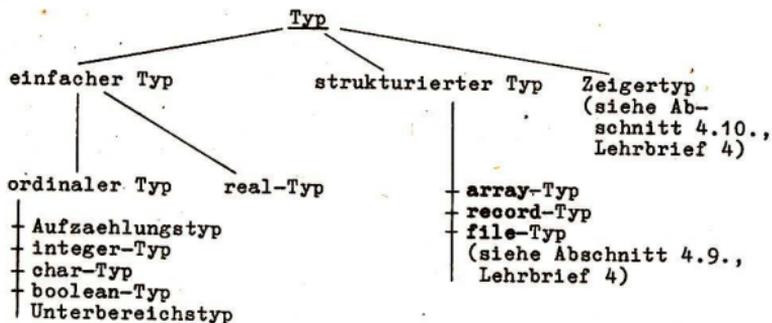
Bemerkung:

Auch in der Laufanweisung kann die auf das do folgende Anweisung wieder eine strukturierte Anweisung sein. Daraus ergibt sich die Moeglichkeit verschachtelter Laufanweisungen zur Bearbeitung von sogenannten mehrdimensionalen Feldern, z. B. von Matrizen.

4.6. Grundsuetzliches zu Datentypen

Nachdem wir den Umgang mit real- und integer-Objekten sowie den Feldern, deren Komponenten aus diesen elementaren Datentypen bestehen, kennengelernt haben, sollen nun die Datentypen von PASCAL allgemein besprochen werden. Prinzipiell unterscheiden wir zwischen sogenannten einfachen und strukturierten Datentypen.

PASCAL klassifiziert die Datentypen wie folgt:



Danach haben wir mit den Zahlenmengen die einfachen Typen integer und real bereits kennengelernt.

PASCAL bietet die Moeglichkeit, ausgehend von den Anforderungen, die ein konkretes Problem stellt, selbst Typen zu definieren.

Die Typvereinbarung hat den folgenden syntaktischen Aufbau:

```
type typname = typ ;
```

Das syntaktische Objekt typname ist eine vom Programmierer frei waelhbare Bezeichnung. Das syntaktische Objekt typ kann ein einfacher oder strukturierter Datentyp sein. Danach sind die Bezeichnungen integer und real vordefinierte Typnamen.

Beispiel:

```
type index=integer;
   element=real;
   matrix=array[1..20,1..20] of integer;
   feld=array[1..10] of real;
```

Diese Typnamen koennen dann in Variablenvereinbarungen in der gleichen Weise verwendet werden, wie wir es bisher mit den vordefinierten Datentypen getan haben.

Beispiel:

```
var i,j,k:index;
    c:array[1..15] of element;
    x,y:element;
    a,b:matrix;
    u:feld;
```

Als weitere Anwendung betrachten wir die Beziehung zwischen ein- und zweidimensionalen Feldern.

Durch die explizite Typdefinition ist folgende Moeglichkeit der Matrixdefinition denkbar:

```
type zeile=array[1..10] of integer;
   mat=array[1..5] of zeile;
var c:mat;
    d:zeile;
```

c ist ein eindimensionales Feld, dessen Elemente wiederum eindimensionale Felder sind.

Mit c[i] kann ein Element von c und damit eine ganze Zeile angesprochen werden, so dass geschrieben werden kann

d:=c[i] (Typgleichheit auf beiden Seiten).

Es ist aber auch moeglich, auf ein einzelnes Element der i-ten Zeile zuzugreifen:

c[i][j] entspricht c[i,j] .

Wir wissen, dass der Typ Wertemengen charakterisiert. Somit ist die einfachste Art der Definition einer Wertemenge die Aufzaehlung ihrer Elemente.

In PASCAL heisst der einfache Datentyp, der durch die Aufzählung seiner Elemente definiert ist, Aufzählungstyp.

Er kann explizit durch die Typvereinbarung erklärt werden:

```
type typname = (w1,w2,...,wn);
```

oder

```
var varname : (w1,w2,...,wn);
```

Die w_i sind dabei Bezeichnungen, die voneinander verschieden sind.

