FB 20 Informatik TU Berlin
SYS Fachgebiet Systemanalyse und EDV EDV

Skript zur Lehrveranstaltung EDV I/II

PASCALVS - Kurzlexikon

Shirley Oei

Eitel v. Maur

1. Auflage 1992

Sekr. FR 6-7
Tel. 314-73260

1000 Berlin 10
Franklinstraße 28/29

Inhaltsverzeichnis

HINWEIS: Alle durch @ gekennzeichneten und kursiv gedruckten Stichworte stellen Verweise auf andere Einträge dar. Bei Bedarf kann also unter diesen Stichworten nachgeschlagen werden.

Anweisung

Eine Anweisung stellt einen Befehl innerhalb eines Programmes dar. Man unterscheidet *@Steueranweisungen* und *@elementare Anweisungen*.

Anweisung, elementare

Eine elementare Anweisung ist eine *@Anweisung*, welche keine *@Steueranweisung* darstellt. Elementare Anweisungen sind *@Zuweisungen*, Prozeduraufrufe, *@Dateioperationen*, *@Ein-/Ausgabeanweisungen* etc.

Beispiele:

```
WRITELN ('Hallo');
betrag := 1.50;
RESET (kundendatei);
```

Array

Eine Array ist ein Feld einer bestimmten Länge, deren Elemente alle Werte eines bestimmten Typs annehmen können. Man kann sich ein Array vorstellen als eine Kommode mit durchnumerierten Schubladen (erste Schublade hat die Nummer Anfangswert, die letzte Endwert - s.u.), welche alle nur jeweils eine Sache desselben Typs aufnehmen dürfen.

Allgemein (Eindimensionale Bereich):

```
ARRAY (. anfangswert .. endwert .) OF Datentyp;
```

oder mit *@Typkonzept*:

```
TYPE BEREICHSTYP = ARRAY (. anfangswert..endwert .) OF
                   Datentyp;

VAR bereich : BEREICHSTYP;
```

Beispiel:

```
TYPE VEKTORTYP = ARRAY (. 1..4 .) OF INTEGER;
VAR vektor : VEKTORTYP;
```

Allgemein (Zweidimensionale Bereiche):

```
ARRAY (.anfangswert..endwert.) OF
    ARRAY (.anfangswert..endwert.) OF Datentyp;
```

vereinfacht:

```
ARRAY (.anfangswert..endwert,anfangswert..endwert.)
    OF Datentyp;
```

oder mit *@Typkonzept*:

```
TYPE MATRIXTYP =   ARRAY (.anfangswert..endwert,
                   anfangswert..endwert.) OF Datentyp;

VAR matrix : MATRIXTYP;
```

Beispiel:

```
TYPE MATRIXTYP = ARRAY (.1..4,1..5.) OF Datentyp;
VAR matix : MATRIXTYP;
```

Arrays können auch in mehr als 2 Dimensionen vereinbart werden; die Dimensionierung erfolgt analog zur Dimensionierung der zweidimensionalen Bereiche.

Datei

Eine Datei (engl.: FILE) ist eine Datensammlung auf einem externen Speicher. Sie kann unstrukturiert als Text (z.B. Programmdateien, EXECs etc.) oder strukturiert (Sammlung von Zahlen, Buchstaben oder *@Datensätzen*) abgespeichert sein. Dateien können mit *@Dateioperationen* manipuliert werden, welche ausführlich im PASCALVS-Erweiterungsscript beschrieben sind.

Dateioperationen

Es sind die folgenden Dateioperationen möglich:

Öffnen der Datei zum Lesen:
```
RESET (dateivariable);
```

Öffnen der Datei zum Schreiben:
```
REWRITE (dateivariable);
```

Öffnen der Datei zum Lesen und Schreiben (nur bei gestreut-direkter DO):
```
UPDATE (dateivariable);
```

Einlesen aus der Datei in Datensatzvariable:
```
READ (dateivariable, datensatzvariable)
```

Schreiben aus der Datensatzvariable in die Datei:
```
WRITE (dateivariable, datensatzvariable)
```

Positionieren des Datensatzzeigers (nur bei gestreut-direkter DO):
```
SEEK (dateivariable, satznummer);
```

Schließen einer geöffneten Datei:
```
CLOSE (dateivariable);
```

Datensatz

Element einer *@Datei*. Siehe auch *@RECORD*.

Datentypen

Man unterscheidet *vordefinierte* und *selbstdefinierte*, einfache *(skalare)* und *nichtskalare Datentypen*.

1. Einfache (skalare) Datentypen

Einfache (skalare) Datentypen in PASCAL sind:

INTEGER, REAL, BOOLEAN, CHAR, (STRING) - s. Anmerkung unter STRING.

1.1 BOOLEAN

Datentyp, welcher nur zwei Elemente enthält: Die beiden konstanten Wahrheitswerte TRUE und FALSE.

Beispiel:

Für eine Deklaration

```
VAR wahrheit : BOOLEAN;
```

sind folgende *@Zuweisungen* möglich:

```
wahrheit := TRUE;
wahrheit := FALSE;
wahrheit := zahl <= 19;
```

1.2 CHAR

Datentyp in PASCAL. Enthält alle Buchstaben und Sonderzeichen wie Semikolon, Punkt, Leerzeichen etc.

Beispiel:

Für eine *@Deklaration von Variablen*

```
VAR aphanum : CHAR;
```

sind beispielsweise folgende *@Zuweisungen* möglich:

```
aphanum := 'f';
aphanum := 'F';
aphanum := ';';
aphanum := ' ';
aphanum := '4';
```

Man beachte, daß in der letzten *@Zuweisung* keine Zahl zugewiesen wird, sondern ein alphanumerischer Wert, mit welchem keine arithmetischen Operationen ausgeführt werden können.

Anmerkung: Auf unserer Rechenanlage werden CHARs bei der Eingabe automatisch in Großbuchstaben konvertiert, falls ein Buchstabe eingegeben wurde.

1.3 INTEGER

Datentyp in PASCAL. Enthält nur ganze Zahlen.

