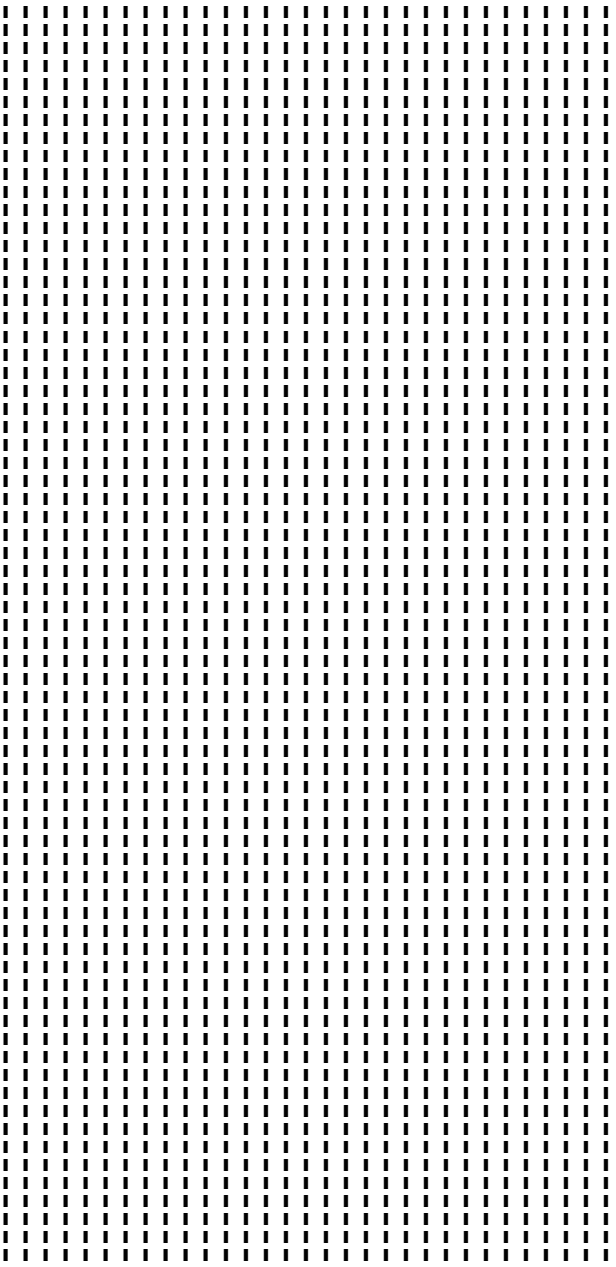
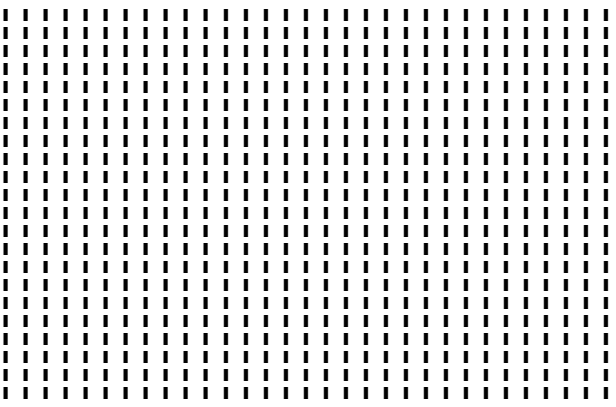


P A S C A L

P. Koop



Vorwort

Ursprünglich war die Programmiersprache PASCAL von N.Wirth nicht für Kleinrechner entwickelt worden. Ihr "Ruhm" aber beruht aber gerade auf ihrer Verbreitung auf diesen Rechnern. An **Schulen** und Hochschulen, Ausbildungsstätten und in der **beruflichen Weiterbildung** ist sie zu der didaktischen Programmiersprache überhaupt erhoben worden. Dieser Lehrtext dient der Orientierung und richtet sich an alle **praktisch orientierten Interessierten,** an Mitmenschen also, die sozusagen auch mit den Händen denken können. Nicht zuletzt richtet sich der Kurs an alle jene Zeitgenossen, die sich mit PASCAL beschäftigen möchten, **ohne in die mathematischen Abgründe der theoretischen Informatik** vordringen zu wollen.

L e r n z i e l e

Dieser Kurs versucht eine intuitive Einführung in die Programmiersprache PASCAL. Der Kurs verzichtet auf mathematische Ableitungen berechenbarer Algorithmen und formaler Sprachen. Am Ende des Kurses sollten Sie kleinere, einfache PASCAL-Programme schreiben und folgende Begriffe erklären können:

- PASCAL Struktur
- Variablen, Konstanten
- Anweisungen, Ausdrücke
- Schleifen
- Prozeduren
- Funktionen
- Bedingte Ausdrücke
- Dateien
- Rekursionen

Eine Einführung in PASCAL entsprechend der Systematik ihrer "Syntax" und "Semantik" mag dem erfahrenen Programmierer

dienen. Für den Anfänger ist dieser Weg unverständlich. "Sprachen" lernt man am besten durch "sprechen". Diese Einführung orientiert sich daher zu Beginn nicht an abstrakten Datentypen und den dann eingeführten auf ihnen arbeitenden Algorithmen sondern einfach an der Alltagserfahrung des Lesers. Von Lernschritt zu Lernschritt kann dann auf das bereits gelernte zurückgegriffen werden; wird die pespektivenreiche Anschaulichkeit mit wachsendem Lernerfolg der notwendigen Abstraktheit weichen.

LITERATUR - Liste

Diese Kursmappe dient einer ersten Einführung in die Programmierung. Bei konsequenter Beachtung der angebotenen Lernhilfe wird keine weitere Literatur benötigt. Für eine weitere Arbeit an der Thematik empfehlen sich die folgenden Monographien:

- allgemeine Einführung -

Hofstadter, D.R. : Gödel, Escher, Bach, Stuttgart 1987

Hofstadter, D.R. : Metamagikum, Stuttgart 1988

Rucker, R. : Der Ozean der Wahrheit
Frankfurt/a.M. 1988-

Rucker,R. : Die Ufer der Unendlichkeit
Frankfurt/a.M. 1989

Minsky,M. : Mentopolis
Stuttgart 1990

Pascal spezifisch -

Wirth, N. : Algorithmen und Datenstrukuren; Stuttgart 1983

Doberkat, E.E.,
Rath, P.,
Rupietta, W. : Programmieren in PASCAL
Wiesbaden 1981

T.Ottmann/

P.Widmayer : Programmierung mit PASCAL
Teubner Studienskripten
Stuttgart 1988

Wenn Sie Zeit haben, sollten Sie einmal einen Besuch in Ihrer Stadtbibliothek beim nächsten Einkauf einplanen, es lohnt sich. Die meisten der hier genannten Titel finden Sie dort bestimmt.

So, und nun geht es in die "Hexenküche".

**Von Briefen; Kochrezepten;
Gebrauchsanleitungen und
anderen nützlichen Dingen**

Nehmen wir einmal an, irgendjemand schriebe einen Brief und dieser Brief beginne folgendermaßen:

> *Aachen, 01.11.1990*
> *Liebe Tante Frieda,*
> *Nun habe ich schon lange nichts mehr von mir hören lassen. Es ist uns halt immer etwas dazwichen gekommen. Aber heute habe ich frei und da will ich endlich einmal schreiben. Mir geht es gut, was ich auch von Dir hoffe. Das Wetter war ja die letzten Tage sehr schlecht, bei Euch in Naumburg sicher auch...*

Wie dieser Brief beginnen viele Briefe, die täglich in Deutschland geschrieben werden. Briefanfänge dieser Art lassen sich auf folgendes Schema bringen:

1. Angabe des ORTES;
2. Angabe des DATUMS;
3. "Lieber/Liebe" als ANREDE;
4. Entschuldigung, so lange nicht geschrieben zu haben als EINLEITUNG;
5. Mitteilung der eigenen Gesundheit;
6. Gesundheitswunsch an den Empfänger als ENDE;

Wir haben also eine feste und häufig verwendete Form vor uns. Man könnte nun fragen, ob diese Form nicht in der Natur der Sache liege, also von selbst verstehe. So aber ist es nicht. Denken wir nur an Briefe der Antike. Die bei uns gebräuchliche Anrede gab es nicht. Da heißt es einfach: *"Gaius an Titus"* und dann kommt der Schreiber zur Sache.
Das Datum steht dann ohne Unterschrift, die steckt sozusagen im Absender, mit dem Gruß *"Vale"* am Ende des Briefes. Oder denken wir nur an Geschäftsbriefe. Hier steht zu Beginn die Anschrift des Briefeempfängers; dann kommen geheimnisvolle Wendungen wie **"Ihr Zeichen ...", "Ihre Nachricht ...", "Unsere Nachricht ...", " Betreff..."**.Ein Werk über die Geschichte des Briefeschreibens würde die **"Formelhaftigkeit"** in jeder Epoche-vom Altertum bis heute- deutlich machen.
Feste Formen aber gibt es nicht nur bei Briefen. **Marcus Porcius Cato(234-149 v.Chr.)** hat in seiner Schrift "Über den Ackerbau" eine ganze Reihe alter Koch- und Backrezepte aufgezeichnet. So lautet ein Rezept des alten Cato:

> Man reibe 2 Pfund Käse gründlich im Mörser. Ist der Käse gut zerrieben, so gebe man 1 Pfund Siligo-Weizenmehl oder, wenn man feiner essen will, nur 1/2 Pfund Similago-Weizenmehl hinzu und verrühre es gut. Forme dann aus dem ganzen einen Kuchen, lege Blätter unter und backe ihn schonend im warmen Ofen unter einer irdenen Schüssel (De agri cultura).

Was fehlt ist eigentlich nur das berühmte: **"Man nehme ... "**. Hier haben wir also eine sprachliche Form, die sich über Jahrhunderte unverändert gehalten hat. Wesentlich für diese Form sind **detaillierte Angaben zu den Zutaten** und eine exakte Beschreibung der genauen **Reihenfolge des Herstellungsvorganges.** Diese Beschreibung des Herstellungsvorganges erfolgt in **Aneinanderreihungen kurzer Sätze, die jeweils Anweisungscharakter haben.**
Kochrezepte sind in der sprachlichen Struktur mit Gebrauchsanweisungen verwandt. Weitere gemeinsame Eigenschaften sind:

1) Sie lösen ein Problem. Im allgemeinen erfassen sie nicht nur ein Problem, sondern eine ganze Problemklasse und liefern zu jedem Problem die richtige Lösung. Die Auswahl eines einzelnen Problems der Problemklasse erfolgt durch die "Zutaten".

2) Der Text des Rezeptes oder der Gebrauchsanweisung hat eine endliche Länge und besteht aus Zeichen eines endlichen Alphabets. Andernfalls könnte man die Anweisungsfolge gar nicht aufschreiben und auch niemand mitteilen.