Beispiel:

```
type tag=(mo,di,mi,don,fr,sa,so);
      form=(dreieck,quadrat,rechteck,trapez);
var a,b:tag;
    x,y:form;
    z:(rot,gruen,gelb,blau);
```

Bemerkung:

Der Aufzählungstyp kann auch implizit in einer Variablenvereinbarung erklärt werden.

Aufzählungstypen genuegen folgenden Axiomen:

1. $w_i \neq w_j$ fuer $i \neq j$ (Eindeutigkeit entsprechend der Mengendefinition);
2. $w_i < w_j$ fuer $i < j$ (durch die Reihenfolge der Aufzählung ist eine Ordnung definiert);
3. nur die Werte w_i ($i = 1, \dots, n$) gehoeren zum definierten Typ `typname`.

Beispiele:

1. `type farbe=(blau, gruen, schwarz);`
`var s:farbe;`

Unter diesen Voraussetzungen wuerde die Ergibtanweisung
`s:=gelb;`
im Widerspruch zu Axiom 3 stehen.

2. Eine Typdefinition
`type tag=(mo,di,mi,don,fr,so,so);`
steht im Widerspruch zu Axiom 1.
3. Die Typdefinitionen
`type tag=(mo,di,mi,don,fr,sa,so);`
`weekend=(sa,so);`
stehen im Widerspruch zur eindeutigen Zuordnung eines Wertes zu einer Wertemenge.

Da eine eindeutige Ordnung der Elemente des Aufzählungstyps definiert ist, wird dem zuerst aufgezählten Element w_1 die Ordnungszahl 0 und dem Element w_n die Ordnungszahl $n-1$ zugewiesen; fuer die Elemente sind die Vergleichsoperatoren $<$, $<=$, $=$, \neq , $>$, $>=$ entsprechend der Ordnung der Wertemenge definiert.

Es existieren Standardfunktionen, die die Ordnungszahl, den Vorgaenger bzw. den Nachfolger eines Elements in der Aufzählungsreihenfolge ermitteln.

$\text{ord}(w_i) = i - 1$ ermittelt die Ordnungszahl des i -ten Elements fuer $1 \leq i \leq n$.

$\text{succ}(w_i) = w_i + 1$ ermittelt den unmittelbaren Nachfolger von w_i fuer $1 \leq i < n$.

$\text{pred}(w_i) = w_i - 1$ ermittelt den unmittelbaren Vorgaenger von w_i fuer $1 < i \leq n$.

Beispiele:

```
type wochentag=(mo,di,mi,don,fr,sa,so);
var tag:wochentag;
    k:integer;
```

Unter diesen Voraussetzungen wuerden sich folgende Zuweisungen ergeben:

Ergibtanweisung

```
tag:=mi;
tag:=succ(tag)
k:=ord(pred(tag))
tag:=pred(succ(tag))
```

Wertzuweisung

```
tag ← mi
tag ← don
k ← 2
tag ← don
```

Beispiel:

```
type progsprache=(fortran,algol,pl1,pascal,ada,modula);
var language:progsprache;
begin
  language:=fortran;
  language:=succ(language);{language hat den wert algol}
  language:=ada;
  language:=pred(language);{language hat den wert pascal}
  if language <>pl1
  then write(output,'wert von language ist nicht pl1')
  else write(output,ord(language))
end
```

Eine Wertzuweisung ueber die Eingabeanweisung ist an Variablen vom Aufzählungstyp nicht moeglich.

Zu den ordinalen Datentypen gehoeren die Datentypen char und boolean.

Datentyp char

Der Datentyp char ist vordefiniert und beschreibt die Menge von endlich vielen Zeichen, die ein Computersystem fuer die Kommunikation ueber das Terminal (oder andere E/A-Geraete) benutzt. Diese Zeichenmenge ist im allgemeinen installationsabhaengig, so dass wir fuer weitere Betrachtungen die folgenden minimalen Voraussetzungen treffen:

1. Zur char-Menge gehoeren die Buchstaben a, ..., z.
2. Zur char-Menge gehoeren die Ziffern 0, ..., 9.
3. Zur char-Menge gehoeren das Leerzeichen sowie einige typische, nicht weiter spezifizierte Sonderzeichen (vgl. Abschnitt 4.2.).