Beispiel:

Für eine *@Deklaration*

```
VAR zahl : INTEGER;
```

sind beispielsweise folgende *@Zuweisungen* möglich:

```
zahl := 0;
zahl := -12345;
zahl := 7;
```

1.4 REAL

Datentyp in PASCAL. Enthält alle *INTEGER* und alle Zahlen mit Nachkommastellen. In PascalVS ist *REAL* kein einfacher Datentyp i. e. S..

Beispiel:

Für eine Deklaration

```
VAR zahl : REAL;
```

sind beispielsweise folgende *@Zuweisungen* möglich:

```
zahl := 0.5;
zahl := -1.8905;
zahl := 3;
zahl := 3.0;
```

Die letzten beiden *@Zuweisungen* bewirken dasselbe.

1.5 STRING

Datentyp in PascalVS und in Turbo-Pascal (nicht in Standardpascal). Enthält alle alphanumerische Zeichenketten der Länge n, wobei n in der *@Deklaration* angegeben werden muß. STRINGs werden intern auf *@ARRAY* abgebildet, somit ist der Datentyp STRING eigentlich kein skalarer Datentyp. Der Einfachheit halber soll er im Rahmen dieser Lehrveranstaltung jedoch als skalar angenommen werden.

Beispiel:

Für eine *@Deklaration*

```
VAR name : STRING(10);
```

sind folgende *@Zuweisungen* möglich:

```
name := 'Meier';
name := '';
name := 'u';
name := 'Freiherr Graf Fritz von Kreuzberg';
```

Anmerkungen: Die *@Variable* name enthält nach der zweiten *@Zuweisung* den leeren *String*, nach der vierten den Wert 'Freiherr G', weil name maximal 10 Zeichen (siehe Deklaration der Variablen name) aufnehmen kann.
Auf unserer Rechenanlage werden *Strings* bei der

Eingabe automatisch in Großbuchstaben konvertiert.

2. Nichtskalare Datentypen

Nichtskalare (komplexe) Datentypen in PASCAL sind (mit beliebiger Schachtelung):

@ARRAY, @RECORD, @FILE.

Weitere Datentypen in PASCAL wie Set, Aufzählungstypen und Pointer sollen hier nicht betrachtet werden.

3. Selbstdefinierte Datentypen

Selbstdefinierte Datentypen sind alle Datentypen, die in der *@Deklaration von Typen* vereinbart wurden. Sie ermöglichen eine Strukturierung der verwendeten Objekte, vereinfachen die Übergabe von Objekten als Unterprogrammparameter und verbessern die Lesbarkeit und Änderbarkeit von Programmen.

4. Vordefinierte Datentypen

Vordefinierte Datentypen in PASCAL sind sämtliche *@skalare Datentypen* und *nichtskalare Datentypen*

Deklaration von Konstanten

Konstanten werden mit

```
CONST KONSTANTENNAME = wert;
```

vereinbart.

Beispiel:

```
CONST
   PROZENTMEHRWERTSTEUER = 0.14;
   PI = 3.14;
```

So sind dann *@Zuweisungen* möglich wie die folgende:

```
mehrwertsteuer := betrag * PROZENTMEHRWERTSTEUER;
```

Konstanten werden -wie alle *@Schlüsselworte* und Typenbezeichner auch- groß geschrieben.

Deklaration von Typen

Typen können vom Programmierer definiert und benannt werden.

Allgemein:

```
TYPE SELBSTDEFINIERTERTYPNAME = Datentyp;
```

Beispiel:

```
TYPE NAMENSTYP =  RECORD
                    vorname   : STRING (20);
                    nachname  : STRING (20)
                  END;

     ADRESSENTYP = RECORD
                     strasse  : STRING (20);
                     plz      : STRING (6);
                     ort      : STRING (20)
                   END;
```

```
KUNDENTYP = RECORD
              name       : NAMENSTYP;
              adresse    : ADRESSENTYP;
              stammkunde : BOOLEAN
            END;
KUNDENDATEITYP = FILE OF KUNDENTYP;
```

Zu dieser Deklaration von Typen können dann bei der *@Deklaration von Variablen* entsprechende Objekt angelegt werden.

Beispiel:

```
VAR kunde       : KUNDENTYP;
    kundendatei : KUNDENDATEITYP;
```

Deklaration von Variablen

Eine *@Variable* wird deklariert durch

```
VAR variablenbezeichner : Datentyp;
```

Beispiel:

```
VAR rechnungsbetrag : REAL;
```

Werden mehrere *@Variablen* deklariert, braucht das *@Schlüsselwort* VAR trotzdem nur einmal verwendet werden:

```
VAR
   variablenbezeichner1 : Datentyp1;
   variablenbezeichner2 : Datentyp2;
   ...
   variablenbezeichnern : Datentypn;
```

Mehrere *@Variablen* des gleichen *@Datentypes* können -durch Kommatas getrennt- aufgezählt werden mit anschließender Nennung des *@Datentyps*:

```
VAR
   variablenbezeichner1,
   variablenbezeichner2,
   ...
   variablenbezeichnern : Datentyp;
```

Deklaration von Unterprogrammen

@Unterprogramm. Achtung: Wird im Unterprogrammrumpf eines Unterprogrammes x ein anderes Unterprogramm y aufgerufen, so muß y textuell vor x deklaraiert worden sein.

Deklarationsteil

Der Deklarationsteil besteht -in der Reihenfolge- aus der *@Deklaration von Konstanten*, der *@Deklaration von Typen*, der *@Deklaration von Variablen* und schließlich der *@Deklaration von Unterprogrammen.*

Durchgangsparameter

Durchgangsparameter werden in der Parameterliste eines *@Unterprogrammes* mit VAR gekennzeichnet. Sie werden ggf. mit neuen Werten an die aufrufende Umgebung zurückgeliefert.