3) Das beschriebene Verfahren ist tatsächlich durchführbar und besteht aus durchführbaren Anweisungen. Bei Wahlmöglichkeiten liegt genau fest, wie eine Wahl an dieser Stelle durchzuführen ist.

4) Die Anweisungsschritte sind determiniert. Bei gleichen Zutaten und Beachtung der Anweisungen kommt immer das gleiche Ergebnis heraus.

5) Es lassen sich Rezepte denken, die die bisher genannten Bedingungen erfüllen, aber bei der "Eingabe" bestimmter Zutaten in der Durchführung nicht zu einem Ende kommen (nicht terminieren). In der Praxis sind solche Rezepte unbrauchbar und müssen vermieden werden.

6) Praktisch interessante Rezepte müssen möglichst effizient sein. D.h., sie müssen ein Problem in möglichst kurzer Zeit und geringem Aufwand an Hilfsmittel lösen.

+--+
¦ **Hinweis:** Viele interessante weitere, ¦
¦ hier aber nicht zu diskutierende Eigen- ¦
¦ schaften werden in der genannten Lite- ¦
¦ ratur behandelt. Für mathematisch inte- ¦
¦ ressierte Leser lohnt sich die ein oder ¦
¦ andere Lektüre sicher. ¦
+--+

Von Sprachproblemen; Ballungen und Compilern

Mathematiker nennen Kochrezepte **ALGORITHMEN**. Bekannt sein dürfte Ihnen der Euklidische Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen. Noch bekannter ist der folgende Algorithmus:

```
REZEPT Spiegelei (Zutaten, Ergebnis);
ZUTATEN Ei,Ei,Ei,Fett,Speck,Zwiebeln : Lebensmittel;
                        Herd, Pfanne: Kuechengeraete;
ANFANG
   Pfanne auf Arbeitstemperatur bringen;
   Fett in der Pfanne schmelzen;
   Speck in der Pfanne schmoren;
```

```
        Zwiebel in der Pfanne schmoren;
          SOLANGE Eier NICHT VERBRAUCHT
               ANFANG
                    Ei aufschlagen;
                    Inhalt in die Pfanne geben;
               END;
          Pfanneninhalt stocken lassen;
               WENN Eier NICHT VERBRANNT DANN
             Fertiges Gericht servieren
          SONST Pfanneninhalt vernichten
    ENDE.
```

Programme in höheren Programmiersprachen, wie PASCAL, haben eine ähnliche Struktur und gehören also zur **"Sprachform der Kochrezepte".**

Wie aber lassen sich diese "Kochrezepte" auf Rechnern zur Ausführung bringen**?**

Ein Vergleich mag uns da weiterhelfen:

Aus der Perspektive des Betrachters ist ein Fernsehschirm äußerst flexibel. Die Menge der darstellbaren Bilder scheint quasi unendlich zu sein. Auf der alleruntersten Ebene aber besteht jedes Bild aus einer endlichen Menge von Farbpunkten, die entweder eingeschaltet oder ausgeschaltet sind. Auf dieser Ebene ist die Menge der darstellbaren Bilder alles andere als unendlich. Sie lässt sich leicht mit den Hilfsmitteln der Kombinatorik aus der Menge aller möglichen Kombinationen ein- und ausgeschalteter Bildpunkte berechnen. Ähnlich ist es mit einem digitalen Rechnersystem. Auf der Ebene des Benutzers ist der Rechner hochflexibel und kann nahezu alle nur denkbaren "Kochrezepte" erledigen. Auf der alleruntersten Ebene aber besteht der Rechner aus kleinsten Speicherelementen, den **Flip-Flops** (zwei gegeneinandergeschaltete Transistoren), die nur zwei Zustände kennen (deshalb sind Computer auch so dumm), die gewöhnlich mit den Ziffern 0 und 1 bezeichnet werden. Diese Ziffern haben die Qualität von Namen und sind keine Zahlen. Die beiden Zustände könnten ebensogut mit "**Max**" und "**Moritz**" bezeichnet werden. Aber dazu fehlt Technikern eben die Phantasie. Ein Programm auf dieser untersten Ebene besteht dann aus Folgen der Zustände der Flip-Flops.

Auf der Ebene der dem Rechner verständlichen Sprachen ist ein Programm für den Menschen sehr unübersichtlich, es besteht eben nur aus einer Folge von Zuständen der einzelnen Flip-Flops etwa:

```
00010001
11011100
10101010          Maschinencode
00110011
11111000
```

Dabei steht eine Folge von etwa 8,16 oder 32 Flip-Flops (= 1Wort) entweder für einen Befehl oder für ein zu verarbeitendes Datum. Und tatsächlich mussten die ersten Rechner, die mehr an Eisenwahrenhandlungen erinnerten als an Datenverarbeitende Automaten so programmiert werden. Würden Rechner heute im Alltag immer noch so unhandlich sein, ihr Siegeszug wäre nicht möglich gewesen. Bald schon aber viel auf, daß in jedem Programm sich immer "Ballungen" befanden wie Einlesen eines Datums, Verarbeiten einer Zeichenkette und Ausgabe eines Datums. Daher ging man bald dazu über, fertige "Kochrezepte" zu schreiben, die diese Tätigkeit erfüllten. Bei der Erstellung eines neuen Programms brauchte der Benutzer diese "Kochrezepte" dann nur noch aufzurufen; etwa

```
WRITE, WRITELN      (* für Ausgabe *)
READ, READLN        (* für Eingabe *)
```

Eine "Kochrezeptsammlung" bildet zusammen eine sogenannte höhere oder problemorientierte Programmiersprache, wie etwa **PASCAL** und **COBOL** oder **LISP** und **LOGO**. Programmiersprachen sind " g e b a l l t e " Kochrezeptbibliotheken mit der flexiblen Fähigkeit, die einzelnen Rezepte neu zusammen stellen zu können. Das kann etwa die **Berechnung des mittleren Alters der Vereinsmitglieder eines Vereines sein,** deren individuelle Altersangaben auf einer Datei gespeichert sind.

Zur besseren Lesbarkeit der Programme dienen Kommentare mit folgender Syntax:

```
(* dies ist ein Kommentar *)
```

Ein Kommentar wird vom Rechner, bei der Außführung des Programmes nicht beachtet und überlesen.

In PASCAL wäre das Programm dann so zu codieren:

```
PROGRAM Mitgliederalter (INPUT,OUTPUT);
  CONST eins = 1;  (* Konstante *)
  VAR   hilf,alter,zaehler : INTEGER;
        (* Variablen *)
BEGIN   (* des Hauptprogramms *)
    zaehler := 0; (* Wertzuweisung *)
    hilf    := 0;
    WHILE NOT EOF DO
    (* Bis Dateiende erreicht *)
    +-- BEGIN
    |     READ(alter);
    |     zaehler := zaehler + eins;
    |     hilf := hilf + alter;
    +-  END;(* Ende der WHILE-Schleife*)
    +-  IF zaehler = 0
    |   THEN (* wenn Datei leer *)
    |     WRITE('Datei ist leer')
    |   ELSE
    | +-- BEGIN
    | |    (*wenn Datei nicht leer ist*)
    | |     WRITE('Mittleres Alter ',
    | |            hilf/zaehler);
    | |     WRITE('der ',
    | |          zaehler,'Mitglieder');
    | |     WRITELN;
    +---- END; (* ELSE *)
END. (* des Hauptprogramms *)
```

Diese Programme aber müssen in ein für den Rechner verständliches Maschinenprogramm übersetzt werden. Programme, die diese Arbeit erledigen werden Interpreter und Compiler genannt. **Interpreter** übersetzen dabei das Programm zur Laufzeit in ein Maschinenprogramm und benötigen daher bei jedem Programmlauf das Programm in einer höheren Programmiersprache. **Compiler dagegen** übersetzen das Programm nur einmal. Der so produzierte Maschinencode ist schneller, da beim Programmlauf die Übersetzungszeit eingespart wird. **PASCAL ist eine typische COMPILER-Sprache**. In den praktischen Übungen dieses Kurses werden wir also mit einem **COMPILER** arbeiten.

Die Struktur eines PASCAL - Programms

Jedes Pascalprogramm folgt einer immer identischen Struktur.

```
PROGRAM beispiel (input,output);
   (* benötigte Zutaten          *)
BEGIN
  anweisung;
  anweisung;
  anweisung
END.
```

Aufbau eines Pascal Programms

Nach dem Schlüsselwort **PROGRAM** folgt der Name des Programms. Dahinter folgt in Klammern die Angabe der Ein- und Ausgabedateien. Im Regelfall sind das die Standartdateien Input und Output. Der Dateibegriff wird weiter unten behandelt.

Ausgaben aus dem Programm oder Eingaben in das Programm aber sind grundsätzlich nur über Dateien möglich. Wir wollen den Dateibegriff aber vorerst noch offen halten.
Es folgt ein Deklarationsteil für benötigte Variablen, Konstanten usw. Geklammert in die Schlüsselworte **BEGIN** und **END.** folgt der Anweisungsblock. Jede Anweisung wird eine nach der anderen abgearbeitet. Kommentare werden in **{...}** oder **(*...*)** eingeschlossen. Jede Anweisung endet mit einem Semikolon **(;)**, das letzte **END** mit einem **(.)**. Vor **END** oder anderen Blockbegrenzern steht kein Semikolon.

Variablen; Arithmetische Ausdrücke;
Vordefiniert Funktionen

Nun kommt aber endlich unser erstes PASCAL -PROGRAMM.

Das folgende PASCAL-Programm berechnet die Summe zweier ganzer Zahlen. Es sind also zunächst die benötigten **VAR**iablen zu deklarieren. Danach sind die **erste_Zahl** und die **zweite_Zahl** ***einzulesen (*READ*)***. Aus beiden Zahlen ist dann eine **Summe** zu bilden. Diese **Summe** ist dann **auszugeben (*WRITE*)**:

```
PROGRAM beispiel_2(input,output);
VAR erste_zahl,zweite_zahl,
    summe        : INTEGER;

BEGIN
  READ(erste_zahl);
```

```
   READ(zweite_zahl);
   summe := erste_zahl + zweite_zahl;
   WRITE(summe)
END.
```

Variablen

Variablen sind Speicherplätze im Rechner, in denen Zahlen, Buchstaben, Worte oder andere Objekte (die nachfolgend näher erläutert werden) gespeichert werden. Eine Variable hat einen Namen (**Variablenname**), einen sogenannten **Datentyp** und einen Dateninhalt. Dabei kann mit einer Variable genauso gearbeitet werden, wie mit den Daten selber. So kann man z.B. mit einer Zahlenvariablen genauso rechnen wie mit anderen Zahlen. Der Name und der Type einer **Variablen** werden im Deklarationsteil des Programms festgelegt.