Die Elemente dieser char-Menge sind geordnet.

Bemerkung:

In Abhaengigkeit vom verwendeten Zeichensatz wird bei den Installationen meist entweder der ASCII- oder EBCDIC-Zeichensatz verwendet, wobei unterschiedliche Ordnungsbeziehungen verwendet werden!

Ein Zeichen der char-Menge, eingeschlossen in Apostrophe, bezeichnet eine Zeichenkonstante.

Beispiele:

'a', '3', 'L'

Aufgrund der auf der char-Menge definierten Ordnung ist die Anwendung der Vergleichsoperatoren <, >, <=, >=, <> und = moeglich.

Die Wertzuweisung zu char-Variablen erfolgt ueber die Eingabe- bzw. Ergibtanweisung.

Beispiele:

```
var ze1,ze2:char;  
.  
.  
.  
begin  
  ze1:=' '  
  read(input,ze2);  
  .  
  .  
  .
```

Der Datentyp char liefert die Grundlage fuer die Zeichenverarbeitung (Textverarbeitung).

Beispiel 14:

Gegeben sei ein Text, der mit einem im Text selbst nicht vorkommenden Zeichen endet, z. B. mit einem '#'. Es sollen alle im Text vorkommenden Leerzeichen verdoppelt werden.

```
program text(input,output);
{ein text wird zeichenweise ein- und ausgegeben;
 leerzeichen werden zweifach ausgegeben}
  var zeil:char;
begin
  writeln(output,'ein text wird ein- und ausgegeben, ');
  writeln(output,'leerzeichen werden zweifach ausgegeben');
  writeln(output);
  writeln(output,'geben sie einen beliebigen text ',
    'fortlaufend ein');
  writeln(output,'beenden sie ihn mit #');
  read(input,zeil);
  while zeil<>'#' do
    begin
      if zeil=' ' then write(output, ' ');
      write(output,zeil);
      read(input,zeil)
    end
  end.
end.
```

Eingabe: HOW ARE YOU?#

Ausgabe: HOW ARE YOU?

Auch auf dem Datentyp char sind die Funktionen ord, succ und pred definiert.

Während die Funktion succ(x) bzw. pred(x) zu einem Zeichen x das Nachfolgerzeichen bzw. das Vorgängerzeichen auf dem zugelassenen Zeichensatz bestimmt, ist ord(x) eine sogenannte Transferfunktion, die zu einem Zeichen x eine integer-Zahl, die Ordnungszahl im Zeichensatz angibt.

Umgekehrt wird durch die Standardfunktion chr(k) zu einer integer-Zahl k das Zeichen x ermittelt, das die Ordnungszahl k besitzt.

Beispiele:

Bezogen auf den ASCII-Zeichensatz, gilt:

```
chr(97) ----> 'a'
chr(122) ----> 'z'
chr(ord('0')) ----> '0'
```

Falls z ein Ziffernzeichen ist, so liefert ord(z)-ord('0') die dem Ziffernzeichen zugeordnete integer-Zahl.

Beispiel:

ord('9')-ord('0')=9

Zur Illustration dient

Beispiel 15:

Gegeben sei eine integer-Zahl mit Vorzeichen als Zeichenkettenkonstante. Das Programm realisiere die zeichenweise Eingabe dieser Zahl und die Konvertierung dieser Zeichenfolge in eine integer-Groesse.

```
program zahl(input,output);
{aus einer folge von ziffern, die mit einem vorzeichen
versehen sein kann, wird eine integer-zahl erzeugt;
die zahl darf den integer-bereich nicht ueberschreiten}
  var ch:char;
      z,vz:integer;
begin
  writeln(output,'aus einer folge von ziffern, die mit ',
             'einem vorzeichen');
  writeln(output,'versehen sein kann,');
  writeln(output,'wird eine integer-zahl erzeugt');
  writeln(output,'die zahl darf den integer-bereich ',
             'nicht ueberschreiten');
  writeln(output);
  writeln(output,'geben sie das vorzeichen ein');
  readln(input,ch);
  if ch='+' then vz:=1
             else vz:=-1;
  writeln(output,'geben sie die ziffern der zahl ',
             'ohne leerzeichen ein');
  writeln(output,'beenden sie mit leerzeichen');
  writeln(output);
  read(input,ch);
  k:=0;
  while ch<>' ' do
    begin
      k:=k*10+ord(ch)-ord('0');
      read(input,ch)
    end;
  k:=k*vz;
  write(output,k) {k definiert den integer-wert der
                  \zeichenkettenkonstanten}
end.
```