Beispiel:

```
PROCEDURE add (    zahl1,
                   zahl2 : INTEGER;
               VAR summe : INTEGER);
BEGIN
  summe := zahl1 + zahl2
END; /* add */
```

Beispiel eines Aufrufs:

```
ergebnis := 0;
add (3,5,ergebnis);
WRITELN (ergebnis);
```

Auf dem Bildschirm wird erwartungsgemäß die Zahl 8 ausgegeben. Ohne das *@Schlüsselwort* VAR, welches nicht zu verwechseln ist mit dem VAR aus der *@Deklaration von Variablen,* wäre ergebnis ein *@Eingabeparameter* gewesen; ausgegeben würde in diesem Fall die Zahl 0.

Ein- / Ausgabeanweisungen

Eingabeanweisungen in PascalVS sind READ und READLN. Unter VM/SP ist jedoch das READLN leider nicht sinnvoll möglich. Deshalb soll im Rahmen dieser Lehrveranstaltung ausschließlich READ verwendet werden. Auch darf als letzte Schreibanweisung (WRITE / WRITELN) vor einem READ kein WRITE stehen.

Beispiel:

```
READ (zeichen);
```

Unter VM/SP gibt es noch weitere Probleme beim Einlesen über die Tastatur. Vor dem Lesen muß immer die Anweisung RESET(INPUT) abgesetzt werden:

```
RESET (INPUT);
READ (zeichen);
```

Ausgabeanweisungen in PascalVS sind WRITE und WRITELN.

Beispiel:

```
WRITE (zeichen);

WRITELN (zeichen);
```

WRITE macht nach der Ausgabe -im Gegensatz zu WRITELN- keinen Zeilenvorschub. Siehe auch *@Formatieren der Ausgabe in PASCALVS.*

Eingangsparameter

Eingangsparameter werden in der Parameterliste eines *@Unterprogrammes* vereinbart. Auch wenn im Rumpf des jeweiligen *@Unterprogrammes* die Werte von Eingangsparametern modifiziert werden, so werden sie trotzdem nicht an die aufrufende Umgebung zurückgeliefert.

Beispiel:

```
PROCEDURE quatsch (    zahl1,
                       zahl2 : INTEGER;
                   VAR zahl3 : INTEGER);
BEGIN
  zahl1 := 1;
  zahl1 := 2;
  zahl1 := 3
END; /* add */
```

Beispiel eines Aufrufs:

```
x := 0;
y := 0;
z := 0;
quatsch (x,y,z);
WRITELN (x,y,z);
```

Auf dem Bildschirm werden die Werte 0,0,3 ausgegeben, da lediglich der Wert von zahl3 (als *@Durchgangsparameter*) an z zurückgeliefert wird.

Entscheidung

Die Entscheidung ist eine *@Steueranweisung*, bei welcher die *@Anweisung* von einer Bedingung abhängt. Man unterscheidet in Pascal die Alternativanweisung, die bedingte Anweisung und das CASE-Konstrukt.

1. Anweisung, bedingte

Eine *@Anweisung* wird nur dann ausgeführt, wenn eine Bedingung erfüllt ist.

Allgemein:

```
IF bedingung
  THEN anweisung;
```

Beispiel:

```
IF zahl = 1
  THEN WRITELN ('Die Zahl besitzt den Wert 1');
```

Man beachte, daß die bedingt auszuführende *@Anweisung* eine *@Sequenz* darstellt, wenn sie aus mehreren logisch zusammengehörenden Unteranweisungen besteht. Sie muß in diesem Fall mit `BEGIN` und `END` geklammert werden:

```
IF bedingung
  THEN
   BEGIN
     anweisung1;
     anweisung2;
     ...;
     anweisungn
   END;
```

2. Alternativanweisung

Eine erste *@Anweisung* wird nur dann ausgeführt, wenn eine Bedingung erfüllt ist; sonst wird eine zweite *@Anweisung* ausgeführt..

Allgemein:

```
IF bedingung
  THEN anweisung1
  ELSE anweisung2;
```

Beispiel:

```
IF zahl = 1
  THEN WRITELN ('Die Zahl besitzt den Wert 1')
  ELSE WRITELN ('Die Zahl ist nicht 1');
```

Noch ein Beispiel:

```
IF zahl = 1
  THEN WRITELN ('Die Zahl besitzt den Wert 1')
  ELSE
    IF zahl = 2
      THEN WRITELN ('Die Zahl besitzt den Wert 2')
      ELSE WRITELN ('Die Zahl ist nicht 1 oder 2');
```

Man beachte, daß vor dem ELSE kein Semikolon als *@Folgeoperator* stehen darf.

ACHTUNG:

Ist die im THEN-Teil einer unbedingten Anweisung eine *@bedingte Anweisung*, so muß sie durch BEGIN und END geklammert werden. Man mache sich klar, worin der Unterschied zwischen den beiden folgenden Anweisungen besteht (Was wird ausgegeben, wenn zahl den Wert -5 besitzt?):

```
IF zahl < 1
  THEN
   BEGIN
     IF zahl = 0
       THEN WRITELN ('Die Zahl besitzt den Wert 0')
   END
  ELSE WRITELN ('Die Zahl ist groesser als 0');
```

und

```
IF zahl < 1
  THEN
   IF zahl = 0
     THEN WRITELN ('Die Zahl besitzt den Wert 0')
     ELSE WRITELN ('Die Zahl ist kleiner als 0');
```

Zur Vertiefung siehe das Kontrollstrukturenskript.

