Wirtschaftsinformatik

Programmierung I

§1 Überblick, Grundbegriffe

Programmierung : 1.) Vorgang der Programmerstellung
2.) Teilgebiet der Informatik, das die Methoden beim Entwickeln von Programmen umfaßt.

zu 1.)

- die exakte Formulierung der Lösung ist notwendig, um das Problem überhaupt angehen zu können. (**Spezifikation**)
- Formulieren von Teilaufgaben (**Zergliedern**)
- Festlegung der **Datenstrukturen**
- evnt. Formulierung in einer „**Beschreibungssprache**“ z.B. Flußdiagramme wie Programmablaufplan (PAP), Datenflußdiagramme, Struktugramnme, Pseudocode. (➔ **Softwareengeneering**)
- **Codierung** in einer Programmiersprache.
- **Test** des Programms

zu 2.)

- Theoretische Informatik
  - formale Sprache
  - Automatentheorie
- Praktische Informatik
  - Programmierung
  - Datenbanken
  - Entwicklungsumgebung und Werkzeuge (Case-Tools (Computer Aided Software Engeneering))
  - Informationssysteme
- technische Informatik
  - Rechnerarchitektur
  - Rechnerorganisation
- angewandte Informatik
  - wichtigstes Teilgebiet der Wirtschaftsinformatik
  - Ergonomie, Mensch-Maschine-Kommunikation
  - Rechtsinformatik
  - medizinische Informatik

Programm: Anweisung, die einen Rechner veranlassen, bestimmte Dinge zu tun.
Software: Alle Programme (System- und Anwendungsprogramme) von Rechner

Forderungen an die Software:

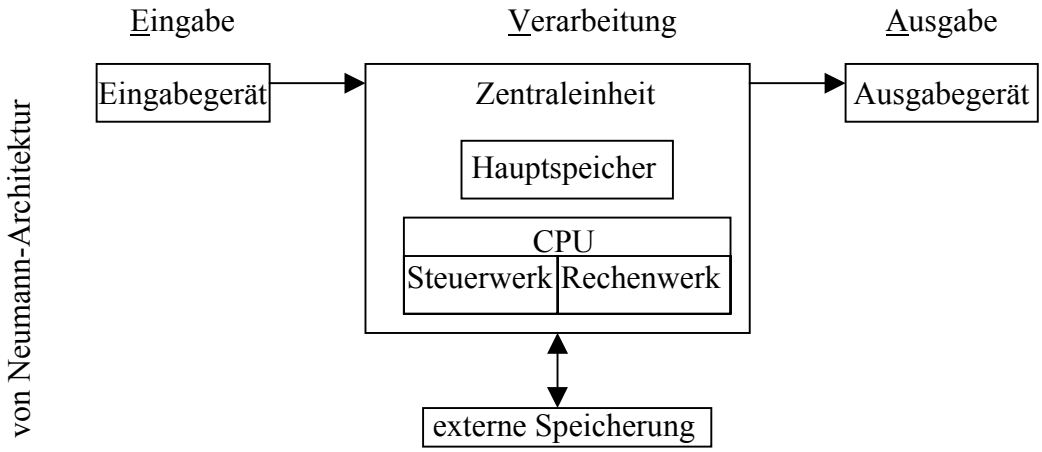
- Zuverlässigkeit
- Benutzerfreundlichkeit
- Ergonomie
- Effizienz
- Wartbarkeit
- Portabilität

§2 Grundfunktionen einer DV-Anlage (Hardware)

Prinzipieller Arbeitsablauf:

- Eingabe von Daten (Input)
- Verarbeiten von Daten
- Ausgabe von Daten (Output)
- (externe) Speicherung von Daten und Programmen

➔ EVA – Prinzip



A) Zentraleinheit

- Hauptspeicher (auch Arbeitsspeicher)
  Speichert die auszuführenden Programme und die dafür benötigten Daten
- CPU
  Central-Prozessing-Unit: interpretiert die Programmbefehle und führt sie aus.
- Interne Datenwege (Datenbusse)
  Datentransfer zwischen Hauptspeicher und Peripherie

B) Hauptspeicher

merkmale des Hauptspeichers gegenüber dem Festspeicher oder externem Speicher:

- jeder Speicherzelle wird genau eine Zahl zugeordnet. Diese Zahl wird verwendet, um die Speicherzelle anzusprechen. Man spricht daher auch von der Adresse der Speicherzelle.

- Die CPU kann nur aus dem Haupspeicher Befehle auslesen. Dazu bedient sie sich der Adresse der entsprechnenden Speicherzelle. Die Adresse des nächsten Befehles ist in dem Instruction-Pointer-Register (IP-Register) gespeichert.

C) Die CPU

Das Steuerwerk regelt die Reihenfolge, in der die Befehle eines Programms ausgeführt werden. Gleichsam werden die Befehle entschlüsselt. Dabei gibt es :

- Übertragungsbefehle
- Sprungbefehle (bedingte Sprünge nach Abfragen oder unbedingte)
- Vergleichsbefehle
- arithmetische Befehle

insgesamt kann der Befehlssatz eines Prozessors je nach Bauart von 150 bis 300 Befehle enthalten (s.a. RISC = „Reduced Instructionset“).

§3 Software

Programmiersprachen dienen der Formulierung einer Lösung zu einem Problem (Algorithmus) in einer der Maschine verständlichen Form.

A) Maschinensprache (1. Generation)

Bei der Maschinensprache wird ein Programm in Befehlen in Binärer Form (0,1) ausgedrückt.
Beispiel: die Addition von 5 und 2 in einem Befehl

0011010	0101	0010
Additionsbefehl	5	2

Ein auf dieser Technik geschriebenes Programm ist
- sehr unübersichtlich
- fehleranfällig
- der Hardware angepasst

B) Assemblersprachen (2. Generation)

Die Assemblersprache stellte ein Hilfsmittel dar, um die Maschinensprache weiter als Sprache verwenden zu können. Dabei wurde jede Binärkombination eines Befehls genau einem Wort zugeordnet, welches die Aufgabe des Befehles am besten beschrieb. Diese Worte heißen Mnemonics (Gedächtnis-stützende Abkürzung). Jeder Code korrespondiert genau mit einem Maschinenbefehl.
Beispiel von oben in Assembler :

```
ADD  5,2
```

Jeder Befehl muß nun mittels eines Assemblers übersetzt werden. Dazu kann der Assembler eine Tabelle verwenden. Assemblerprogramme sind optimal (wenn gut Programmiert) hinsichtlich Speichernutzung und Geschwindigkeit, da sie maschinennah und auch Maschinenorientiert sind. Das ist aber auch der Grund, warum auch Assemblerprogramme schlecht zu portieren sind.