Datentyp boolean

In den Bedingungen fuer die Alternativen oder Schleifen sind bereits die Wahrheitswerte true und false benutzt worden.

true und false sind Konstanten des Datentyps boolean (sogenannte logische Konstanten). Dieser Datentyp umfasst nur diese beiden Werte, so dass folgende Definition gilt:

```
type boolean = (false,true);
```

Damit gilt die Ordnung false < true.

Die Definition von Variablen dieses Typs (die in der gleichen Weise wie die Vereinbarung der integer-, real- oder char-Variablen erfolgt) ermöglicht die Verwendung von logischen Variablen im Programm und deren Wertzuweisung und Ausgabe.

Beispiel:

```
var bc:boolean;
    .
    .
    bc:=false;
    write(output,bc);
```

Achtung:

Wahrheitswerte koennen nicht direkt eingelesen werden!

Werden die logischen Konstanten und Variablen mit den zu diesem Datentyp gehoerenden logischen Operatoren and, or und not verknuepft, so koennen logische Ausdruecke gebildet werden, wobei die Konstanten und Variablen selbst einfache logische Ausdruecke sind.

Bedeutung der Operatoren

<u>Operator</u>	<u>Operand 1</u>	<u>Operand 2</u>	<u>Ergebnis</u>
not		true	false
(einstellig)		false	true
and	true	true	true
	alle anderen Kombinationen		false
or	false	false	false
	alle anderen Kombinationen		true

Sind a,b,c vom Typ boolean, dann entspricht

not a or b and c (not a) or (b and c),

so dass fuer die Operatoren die Prioritaetsreihenfolge not, and, or gilt.

Wird eine andere Abarbeitungsreihenfolge gewuenscht, so muss diese durch explizite Klammerung erzwungen werden.

Beispiele fuer logische Ausdruecke:

- true, false
 - a,b,c mit var a,b,c:boolean;
 - a or b, not c and true
 - (a or b) and c
- Wuerden hier die Klammern weggelassen, so erfolgte die Auswertung nach Vorrangregeln, also a or (b and c).
- (x>y) or (y>0)
 - (0<=x) and (x<=15)

Unterbereichstypen

In manchen Anwendungsfällen nimmt eine Variable Werte nur innerhalb einer gewissen Teilmenge des Wertebereichs an. Für den Fall, dass die Teilmenge ein Intervall einer ordinalen Wertemenge ist, kann diese durch die Definition eines Unterbereichstyps vereinbart werden:

```
type typname = min..max;
```

Dabei sind min und max Konstanten (mit $\text{min} \leq \text{max}$) eines ordinalen Datentyps.

Beispiele:

```
type tag=(mo,di,mi,don,fr,sa,so);
  arbeitstag=mo..fr; {unterbereich vom typ tag}
  index=1..100; {unterbereich vom typ integer}
  buchst1='a'..'l'; {unterbereich vom typ char}
  buchst2='m'..'z'; {unterbereich vom typ char}
```

Unterbereiche vom Typ integer finden z. B. in der Feldtypdefinition als Indexbereich Anwendung.

Unter Verwendung der oben definierten Typen sind beispielsweise folgende Variablendefinitionen möglich:

```
var x:index;
    y:array[index] of real;
    z:buchst1;
```

Bemerkung:

Die Zuweisung $x:=0$ bzw. $z:='r'$ würde dann bei o. g. Vereinbarung zu einem Laufzeitfehler führen, da die zugewiesenen Werte nicht im vereinbarten Wertebereich liegen.