3. CASE - Konstrukt

Es wird diejenige Anweisung ausgeführt, bei der ein bestimmter Wert dem Wert einer Variablen entspricht, welche nach dem Schlüsselwort CASE angegegeben ist. Allgemein (Man beachte, daß die linke CASE-Anweisung äquivalent zur rechten ist - siehe Beispiel unten):

```
CASE variable OF
  wert1 : anweisung1;
  wert2 : anweisung2;
  wert3 : anweisung3;
  ....
  wertn : anweisungn
  OTHERWISE
    anweisungx
END; /* CASE */
```



```
IF variable = wert1
 THEN anweisung1
 ELSE
  IF variable = wert2
   THEN anweisung2
   ELSE
    IF ...
          ELSE
           IF variable = wertn
             THEN anweisungn
             ELSE anweisungx;
```

Beispiel:

```
CASE eingabe OF
  1 : WRITELN ('eingabe hat den Wert 1.');
  2 : WRITELN ('eingabe hat den Wert 2.');
  3 : WRITELN ('eingabe hat den Wert 3.');
  4 : WRITELN ('eingabe hat den Wert 4.');
  5 : WRITELN ('eingabe hat den Wert 5.');
  OTHERWISE
    WRITELN ('eingabe nicht zwischen 1 und 5.')
END; /* CASE */
```

Angenommen, in oberem Beispiel hat die Variable eingabe den Wert 2. Es wird nun in der ersten Alternative der CASE-Anweisung abgefragt, ob eingabe den Wert 1 besitzt. Da das nicht der Fall ist, wird die zweite Alternative abgetestet: Hat eingabe den Wert 2? Da das der Fall ist, wird die Anweisung `WRITELN ('eingabe hat den Wert 2.');` ausgeführt. Die restlichen Alternativen 3,4,5 bzw. OTHERWISE werden dann nicht mehr weiter überprüft, sondern der Programmlauf nach dem `END; /* CASE */` fortgesetzt. Hätte eingabe den Wert 6, so würden in obigem Beispiel alle Alternativen durchlaufen und dann die OTHERWISE-Alternative `WRITELN ('eingabe nicht zwischen 1 und 5.')` ausgeführt.

Es können bei CASE-Anweisungen auch Intervalle für Werte angegeben werden:

```
CASE kaufbetrag OF
     0..  100 : rabattsatz = 0;
   101.. 1000 : rabattsatz = 10;
  1001.. 5000 : BEGIN
                  WRITELN ('Rabattsatz = 20% ');
                  rabattsatz = 20
                END /* 1001..5000 */
  5001..10000 : rabattsatz = 30
  OTHERWISE
    WRITELN ('Bitte zum Geschäftsführer!')
END; /* CASE */
```

Anmerkung:

Bei dem CASE-Konstrukt muß das abgefragte Kriterium ein einfacher (skalarer) Typ sein. In PASCALVS kann es sogar nur entweder vom Typ INTEGER, CHAR oder BOOLEAN sein; REAL ist also nicht funktionsfähig. Abgetestet kann der Wert der angegebenen Variablen entweder auf Gleichheit oder auf die Zugehörigkeit zu einem Intervall. Boolsche Abfragen wie "variable< = 20" bzw. booolsche Verknüpfungen mit AND oder OR sind also nicht erlaubt. Intervallangaben sind durch zwei Punkte .. darstellbar. OTHERWISE ist optional, wogegen END zwingend zur Syntax gehört. OTHERWISE sollte jedoch in keiner CASE-Anweisung fehlen, denn besitzt die abgetestete Variable einen Wert, welcher durch keine Alternaltive innerhalb der CASE-Anweisung abgedeckt wird, so gibt es einen Laufzeitfehler.

Beispiel:
```
CASE kaufbetrag OF
     0..  100 : rabattsatz = 0;
   101.. 1000 : rabattsatz = 10;
  5001..10000 : rabattsatz = 30
 END; /* CASE */
```

Ist in diesem Fall kaufbetrag = 15000, dann ist keine Alternative gegeben und es gibt eien Laufzeitfehler.

EXEC

Ein EXEC (auch batch-file genannt) besteht aus einer Ansammlung, in der Regel auch separat ablauffähiger Befehle des Betriebssystemes. Unserer Anlage läuft unter dem Betriebsystem VM/SP - somit können in einem EXEC CMS-Kommandos zu einem benutzerdefiniertem, quasi neuem Befehl zusammengefaßt werden.

Beispiel:
```
TYPE 112A3PRG PASCAL
TYPE 112A3PRG LISTING
```

Werden diese beiden Befehle in eine EXEC-Datei geschrieben (z.B. AUFBILD EXEC), so kann man sie lediglich über die Eingabe des Dateinamens (in diesem Beispiel AUFBILD) starten (näheres s.u.).

EXECs kann man mithilfe einiger PASCAL-ähnlichen Steueranweisungen genauso wie in einer höheren Programmiersprache 'programmieren'. Der Aufruf des EXEC's in der CMS-Ebene durch Eingabe des filenamen (nur dort kann ein EXEC aufgerufen werden) bewirkt, daß die im EXEC angegeben Befehle nacheinander ausgeführt werden.
Werden aus einem EXEC heraus EXEC's aufgerufen, was durchaus üblich ist, so muß das Wort "EXEC" vor den filenamen geschrieben werden.

Beispiel:

```
EXEC PASCCOMP 112A3PRG
```

Man kann ein EXEC als ein aus Betriebssystembefehlen bestehendes Programm bezeichnen.

Erstellung und Start eines EXEC's

Ein EXEC wird wie ein PASCAL Programm in der XEDIT - bzw. INPUT-Ebene erstellt. In der CMS-Ebene muß der Editor mit folgendem Kommando aufgerufen werden:

```
XEDIT filename EXEC.
```

Als 'filetype' wird also immer 'EXEC' angegeben. Wenn das EXEC auf dem Storus abgespeichert werden soll, muß der 'filename' immer den im Tutorium besprochenen Namenskonventionen entsprechen.

Ein EXEC muß nicht übersetzt werden! Es kann sofort nach dem Verlassen des Editors und der Rückkehr in die CMS-Ebene ausgeführt werden. Das Betriebssystem kennt alle im EXEC verwendeten Befehle und übersetzt diese 'intern', ohne daß man sich als Anwender darum kümmern muß.

Gestartet wird ein EXEC durch eingeben des EXEC-Namens (filename) in der CMS-Ebene.

Eine Auswahl, für PASCALVS notwendige Befehle.

PASCCOMP Kompilert ausschließlich Programme der Sprache PASCAL. Daher muß nur der Name des Programmes angegeben werden, die Angabe des Dateitypen 'PASCAL' erübrigt sich.

PASCRUN Startet durch PASCCOMP übersetzte PASCAL-Programme. Sollen mit einem Programm ausgelagerte SEGMENTE gleichzeitig gestartet werden, so muß der filename des Segmentes hinter dem des aufrufenden Programmes stehen.