C) Prozedurale Sprachen (3. Generation)

- höhere (d.h. auch Maschinenunabhängige) Programmiersprache
- zur Problemlösung werden Allgorithmen formuliert
- eine Anweisung einer höheren Sprache kann viele Befehle in Maschinensprache ergben
- Programme sind auf Maschinenebene (Compilat) nicht so optimal, wie die direkt in Maschinensprache (bzw. Assembler) entwickelten Programme, jedoch kann der Programmierer seine Aufmerksamkeit auf die Problemlösung beschränken.

- durch Verwendung verschiedener Compiler *kann* der Quelltext eines Programms auf einer anderen Plattform nach einer Neucompilation funktionieren. Es gibt daher möglichkeiten der Portabilität.
- ebenfalls zur dritten Generation der Programmiersprachen gehören auch die Interpreter, welche das Programm zur Laufzeit *interpretieren*. Eine Übersetzung in eine andere Sprache geschieht nicht, der Programmierer daher interaktiv. Interpreterprogramme sind aber lange nicht so schnell, wie Compilate.

Beispiele für Programmiersprachen der 3. Generation :

- Cobol (Common Business Oriented Language)
  - *noch* die häufigste Programmiersprache
  - für betriebswirtschaftliche Anwendungen
  - Vorteile :
    relativ gut lesbar
    gut dokumentierbar
    gut, um große Datenmengen zu verarbeiten

- Fortran (Formular Translator)
  - technisch, wissenschaftliche Sprache
  - numerische Algorithmen
  - weniger gut für Texte und Datenmengen geeignet

- Algol (Algorithmic Language)
  - wie bei Fortran

- Basic (Beginners All purpose Symbolic Instructioncode)
  - schlecht zu strukturieren (früher, -> Visual Basic)
  - normalerweise eine Interpretersprache

- Pascal (nach Blair Pascal, erfunden aber von Niklaus Wirth)
  - gut zu strukturieren
  - viele Datenstrukturen
  - Einsatz im Hochschulbereich

- Modula 2
- PL/1 (Programming Language, das Beste aus Cobol, Fortran und Algol)
- ADA
- C
  - Hardwarenahe Arbeit möglich
  - strukturiert und daher im Kommerziellen Bereich einsetzbar
  - für fast alle Rechnerarchitekturen stehen C-Compiler zur Verfügung
  - Unix z.B. ist zu 90% in C geschrieben worden, was ein Grund für die hohe Verbreitung sein dürfte.
  - ANSI-Standard (1989) (ANSI=American National Standarisation Institut)

D) Objektorientierte Sprachen (4. Generation oder auch 4GL)

Nichtprozedurale Sprache, d.h. es wird dem Rechner *nicht* mitgeteilt, wie er etwas machen soll, sondern nur, wie das Ergebnis aussehen soll, bzw. was geschehen soll.

Umgangsprachliches Beispiel:

Prozedural:

1. Lies Mitarbeiterdaten
2. Prüfe, ob Monatsgehalt > 6.000,-
3. Falls ja, gib Namen aus
4. Falls nein, ignorieren
5. Zurück zu 1, wenn nicht EOF

Objektorientiert:

Finde alle Mitarbieter, deren Monatsgehalt höher liegt, als 6.000,-

Beispiele für Objektorientierte Sprachen :

- SQL (Structured Query Language, eine Datenbank-Abfrage-Sprache)
- CSP
- Informix
- Natural
- SAS

Objektorientierte Sprachen sind gut geeignet, große Datenbankabfragen zu erledigen, leider aber ist der Programmcode um 50%-100% langsamer, als in der prozeduralen Version, da der objektorientierte Compiler sozusagen den Prozeduralen Code nachschaltet, der hinter der Anweisung steht. Dieser Code ist aber nicht mehr Optimierbar und daher möglicherweise nicht so gut auf ein Problem zugeschnitten, wie es ein Mensch vielleicht Codiert hätte.

In traditionellen Programmiersprachen wurde eine Trennung zwischen Code und Daten vorgenommen.

In objektorientierten Programmiersprachen (OOP) wird diese Trennung aufgehoben. Objekte enthalten die benötigten Daten und den zugehörigen Code, um diese Daten manipulieren und das Objekt verändern zu können.

Bei graphischen Oberflächen können folgende Dinge Objekte darstellen :
Fenster, Eingabefelder, Textfelder, Dropdown-Listen, Menüs ...

Man unterscheidet OOP unter folgenden zwei Gesichtspunkten:

1. *Reine* OOP => geringe kommerzielle Bedeutung. Beispiel: Smalltalk oder Eiffel
2. Angereicherte traditionelle Programmiersprachen, die um die objektorientierte Methode erweitert wurden.

Beispiel :
„C“ -> „C++“
Pascal -> Objected-Pascal

Delphi z.B. baut auf dem „Objected-Pascal“ auf. Es sind aber im Großen und Ganzen Erweiterungen von Borland-Pascal-Versionen und Turbo-Pascal-Versionen. Das Standard-Pascal von Wirth ist nur noch in (kleinen) Teilmengen enthalten.

Die 3 Phasen der Programmierung:

1. Phase
Finden des Lösungsverfahrens für das gegebene Problem

2. Phase
Formulieren des Programms, also Erstellen des Quellcodes

3. Phase
Übersetzen (compilieren) des Programmes und laufen lassen. Dabei muß zu erst auf korrekte Syntax (bei der Compilation) und dann auf inhaltliche Korrektheit durch testen geachtet werden. Möglicherweise ergeben sich dabei Wiederholungen der ersten und zweiten Phase.

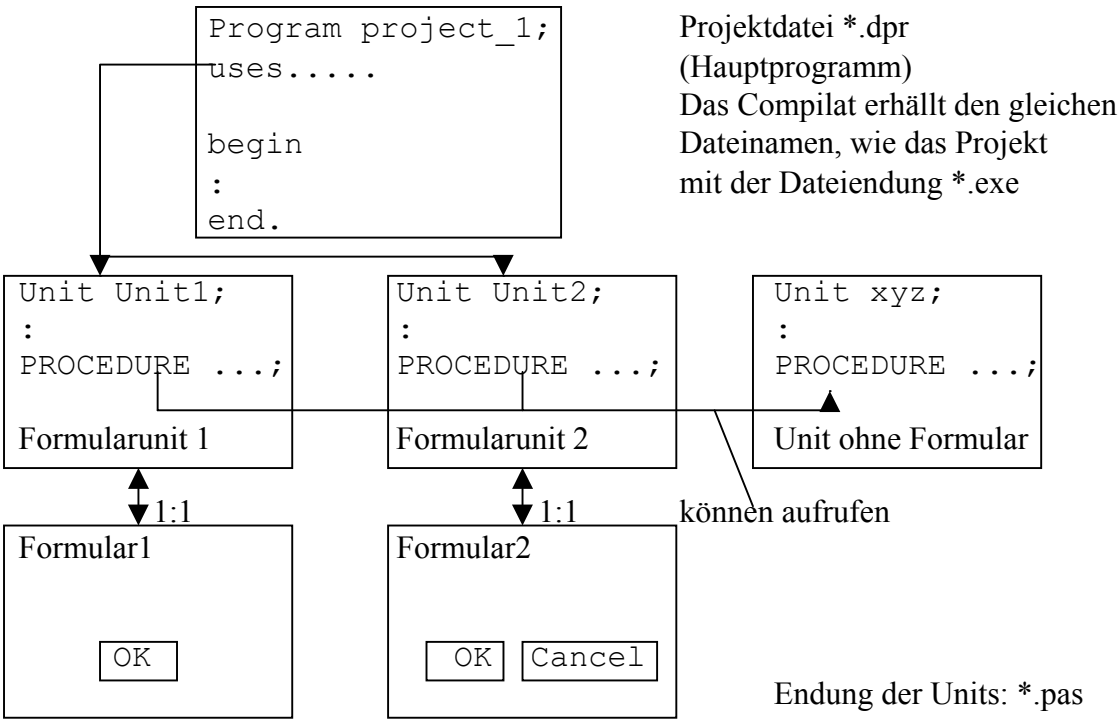
Beispiel der MwSt-Berechnung:
gegeben: Rechnungsbetrag Netto in DM
gesucht: MwSt-Betrag dieses Rechnungsbetrages in DM