Beispiel 16:

Gegeben ist ein Text, der mit '#' abgeschlossen wird. Gesucht ist die Anzahl der im Text auftretenden Vokale (einzeln).

```
program vokal(input,output);
{bestimmung der anzahl der vokale in einem gegebenen text}
  var tex:char;
      {tex ist die variable, die nacheinander alle zeichen
      des vorgegebenen textes aufnimmt}
      a,e,i,o,u:integer;
      {a,e,i,o,u nehmen jeweils die zahlen auf, die die
      anzahl der vokale angeben}
begin
  writeln(output,'bestimmung der anzahl der vokale in ',
            'einem vorgegebenen text');
  writeln(output);
  writeln(output,'geben sie einen text fortlaufend ein ',
            'beenden sie ihn mit #');
  writeln(output);
```

```

a:=0; e:=0; i:=0; o:=0; u:=0;
read(input,tex);
while tex<>'#' do
  begin
    if tex='a' then a:=a+1;
    if tex='e' then e:=e+1;
    if tex='i' then i:=i+1;
    if tex='o' then o:=o+1;
    if tex='u' then u:=u+1;
    read(input,tex)
  end;
writeln(output,'anzahl der ''a''=',a);
writeln(output,'anzahl der ''e''=',e);
writeln(output,'anzahl der ''i''=',i);
writeln(output,'anzahl der ''o''=',o);
writeln(output,'anzahl der ''u''=',u)
end.

```

Die Folge der unvollstaendigen Alternativen koennte auch in eine verschachtelte vollstaendige Alternative umgewandelt werden:

```

if tex='a'
  then a:=a+1
  else if tex='e'
    then e:=e+1
    else if tex='i'
      then i:=i+1
      else if tex='o'
        then o:=o+1
        else if tex='u'
          then u:=u+1;

```

Dabei wird der Aufwand fuer die Auswertung der Bedingungen verringert (es braucht nicht jede Bedingung ausgewertet zu werden, wenn bereits eine Gleichheit ermittelt worden ist), aber das geht zu Lasten der Uebersichtlichkeit.

Um solche verschachtelten if-Anweisungen uebersichtlicher zu gestalten, kennt PASCAL die Fallaweisung.

Diese Anweisung hat folgende syntaktische Form:

```

case fallausdruck of
  fallkonstante1 : anweisung1 ;
  .
  .
  fallkonstanten : anweisungen
end

```

Dabei sind die syntaktischen Objekte fallausdruck und fallkonstante1 vom gleichen einfachen ordinalen Typ.

Die Wirkung dieser Anweisung kann charakterisiert werden durch folgende verschachtelte Alternative:

```
if fallausdruck = fallkonstante1
then anweisung1
else if fallausdruck = fallkonstante2
then anweisung2
else if ...
else if fallausdruck = fallkonstanten
then anweisungen ;
```

Jedoch ist zu beachten:

1. Ist fuer einen Wert eines fallausdrucks keine entsprechende fallkonstante angegeben, so fuehrt das zum Fehler!
2. Eine fallkonstante darf in der case-Anweisung nur einmal auftreten!

Das bedeutet fuer das Beispiel 16, dass die if-Konstruktion durch die Anweisung

```
case tex of
  'a':a:=a+1;
  'e':e:=e+1;
  'i':i:=i+1;
  'o':o:=o+1;
  'u':u:=u+1
end
ersetzbar ist.
```

Bemerkung:

Anstelle der Anweisung if bedingung then A1 else A2 gilt:

```
case bedingung of
  true:A1;
  false:A2
end
```

Erweitert man nun die Aufgabenstellung von Beispiel 16 dahingehend, dass das Auftreten aller Buchstaben gezaehlt werden soll, so empfehlen sich folgende Ueberlegungen. Betrachtet man die Definition einer Feldvariablen

```
var anzahl:array[index] of integer;
```

deren Elemente die Anzahlen des Auftretens der einzelnen Buchstaben enthalten sollen, so koennte man den Indextyp index als 1..26 wahlen, dann wuerde anzahl[1] die Anzahl der aufgetretenen a und anzahl[26] die Anzahl der aufgetretenen z enthalten. Die Beziehung zwischen Zeichen und Index ist dann ueber die ord-Funktion herzustellen.