Im Deklarationsteil werden **VAR**iablendeklarationen mit dem Schlüsselwort **VAR** eingeleitet. Mehrere Variablen des gleichen **TYPE**s werden durch Kommas getrennt. Nach dem Doppelpunkt folgt der Variablentyp. Der Dateninhalt wird der Variablen aber erst im Programm zugewiesen und kann jederzeit geändert werden.

Datentypen

INTEGER : Ganze Zahlen. Bei jedem Rechner ist darauf zu achten, das der Zahlenbereich beschränkt sein kann. Die Zahlen können positiv oder negativ sein.

REAL : Dezimalzahlen. Eine Dezimalzahl kann auf einem Computer nur mit einer beschränkten Genauigkeit angegeben werden. Die Zahlen können mit einem Komma oder mit einer Zehnerpotenz dargestellt werden. z.B.: 300000000000 = 3E11 (sprich: 3 mal 10 hoch 11).

CHAR : Zeichen. Dies können alle Buchstaben, Satzzeichen, Ziffern oder Sonderzeichen sein.

BOOLEAN : Logische Variable. Dieser Variablentyp kennt nur zwei mögliche werte: Wahr oder Falsch. In Pascal heißen diese Werte:

TRUE - Wahr
FALSE - Falsch

STRING : Zeichenkette. Eine Zeichenkette ist eine Folge von beliebigen Zeichen. Ein STRING kann z.B. ein Wort, ein Name oder ein Satz sein. Dieser Datentyp wird hier nur der vollständigkeit halber genannt. Er ist nicht in allen Pascal-Dialekten vorhanden!

Warum eigentlich Typen?

Nun, hier muß man wissen, daß der Computer alle Werte intern in gleicher Weise (als Binärzahlen) speichert. Wenn aber keine Typunterscheidung gemacht würden, dann würde der Computer ohne Skrupel einen Buchstaben zu einer Zahl addieren. Dies kann dann zu einem wahren Chaos führen.

Ebenso kann es vorkommen, daß der Programmierer nicht mehr weiß, wofür er eine Variable eingeführt hat und eine Zeichenkette z.B. in eine Berechnung mit einsetzt. Hier führt die klare Typabgrenzung von PASCAL zu einem Schutz vor solchen Fehlern. Findet der Computer eine Typkollision, so gibt er eine Fehlermeldung aus. Diese weist den Benutzer darauf hin, daß die Typen nicht zueinander passen und das der Fehler beseitigt werden muß, bevor er das Programm starten kann.

Bitte beachten Sie folgende Regel:

```
+-----------------------------------------------------------+
¦Einer Variablen dürfen nur Werte des gleichen Typs zuge-   ¦
¦wiesen werden. Die Werte müssen von dem Typ sein, der      ¦
¦für die Variable vereinbart wurde.                         ¦
+-----------------------------------------------------------+
```

Konstanten

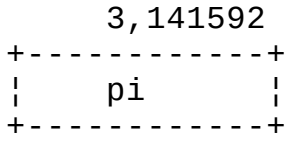
Eine Konstante ist ähnlich der Variablen ein Speicherplatz, der einen Namen und eine Dateninhalt hat. Jedoch läßt sich der Dateninhalt der Konstanten nicht mehr im Programm verändern. Der Inhalt wird im Deklarationsteil festgelegt.

Die Konstantendeklaration wird mit dem Schlüsselwort **CONST** eingeleitet. Nach dem Namen der Konstante folgt ein **(=)** und der Dateninhalt. Konstanten sind z.B. sehr nützlich, wenn

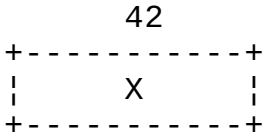
man lange Zahlen abkürzen möchte. So kann man die Zahl 3.141592654 als Konstante PI definieren und im Programm braucht man dann nicht immer den genaueren Wert sondern nur PI eingeben.

Konstanten 137 und -24

Benannte Konstante



Variable



Die Zuweisung unter PASCAL

In PASCAL werden Variablen mit **(:=)** Werte zugewiesen. Dies können auch Ergebnisse aus Arithmetischen Ausdrücken sein.

z.B.: zahl := 10; oder Ergebnis := zahl1 + zahl2;

Arithmetische Ausdrücke

In PASCAL sind die folgenden Operationen definiert:

+	*Addition*
-	*Subtraktion*
*	*Multiplikation*
/	*Division (REAL)*
DIV	*Division (INTEGER) ohne Rest*
MOD	*Rest bei INTEGER Division*

Außerdem sind folgende Funktionen vordefiniert:

ABS(x)	*Betrag*
ARCTAN(x)	*arcustangens*
COS(x)	*cosinus*
EXP(x)	e^x
LN(x)	*ln x*
ROUND(x)	*runden*
SIN(x)	*sinus*
SQR(x)	x^2
SQRT(x)	*Quadratwurzel*
TRUNC(x)	*ganzzahliger Teil*

Erklärung Beispiel Programm 2

Im obigen Beispiel werden die Variablen **erste_zahl, zweite_zahl** und **summe** als **INTEGER** deklariert, d.h. sowohl die eingegebenen Variablen als auch die Ausgabe sind vom Typ **INTEGER** (ganze Zahlen).

READ weist einer Variablen ein Wert zu, der z.B. von der Tastatur aus eingeben wurde.

In der Anweisung *(summe := erste_zahl + zweiten_zahl)* wird das Ergebnis aus der Addition der *ersten_zahl* und der *zweiten_zahl* der Variablen *summe* zugewiesen.

WRITE gibt den Wert einer Variablen (hier von summe), z.B. auf den Bildschirm, aus.

In arithmetischen Ausdrücken gelten die bekannten Prioritäten und Klammerregeln;

Der Bezeichner

Bezeichner sind Namen für Objekte (z.B. Variablen, Konstanten), die in PASCAL-Programmen deklariert oder definiert werden. Dabei ist ein Bezeichner eine Folge von Buchstaben und Ziffern, die mit einem Buchstaben beginnt. Bei der Wahl des Bezeichners sind Sie fast vollkommen Ihrer Phantasie überlassen. Lediglich folgende PASCAL-Schlüsselwörter sowie Standardbezeichner dürfen nicht verwendet werden.

Schlüsselwörter von PASCAL:

AND	ARRAY	BEGIN	CASE	CONST	DIV
DO	DOWNTO	ELSE	END	FILE	FOR
FORWARD	FUNCTION	GOTO	IF	IN	LABEL
MOD	NIL	NOT	OF	OR	PACKED
PROCEDURE	PROGRAM	RECORD	REPEAT	SET	THEN
TO	TYPE	UNTIL	VAR	WHILE	WITH

Stadardbezeichner von PASCAL

Einige Bezeichner haben eine vordefinierte, **feste** Bedeutung. Dazu gehören insbesondere die Namen der Standardprozeduren und- funktionen wie sie nachfolgend aufgeführt sind:

ABS	ARCTAN	BOOLEAN	CHAR	CHR	COS	DISPOSE
EOF	EOLN	EXP	FALSE	GET	INPUT	INTEGER
LN	MAXINT	NEW	ODD	ORD	OUTPUT	
PACK	PAGE	PRED	PUT	READ	READLN	
REAL	RESET					
REWRITE	ROUND	SIN	SQR	SQRT	SUCC	
TEXT	TRUE	TRUNC	UNPACK	WRITE	WRITELN	

Dazu können, je nach PASCAL Dialekt, noch andere Schlüsselwörter und Standardbezeichner kommen. Sehen Sie bitte in Ihrem Handbuch zum PASCAL Compiler nach.

Bitte beachten Sie diese Regel:

```
+-----------------------------------------------------------+
¦Kein Bezeichner darf genau so wie ein Schlüsselwort oder    ¦
¦ein Standardbezeichner geschrieben sein, um Bezeichner      ¦
¦grundsätzlich von PASCAL-Schlüsselwörtern und Stadard-      ¦
¦bezeichnern zu unterscheiden.                               ¦
+-----------------------------------------------------------+
```

Programmschleifen

Konstante Werte spielen in "Alltagsalgorithmen eine häufig bestimmende Funktion; der Wagen wird solange von neuem gestartet bis er endlich anspringt; der Nagel wird solange geschlagen bis er im Holz steckt, solange fliegt das Flugzeug eine Schleife, bis es landen kann. Sind Anweisungen oder Anweisungsblöcke mehrfach auszuführen werden sie in Schleifenkonstruktionen angefaßt,
deren Abbruchbedingung häufig von Konstanten abhängig ist:

Die FOR-SCHLEIFE

```
FOR variable := n1 TO/DOWNTO n2
  DO
    BEGIN
      anweisung;
       anweisung
      END
```

Die **FOR** - Schleife wird so oft durchlaufen, d.h. die Anweisungen innerhalb von BEGIN...END werden so oft durchlaufen, bis die Zählvariable einen maximalen/minimalen Wert (n2) erreicht hat.

Die REPEAT-UNTIL SCHLEIFE

```
REPEAT
   anweisung;
  anweisung
UNTIL bedingung;
```

Die **REPEAT** - Schleife wird so oft durchlaufen, mindestens aber einmal, bis die Bedingung erfüllt ist.

Die WHILE-SCHLEIFE

```
WHILE bedingung;
 DO
   BEGIN
      anweisung;
```

```
    anweisung
END;
```

Die **WHILE** - Schleife wird nur durchlaufen, wenn und solange die Bedingung erfüllt ist.

Wichtig!

Bei der Anwendung der **WHILE**- und **REPEAT**- Schleife innerhalb von Programmen ist immer darauf zu achten, daß innerhalb der Schleife eine Anweisung definiert ist, die bei widerholter Ausführung mit Sicherheit einmal die Abbruchbedingung erfüllt. D.h. es muß sichergestellt werden, daß die Bedingung für das Beenden der Schleife erreicht wird. Ansonsten kann es zu sogenannten Endlosschleifen kommen, die nicht abbrechen (nicht terminieren).

Ausgabeformate; Zeichenvariable

In den letzten Kapiteln haben wir uns wenig um die Gestaltung der Ausgabe gekümmert. Wichtig war bisher nur eine allgemeine Ausgabeanweisung. Werden aber etwa Listen von Zahlen über einen Drucker ausgegeben, ist die eindeutige rechtsbündige Ausgabe erforderlich. In diesem Fall ist die WRITE-Anweisung die folgende Form:

```
WRITE(zahl:zeichen:nachkomma);
```

<zeichen> *steht für die anzahl der auszugebenden Zeichen einschließlich des Dezimalpunktes und der nachkommastelle.*

<nachkomma> *eine Angabe der Nachkommastellen ist nur bei REAL-Variablen sinnvoll.*

```
WRITE(10.12345:10:2)
```

führt zu folgender Ausgabe: _____10.12

(_) steht für Leerzeichen

Ohne diese Angabe werden Dezimalzahlen in Gleitkommadarstellung ausgegeben. Um Gleitkommazahlen brauchen Sie sich in diesem Kurs nicht zu kümmern. Hier nur ein kleines Beispiel:

1.2345E7 = 1.2345*10^7 = 12345000

Logische Variablen führen bei der Ausgabe zur Ausgabe der Zeichenkette 'TRUE' oder 'FALSE'.Die Ausgabe jeder anderen Zeichenkette erfolgt in Hochkommata:

```
WRITE('dies ist ein text':20);

Ausgabe:  ___dies ist ein text
```

Jede Ausgabe schließt an das letzte Zeichen der vorherigen Ausgabe an. Ein Zeilenumbruch wird mit WRITELN (WRITE LiNe) eingefügt.