zu 1.) Lösungsverfahren :
MWST_BETRAG:=(RECHNUNGSBETRAG*15)/100;
zu 2.) Frame bauen und Code eingeben.
zu 3.) Compilieren lassen und Fehler entfernen.

1. die integrierte Delphi-Entwicklungsumgebung

1.1 Formulare, Units und Programme

1.1.1 Die Struktur eines Delphi-Programmes



- zu jedem Formular gehört genau eine Formularunit
- es gibt auch Units ohne Formular, welche Prozeduren enthalten, die von anderen Units aufgerufen werden können

Aufbau einer Unit:

```
Unit xyz;
     :
TYPE
     :
CONST
     :
VAR
     :
IMPLEMENTATION
     :
PROCEDURE Tform1.ButtonClick
VAR
   ...
BEGIN
      (eigener Programmcode)
END;

PROCEDURE(s)
     :
FUNCTION(s)
     :
END.
```

Speichern von Programmen:

Datei→alles Speichern
3 Dateien sind wirklich nötig für eine Anwendung:
- *.pas (Code der Unit eines Formulars)
- *.dfm (Formular und Komponente)
- *.dpr (Projektdatei, die sozusagen alles zusammenhält)

Achtung:
Die Projektdatei darf nicht den gleichen Namen erhalten, wie die Unit.
Die EXE-Datei erhällt schließlich den Namen der Projektdatei.

1.1.2 Vereinbarungen, bzw Definition von Variablen

Jede Variable hat einen Namen und einen Typ:

```
PROCEDURE Test;
VAR
   Name1,Name2:Typ;
BEGIN
     :
END;
```

Elementare Datentypen:

integer	(ganze Zahlen)	4 Byte (32Bit)
smallint	(ganze Zahlen)	2 Byte (16Bit)
single	(Fließkommazahl)	4 Byte
double	(Fließkommazahl)	8 Byte
real	(Fließkommazahl)	6 Byte
char	(Zeichen)	1 Byte
string	(Zeichenkette)	Variable Länge (dynamisch)
string[x]	(Zeichenkette)	x+1 Byte
boolean	(logischer Wert)	1 Byte (TRUE/FALSE)

Namen von Variablen:

Beginnt mit einem Buchstaben, auf den andere Buchstaben, Ziffern oder der Unterstrich folgen können.
Groß- und Kleinschreibung wird **nicht** unterschieden.

Variablenvereinbarungen:

Beispiel:

a)
```
max,min:integer;
x,y,z:real;
Zeichen:char;
```

b)
```
i,j,k:integer;
x:single;
ok:boolean;
str1:string[6] (s.u.)
str2:string; „beliebige“ Länge (max.2GB)
```

5	H	a	l	l	o														

`string[5]` = 6 Byte;
aktuelle Länge mit : `length(str1);`

Variablen können nun Werte zugewiesen werden durch := (Zuweisungsoperator)

z.B. (zu a)
```
max:=17;
y:=237.01;
Zeichen:='a';
```

nicht erlaubt
```
max:=17.4;
Zeichen:=a;  falls a nicht Variable vom Typ char;
```

(zu b)
```
x:=19.8;
ok:=TRUE; ok:=FALSE;
str1:='Hallo';
str2:='hier bin ich';
```

in einer Unit findet man die Deklaration der Variablen zwischen PROCEDUREN und BEGIN der Prozedur.

1.1.3 Konstanten

Im *Konstanten-Definitionsteil* (vor VAR) können Namen für Konstanten und deren Werte angegeben werden.

A) Konstante *ohne* Typ

```
CONST Name1:Konstante1;
      Name2:Konstante2;
```

Die Werte können nie verändert werden. Sie werden wie #defines in C per Präprozessor eingesetzt (→ „suchen und ersetzen“), was auch der Grund für die Typenlosigkeit ist.

B) Konstante *mit* Typ

```
CONST Name1:Typ=Konstante1;
```

Die Werte können wieder verändert werden. Diese Konstantendeklaration dient nur der Deklaration einer Variable inclusive gleichzeitiger Initialisierung mit einem Wert. Man könnte auch alle nötigen Initialisierungen im Hauptprogramm erledigen, aber unter Delphi ist dieses Hauptprogramm nicht mehr zugänglich, da es im Projekt steht. Units haben im Gegensatz zu ADA keinen Initialisierungsteil.

1.1.4 Arithmetik und Standardfunktionen

Beispiel:

```
VAR  i,j,k:INTEGER;
     x,y,z:SINGLE;
BEGIN
    i:=1+2;
    j:=3+i;
    k:=i+j+5;

    x:=1.0+j;
    y:=3.0/i;
    z:=(i+k)/(3*i)
END;
```

Integerausdrücke

Singleausdrücke

Arithmetische Operatoren:

```
+ - * /
```

DIV Division ohne Rest
z.B. k:=7 div 3; => k=2;

MOD Rest einer Division
z.B. k:= 5 mod 2; => k=1;

Standardfunktionen:

```
abs(x)
cos(x)
```

Betragwert, Absolutwert von x oder $|x|$

`sin(x)`	
`exp(x)`	e^x
`sqr(x)`	x^2
`sqrt(x)`	$\sqrt{x}$
`ln(x)`	
`round(x)`	Kaufmännisches Runden (auf- und abrunden!)