Der Indextyp kann auch Unterbereich eines beliebigen anderen ordinalen Typs sein, beispielsweise koennen wir definieren:

```
type index='a'..'z';
zahl=array[index] of integer;
```

Damit wuerde sich folgende Realisierung der erweiterten Aufgabenstellung ergeben (Beispiel 17).

Beispiel 17:

```
program buchstabenzahl(input,output);
{bestimmung der anzahl der einzelnen buchstaben in einem
 vorgegebenen text}
  type index='a'..'z';
  var anzahl:array[index] of integer;
      tex:char;
      lauf:char;
begin
  writeln(output,'bestimmung der anzahl der einzelnen
    'buchstaben');
  writeln(output,'in einem vorgegebenen text');
  writeln(output);
  writeln(output,'geben sie den text fortlaufend ein,
    'beenden sie ihn mit #');
  for lauf:='a' to 'z' do
    anzahl[lauf]:=0;
  {damit werden die feldelemente mit dem anfangswert 0
    initialisiert}
  read(input,tex);
  while tex<>'#' do
    begin
      if (tex>='a') and (tex<='z')
        then anzahl[tex]:=anzahl[tex]+1;
      read(input,tex)
    end;
  writeln(output);
  lauf:='a';
  while lauf<='z' do
    begin
      write(output,'anzahl der ',lauf,'=',anzahl[lauf]);
      lauf:=succ(lauf);
      write(output,' ':15);
      writeln(output,'anzahl der ',lauf,'=',anzahl[lauf]);
      lauf:=succ(lauf);
    end
  end.
end.
```

Beachten Sie:

Bei dieser Verfahrensweise muss jedoch eine Voraussetzung gelten: Der Datentyp char muss so geordnet sein, dass die Buchstaben nacheinander ohne Luecke angeordnet sind!

Um sich bei einer gegebenen Implementation ueber die Anordnung spezieller Zeichen des Typs char zu informieren, kann man sich mit folgendem Programm die Tabelle der Buchstaben und Ziffern und ihrer Ordnungszahlen ausdrucken lassen (Beispiel 18).

Beispiel 18:

```
program ordnung(output);
{druck der ordnungszahlen von zeichen vom typ char}
var i,j:integer;
begin
writeln(output,'tabelle der ordnungszahlen von zeichen ',
'vom typ char');
writeln(output);
writeln(output,' zeichen ordnungszahl zeichen ',
' ordnungszahl');
lauf:='a';
while lauf<='z' do
begin
write(output,lauf:5,ord(lauf):12);
lauf:=succ(lauf);
writeln(output,lauf:10,ord(lauf):12);
lauf:=succ(lauf)
end;
for lauf:='0' to '9' do
writeln(output,lauf:5,ord(lauf):12)
end.
```

Da in diesem Programm das Druckformat eine wesentliche Rolle spielt, wurde die um eine Faehigkeit erweiterte Ausgabeanweisung angewendet.

Sie hat die Form: write(output,x:m)
Dabei ist x der auszudruckende Wert, und m ist die Laenge des Stellenbereiches, in den x rechtsbuendig gedruckt wird.

Fehlt m, so wird eine implementationsabhaengige Stellenbereichs-laenge verwendet.

Ist x vom Typ char, so werden m - 1 Leerzeichen gedruckt, dann x.
Fehlt m, so wird in der Regel m = 1 gesetzt.

So bewirkt zum Beispiel

```
write(output,'*:2) den Druck von  * ,
und
write(output,'_:3) den Druck von  _ _ _ .
```

Ist x vom real-Typ, so kann nach m noch ein zweiter Parameter fuer die Formatierung angegeben werden:

```
write(output,x:m:n)
```

Diese Angabe bewirkt die Ausgabe des real-Wertes nicht in halb-logarithmischer Darstellung, sondern als Dezimalzahl mit n Stellen nach dem Dezimalpunkt.

Beispiel:

```
write(output,x:10:5)
```

bedeutet, dass der Wert von x auf insgesamt 10 Stellen (einschliesslich des Dezimalpunktes) mit 5 Stellen nach dem Dezimalpunkt ausgedruckt wird.

Anhang: PASCAL-Syntaxdiagramme