Einzelne Zeichen aus dem Zeichenvorrat des Rechners werden Variablen vom Typ CHAR zugewiesen. Die Zuweisung erfolgt in Hochkommata: **buchstabe := 'K';**

Der ASCII-Code

Alle Zeichen sind aufsteigend sortiert und einem Zahlencode (beim ASCII 0-255) zugeordnet. Da die Zeichen bei einem solchen Code "abzählbar" sind, sind die Vergleichsoperatoren <,>=,=,<= und > sinnvoll definiert. Auf den Zeichenvorrat sind weiter die Funktionen **ORD** (liefert den Zahlencode eines Zeichens) und **CHR** (liefert das zum Code gehörende Zeichen definiert:

Bsp. ORD('a')

gibt die Zahl 65 zurück

CHR(65)

gibt das Zeichen 'a' zurück

ORD('9') - ORD('0')

gibt die Zahl 9 zurück

Verzweigungen; Boolsche Variable;

Unsere bisherigen Programme bestanden aus sequenziellen Anweisungen; wie die Perlen einer Schnur. Eine Anweisung nach der anderen. Die Ausführung von Anweisungen kann aber von Bedingungen abhängen. In diesem Fall wird dann je nach Bedingung eine Perle übersprungen. Die Struktur der bedingten Anweisung ist:

```
WENN bedingung DANN anweisung1
               SONST anweisung2;
oder:

WENN bedingung DANN ANFANG
                      anweisung;
                      anweisung;
                    ENDE
               SONST
                    ANFANG
                      anweisung;
                      anweisung
                    ENDE;
```

Die Bedingung ist ein logischer Ausdruck der also entweder wahr oder falsch ist (TRUE,FALSE) vom TYPE BOOLEAN.

```
PROGRAM beispiel_7(INPUT,OUTPUT);
VAR a,b : INTEGER;
    ungleich : BOOLEAN;
BEGIN
  READ(a);
  READ(b);
  ungleich := a<>b;
  IF ungleich THEN
              WRITE('ungleich')
ELSE
BEGIN
```

```
  WRITE('gleich');
  WRITE('ok')
END
END.
```

Es gelten folgende logische Operationen:

a = b	Test auf Gleichheit
a <> b	Test auf Ungleichheit
a >= b	" "
a <= b	" "
a < b	" "
a > b	" "
NOT(a)	Ergebnis TRUE wenn a = FALSE sonst umgekehrt

a und b beide = TRUE	
a AND b	NOT(a) AND NOT(b) = FALSE a AND NOT(b) = FALSE NOT(a) AND b = FALSE a AND b = TRUE
a OR b	NOT(a) OR NOT(b) = FALSE a OR NOT(b) = TRUE NOT(a) OR b = TRUE a OR b = TRUE

Eingabe von Daten

Eingaben erfolgen über die Prozedur **READ**.

Wie für die Prozedur WRITE gelten für die Prozedur READ einige Sonderfälle, die wir uns kurz ansehen sollten. Wird eine **ganze Zahl** oder eine **Dezimalzahl von INPUT gelesen**, ist der Lesevorgang beendet, sobald ein nicht zu einer Zahl gehörendes Zeichen gelesen wird. Die Prozedur **READLN** sorgt für ein überlesen aller Zeichen bis unmittelbar hinter den nächsten Zeilenumbruch.

Die CASE - Anweisung

Keine Besonderheit aber eine besonders feine Sache ist die Steuerung eines Programms über eine Eingabedatei. Häufig kann es zum Beispiel sinnvoll sein, eine Zeile mit verschiedenen Steuerzeichen zu versehen, die eine differenzierte Bearbeitung der Zeilen zulassen. Wird ein Steuerzeichen gelesen, so verzweigt das Programm dann in ein Mehrfachauswahl. Die Mehrfachauswahl hat folgende Struktur:

```
CASE steuerzeichen OF
'A' : anweisung_a;
'B' : anweisung_b;
 .
 .
'n' : anweisung_n
END;
```

Kann die Variable <steuerzeichen> keiner in der CASE - Anweisung möglichen Konstanten zugeordnet werden, kommt es zum Programm-abbruch mit Fehlermeldung.

Beispiel für die CASE - Anweisung:

```
PROGRAM Rechner (INPUT,OUTPUT)
VAR zahl1,zahl2 : INTEGER;
    operator    : CHAR;
BEGIN
  READLN(operator,zahl1,zahl2);
  CASE operator OF
  '+' : WRITELN(zahl1 + zahl2);
  '-' : WRITELN(zahl1 - zahl2);
  '*' : WRITELN(zahl1 * zahl2);
  '/' : WRITELN(zahl1 / zahl2);
  ELSE WRITELN('Fehler');
  END (* CASE *)
END.
```

Felder und Typen_Vereinbarungen

Neben den bisher benutzten Daten**TYPE**n (INTEGER; REAL; CHAR; BOOLEAN;) ist es in Pascal möglich, eigene Datentypen zu definieren. Dazu einige Beispiele:

Am einfachsten ist wohl die Definition des neuen Daten**TYPE**s durch aufzählen seiner Elemente:

```
TYPE Ampel = (rot,gelb,grün,braun,weiss,schwarz);
```

Allgemein ist die Syntax der Typdefinition gegeben durch:

TYPE typbezeichner = neuer Datentyp.

Zugelassen sind Aufzählungen neuer Datentypen oder Begrenzungen von Unterbereichstypen.

Etwa:**TYPE wochentag = (MO,DI,MI,DO,FR,SA,SO);**
oder:**Type ziffern = 0..9;**

Bei Aufzählungstypen sind die folgenden Operationen definiert:

ORD Position des Argumentes

```
ORD(MO) = 0
ORD(DI) = 1
```

SUCC Nachfolger des Argumentes

```
SUCC(DI) = MI
```

PRED Vorgänger des Argumentes

```
PRED(DI) = MO
```

Erlaubt sind auch strukturierte Daten**TYPE**n.

Häufig ist es sinnvoll, mit indizierten Variablen zu arbeiten. Eine 80-Zeichen lange Zeile aus einem Brief etwa läßt sich als eine Variable vom **TYPE** Zeile mit den möglichen Indizies 1-80 interpretieren. In Pascal läßt sich ein neuer Variablen**TYP** im Deklarationsteil eingeleitet durch das Schlüsselwort **TYPE** definieren:

```
TYPE zeile = ARRAY (.1..80.) of CHAR;
```

Neben eindimensionalen Feldern sind auch mehrdimensionale Felder möglich:

```
TYPE string_80 = ARRAY(.1..80.) OF CHAR;
TYPE Seite = ARRAY(.1..24.) OF string_80;
```

Sind die Komponenten eines Feldes Skalar, so sind die Vergleichsoperationen anwendbar:

Bsp. : ARRAY (.1..6.) OF CHAR;
'SCHALE' < 'SCHULE'

Mit **TYPE** definierte Daten**TYPE**n werden im Programm dann wie die Standarttypen bei Variablendefinitionen verwendet:

```
TYPE wochentage = (mo,di,mi,do,fr,sa,so);
VAR  meine_woche, deine_woche,
     seine_woche : wochentage;
```

erlaubt ist dann etwa die Zuweisung

```
meine_woche := mi;
```

Unterbereiche

Bei dem weiter oben gewählten Variablen Ampel ist der Datentyp Farbe sicher nicht glücklich gewählt. Besser wäre es, nur die Farben rot, gelb, gruen zuzulassen. Da aber auch Gründen der Eindeutigkeit Bezeichner nicht doppelt verwendet werden dürfen, können wir nicht einen weiteren Ampeltyp deklarieren. Es gibt allerdings in PASCAL die Möglichkeit, von jedem aufzählbaren Datentyp Unterbereiche zu verwenden.

Beispiel:

```
TYPE              Farbe                                    =
(rot,gelb,gruen,braun,weiss,schwarz);
     Ampel  = rot..gruen;
     SWFarbe= weiss..schwarz;
```

Ein Unterbereich wird also dadurch angegeben, daß Angangs- und Endelement des Bereichs durch zwei (!) Punkte getrennt aufgeschrieben werden.

```
PROGRAM VAmpel (INPUT,OUTPUT);
TYPE Farbe = (rot,gelb,gruen,weiss,schwarz);
     Ampel = rot..gruen;
     SWFarbe = weiss..schwarz;
VAR Ampelfarbe : Ampel;
    Schalter   : SWfarbe;
    ch : CHAR;
BEGIN
  Ampelfarbe := rot; Schalter := weiss;
  WHILE Schalter = weiss DO
  BEGIN
    IF Ampelfarbe = gruen THEN
      WRITELN('Grün -> Bitte fahren !')
    ELSE IF Ampelfarbe = gelb THEN
           WRITELN('Gelb -> Achtung !')
    ELSE IF Ampelfarbe = rot THEN
           WRITELN ('Rot -> Bitte warten!');
    Ampelfarbe := SUCC(Ampelfarbe);
    WRITE('Soll die Ampel ausgeschaltet ');
    WRITELN('werden ?');
    WRITELN('Geben Sie ein J/N');
    READLN(ch);
    IF ch='J' THEN Schalter := schwarz;
  END;
  WRITELN('Ampel ist ausgeschaltet.');
END.
```

Mengen

Selbst definieren kann man in Pascal auch Mengen. Nützlich sind sie besonders bei der Überprüfung von Bedingungen. Bedingungen bedingter Anweisungen und -Schleifen sind häufig recht komplex.

Die bedingte Anweisung

```
IF ((a >= 'A') and ( a <= 'Z')) or c = d
THEN anweisung;
```

wäre übersichtlicher zu formulieren, wenn in pascal die Zugehörigkeit zu einer Menge überprüft werden könnte, etwa

```
IF a IN buchstaben or c = d then anweisung;
```

Ist etwa die Variantensuche in einer **CASE**-Anweisung erfolglos, bricht das Programm ab. Es wäre sinnvoll, einem solchen Abbruch durch folgende Konstruktion verhindern zu können:

```
If zeichen IN buchstaben
THEN
     CASE zeichen OF
     'A' : anweisung_a;
      .
     'Z' : anweisung_z
     END;
```

Offensichtlich wird die Zugehörigkeit des Zeichens zur Menge der Buchstaben vor einem Durchlauf der CASE-Anweisung überprüft.