Frage: Wie Rechne ich x^{75} aus?
Antwort: $x^y = e^{y \cdot \ln x} \left(= e^{\ln x^y} = x^y \right)$ und in Pascal:

```
exp(y*ln(x));
```

Beispiele für arithmetische Ausdrücke in Pascal:

mathematischer Ausdruck	**Pascal**
$\sqrt{a}$	`sqrt(a)`
$\frac{1}{\sqrt{a}}$	`1/sqrt(a)`
$\frac{x}{y \cdot z}$	`x/(y*z)`
$\frac{1}{a} + \frac{4}{2+a^2}$	`(1/a)+(4/(2+sqrt(a)))=` `1/a + 4/(2+sqrt(a))`

Hierarchie von arithmetischen Operatoren:

+, - als Vorzeichen	*hohe Priorität*
( )	
*, /	
+, -	*niedrige Priorität*

Beispiel:

```
x:=-3;
z:=1—3;
u:=1-+2;
A:=3*4+5;
B:=3*(4+5);              (A<>B)
```

Logische Ausdrücke:

Logikelemente haben einen Wert vom Typ Boolean (FALSE / TRUE)

Verknüpfungsoperatoren:

`NOT`	Negation
`AND`	und
`OR`	oder

`XOR` exclusives oder

mathematische Form	**Pascal**
$0 < x < 5$	`(0<x) AND (x<5)`
$\overline{A \vee B}$	`NOT (A OR B)`
$A \vee \overline{B}$	`A OR (NOT B)`

Logische Vergleichsoperatoren:

= <>(ungleich) < > <= >=

werden bei bedingten Anweisungen benutzt, also z.B.

```
IF a>b THEN c:=27;
IF (0<x) AND (x<5) THEN Label1.Text:='Hallo';
```

2.4 Anweisungen (Statements)

Anweisung: Aktionen, die das Programm asuführen soll.
Steuerung des Programmablaufes. (Linear (sequentiell), bedingt oder geschleift)

Gesamtübersicht von Anweisungen:

- Zuweisung :=
- Sprungbefehl `GOTO` (iiiiiiiiih!!!!!)
- zusammengesetzte Anweisung (`BEGIN...END;`)
- Prozedur- oder Funktionsaufrufe
- `IF-THEN-ELSE` } Bedingte Anweisung
- `CASE` }
- `WHILE...DO` } iterative Anweisungen (Schleifen)
- `REPEAT...UNTIL` }
- `FOR...DO` }
- `WITH` }
- Leere Anweisung

2.4.1 Zuweisungen:

Variable := Ausdruck
Ausdruck und Variable sollten dabei vom gleichen Typ sein. Variable muß auf der linken Seite eines Ausdruckes stehen dürfen, darf also z.B. keine typlose Konstante sein.

2. zusammengesetzte Anweisung:

Beschreibt die Technik, mehrere Anweisungen zu einer einzelnen Anweisung zusammen zu fassen. Wenn ein Anweisungsblock in `BEGIN...END`

eingefasst wird, wird er als Einheit und somit als einzelne Anweisung betrachtet.

Beispiel:

a)
```
BEGIN
     a:=5/2;
     b:=a/4
END;
```
kein Semikolon an dieser Stelle, da sonst eine leere Anweisung erzeugt wird!

- überall dort, wo nur eine Anweisung gestattet ist, kann man so mehrere unterbringen.
- Klammerung durch `BEGIN...END`. (Ausnahme hiervon ist `REPEAT...UNTIL`)
- Zusammengefasste Anweisungen können beliebig verschachtelt werden.

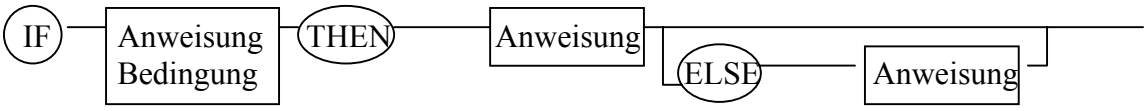
b) `BEGIN ;;;; END;` ergibt 4 leere Anweisungen!

c)
```
BEGIN
     IF a<2 THEN
     BEGIN
          x:=2;
          y:=5
     END;
     z:=4
END;
```
erster Block

zweiter Block

2.4.5 IF-Anweisung

Syntax:



Beispiel:

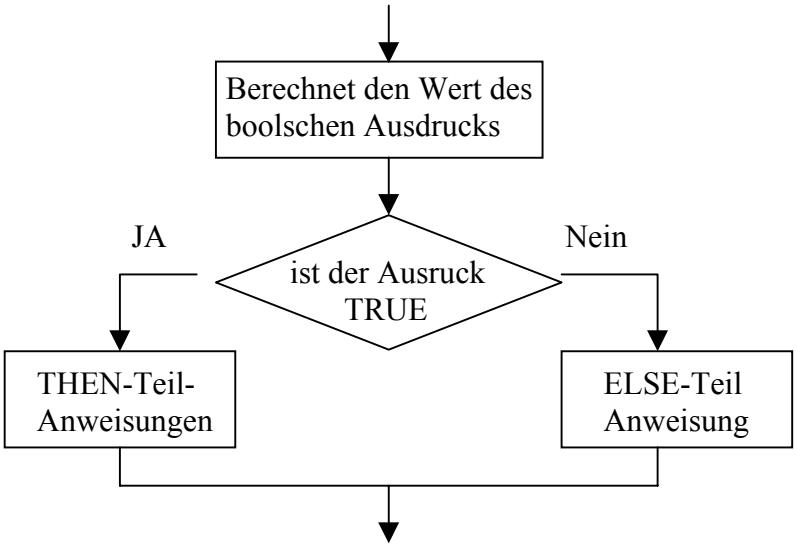
a)
```
IF zahl>0 THEN
     text:='positiv';
[ELSE
     text:='negativ';]
```

!!! **Vor ELSE aber kein Semikolon** !!!

b)
```
IF Name='Fritz' AND Nachname='Meyer' THEN
BEGIN
     x:=x+5;
     y:=z-3
END;
```

- Vor `ELSE` kein Semikolon!
- Der Ausdruck nach `IF` muß vom Typ boolean (also ein logischer Ausruck) sein. Bei TRUE wird die Anweisung hinter `THEN` ausgeführt, andernfalls die Anweisung hinter `ELSE`, sofern eines angegeben wurde.
- geschachtelte `IF` sollten vermieden werden, da möglicherweise Probleme bei der Zuordnung von `ELSE` oder Anweisungsblöcken auftreten

Flußdiagramm für die `IF` Anweisung:



Kommentare in Pascal:

// Text : Kommentar bis zum Zeilenende
{...} : Kommentar von bis (auch über mehrere Zeilen)
/*...*/ : wie oben.