Allgemein: PASCRUN f n des Programmes f n des Segmentes

Beispiel: PASCRUN 112A3PRG 112A3SEG

FILEDEF Verbindet einen physischen Dateinamen unter VM/SP mit dem logischen Dateinamen im Pascal-Programm (*@Programmkopf*)

Abkürzung: FI

Optional kann im FILEDEF *d*urch XTENT m *(extension)* bei gestreut-direkten Dateitypen die Anzahl der Datensätze (m) angegeben werden, wenn mehr als 50 Datensätze angelegt werden sollen. Sonst kann man über den Dateischlüssel nur auf die Satznummern 0 - 49 zugreifen.

Bei sequentiellen Dateien kann außerdem noch die Option DISP MOD angegeben werden, wenn die neu geschriebenen Sätze an bereits eingegebene Sätze angehängt werden sollen. Andernfalls (was manchmal jedoch gewünscht sein kann) werden alle zuvor eingegebenen Sätze gelöscht!

Beispiel:

```
FI BOOTEDATEI DISK 99BOOTE DATA (DISP MOD
```

Anmerkung: Man beachte, daß die Klammer für die Optionen nicht wieder geschlossen werden darf.

Dateitypen (filetypen) in CMS

PASCAL	Pascal-Proggramm
LISTING	Datei, die beim Kompilieren eines PASCALVS Programmes. In dieser Datei steht der Programmcode mit den Compilerfehlern als Randbemerkungen. Geeignet um bei vielen Compilerfehlern dieArbeit zu erleichetern.
TEXT	Das in Maschinensprache übersetzte PASCALVS Programm.
MAP	Datei, die den Pfad zur Verknüpfung zwischen *@Segment* und *@Hauptprogramm* angibt.
DATA	Datei, welche eine strukturierte Sammlng von Datensätzen enthält.
CONSPROT	Konsolenprotokoll
EXEC	Ansammlung von Betriebssystembefehlen unter CMS.

Beispiel:

```
FI KUNDENDATEI DISK 99KUNDEN DATA (DISP MOD XTENT 100
FI BOOTEDATEI DISK 99BOOTE DATA
EXEC PASCCOMP 028A4PRG
EXEC PASCCOMP 028A4SEG
EXEC PASCRUN 028A4PRG 028A4SEG
```

Folgeoperator

Der Folgeoperator bezeichnet das Ende einer *@Anweisung*. In Pascal ist der Folgeoperator das Semikolon.

Formatieren der Ausgabe in PascalVS

Die Ausgabe von *@Variablen* unterschiedlicher *@Datentypen* kann formatiert erfolgen. Es kann linksbündig oder rechtsbündig in einem bestimmten Bildschirmspaltenbereich ausgegeben werden.

Sei n die Anzahl der zu verwenden Bildschirmspalten. Wird dem n ein Minuszeichen vorangestellt, so erfolgt die Ausgabe in diesem Bildschirmspaltenbereich linksbündig, sonst rechtsbündig.

Man muß unterscheiden:

Ausgabe von STRINGs

Falls n kleiner als die Anzahl der Zeichen der Zeichenkette, so werden nur die ersten n Zeichen der Zeichenkette ausgegeben.

Ausgabe von numerischen Werten

Bei der Ausgabe ganzer Zahlen oder reeller Zahlen wird die Ausgabe vom Laufzeitsystem ggf. korrigiert - die auszugebende Zahl wird vollständig auf dem Bildschirm ausgegeben.

Beispiel:

Der Datentyp STRING : "WRITELN(zeichen : n);"

```
WRITELN ('Hallo': -8);
```

Als Ausgabe:

```
Hallo
```

```
WRITELN ('Hallo': 8);
```

Als Ausgabe:

```
...Hallo
```

(die ersten drei Stellen enthalten Leerzeichen)

```
WRITELN ('Hallo': 3);
```

Als Ausgabe:

Hal

Der Datentyp INTEGER: "WRITELN(zahl : n);"

```
zahl := 15;
WRITELN (zahl : -5);
```

Als Ausgabe:

15... (drei Leerzeichen nach der 15)

```
WRITELN (zahl : 5);
```

Als Ausgabe:

...15 (die ersten drei Stellen enthalten Leerzeichen)

```
WRITELN (zahl : 1);
```

Als Ausgabe:

15

Ist n wie in diesem Beispiel kleiner als die Länge der Ziffer, so wird die Ziffer trotzdem vollständig ausgegeben. Hier ist der Unterschied zum STING, bei dem alle Zeichen > n abgeschnitten werden.

Der Datentyp REAL: "WRITELN(zahl : n1 : n2);"

n1 = Anzahl der Bildschirmspalte. Davon n2 Spalten für die Nachkommastellen und eine für den Dezimalpunkt und u. U. eine für die Kennzeichnung, daß es sich um einen negativen Wert handelt (-).

```
zahl := 200;
WRITELN (zahl : -8 : 2);
```

Als Ausgabe:

```
200.00
```

```
WRITELN (zahl : 8 : 2);
```

Als Ausgabe:

```
..200.00
```

(die ersten beiden Stellen enthalten Leerzeichen)

Funktion

Eine Funktion ist ein *@Unterprogramm*, welches genau ein skalares Ergebnis (nur einfache *@Datentypen*) liefert; daraus folgt, daß Funktionsdeklarationen keine *@Durchgangsparameter* enthalten dürfen (vgl. *@Prozedur*).

Eine Funktion beginnt mit einem Funktionskopf. Der Funktionskopf besteht wiederum aus dem *@Schlüsselwort* FUNCTION, dem Funktionsnamen, ggf. aus einer Liste von *@Eingangsparametern,* einem Doppelpunkt und dem *@Datentyp* des Funktionswertes:

```
FUNCTION name (parameter : DATENTYP1) : DATENTYP2;
```

Dem Funktionskopf folgt ggf. eine *@Deklaration von lokalen Konstanten*, eine *@Deklaration von lokalen Typen* und eine *@Deklaration von lokalen Variablen,* welche dann lokal in der Funktion gelten.