Mengen können in Pascal im Deklarationsteil definiert werden:

```
TYPE buchstaben = 'A'..'Z' (*Menge der großen Buchstaben*)
VAR letter : SET of buchstaben;
```

Auf Mengen sind folgende Operationen definiert:

Logische Operationen

IN	Element (prüft ob Element in einer Menge enthalten ist)
<	echte Teilmenge
<=	Teilmenge
>	echte Obermenge
>=	Obermenge
=	Gleichheit
<>	Ungleichheit

Beispiel: der Audruck **kinder < menschen** ist **TRUE**

Arithmetische Operationen

*	Durchschnitt
+	Vereinigung
-	Differenz

Beispiel: sinnvoll ist der Ausdruck

```
familie := kinder + eltern
```

wenn *kinder* und *eltern* Mengen sind, die aus Elementen vom **TYPE** menschen bestehen.

Verbunde (RECORD)

Weiter oben lernten wir die indizierten Felder kennen. Solche Felder zeichnen sich durch Komponenten gleichen Typs aus. In der Praxis aber, etwa in einem Personalbüro werden zu einer Person Angaben unterschiedlichen Datentyps zusammengefasst. Solche Datenzusammenfassungen finden sich traditionell in Karteien. In Pascal ist es möglich einen weiteren Datentyp, den **RECORD**, für solche Zusammenfassungen zu verwenden. Für ein Personalbüro könnte die Definition lauten:

```
TYPE person =       RECORD
                    nr,
                   name,
                   ort,
                   konto,
                   Str     : string_20;
                   PLZ     : string_6
                   END;

VAR arbeiter : Person;
```

Handelt es sich in Grenznahen Bereichen um ausländische Beschäftigte, so kann das Feld PLZ zu kurz sein, und die Landesangabe wichtig werden. In diesem Fall wird ein varianter RECORD definiert:

```
TYPE Karte  = RECORD
              nr,
              ort,
              konto,
              str     : string_20;
               CASE brd:BOOLEAN OF
                     TRUE     :(PLZ:string;ort          :
string_20);
                                FALSE  :(land,ort  :  string_20;

       PLZ:string_10);
               END;
```

```
VAR person : Karte
```

Funktionen und Proceduren

Funktionen kennen Sie noch aus dem Mathematikunterricht, etwa die Normalparabel (y = f(x), y = x^2) oder den sinus (y = sin(x)) Jeder Zahl x wird genau eine Zahl y zugeordnet. Wir wollen hier den Funktionsbegriff nicht genauer definieren. Wenn Sie Probleme haben, schauen Sie doch bitte in Ihrem Mathematikbuch nach. Das folgende Programm berechnet die Quadratwurzel einer Zahl.

```
PROGRAM beispiel_3(INPUT,OUTPUT);
VAR zahl : REAL;

BEGIN
  READ(zahl);
  zahl := SQRT(zahl);
  WRITE(zahl)
END.
```

Bei der Funktion **SQRT** handelt es sich um eine vordefinierte Funktion. Weitere vordefinierte Funktionen sind auf der Seite 12.

Selbstdefinierte Funktionen

In PASCAL ist es aber auch möglich, selbst solche FUNKTIONEN zu definieren. Funktionen haben immer ein Ergebnis, das z.B. einer Variabeln zugeordnet oder ausgegeben werden kann.

Selbstdefinierte Funktionen werden im Deklarationsteil eines PASCAL-Programms erklärt. Diese Funktionen werden dann im Anweisungsteil wie vordefinierte Funktionen benutzt.

Das folgende Programm zeigt dies an einem simplen Beispiel. Im Programm wird eine Additionen mit einer selbstdefinierten Funktionen durchgeführt.

```
PROGRAM beispiel_5(INPUT;OUTPUT);
VAR zahl1, zahl2, zahl3 : INTEGER;
FUNCTION add(z1,z2:INTEGER):INTEGER;

  BEGIN
     add := z1 + z2
  END;

BEGIN
  zahl1 := 6;
  zahl2 := 4;
  zahl3 := add(zahl1,zahl2)
END.
```

Dieses Programm zeigt eine selbstdefinierte Funktion. Funktionsdefinitionen dienen der STRUKTURIERUNG von Programmen. Sie werden einmal definiert und können dann beliebig oft im Programm verwendet werden. Dies macht Pascal-programme Übersichtlich. Ein Programm kann dann Schrittweise vom "groben Hauptprogramm" bis zur detailierten Unterprogrammen erstellt werden. Dieses Vorgehen wird "TOP DOWN" oder "schrittweise Verfeinerung" genannt. Die Struktur der entstehenden Programme ist dann blockartig BLOCKSTRUKTUR.

Funktionsdefinitionen beginnen nach den **VAR**iablendeklarationen mit dem Schlüsselwort **FUNCTION** gefolgt von einer geklammerten Parameterliste und deren Typedefinition. Da die Funktion selbst einen Wert zurück gibt muß auch sie selbst einem Datentyp zugeordnet werden. Dem Kopf folgt im Hauptteil ein Deklarationsteil, der selbst wieder lokale Variablen und Funktionen enthalten kann. In **BEGIN** und **END** eingeschlossen folgt dann der Anweisungsteil Das Ergebnis wird dann einem Bezeichner übergeben, der mit dem Namen der Funktion identisch ist.

Die Parameterübergabe ist eine schwierige Sache. Lesen Sie die folgenden Sätze sehr genau.

Die Definition

FUNCTION f(VAR x:INTEGER):INTEGER;

unterscheidet sich von der Definition

FUNCTION f(x:INTEGER):INTEGER;

durch die Art der Parameterübergabe. Im ersten Fall wird beim Aufruf der FUNCTION f an **VAR** x der aktuelle Parameter (sozusagen das "ORIGINALKÄSTCHEN", die Original**VAR**iable) übergeben im zweiten Fall an x nur eine Kopie (sozusagen

ein zweites Kästchen mit identischem Inhalt). Im ersten Fall führt jede Veränderung in der Funktion zu Veränderungen der Original**VAR**iable, im zweiten Fall bleibt das Original unverändert, denn in der Funktion kann nur seine Kopie verändert werden.
Ein Beispiel mag die damit verbundenen Konsequenzen näher erleutern:

```
PROGRAM beispiel_5(INPUT,OUTPUT);
VAR zahl1,zahl2,zahl3 : INTEGER;

FUNCTION add(VAR z1,z2:INTEGER):
                                INTEGER;

  BEGIN
    z1 := 1;
    add := z1 + z2
  END;

BEGIN
  zahl1 := 6;
  zahl2 := 4;
  zahl3 := add(zahl1,zahl2);
  WRITELN (zahl3)
END.
```

In dem obigen Programm werden die beiden Variablen zahl1 und zahl2 übergeben und dort wird der Variablen, die innerhalb der FUNCTION z1 heißt die 1 zugewiesen (:=) Nach Beendigung des Programms ist zahl1= 1.

CALL-BY-REFERENCE

Änderung des Parameters in der Prozedur ändern gleichzeitig die Umgebung.

```
PROGRAM beispiel_6(INPUT,OUTPUT);
VAR zahl1,zahl2,zahl3 : INTEGER;

FUNCTION add(z1,z2:INTEGER):INTEGER;

BEGIN
  z1  := 1;
  add := z1 + z2
END;

BEGIN
  zahl1 := 6;
  zahl2 := 4;
  zahl3 := add(zahl1,zahl2);
  WRITELN (zahl1, zahl3)
END.
```

Im Nebenstehenden Programm werden im Vorigen Programm wieder die Variablenzahl1 und zahl2 in lokale Variablen übergeben. Im Gegensatz zum vorigem Programm wird die globale Variable zahl1 in der FUNCTION nicht verändert.

CALL-BY-VALUE/RESULT

Der Parameter ist sowohl Ein- als auch Ausgabenparameter. d.h. der Wert geht in die FUNCTION ein, wird dort eventuell verändert, außerhalb der FUNCTION bleibt die Variable unverändert.

Selbstdefinierte PROZEDUREN

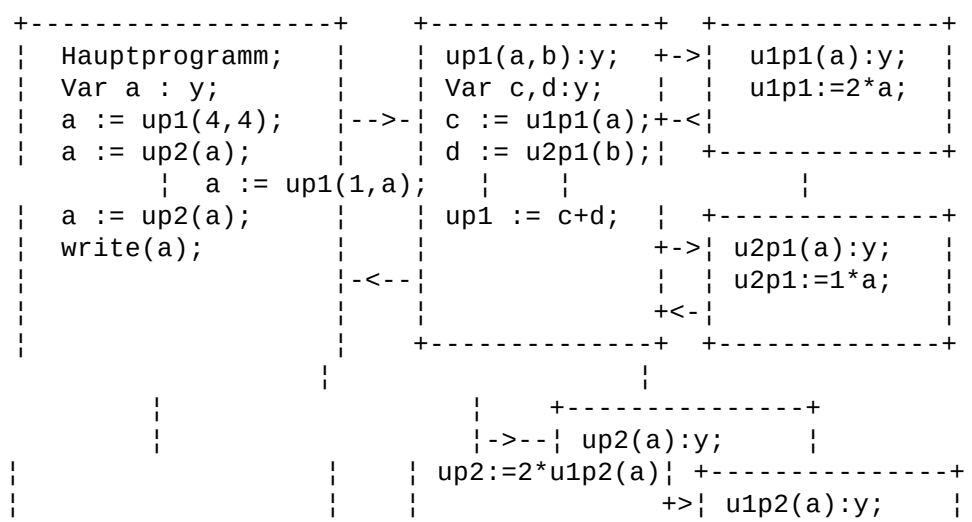
Funktionen sind Unterprogramme, die definierte Werte zurückgeben. Unterprogramme, die Teilanweisungen oder Anweisungen oder Anweisungsblöcke ausführen werden Prozeduren genannt. Sie werden wie Funktionen im Deklarationsteil definiert und mit ihrem Bezeichner, gegebenenfalls erweitert um aktuelle Parameter, im Hauptpro-

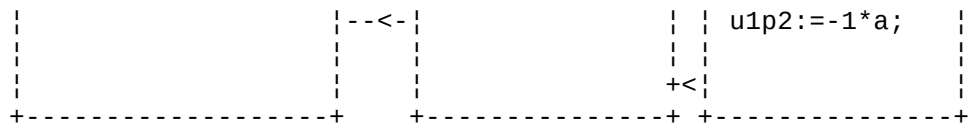
gramm wie Anweisungen aufgerufen. Vordefinierte Prozeduren sind WRITE und READ. Das folgende Programm benutzt eine Prozedur um einen Wert auszugeben.

```
PROGRAM beispiel_8(INPUT,OUTPUT);
VAR zahl : INTEGER;

PROCEDURE schreibe(zahl:INTEGER);
BEGIN
  WRITE(zahl);
END;

BEGIN
  READ(zahl);
  schreibe(zahl)
END.
```

Bei der Parameterübergabe finden die selben Regeln Anwendung, wie sie auch bei Funktionen gelten.