2.4.4 Die FOR-Schleife:

a) Berechnung der Summe von 1-10

```
:
VAR i, Summe : Integer;
BEGIN
   Summe:=0; // Initialisierung!
   FOR i:=1 TO 10 DO
      Summe:=Summe+i
END;
```

i	Summe
1	1
2	3
3	6
:	
10	55

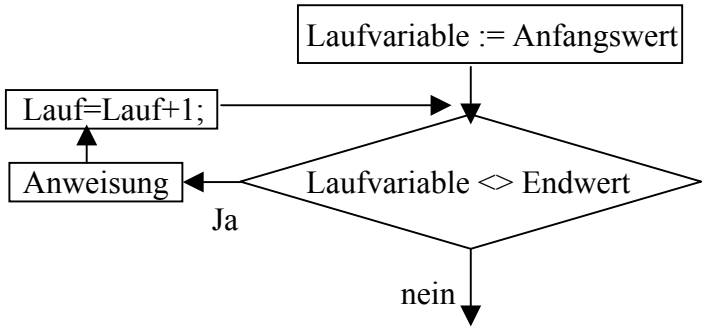
Alternativ ist `DOWNTO`:

```
FOR i:=10 DOWNTO 1 DO
```

b) Einlesen von 10 REAL-Zahlen und aufsummieren

```
:
VAR Summe,z : REAL;
    i       : Integer;
BEGIN
   Summe:=0; // Initialisierung!
   FOR i:=1 TO 10 DO
   BEGIN
      z:=StrToFloat(InputBox(´Eingabe´,IntToStr(i)+ ´. Zahl bitte:´));
      Summe:=Summe+z
   END;
   ShowMessage(´Der Wert ist: ´+FloatToStr(Summe));
END;
```

Das Flußdiagramm der FOR-Schleife:



2.4.2 Die WHILE-Schleife:

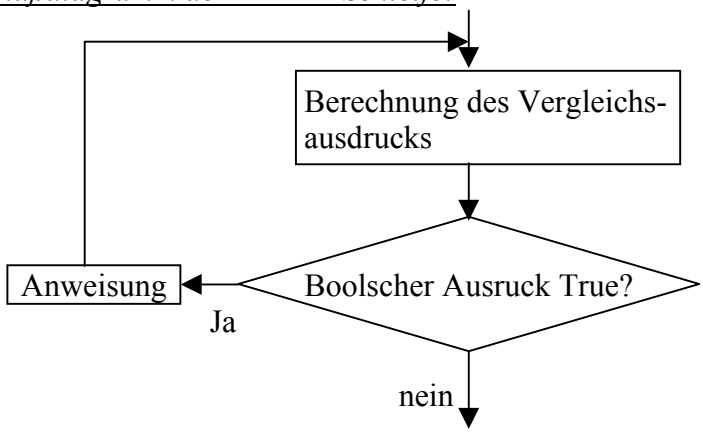
Beispiel:

```
a:=1;  //Typ soll Integer sein
WHILE a<=10 DO
   a:=a+1;
// hier ist a=11
```

oder auch, wenn nicht nur eine Anweisung verwendet werden soll:

```
a:=1;  //Typ soll Integer sein
WHILE a<=10 DO
BEGIN
   a:=a+1;
   b:=b+1
END;
// hier ist a=11
```

Das Flußdiagramm der `WHILE`-Schleife:



Die While-Schleife testet die Bedingung am Anfang der Anweisung. Der Rumpf wird also möglicherweise nie durchlaufen.
Das nennt man auch Kopfgesteuerte oder Abweisende Schleife.

2.4.3 `REPEAT-UNTIL`-Schleife:

Beispiel:

```
a:=1;  //Typ soll Integer sein
REPEAT
   a:=a+1
UNTIL a>10;
// hier ist a=11
```

oder auch, wenn nicht nur eine Anweisung verwendet werden soll:

```
a:=1;  //Typ soll Integer sein
REPEAT
   a:=a+1;
   b:=b+1
UNTIL a>10;
// hier ist a=11
```

Bei `REPEAT-UNTIL` benötigt man die genau negierte Bedingung, als bei einer vergleichbaren `WHILE`-Schleife.
Diese Schliefe ist Fußgesteuert oder nicht-abweisend.

Unterschiede `WHILE - REPEAT-UNTIL`

`WHILE`	`REPEAT-UNTIL`
Schleife wird so lange wiederholt, wie Bedingung erfüllt ist. Die Schleife ist aktiv, wenn die Bedingung `TRUE` ist. (Ausführungsbedingung)	Schleife wird so lange wiederholt, bis die Bedingung erfüllt ist. Die Schleife ist aktiv, wenn die Bedingung `FALSE` ist. (Abbruchsbedingung)
Bedingung wird am Anfang getestet. (Kopfgesteuert, abweisende Schleife)	Bedingung wird am Ende getestet, d.h. es findet genau ein Druchlauf statt. (Fußgesteuerte, nichtabweisende Schleife)

2.4.6 Die CASE Anweisung:

Die CASE-Anweisung ist mit der IF-Anweisung verwand und beide Konstrukte lassen sich gegeneinander austauschen, das eine ist dann immer etwas aufwendiger, als das andere.
Im Gegensatz zu IF gibt es bei CASE die Möglichkeit, eine unbegrenzte Anzahl von Verzweigungen zu schalten.

Beispiel:
Berechnung der Anzahl Tage eines Monats:

```
CASE Monat OF
   1,3,5,7,8,10,12 : AnzahlTage:=31;
   4,5,9,11        : Anzahl Tage:=30;
   2               : IF Jahr MOD 4 THEN
                        AnzahlTage:=29
                     ELSE
                        AnzahlTage:=28
END;
```

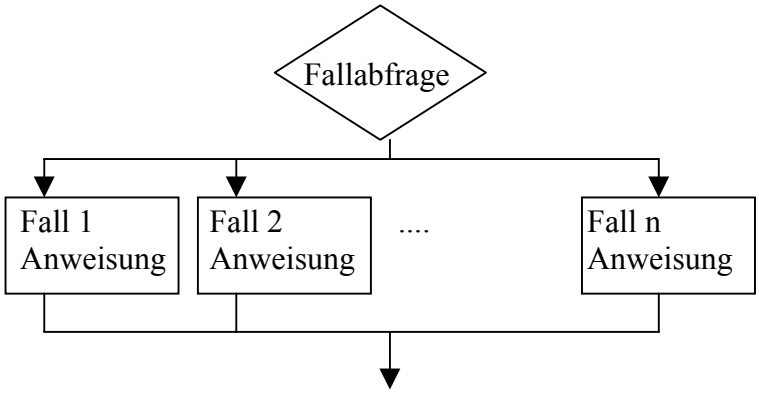
Die Variable „Monat“ muß von einem sog. „Ordinalen“ Typ sein. Unter diesen Typen versteht man Variablen, deren Vorgänger und Nachfolger bekannt sind. Also Daten von Typ Integer, char, boolean, oder selbstdefinierte Aufzählungstypen, da jedem bekannt ist, daß der Vorgänger von 2 = 1 ist und der Nachfolger ist 3. Real dagegen ist kein ordinaler Typ.