Nach diesen Deklarationen folgt eine *@Sequenz*, welche den Funktionsrunpf darstellt. Innerhalb dieser Sequenz wird dem Funktionsnamen der Funktionswert zugewiesen.

Beispiel (vgl. *@Prozedur*):

```
FUNCTION fakultaet (wert : INTEGER) : INTEGER;
VAR zaehler,
    ergebnis : INTEGER;
BEGIN
  ergebnis := 1;
  FOR zaehler := 1 to wert DO
     ergebnis := ergebnis * zaehler;
  fakultaet := ergebnis
END; /* fakultaet */
```

Beispiel eines Aufrufes (vgl. *@Prozedur*):

```
x := fakultaet (fakultaet (3));
```

Hauptprogramm

Das Hauptprogramm stellt die Hauptsequenz des Programmes dar. In dieser *@Sequenz* sollten bestenfalls nur Unterprogrammaufrufe stehen. Nach dem letzten END steht in *@PASCAL-Programmen* kein Semikolon als *@Folgeoperator*, sondern ein Punkt, welcher das Programmende kennzeichnet.

Konstante

Eine Konstante ist -im Gegensatz zu einer *@Variablen*- ein unveränderlicher Wert.

Beispiel:

```
zahl := 1;
```

Die *@Variable* zahl erhält den Wert der Konstanten 1.

Konstanten können auch durch die *@Deklaration von Konstanten* benannt werden.

Parameter

Parameter ermöglichen *@Unterprogrammen* die Kommunikation mit der aufrufenden Umgebung (z.B. das *@Hauptprogramm*). Man unterscheidet *@Durchgangsparameter* und *@Eingangsparameter*, *@formale* und *@aktuelle Parameter*.

Parameter, aktuelle

Aktuelle Parameter sind diejenigen Parameter, welche beim Aufruf eines *@Unterprogrammes* in der aufgeführten Reihenfolge an die entsprechenden *@formalen Parameter* übergeben werden. Die Zahl der aktuellen Parameter muß gleich der Anzahl der *@formalen Parameter* sein. Ihre Zuordnung erfolgt über die Anordnung der Reihenfolge. Der jeweilige Datentyp von *@formalem Parameter und* aktuellem Parameter muß identisch sein.

Beispiel: (Dazugehörige formale Parameter siehe bei Beispiel in *@formale Parameter*)

```
fakultaet (3,erg);
```

3 und erg sind aktuelle Parameter.

Parameter, formale

Formale Parameter sind diejenigen *@Parameter*, welche bei der *@Deklaration eines Unterprogrammes* im *@Prozedurkopf* in Form einer Liste angegeben werden.

Beispiel: (beispielshafte aktuelle Parameter siehe bei Beispiel in *@aktuelle Parameter*)

```
PROCEDURE fakultaet (    wert     : INTEGER;
                     VAR ergebnis : INTEGER);
```

wert und ergebnis sind formale Parameter.

PASCAL-Programm

Ein PASCAL-Programm besteht aus dem *@Programmkopf*, dem *@Deklarationsteil* und dem *@Hauptprogramm*.

Programmkopf

In PASCAL sieht ein Programmkopf wie folgt aus:

```
PROGRAM progname (INPUT, OUTPUT, datei1, ..., datei n);
```

Alle verwendeten *@Dateien* müssen in der geklammerten Liste aufgezählt werden. Werden keine *@Dateien* benötigt, so enthält die Liste nur INPUT für die Tastatur und OUTPUT für den Bildschirm.

Prozedur

Eine Prozedur ist ein *@Unterprogramm*. Sie beginnt mit einem *Prozedurkopf*. Der Prozedurkopf besteht wiederum aus dem *@Schlüsselwort* PROCEDURE, dem Prozedurnamen und ggf. aus einer Liste von *@Parametern*.

Den Deklarationen folgt eine Sequenz, welche den Prozedurrumpf darstellt.

Beispiel (vgl. *@Funktion*):

```
PROCEDURE fakultaet (    wert     : INTEGER;
                     VAR ergebnis : INTEGER);
VAR zaehler : INTEGER;
BEGIN
  ergebnis := 1;
  FOR zaehler := 1 to wert DO
     ergebnis := ergebnis * zaehler
END; /* fakultaet */
```

Beispiel eines Aufrufes (vgl. *@Funktion*):

```
fakultaet (3, x);
```

RECORD

Ein Record ist eine *@Datenstruktur* in Pascal, mit welcher logisch zusammengehörende Eigenschaften eines Objektes zusammengefaßt werden können.

Beispiel:

```
TYPE NAMENSTYP =  RECORD
                    vorname    : STRING (20);
                    nachname   : STRING (20)
                  END;

     ADRESSENTYP = RECORD
                     strasse   : STRING (20);
                     plz       : STRING (6);
                     ort       : STRING (20)
                   END;

     KUNDENTYP = RECORD
                   name        : NAMENSTYP;
                   adresse     : ADRESSENTYP;
                   stammkunde  : BOOLEAN
                 END;

VAR kunde : KUNDENTYP;
```

Auf die einzelnen Attribute eines Record-Objektes kann über den sogenannten Qualifikator (in Pascal der Punkt) zugegriffen werden. Bei der Qualifikation wird die Satzvariable durch einen Punkt von seinen Komponenten getrennt.

Allgemein:

```
satzvariable.attribut1 := ausdruck;
satzvariable.attribut2 := ausdruck;
```

Für das obige Beispiel:

```
kunde.name.vorname := 'Hans';
kunde.name.nachname := 'Meier.';
kunde.adresse.strasse := 'Holweg 3';
kunde.adresse.ort := 'Berlin';
```

etc.

Schleife

Eine Schleife (engl. loop) ist eine *@Steueranweisung*, in welcher eine andere *@Anweisung* wiederholt ausgeführt werden kann. Man unterscheidet ***Precheck-Loops*** und ***Postcheck-Loops***.

1. Precheck-Loops

Precheck-Loops werden in zwei verschiedene Schleifenarten differenziert: WHILE- und Zählschleifen.