EXKURS
Blockstruktur und strukturierte Programmierung;
Selbstdefinierte **Funktionen** und **Prozeduren**, wie wir sie im letzten Kapitel kennenlernten sind das Salz in der Suppe **PASCAL**s. Sie machen Pascal-Programme übersichtlich und strukturieren die Programmerstellung von Anfang an. Die Zauberworte heißen **schrittweise Verfeinerung** oder **Top down Programmierung**, wie es im Fachchinesisch heißt. Ein Problem, daß zur Programmierung ansteht, wird zunächst grob beschrieben. Schritt für Schritt werden dann die Feinheiten immer genauer bestimmt. Aufgabenstellungen für Programme können sehr komplex sein. Damit sie übersichtlich und verständlich werden, gliedert man sie auf verschiedenen **hirarschischen** Schichten in **Bausteine**, deren Inhalt und Verknüpfungen (**Schnittstellen**) mit anderen Bausteinen man definiert. Eine zu lösende Aufgabe wird zunächst als Block (Baustein) in den Programmentwurf aufgenommen. Danach wird dieser Block in immer feinere Blöcke unterteilt. Über die Schnittstellen sind dann die Ein- und Ausgaben eines Blocks definiert. Die Eingaben kommen aus einer höheren Schicht in einen untergeordneten Block. Die Ausgaben aus einem Block niederer Ordnung wandern in einen Block höherer Ordnung. Auf dieser Ebene sind die Blöcke noch rein nach ihrer zu erfüllenden Funktion, also funktional definiert. Erst auf der Untersten Ebene, in der Regel bereits der Ebene der Codierung, etwa in Pascal, erfolgt die technische Definition, also die Definition der Durchzuführenden Schritte, um die geforterte Funktion zu erfüllen.
Das folgende grafische Beispiel mit Pseudo-Pascal-Notation mag den Gedankengang anschaulich machen:





Dateien

Dateien sind in PASCAL im Hintergrund gespeicherte Ketten aus Elementen gleichen Datentyps. Zwei Vertreter eines solchen Dateityps haben wir bereits kennengelernt. **INPUT** und **OUTPUT** sind **TEXT**-Dateien. Auf diese Dateien sind die Prozeduren **WRITE, WRITELN, READ, READLN** in der bereits bekannten Art definiert. **INPUT** und **OUTPUT** sind die standartisierte Dateien zur Ein- und Ausgabe. Für sie gelten Besonderheiten, die für selbstdefinierte Dateien nicht definiert sind. So ist **INPUT** ausschließlich Eingabedatei, **OUTPUT** ausschließlich Ausgabedatei. Darüberhinaus können über **INPUT** nicht nur Texte sondern auch Zahlen eingelesen werden, analoges gilt für **OUTPUT**. Auf selbstdefinierten Textdateien **DatenTYP TEXT**;) sind die bekannten Schreib- und Leseprozeduren definiert. Jede Textdatei enthält Markierungen für das Zeilenende und eine Markierung für das DateiDateiende. PASCAL kennt die vordefinierten logischen Funktionen **EOF** (Ende der Datei) und **EOLN** (Zeilenende) Die Funktionen geben TRUE zurück, wenn die jeweilige Markierung erreicht wurde sonst FALSE. Das folgende Programm liest einen Text und gibt ihn unverarbeitet wieder aus.

```
PROGRAM bsp_10(eingabe,ausgabe);
VAR
  eingabe,
  ausgabe : TEXT;
  ch      : CHAR;
BEGIN
   RESET(eingabe)
   REWRITE(ausgabe)
   WHILE NOT EOF(eingabe)
      DO
      BEGIN
       WHILE NOT EOLN(eingabe)
        DO
        BEGIN
         READ(eingabe,ch);
         WRITE(ausgabe,ch)
        END;
       READLN(eingabe);
       WRITELN(ausgabe)
     END
END.
```

```
¦ ¦D  ¦I  ¦E  ¦*  ¦I  ¦S  ¦T  ¦*  ¦E  ¦I  ¦
+-----------------_-----_------------------+
  READ(ch); (* die Variable ch ist nun gleich 'I'*)
+--------------------_-----_---------------+
¦ ¦D  ¦I  ¦E  ¦*  ¦I  ¦S  ¦T  ¦*  ¦E  ¦I  ¦
+--------------------_-----_---------------+
```

Wie durch ein Fenster schaut Pascal immer nur auf ein Element einer Datei. Ist dieses Element gelesen springt das Fenster automatisch zum nächsten Element, auf das es wie ein "Zeiger" hinweist.

Mit Ausnahme der Prozeduren **Readln** und **Writeln**, die nur auf Textdateien sinnvoll definiert sind, arbeiten die Schreib und Leseprozeduren auf Dateien anderen Typs identisch:

```
+---------------------------------------+
¦PROGRAM bsp_11(eingabe,ausgabe);       ¦
¦VAR                                    ¦
¦  eingabe,                            ¦
¦  ausgabe : FILE OF INTEGER;          ¦
¦  zahl    : INTEGER;                  ¦
¦BEGIN                                  ¦
¦  RESET(eingabe);                      ¦
¦  REWRITE(ausgabe);                    ¦
¦  WHILE NOT EOF(eingabe)               ¦
¦   DO                                  ¦
¦   BEGIN                               ¦
¦     READ(eingabe,zahl);               ¦
¦     WRITE(ausgabe,zahl)               ¦
¦   END;                                ¦
¦END.                                   ¦
+---------------------------------------+
```

Zeiger - Typen

Bis zu diesem Punkt sind nun die wesentlichen Merkmale Pascals behandelt. Mit den Themen einschließlich des letzten Kapitels müssen Sie in Abschlußklausur rechnen. PASCAL aber verfügt über eine weitere Eigenschaft, die diese Sprache recht interessant macht. Da das Verständnis dieser Eigenschaft der Sprache einen mühsamen "Kreuzweg" voraussetzt, werden wir uns hier darauf beschränken, in den folgenden beiden Kapiteln Ihr Interesse für diese Eigenschaft zu wecken. Wir erwarten aber nicht, daß Sie diese Spracheigenschaft bei Ende dieses Kurses perfekt beherrschen. Die Arbeit mit dieser Spracheigenschaft bleibt einem eigenen Kurs vorbehalten.Vergessen Sie zunächst diesen Kurs. Stellen Sie sich vor, Sie verbringen Ihren Jahresurlaub in einem verschlafenen Nest an der Südküste Kretas. Zwischen den steilen Wänden der gebirgigen Küste haben Sie

eine einsame Bucht gefunden. Die Sonne brennt Ihnen aufs Hirn. Da kommt Ihnen der Wahnsinnige Einfall, direkt am Wasser mit dem feuchten Sand ein Modell der Küstenlinie Kretas anzufertigen. Ihre Begeisterung treibt sie voran. Nach Stunden mühevoller Arbeit sind sie fertig. Stolz zeigen Sie Ihren Freunden das Werk. Man lobt sie, stellt aber fest, daß das Modell nun unvollständig wäre. Durch die Schaffung des Modells sei die Küstenlinie am Ort so verändert worden, daß im Modell selbst an der äquivalenten Stelle ein Modell des Modells anzufertigen sei und so fort. Erschrocken stellen Sie fest, daß diese Aufgabe wohl kein Ende findet und erinnern sich an die Ihr Gesicht links und rechts begrenzenden und ins Unendliche reflektierenden Toilettenspiegel bei der morgendlichen Rasur im Hotel.Kehren wir an dieser Stelle zu PASCAL zurück. Vernachlässigen wollen wir auch die sich anbietende philosophische und mathematische Diskussion des Themas. Es bleibt dann die Feststellung, vor uns zwei anschauliche Beispiele einer Selbst- oder Rückbezüglichkeit vor uns zu haben. Ähnlich lassen sich Texte definieren:

Ein Text ist entweder ein Buchstabe oder ein Buchstabe gefolgt von einem Text.

Diese Definition ist noch etwas holprig, weil sie wie unser Kretamodell im Unendlichen verschwindet. Aber vielleicht geht es so:

> *Ein n Zeichen langer Text ist leer wenn n = 0 oder einfach ein Zeichen wenn n=1 oder ein Zeichen gefolgt von einem n-1 Zeichen langen Text, wenn n > 1.*

Teufel das ist nicht einfach zu verstehen; besonders dann, wenn die Nachfolger explosionsartig wachsen:

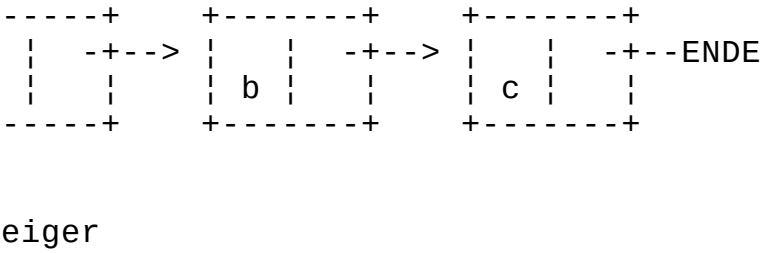
> *Eine n-blättrige Eiche ist ein blattloser Stamm wenn n=0 sonst ist sie ein eichenblatt gefolgt von einer Astgabel und einer*

In Pascal werden solche Datenstrukturen mit "Zeigertypen" definiert. Für das Beispiel mit dem n-Zeichen langen Text wird die Datenstruktur wie folgt definiert:

```
TYPE    zeiger     = ^buchstaben;
        buchstaben = RECORD
                       ch   : CHAR;
                       nach : zeiger
                     END;
VAR textanfang : zeiger;
```

```
        vorgaenger,
        nachfolger,
        neu        : buchstaben;
```

Es ist kein Zufall, daß diese Datenstruktur der einer Textdatei ähnlich ist. Wie dort verweist der Vorgänger auf seinen jeweiligen Nachfolger:



Wie dort wird auf ein Element über Zeiger zugegriffen
Da erst während der Laufzeit des Programms die aktuelle Datenstruktur aufgebaut wird, existiert erst zur Laufzeit auch ein Bereich für die Variable im Speicher des Rechners.
Definierte Operationen sind dann:

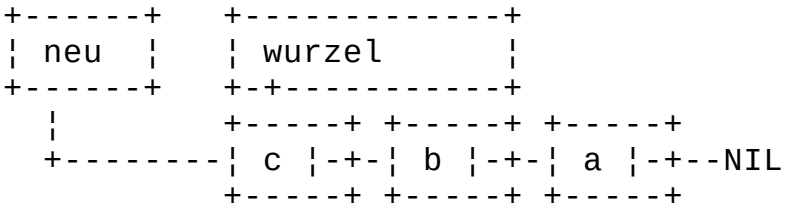
NEW(zeiger) *für die Variable wird Speicherplatz reserviert*