Beispiele

```
1)   VAR i:integer;
     :
     i:=(Zuweisung eines Wertes)
     :
     CASE i<20 OF
        TRUE : i:=i*10;
        FALSE: i:=i*2
     END;
```

```
2)   VAR c:char;
     :
     CASE c of
        ´0´..´9´ : Edit1.Text:=´c ist eine Ziffer´;
        ´a´..´z´ : Edit1.Text:=´c ist ein Kleinbuchstabe´;
        ´A´..´B´ : Edit1.Text:=´c ist ein Großbuchstabe´;
        ELSE     : Edit1.Text:=´c ist nichts´
     END;
```

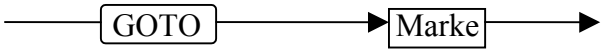
Das Flußdiagramm der CASE-Anweisung:



2.4.7 Die GOTO-Anweisung:

Normalerweise ist die Ausführungsreihenfolge von Anweisungen sequenziel entsprechend ihrer Anordung im Programmtext. Eine GOTO-Anweisung unterbricht diesen Anweisungstrom und springt an eine definierte Stelle (Label) im Programmtext.. Grundsätzlich gilt, daß jedes GOTO druch andere Konstrukte (REPEAT-UNTIL, DO-WHILE) ersetzbar ist. Manchmal ist aber gerade bei der Programmentwicklung das GOTO eine letzte und elegante Möglichkeit, bei Ausnahmen oder anderen spontanen Dingen einen Sprung auszuführen.

Synrax:



„Marke“ ist das namentliche Label, wobei in diesem Fall (anders als bei Variablen) Ziffern und Buchstaben gefolgt von einem Doppelpunkt zulässig sind.
Sie müssen mit dem Schlüsselwort „LABEL“ deklariert werden und können dann im Programmtext an die entsprechende Stelle mit einem Doppelpunkt plaziert werden.

Beispiel:

```
   VAR i:integer;
       r:real;
   LABEL 10, schluss;
            :
   IF r=10.0 THEN GOTO 10;
   IF r<0 THEN GOTO schluss;
            :
10:
   text:=´Hallo 10´;
            :
schluss:
   text:=´x<0´;
```

2.5 Datenstrukturen

2.5.1 Arrays und Strings

Die bisher einfachen Datenstrukturen (Integer, Real, Boolean, Char,...) werden wir nun um den Datentyp Array erweitern. Unter einem Array versteht man die Zusammenfassung mehrer Daten eines Typs in einer Variablen. Arrays werden auch als Felder bezeichnet. Es liegt ferner der Vergleich „Vektor“ oder „Matrix“ nahe, ist aber nicht gebräuchlich.

Beispiel:

```
VAR ZahlenArray : ARRAY[1..10] OF integer;
```

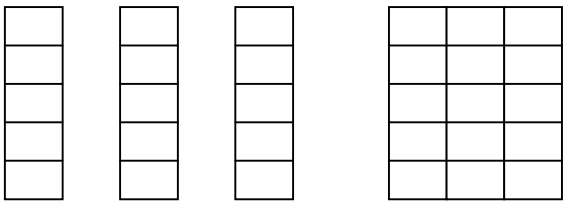
Diese Deklaration stellt die Variable ZahlenArray als zehnelementiges Feld zur Verfügung, auf deren einzelne Elemente über den Index zugegriffen werden können:
ZahlenArray[1], ZahlenArray[2]...
Arrays können wie gewöhnliche Variablen des gleichen Typs wie die Einzelelemente verwendet werden. Ebenso kann als Index auch eine Variable vom Typ Integer dienen. Dies stellt auch die häufigste Anwendung dar:

```
VAR z:ARRAY[1..100] OF real; // 100 Realvariablen...
    i:integer;               // unser Index
:
FOR i:=1 TO 100 DO
   z[i]:=i*3;
```

Mit einem Array deklariert man also eine Anzahl Variablen gleichen Typs, die unter einem einzigen Namen ansprechbar sind und über den Index unterschieden werden. Man kann sich ein Array auch als einen Stapel Variablen vorstellen.

2.5.2 Mehrdimensionale Arrays

Beispiel:
Die Zusammenfassung von 3 Spalten zu einem 2-Dimensionalen Array:




```
VAR m:Array[1..4, 1..5] OF real;
```

Benutzung z.B. `m[2,3]:=7,4;`
wobei der 1. Index = der Zeile und der 2. Index = der Spalte ist.

oder auch `m[4,3]:=m[2,1]+m[1,2];`

Auch mehrdimensionale Arrays kann man wie ganz normale Variablen behandeln.

Beispiel für Deklarationen:

```
VAR  Zahlen:ARRAY[5..10] OF Integer;
     Word:ARRAY[1..10] OF Char;
     Tabelle:ARRAY[1..5,1..10] OF Real;
```

Zugriff auf die Komponente:

```
     Zahlen[6]:=27;
     Word[9]:='A';
     IF Tabelle[5,7]>3,85 THEN
```

In Standard Pascal gab es damals noch nicht den Datentyp STRING. Daher mußte man sich dort (genau wie auch noch heute bei C) mit einem Array[0..X] OF Char (wobei X<256) behelfen. Man kann sich nun streiten, bei welcher Methode man mehr Freiheiten hat, bzw. sicherer Programmieren kann.

Der Datentyp String:

Zeichenketten bestimmter Länge, verwandt mit Arrays.
Deklaration:

`String[Konst.]` $0 \leq$ Konst. ≤ 255

Beispiele:

```
VAR  Zeile:String[80];
     s,t:String[25];
     Wort:String[10];
```

Konkadinieren von String mit +, z.B. ´Hallo´ + ´_Du´ oder s + t, concat(s, t);

Arrays OF Char sind nicht das gleiche, wie der Datentyp String, dennoch kann man einen String als ein solches Array behandeln.
Beispiel:

```
Name[1]:='W';
```

Läßt man die Längenangabe bei dem Datentyp String weg, so erhält man bei Turbo Pascal einen String mit maximaler (255) Länge, bei Delphi dagegen erhält man einen Dynamischen String, der bis zu 2GByte Daten aufnehmen könnte.

2.7 Prozeduren + Funktionen

Prozeduren und Funktionen geben Anweisungsfolgen einen Namen. Mit diesem Namen werden die Anweisungsfolgen aufgerufen. Ein Programm kann dann in Teilen (Modulen) zerlegt werden.