1.1 WHILE-Schleife

Solange eine Bedingung erfüllt ist, soll eine *@Anweisung* wiederholt werden:

```
WHILE bedingung DO
   anweisung;
```

Beispiel

```
betrag := 0;
WHILE NOT betrag > 1000 DO
 BEGIN
   RESET (input);
   READ (betrag);
   alter_betrag := betrag + alter_betrag
 END;

/* Beachte die Notwendigkeit der Initialisierung! */
```

Beispiel: Ausgabe der Kundennamen, welche in einer Kundendatei gespeichert sind. Die Kundensätze sind vom Typ KUNDENTYP, welches als Beispiel bei *@RECORD* angegeben ist:

```
RESET (kundendatei);
WHILE NOT (EOF (kundendatei)) DO
 BEGIN
   READ (kundendatei, kundensatz);
   WITH kundensatz DO
     WRITELN (name.vorname, name.nachname)
 END; /* WHILE */
CLOSE (kundendatei);
```

Der Schleifenrumpf einer WHILE-Schleife wird minimal 0, maximal unendlich oft abgearbeitet. Damit die Schleife terminiert, muß im Schleifenrumpf dafür gesorgt werden, daß die Schleifenbedingung irgendwann einmal erfüllt wird.

1.2. Zählschleife

Eine Zählschleife ist eine *Precheck-Loop*, deren Schleifenrumpf genau n mal ausgeführt wird. n ist abhängig vom Anfangswert und Endwert der Schleifenbedingung:

n = Anfangswert - Endwert + 1 für Anfangswert > = Endwert;
sonst n = 0 (Man kann einer Schleife nicht -x mal durchlaufen).

Allgemein:

```
FOR zählvariable := anfangswert TO endwert DO
 anweisung;
```

Die Zählvariable wird automatisch nach jedem Durchlauf um 1 erhöht.

Beispiel: Ausgabe der Zahlen 1234 bis 4321:

```
FOR zaehler := 1234 TO 4321 DO
  WRITELN (zaehler);
```

ACHTUNG: Auf die Zählvariable darf innerhalb des Schleifenrumpfes nur lesend zugegriffen werden, d.h. eine *@Zuweisung*

```
zaehlvariable := ausdruck;
```

ist nicht erlaubt. Die Zählvariable muß deklariert werden, ist jedoch nur innerhalb der Zählschleife gültig, d.h. die Zählvariable ist nach Abarbeitung der Schleife undefiniert.

2. Postcheck-Loop

Eine *@Anweisung* soll solange wiederholt werden, bis eine Bedingung erfüllt wird:

```
REPEAT
  anweisung
UNTIL bedingung
```

Beispiel:

```
REPEAT
  WRITELN ('Bitte J oder N eingeben:');
  RESET (INPUT);
  READ (antwort);
  eingabe_ok :=(antwort = 'J') OR (antwort = 'N');
  IF NOT (eingabe_ok)
    THEN WRITELN ('Bitte nur J oder N eingeben...')
UNTIL eingabe_ok;
```

Hinweis: REPEAT - UNTIL wirkt wie eine Klammer; eine *@Sequenz* muß deshalb ausnahmsweise nicht mit BEGIN - END geklammert werden, wenn sie als Schleifenrumpf einer Postcheck-Loop fungiert.

Der Schleifenrumpf einer REPEAT-Schleife wird minimal 1 mal, maximal unendlich oft abgearbeitet. Damit die Schleife terminiert, muß im Schleifenrumpf dafür gesorgt werden, daß die Schleifenbedingung irgendwann einmal erfüllt wird.

Schlüsselwort

Schlüsselworte werden groß geschrieben. Sie sind reserviert und dürfen deshalb nicht zur Bezeichnug von Variablen, Prozeduren, Typen etc. bennutzt werden!

Segment

Ein Segment stellt eine Sammlung von 1 bis n Unterprogrammen dar. Diese Unterprogramme können von einem "normalen" Pascalprogramm aus aufgerufen werden, wenn bestimmte Bedingungen (s.u.) erfüllt sind. Da ein Segment lediglich eine Sammlung von Unterprogrammen darstellt, ist ein Segment alleine nicht lauffähig.

Segmente kann man sinnbildlich mit Bibliotheken vergleichen: Ein Segment könnte praktisch alle Unterprogramme beinhalten, welche mit Dateiverarbeitung zu tun haben. Ein anderes Segment hingegen könnte alle

Unterprogramme versammelt haben, welche im Aufgabenbereich Auftragsannahme arbeiten sollen.

Es ist möglich, von beliebigen Pascal-Programm auf Segmente zuzugreifen. Durch dieses Konstrukt wird bei Großprojekten eine Arbeitsteilung ermöglicht. Das Segment kann wie gewöhnliche Pascal-Programme compiliert werden.

Aufbau des Segments

a) Segmentkopf:

Statt wie im Pascal-Programm bisher üblich lautet die Kopfzeile

Allgemein: SEGMENT segmentname;
Beispiel: SEGMENT berechnen;

b) Deklaration im Segment:

- Typen, die im Segment benutzt werden, müssen zuerst deklariert werden.

  Beispiel:

```
TYPE KUNDENTYP = RECORD
                   nummer  : INTEGER;
                   name    : STRING(10);
                   adresse : STRING(10)
                 END; /* KUNDENTYP */
```

- Dateien, mit denen im Segment gearbeitet wird, müssen mit STATIC deklariert werden.

  Beispiel:

```
STATIC kundendatei : KUNDENDATEITYP;
```

c) Kommunikation Pascal-Programm mit Segment:

- Wie oben gesagt, ist es unter bestimmten Bedingungen möglich, von einem Pascal-Programm aus auf die gesammelten Unterprogramme eines Segmentes zuzugreifen. Diese Bedingungen sind die folgenden:
  1. Das Segment muß bekannt geben, welche der im Segment gesammelten Unterprogramme von Pascal-Programmen überhaupt benutzt werden dürfen.