DISPOSE(zeiger) *Speicherplatz wird wieder freigegeben.*

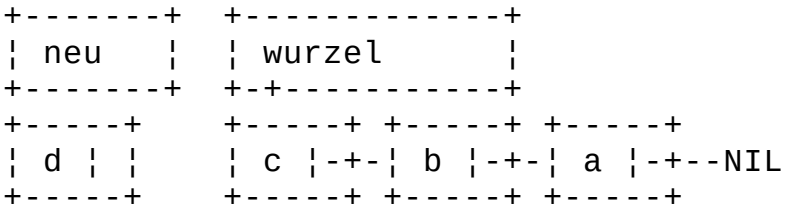
zeiger_a := zeiger_b;	*Zeiger_a zeigt nun auf den selben Speicherbereich wie zeiger_b*
zeiger_a^ := zeiger_b^;	*Beide Zeiger zeigen weiterhin auf verschiedene Speicherbereich. Der Inhalt dieser Speicherbereich ist nun aber identisch*
zeiger_a := NIL;	*zeiger_a zeigt ins leere*

grafisch läßt sich der Aufbau einer Kette (Liste) von Elementen dann offensichtlich so veranschaulichen:

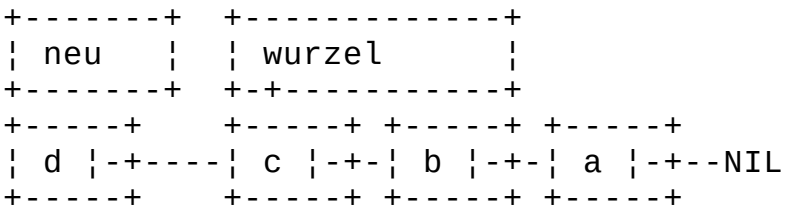
1. eine Liste mit 4 Elementen



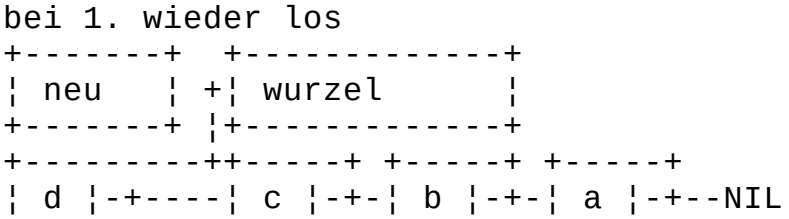
2. ein neues Element wird angelegt (NEW(neu))



3. neu zeigt auf das element, auf das auch die wurzel zeigt



4. die wurzel zeigt auf das neu eingefügte element. es geht bei 1. wieder los



Das folgende Beispiel liest einen Text ein und fügt ihn in eine Liste ein, die dann ausgegeben wird:

```
PROGRAM bsp_12(INPUT,OUTPUT);
TYPE
  zeiger  =  ^zeichen;
  zeichen =  RECORD
               ch : CHAR;
               next : zeiger
             END;
VAR
  wurzel,
  aktuell : zeiger;
  neu     : zeichen;
```

```
BEGIN
  wurzel := NIL;
  WHILE NOT EOF
  DO
  BEGIN
     NEW(neu)
     READ(neu^.ch);
     neu^.next := wurzel;
     wurzel := neu
  END;
  IF wurzel <> NIL THEN
  BEGIN
    aktuell := wurzel;
    WHILE aktuell <> NIL
    DO
    BEGIN
      WRITE(aktuell^.ch);
      WRITELN;
      aktuell := aktuell^.next;
    END
  END
END.
```

Es ist noch kein Meister vom Himmel gefallen. Sollten Sie noch Schwierigkeiten mit dem Datentyp Zeiger haben, so verlieren Sie nicht den Mut. Ansonsten verweise ich auf die weiterführende Literatur.

Im nächsten Kapitel werden wir einen kleinen Ausflug in die Arbeit mit rekursiven Prozeduren und Funktionen machen. Dieses Kapitel dient aber hauptsächlich zur Information, was in PASCAL möglich ist.

Rekursive Funktionen und Prozeduren

Zu selbstbezüglichen, also **rekursiven** Datenstrukturen, wie wir sie im letzten Kapitel kennenlernten passen rekursive Prozeduren und Funktionen wie der richtige Hut zum richtigen Kopf.
Zur übersichtlichkeit des Programms bsp_12 würde es gut passen, ließe sich die WHILE- Schleife durch eine elegante Prozedur ersetzen. Nur versuchen wir es doch einfach einmal:

```
PROGRAM bsp_12(INPUT,OUTPUT);
TYPE zeiger = ^zeichen;
     zeichen = RECORD
                ch : CHAR;
                next : zeiger
               END;
VAR
 wurzel,aktuell : zeiger;
 neu : zeichen;
PROCEDURE ausgabe(z:zeichen);
BEGIN
  WRITE(z^.ch);
  WRITELN;
  IF z^.next <> NIL
  THEN ausgabe(z^.next)
END;
BEGIN (*HAUPTPROGRAMM*)
  wurzel := NIL;
  WHILE NOT EOF DO BEGIN
     NEW(neu);
     READ(neu^.ch);
     neu^.next := wurzel;
     wurzel := neu
  END;
  IF wurzel <> NIL
  THEN BEGIN
    aktuell := wurzel;
    ausgabe(aktuell)
  END
END.
```

Es fällt auf, daß die Prozedur-Ausgabe sich mit dem Nachfolger von aktuell keinen Nachfolger mehr hat. Sollte sich hier Münchhausen doch am eigenen Schopf aus dem Sumpf ziehen können? Sicher nicht. Erinnern wir uns an die im letzten Kapitel angelegte Liste. Auch dort wurde eine Rückbezüglichkeit definiert. Rückbezüglich waren die Elemente der Liste aber nicht auf sich selbst, sondern auf einen identischen Nachfolger. Ein Beispiel mag dies Verdeutlichen.

Welche Ausgabe gibt wohl folgendes Programm zurück?

```
PROGRAM na_nuh(INPUT,OUTPUT);
PROCEDURE ein;
     VAR ch : CHAR;
     BEGIN
       READ(ch)
       IF ch <> ' ' THEN ein;
       WRITE(ch)
     END;
BEGIN
  ein
END.
```

Rekursiv können auch Funktionen definiert werden. Mit dem letzten Beispiel soll dann dieser erste Einblick in rekursive Definitionen abgeschlossen sein. Das Programm berechnet die Fakultät eines Argumentes:

```
PROGRAM faku(INPUT,OUTPUT);
VAR n : INTEGER;
FUNCTION f(n:INTEGER):INTEGER;
BEGIN
  IF n=1 THEN f:=1
         ELSE f:=n*f(n-1)
END;
BEGIN
  READ(n);
  WRITE(f(n))
END.
```

Vom Problem zum Programm

Diese Kursmappe schließt mit einem Beispiel zur schrittweisen Programmerstellung. Der Mappe entsprechend kann es nur ein kleines Beispiel sein. Wir hoffen dennoch, daß es Sie zur Vertiefung Ihrer Kenntnisse (etwa in der weiter vorne genannten Literatur) anregt:

Eine Institutsbibliothek erstellt monatlich für die Angehörigen des Institutes von den neu angeschafften Monographien drei alphabetisch sortierte Listen nach Autor, Signatur und Titel. Die Erstellung der Listen soll nun automatisiert werden. Am Monatsende erhält die Institutsbibliothek über das lokale Netz eine Kopie der

Datei mit den Neuanschaffungen von der Zentralbibliothek. Aus dieser Datei. ist die Liste zu erstellen. Die Datei ist als file organisiert, dessen Komponenten je drei zeichenketten der Länge 20 bilden; je ein Feld für Autor, Signatur und Titel. Die Anzahl der neu angeschafften Titel variiert monatlich.

Das Problem ist nun bekannt. Wie läßt es sich lösen? Die Struktur der Eingabedaten liegt bereits fest. Für die Ausgabe bietet es sich an, zunächst drei dynamische Listen für Autor, Signatur und Titel anzulegen, da bei Programmstart die Anzahl der zu verarbeitenden Titel nicht festliegt. Da alle drei Komponenten aus identisch langen Zeichenketten bestehen, bietet sich als Datenstruktur ein ARRAY(.autor,signatur,titel.) OF string_20 an. Die gleiche Struktur bietet sich für die drei Wurzeln der Listen an: ARRAY(.autor,signatur,titel.) of zeiger. Die Ausgabe gestaltet sich problemlos. Jede der drei Listen wird vom ersten bis zu letzten Element ausgegeben, dabei wird nach jeder Ausgabe zum Nachfolger "weitergeschaltet".
Schwieriger sieht es schon beim Einfügen aus. Wir haben vier mögliche Fälle je Liste zu unterscheiden:

1. Die Liste ist noch leer. In diesem Fall wird das neue Element einfach an die Wurzel gehängt.

2. Das neue Element hat keinen Nachfolger. Dann wird es an das letzte Element der aktuellen Liste gehängt.

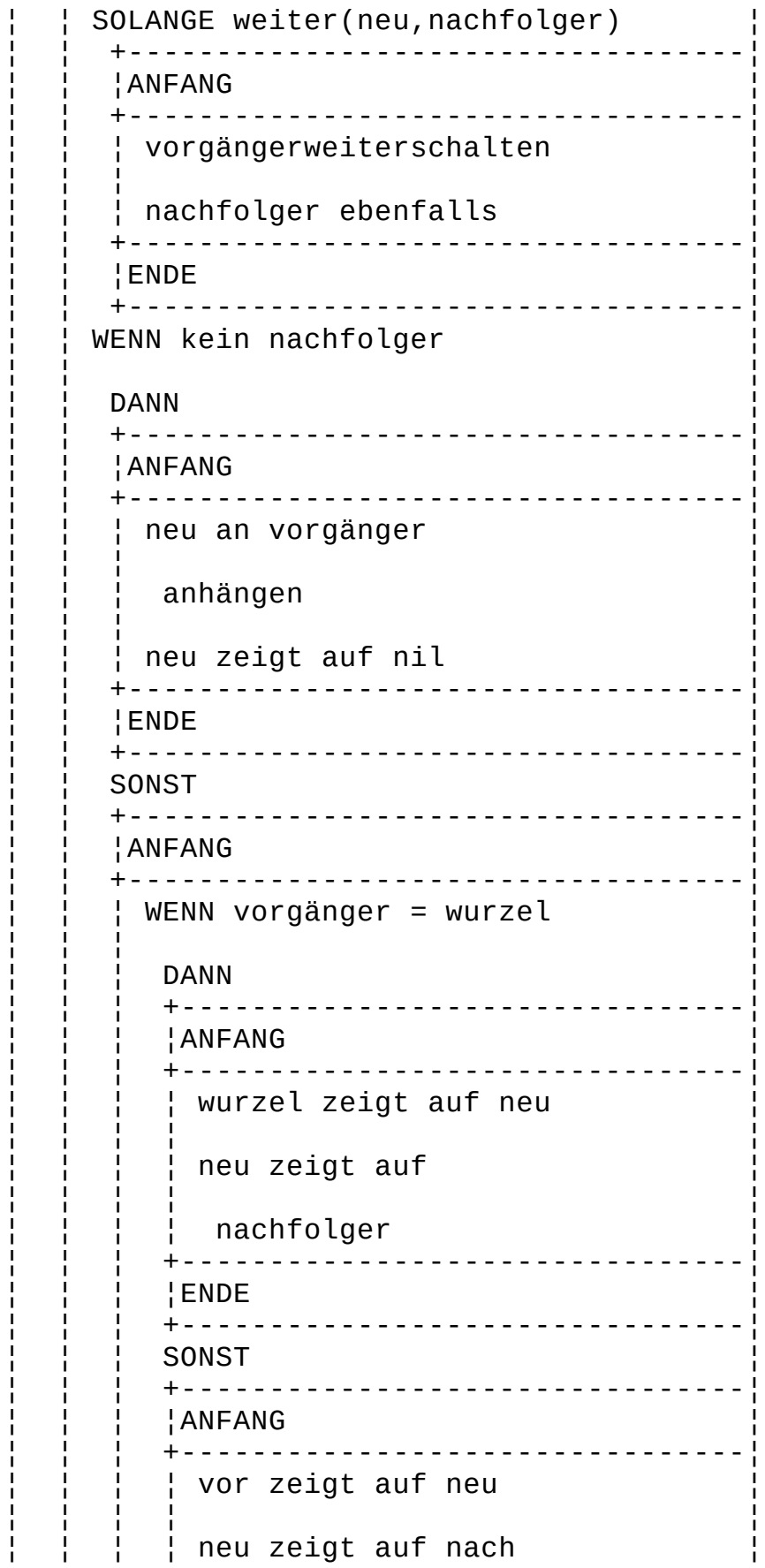
3. Das neue Element hat keinen Vorgänger. Dann wird es vor seinen Nachfolger gesetzt und an die Wurzel gehängt.