Beispiel:

a) Ohne Prozedur!

```
Program Demo1;
Var x,y: Integer;

Begin
   :
  x:=5; y:=10;
   :
  x:=x-1; y:=y-1;
   :
  x:=x-1; y:=y-1;
   :
End;
```

b) Mit Prozedur!

```
Program Demo2;
Var x,y: Integer;

PROCEDURE subtract;
Begin
   x:=x-1; y:=y-1
End;

Begin
   :
  x:=5; y:=10;
   :
  subtract;
   :
  subtract;
   :
End;
```

Übergabe von Parameter an Prozeduren

Obiges Beispiel funktioniert nur mit den Variablen x und y. Will man andere Variablen (Vor allem nicht global deklarierte) verwenden, so muß man diese als Parameter an die Prozedur übergeben. In der Deklaration im Procedure-Kopf nennt man die Variablen „Formale Parameter“, beim Aufruf dann nennt man sie „aktuelle Parameter“.
Dabei unterscheidet man den Call-By-Reference und den Call-By-Value.

Beispiel:

```
Program demo3;
VAR x,y,q,w:Integer;
                        nackter Wert          Adresse (reference)
PROCEDURE subtract (a:integer; VAR b:Integer);
Begin
   a:=a+1;
   b:=b+1
End;

Begin
   x:=5;   y:=10;   q:=1;   w:=4;
   :
   subtract(x, y);
   :
   subtract(q,w);
   :
End;
```

b ist Referenzparameter (VAR b). y wird an den Parameter b der Prozedur übergeben. Am Prozedurende wird der Wert von b (jetzt 9) zurück an das hauptprogramm übergeben. (y hat jetzt auch den Wert 9)

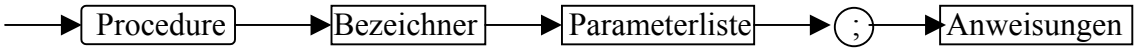
a ist Wertparameter. Beim Aufruf wird eine temporäre Kopie von x erzeugt. Änderungen am Parameter haben nur Auswirkungen innerhalb der Prozedur. Der aktuelle Wert außerhalb der Prozedur bleibt unverändert. Am Ende hat x noch den ursprünglichen Wert (5).

Die Übergabe von Parametern an eine Prozedur bewerkstelligen Hochsprachen über den Stapel (auch Stack oder Keller genannt), den wir bei Assemblerunterprogrammen bereits kennengelernt haben. Es handelt sich dabei um einen Speicher, der mit einem Zeiger (Pointer) auf das zuletzt eingestellte Element ausgestattet ist. Praktisch wird immer das zuletzt abgelegte Element als erstes wieder ausgelesen. Auf diesem Stapel wird bei einem Unterprogrammaufruf (und nichts anderes stellt ein Prozeduraufruf dar!) die Rücksprungadresse abgelegt und in Hochsprachen dazu auch noch die Parameter. Dabei wird bei einem Call-By-Value nur der Wert der Variablen auf den Stapel gebracht, bei einem Call-By-Reference nur die Adresse der Variablen, die den Wert enthällt und vom gleichen Typ sein *muß*, wie der Prozedur-Parameter. Dadurch wird klar, warum Manipulationen an einem Call-By-Reference-Parameter sofort die Variable im Aufrufprogramm mit verändert. Bei einem Call-By-Value dagegen hat niemand auch nur eine entfernte Chance, den Variableninhalt zu verändern, da der Ursprung nicht bekannt ist. Ob es sich bei einer Zahl auf dem Stapel um eine Adresse

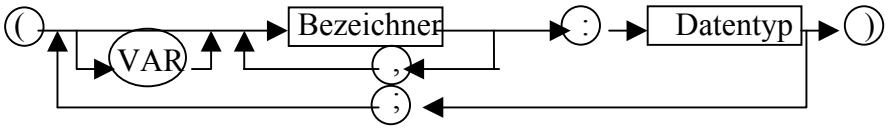
oder um einen Wert handeln soll, legt übrigens die Deklaration der Prozedur fest. Daher ist es auch entscheidend wichtig, Prozeduren vor ihrem Aufruf immer zu deklarieren, also ihren Aufruf festzulegen.

Allgemein: Änderungen in einer Prozedur an Wert-Parametern haben nur innerhalb der Prozedur auswirkungen. Änderungen in einer Prozedur an einem Reference-Parameter haben auch Auswirkungen auf das Hauptprogramm.

Syntax der Prozedurndeklaration



Die Parameterliste:



Anmerkungen zu Prozeduren:

1.) SQR, SQRT, Sin, Cos,... sind Standardfunktionen. Daneben existieren auch Standardprozeduren, welche dann zum Pascal-System gehören und nicht Bestandteil des Compilers selbst sind (wie die Schlüsselworte => IF, WHILE, ...), sondern stehen ein einer Standardbibliothek, d.h. einer Sammlung von Prozeduren und Funktionen, die mitgeliefert wurden.
Nach dem compilieren werden sie vom „Linker“ (oder auch Binder) in den Maschienencode eingebunden.
Prozeduren wie IntToStr, InputBox,--- stehen in eigenen Prozedur-Bibliotheken, die sich dem Programmierern in Form der Units zeigen. [Schlüsselwort „USES“]

2.) Jede Prozedur ist prinzipell ein eigenes programm mit eigenen Variablen. Sie steht nur mit den Übergabeparametern in Kontakt mit der Außenwelt (auch genannt: Schnittstelle, Interface)

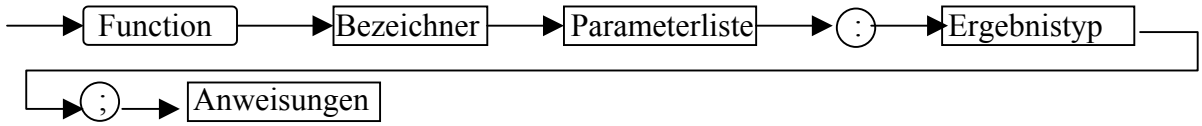
3.) Mittels Prozeduren und Funktionen können alle eigenständigen Teilprobleme getrennt von einander Programmiert werden, in Bibliotheken abgelegt und bei Bedarf benutzt werden.

2.8 Die Funktion

Eine Prozedur, die nur einen Ausgangswert hat, kann man als Funktion auffassen. Eine Funktion ordnet einem (oderer mehreren) Eingangswert(en) *einen* Ausgangswert zu. (wie in der Mathematik). Z.B. die Standard-Pascal-Funktionen: SQRT, Sin, Cos...
Der Funktionsname stehtbei dem Aufruf für den Funktionswert.

Einziger Unterschied von Funktionen zu Prozeduren:
Eine Funktion gibt mit ihrem Namen einen bestimmten Wert zurück, was Prozeduren nicht tun.

Syntax der Funktionsdeklaration



Die Parameterliste entspricht der bei Prozeduren.