2. Das Pascal-Programm muß bekannt geben, welche Unterprogramme es aus einem Segment benutzen möchte.

zu 1.: Die Prozeduren, die von einem Segment für Pascal-Programme zur Verfügung gestellt werden sollen, müssen in diesem Segment gesondert gekennzeichnet werden. Dies erfolgt, indem man nach dem Prozedurennamen die verwendeten Parameter auflistet und darunter ein EXTERNAL setzt. Dadurch wird erreicht, daß diese Prozeduren aus dem Segment nach außen für beliebige Pascal-Programme exportiert werden.

Beispiel:

```
PROCEDURE einlesen (VAR kunde : KUNDENTYP);
EXTERNAL;
```

Es folgen alle benötigten Prozeduren und Funktionen, zunächst diejenigen, die innerhalb der zu exportierenden Prozedur verwandt werden. Sie bleiben jedoch nach außen unsichtbar, man kann sie im Grunde als lokale Prozeduren bezeichnen. Bzgl. der Reihenfolge gilt das unter Prozeduren gesagte (*@Prozedur*). Schließlich wird die bereits oben als zu exportierende gekennzeichnete Prozedur implementiert. Dabei wird beim Prozedurenkopf nur noch der Name der Prozedur genannt. Der Rumpf des Segments wird durch ein END; abgeschlossen, ein Hauptprogramm gibt es nicht (Segmente sind ja auch nicht alleine lauffähig).

Beispiel:

```
PROCEDURE  einlesen;
BEGIN
  WRITELN ('Bitte die Kundennummer eingeben!');
  RESET (INPUT);
  READ (kunde.nummer);

  WRITELN ('Bitte den Kundennamen eingeben!');
  RESET (INPUT);
  READ (kunde.name);

  WRITELN (Bitte die Kundenadresse eingeben!');
```

```
    RESET (INPUT);
    READ (kunde.adresse)
END; /* Segment */
```

zu 2.: **Aufbau der Pascal-Programme, welche Unterprogramme aus einem Segment verwenden wollen.**

Wenn ein Pascal-Programm ein Unterprogramm aus einem Segment benutzen möchte, so muß es dieses zunächst einmal bekannt geben. Dies geschieht analog zur Kennzeichnung dieser Prozedur im Segment mit EXTERNAL.

Beispiel: Angenommen, ein Pascal-Programm möchte die obige Segment-Prozedur einlesen benutzen. Dazu muß es zuerst bekannt geben, daß es diese Prozedur verwenden möchte:

```
PROCEDURE einlesen (VAR kunde : KUNDENTYP);
EXTERNAL;
```

Nun steht dem Pascal-Programm die Prozedur einlesen zur Verfügung, ohne daß jene Prozedur in diesem Pascal-Programm implementiert werden muß (... denn das ist sie ja schon im Segment...). Diese importierte Prozedur kann nun im Programm genauso aufgerufen werden wie bisher üblich, d.h. durch Angabe des Prozedurennamens und der verwendeten Variablen (*@aktuellen Parametern*).

Starten des Programms, bei dem importierte Prozeduren verwandt werden

Dem System muß bei der Verwendung von den als EXTERNAL gekennzeichneten Unterprogrammen bekannt gegeben werden, in welchen Segmenten diese Unterprogramme zu finden sind.

PASCRUN *fn des Programmes fn des Segmentes*

Beispiel: Angenommen, die Prozedur einlesen ist im Segment *123a4seg PASCAL* definiert und das Pascal-Programm *123a4hau PASCAL* möchte es verwenden. Der Start erfolgt dann mit (natürlich nach erfolgreicher Compilierung der beiden Teile):

```
PASCRUN 123a4hau 123a4seg
```

Sequenz

Eine Sequenz ist eine Folge von *@Anweisungen*, welche logisch zusammen gehören und deshalb alle abgearbeitet werden sollen. Man spricht deshalb bei einer Sequenz auch von einem Block.

Allgemein:

```
BEGIN
  Anweisung1;
  Anweisung2;
  ...
  Anweisungn
END;
```

Beispiel:

```
BEGIN
  WRITELN ('Bitte zahl eingeben:');
  RESET (INPUT);
  READ (zahl)
END;
```

Man beachte, daß eine Sequenz selbst eine *@Anweisung* (in diesem Fall eine Steueranweisung) darstellt.

Steueranweisungen

In Pascal gibt es -wie in anderen algorithmischen Programmiersprachen auch- drei verschiedene Arten von Steueranweisungen, mit welchen der Programmablauf gesteuert werden kann: *@Sequenz, @Entscheidung, @Schleife*.

Unterprogramm

Unterprogramme in Pascal sind *@Prozeduren* und *@Funktionen*. Sie können intern oder extern deklariert sein. (*@Segment*)

Variable

Eine Variable stellt einen Platzhalter dar, welcher verschiedene Werte aus einem bestimmten Wertebereich annehmen kann. Der Wertebereich muß als *@Datentyp* bei der *@Deklaration von Variablen* angegeben werden. Variablen werden in PASCAL klein geschrieben.

Man unterscheidet lokale und globale Variablen in Bezug auf die Unterprogrammtechnik. Eine Variable ist lokal in einem *@Unterprogramm*, wenn sie in diesem *@Unterprogramm* deklariert wurde; sonst ist sie global. Die Verwendung von globalen Variablen ist fehleranfällig und zeugt von einem schlechten Programmierstil; aus diesem Grunde sollen in dieser Lehrveranstaltung in Unterprogrammen keine globalen Variablen benutzt werden; im Hauptprogramm können natürlich nur globale Variablen verwendet werden.

Zuweisung

Die Zuweisung ist eine *@elementare Anweisung*. Sie besteht aus einem Variablenbezeichner, dem der Zuweisungsoperator ": = " und einem Ausdruck, dessen Wert zunächst berechnet und dann der links vom Zuweisungsoperator stehenden Variablen übergeben wird.

```
Variablenbezeichner := ausdruck;
```

Beispiel:

```
name   := 'Meier';
ganzezahl := 12 + 5;
reellezahl := 1.5;
zeichen  := 'A';
gefunden := FALSE;
```