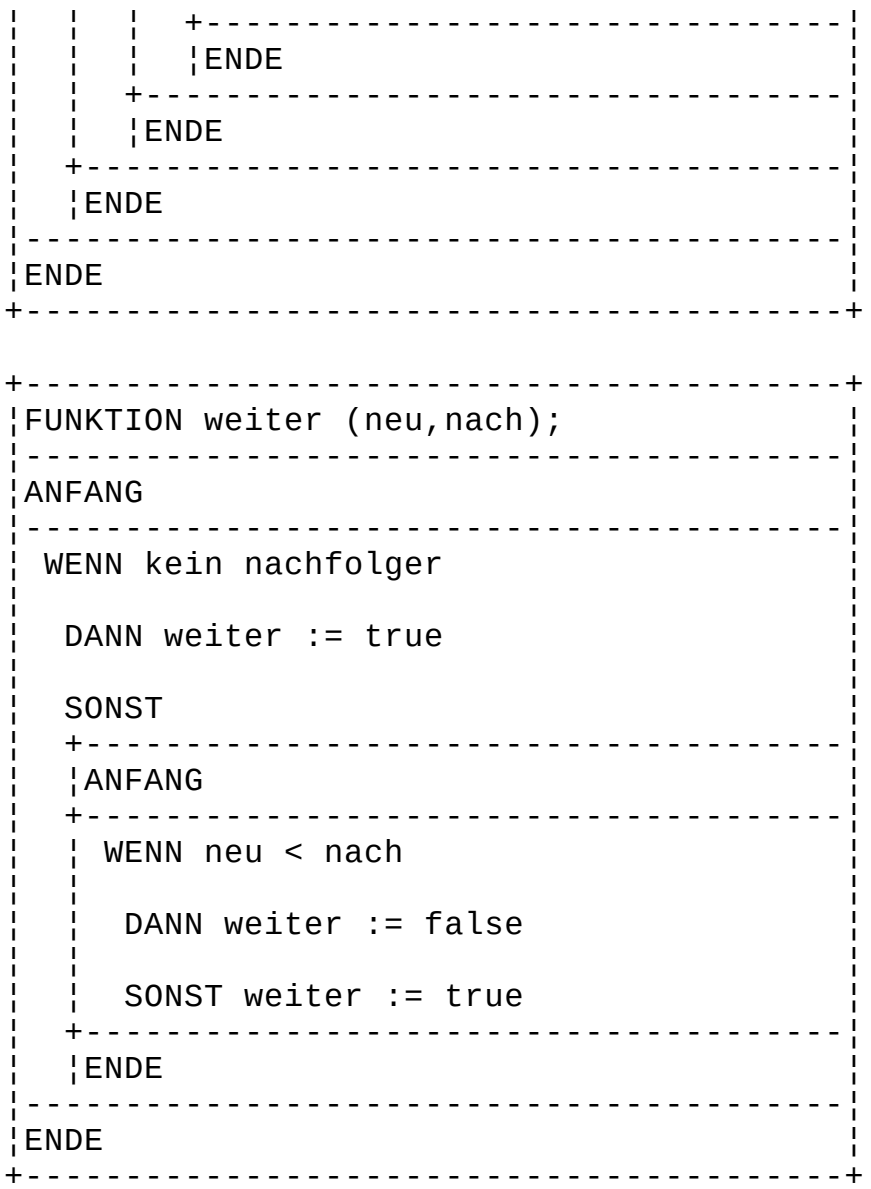
4. Das neue Element hat einen Vorgänger und einen Nachfolger. In diesem Fall wird es an den Vorgänger gehängt und an es selbst sein Nachfolger.

Neben einer Daten struktur für die Wurzel benötigen wir also noch eine Datenstruktur für das neue Element, bestehend aus den drei Strings für die Zeichenkette und einem ARRAY of zeiger für die jeweiligen Nachfolger. weiter benötigen wir je einen zeiger für den vorgänger und den nachfolger. Die Datenstrukturen sind nun bekannt. Weiter benötigen wir nach eine Prozedur für den Einfügevorgang und eine Prozedur, die das Weiterschalten der eiger zur Suche der Einfügestelle steuert:

+--+


bibliothek(datei,ausgabe);
ANFANG
SOLANGE nicht(datei(ende))
ANFANG
lies nächstes dateielement
einordnen(element)
ENDE
FÜR autor,signatur,titel
ANFANG
schreibe(listenelement)
nächstes element suchen
ENDE
ENDE.
PROZEDUR einordnen(element)
ANFANG
WENN wurzel leer
DANN
ANFANG
wurzel zeigt auf neu
neu zeigt auf nil
ENDE
SONST
ANFANG


SOLANGE weiter(neu,nachfolger)
ANFANG
vorgängerweiterschalten
nachfolger ebenfalls
ENDE
WENN kein nachfolger
DANN
ANFANG
neu an vorgänger
anhängen
neu zeigt auf nil
ENDE
SONST
ANFANG
WENN vorgänger = wurzel
DANN
ANFANG
wurzel zeigt auf neu
neu zeigt auf
nachfolger
ENDE
SONST
ANFANG
vor zeigt auf neu
neu zeigt auf nach



Die benötigten Datenstrukturen und der auf ihnen arbeitende Algorithmus ist nun bekannt. Wir können mit der Kodierung in Pascal beginnen:

```
(*****************************************)
      program buecher ( titeldatei,output);
(*========DEKLARATIONEN=====================================*)
(*________TYPE-Deklaration_____________________________*)
      type    zeiger    =     ^buch ;
              merkmal   =     ( autor , titel , signatur );
              zeigerfeld=     array(.merkmal.) of zeiger;
```

```
              string_20 =     array(.1..20.) of char;
              buch      =     record
                              text:array(.merkmal.) of
                                   string_20;
                              naechster:zeigerfeld
                              end;
              dateibuch =     record
                              text:array(.merkmal.) of
                                   string_20;
                              end;

(*________VAR-Deklaration_____________________________________*)
      var
          wurzel          : zeigerfeld;
          neubuch ,
          aktuell         : zeiger;
          titeldatei      : file of dateibuch;
          werk            : dateibuch;
          art ,
          zaehler           : merkmal;
(*=====PROZEDUR=========================================*)



      Procedure einordnen ( art         : merkmal    ;
                            var wurzel  : zeigerfeld ;
                            var neubuch : zeiger       ) ;
           var
              vorgaenger,
              nachfolger  : zeiger;
              (*____lokale___Funktion_____________________*)
              function weiter ( var neu ,
                                nach : zeiger ) : boolean ;
              begin
                    if nach = nil
                     then
                      weiter  := false
                    else
                     begin
                      if nach^.text(.art.) <
                         neu^.text(.art.)
                         then
                          weiter := true
                         else
                          weiter := false
                     end
              end;
              (*_________________Ende_lok_Funktion_*)

           begin
            if wurzel(.art.) = nil
             then
              begin
               wurzel(.art.)             := neubuch;
               neubuch^.naechster(.art.) := nil
```

```
              end
             else
              begin
               vorgaenger := wurzel(.art.);
               nachfolger := wurzel(.art.);
               while weiter ( neubuch , nachfolger )
                do
                 begin
                  vorgaenger := nachfolger;
                  nachfolger := nachfolger^.naechster(.art.);
                 end;
               if nachfolger = nil
                then
                 begin
                  vorgaenger^.naechster(.art.) := neubuch;
                  neubuch^.naechster(.art.)    := nil
                 end
                else
                 begin
                  if vorgaenger = nachfolger
                   then
                    begin
                     wurzel(.art.) := neubuch;
                     neubuch^.naechster(.art.) := nachfolger
                    end
                   else
                    begin
                     vorgaenger^.naechster(.art.) := neubuch;
                     neubuch^.naechster(.art.)    := nachfolger
                    end
                  end
                end
            end;



(*-------------------Hauptprogram-------------------------------*)
            begin
             reset(titeldatei);
             for art := autor to signatur
              do
               wurzel(.art.) := nil;
             while not eof(titeldatei)
              do
               begin
                read(titeldatei,werk);
                new(neubuch);
                for art := autor to signatur
                 do
                  begin
                   writeln(werk.text(.art.));
                   neubuch^.text(.art.) := werk.text(.art.);
                  end;
                for art := autor to signatur
```

```
                do
                einordnen ( art ,
                                      wurzel ,
                                      neubuch          );
              end;
                for art := autor to signatur
                 do
                 begin
                  case art of
                   autor    : write('----autoren---');
                   titel    : write('----titel-----');
                   signatur : write('----signatur--')
                  end;
                  writeln;
                  aktuell := wurzel(.art.);
                  while aktuell <> nil
                   do
                    begin
                     writeln(aktuell^.text(.art.));
                     for zaehler := autor to signatur
                      do
                       if zaehler <> art then
                        write(aktuell^.text(.zaehler.),' ');
                     writeln;
                     write('==============================');
                     writeln;
                     aktuell := aktuell^.naechster(.art.)
                    end
                end;
(*********************************************************)
(*         ENDE                                          *)
(*********************************************************)

           end.
```