Beispiel 1:

```
(*Diese Funktion funktioniert ähnlich dem DIV-Befehl, *)
(*doch wird hier auf null abgetestet und in flag das  *)
(*Ergebnis gespeicher.                                *)
FUNCTION DIVIDE(a,b:integer; VAR flag:BOOLEAN):INTEGER;
BEGIN
   if b=0 THEN
   BEGIN
     flag:=FALSE;
     DIVIDE=0
   END
   else
   BEGIN
      flag:=TRUE;
      DIVIDE:=a DIV b
   END
END;
```

Beispiel 2:

```
(*Diese Funktion ist eine Mischung aus DIV und MOD *)
FUNCTION DIVMOD(a,b:integer; VAR rest:integer):INTEGER;
   help:integer;
BEGIN
   help:=a DIV b;
   rest:=a-(help*b);
   DIVIDE:=help
END;
```

weitere Beispiele:

1.
```
PROCEDURE quadrat(x:real; var x2:real);
BEGIN
   x2:=x*x
END;
```

Aufruf:
```
quadrat(5,x2);
IF x2>25 THEN ...
```

2.
```
FUNCTION quadrat(x:real):real;
BEGIN
   quadrat:=x*x
END;
```

Aufruf:
```
IF quadrat(5)>25 THEN...
```

Wird das Ergebnis einer Funktion häufig benötigt, so ist es natürlich zweckmäßig, das Ergebnis in einer Variablen zu speichern, um es ohne weitere Funktionsaufrufe zur Verfügung zu haben. Schließlich kann ein Funktionsaufruf eine nicht unwesentliche Zeitspanne in Anspruch nehmen.

Die Rückgabe des Funktionswertes ermöglicht eine Zuweisung des Wertes an den Funktionsnamen, der nur im Rahmen dieser Zuweisung wie eine Variable behandelt werden kann. Man sollte damit nie innerhalb der Funktion rechnen, sondern statt dessen lieber eine Hilfsvariable deklarieren, deren Inhalt dann vor dem Rücksprung aus der Funktion an den Funktionsnamen via Zuweisung übergeben wird.

Bei Funktionen *muß* eine Zuweisung eines typgerechten Wertes an den Funktionsnamen erfolgen, bevor die Funktion verlassen wird.

Vorteile der Funktion gegenüber einer Prozedur:

- Übersichtlichkeit
- Man spart Variablen
- Man kann sie in mathematischen Ausrücken direkt verwenden

Fazit: Die Lösung als Funktion ist vorzuziehen, wenn aus den Eingangswerten nur ein Ausgangswert produziert wird.

Exkurs Rekursion:

Das Programm P36 verwendet Rekursion, um die Fakultät zu berechnen. Wie man in dem Programm erkennt, handelt es sich bei der Rekursion um einen wiederholten Aufruf der Funktion (oder auch Prozedur). Diese wiederholten Aufrufe sind nicht in allen Hochsprachen möglich, jedoch in Pascal, C, ADA und vielen anderen schon.

Die Rekursion (Gegenstück: Iteration) beitet sich immer dann an, wenn die gesamte Funktion (oder Prozedur) als Lösung für ein Teilstück, wie auch für das ganze Problem steht. Die Faktultätsberechnung z.B. wiederholt sich als Aufgabe für jedes Element. Die Fakultät von 5 ist eben 5*4!=5*4*3! und so weiter.

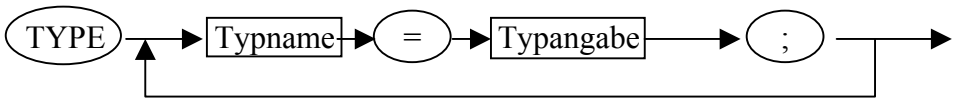
Es gibt gerade im Bereich der Binären Bäume weitere Gebiete, wo sich die Rekursion als Programmierstil für einen Algorithmus anbietet. Man kann grundsätzlich sagen, daß sich jedes Problem sowohl iterativ, wie auch rekursiv lösen läßt. Eine der beiden Möglichkeiten ist dabei jeweils die Bessere.

2.9 Strukturierte Datentypen und Typdeklarationen

Strukturierte Datentypen sind:

- Strings
- Arrays
- Rekords
- Sets
- File-Typ

Typdefinition eigener Datentypen:



Eine Typangabe kann sein: einfacher Typ (integer, real)
strukturierter Typ (Array)
Zeigertypen

Beispiele:

```
Type Tagestyp=1..31;
     namenstyp=string[30];
     Tabelle=Array[1..5,1..3] OF real;
     Buchstabe='A'..'Z';
     Intervall=-100..100;
     Temparaturscala=-20..40

VAR  Tag:Tagestyp;
     Vname,Nname:namenstyp;
     Matrix:Tabelle;
     MessWerte:Array[Temparaturscala] OF integer;
     Zeichenzahl:Array[Buchstabe] OF integer
:
BEGIN
:
   IF Tag>5 THEN...
   x:='B';
   Vname:='Hans';
   MessWerte[23]:=5;
   Zeichenzähler['X']:=1;
:
END.
```

***Records** (Verbundtyp)*

Bei einem Record handelt es sich um einen Datentyp, der aus einer festen Anzahl Komponenten, evnt. unterschiedlichen Typs besteht.

Beispiel:

```
Type datum = RECORD
                TAG:1..31;
                Monat:1..12;
                Jahr:0..99;
             END;
     Person = RECORD
                 Name:String[20];
                 Vorname:String[15];
              END;
     VAR
       p, Kunde:person;
     Bestelldatum, Lieferdatum:Datum;
```

Damit hat man definiert:

```
p: name [....................] und
   vorname [...............]
```

also zwei Komponenten, die unter der Variablen p zusammengefasst sind.

```
Bestelldatum: Tag [mit Werten von 1 - 31]
              Monat [mit Werten von 1 - 12]
              Jahr [mit Werten von 0 - 99]
```

Die Zuweisung:

Auf die Elemente eines solchen Verbundes greift man mit Hilfe der deklarierten Verbundvariable zu. Um die Elemente adressieren zu können, schreibt man sie hinter die Verbundvariable mit einem Punkt getrennt:

```
p.Name:='Hose';
p.Vorname:='Kord';
Kunde.Name:='Schultze & Schulze';
IF Lieferdatum.Jahr>98 THEN ...
Bestelldatum.Monat:=3;
```

Sollte man nun sehr häufig auf eine Verbundvariable zugreifen müssen, da man z.B. alle Elemente initialisieren will, so kann man sich dem `WITH..DO` bedienen:

```
WITH p DO
BEGIN
   Name:='Meier';
   Vorname:='Hans'
END;
```

Selbstverständlich kann ich einen Record auch als Array deklarieren:

```
VAR p:ARRAY[1..10] OF Name;
```

oder

```
VAR p:ARRAY[1..10] OF RECORD
                        Name,Vorname:String[80];
                        Steuerklasse:CHAR;
                        Steuerschuld:REAL;
                        :
                      END;
```

Und mit dem Zugriff:

```
p[1].Name:='Nicht-Ich';
p[1].Vorname:='Horst';
p[1].Steuerklasse:=3;
p[1].Steuerschuld:=37465981.83;
```