Algorithmen und Programmierung

(Studienjahr 2005/2006)

Doz.Dr.habil.Werner Vogt

Technische Universität Ilmenau
Fakultät für Mathematik und Naturwissenschaften
98684 Ilmenau, Germany

e-mail: werner.vogt@tu-ilmenau.de

Teil 3/3

Inhaltsverzeichnis

6 Datensätze und Dateien **252**

6.1 Algorithmen auf Datensätzen (records) 252

6.1.1 Recorddeklaration und -verarbeitung 252

6.1.2 Anwendungen von Records 259

6.2 Algorithmen auf Dateien (files) 269

6.2.1 Bearbeitung typisierter Dateien 269

6.2.2 Grundaufgaben der Dateiarbeit 278

6.2.3 Textdateien 282

6.2.4 Anwendung zur Ergebnis-Visualisierung 287

7 Modulare Programmierung **296**

7.1 Deklaration von Moduln 297

7.2 Aktivieren und Einbinden von Moduln 302

7.3 Standard-Moduln 310

7.4 Dynamische Linkbibliotheken (DLL) 313

7.5 Mathematische Modul-Bibliotheken 318

8 Rekursive Algorithmen **319**

8.1 Rekursive Funktionen 321

8.2 Entwurfsprinzipien rekursiver Prozeduren . . . 329

8.3 Komplexität und Sortierverfahren 334

8.4 Rekursion im wissenschaftlichen Rechnen . . . 343

9 Algorithmen auf dynamischen Datenstrukturen 349
9.1 Zeiger (Pointer, Verweise) 349
9.2 Lineare Listen (Ketten) 353
9.2.1 Definition und Aufbau 353
9.2.2 Operationen auf linearen Listen 355
9.3 Anwendung dynamischer Datenstrukturen . . . 361

10 Elemente objektorientierter Programmierung 369
10.1 Überladen von Operatoren 372
10.1.1 Operatordefinition in GNU-PASCAL 372
10.1.2 Abstrakte Datentypen 378
10.2 Objekte und Methoden 383
10.2.1 Objekte und abgeleitete Objekte 383
10.2.2 Methoden und deren Gültigkeitsbereich 388
10.3 Vererbung und Erweiterung von Objekten . . . 394

Kapitel 6

Datensätze und Dateien

In vielen Anwendungen der Informationsverarbeitung ist es zweckmäßig, Datenelemente unterschiedlichen Typs zu logischen Einheiten, den Datensätzen (records), zusammenzufassen. Damit lassen sich Dateien (files) günstig aufbauen. Algorithmen auf diesen 2 Datentypen unterscheiden sich z.T. erheblich von der Verarbeitung der Felder.

6.1 Algorithmen auf Datensätzen (records)

Ein Datensatz (record) ist ein Datentyp, der aus einer festgelegten Anzahl von Komponenten, den Datenfeldern besteht. Diese Datenfelder können innerhalb eines Records von verschiedenem Typ sein.

Beispiele:

Datum	:	Tag, Monat, Jahr
Name	:	Vorname, Zuname
Komplexe Zahl	:	Realteil, Imaginärteil
Adresse	:	PLZ, Ort, Strasse, Nr
Student	:	Matrikel-Nr., Name, Adresse, Geb.-Datum, Fächer

6.1.1 Recorddeklaration und -verarbeitung

Die Typdeklaration legt (i) die Struktur, d.h. die Reihenfolge der Datenfelder, sowie (ii) die Bezeichner und Typen der Datenfelder fest.

Typ- und Variablendeklaration

Syntax:

⟨Typname⟩ = record
⟨Liste 1 von Datenfeld-Bezeichnern⟩ : ⟨Typ 1⟩;
⟨Liste 2 von Datenfeld-Bezeichnern⟩ : ⟨Typ 2⟩;
.....
⟨Liste n von Datenfeld-Bezeichnern⟩ : ⟨Typ n⟩
end;

Beispiele:

```
Type  Datum   =  record          { Bsp.1 }
                   Tag   : 1..31;
                   Monat : 1..12;
                   Jahr  : integer
                 end;
      Name    =  record          { Bsp.2 }
                   Vorname : string[20];
                   Zuname  : string[40]
                 end;
      komplex =  record          { Bsp.3 }
                   re, im  :  real
                 end;
Var   Geburtstag, Einstellungstag : Datum;
      Mitarbeiterin, Azubi, Chef  : Name;
      z1, z2, z3, wurzel, alpha   : komplex;
```

Semantik:

- Der Typ der Datenfelder kann beliebig sein - also auch strukturiert (string, array, record etc.)
- Die Abspeicherung der Datenfelder erfolgt linear in Reihenfolge der Deklaration

- Der Gültigkeitsbereich der Datenfeldbezeichner (Tag, Vorname etc.) ist die Recorddeklaration selbst!
- Records sollten in einer Typdeklaration vereinbart werden; die Variablendeklaration wird dann übersichtlicher.

Strukturierte Typen in Datensatzabschnitten

Innerhalb der Datensatzabschnitte dürfen auch strukturierte Typen stehen. Es ist ratsam, diese Typen zuvor explizit zu deklarieren und dann die Typbezeichner zu verwenden. Damit können hierarchische Strukturen aufgebaut werden.

Beispiel 1 Muster eines Studierenden

Ein Record „Studenten“ ist zu definieren, der die wesentlichen Personendaten und die belegten Fächer enthält.

```
const Fachzahl =  25;  { max. Anzahl von Fächern }
type  Faecher  =  array[1..Fachzahl] of string[30];
      Namen    =  record
                      Vorname     :  string[20];
                      Zuname      :  string[40];
                  end;
      Datum    =  record
                      Tag         :  1..31;
                      Monat       :  1..12;
                      Jahr        :  integer;
                  end;
      Adressen =  record
                      PLZ         :  0..99999;
                      Ort         :  string[20];
                      Strasse     :  string[40];
                      Nr          :  1..999;
                  end;
```

```
Studenten =  record
                Matrikel_Nr :  integer;
                Name        :  Namen;
                Adresse     :  Adressen;
                Geb_Datum   :  Datum;
                Fach        :  Faecher;
             end;

var  TU_Studentin, Stipendiat          : Studenten;
     Immatrikulation, Exmatrikulation  : Datum;
     Hauptwohnung, Nebenwohnung        : Adressen;
```

Recordzugriff und -verarbeitung

- Zuweisung ganzer Records:
  Zuweisung ganzer Datensätze desselben Typs ist zulässig. Sämtliche Werte der entsprechenden Datenfelder werden (physisch) kopiert.

```
Hauptwohnung    :=  Nebenwohnung;
Exmatrikulation := Immatrikulation;
TU_Studentin    := Stipendiat;
```

- Recordkomponente und Komponentenselektor:
  Der Zugriff auf eine Recordkomponente geschieht durch
  – Angabe der Recordvariablen,
  – gefolgt vom Datenfeldbezeichner (Komponentenselektor),
  – getrennt durch eine Punkt.
  Es entsteht ein 2-stufiger Bezeichner.

```
Stipendiat.Matrikel_Nr
Hauptwohnung.Ort
Hauptwohnung.Strasse
Immatrikulation.Jahr
```

- Prozedur- und Funktionsparameter:
  Recordvariablen können als formale und als aktuelle Parameter stehen. Deklaration als Var-Parameter ist sinnvoll (Speicherökonomie!).

Beispiel 1 **(s.o.) Adressenänderung bei Umzug**

```
Program Studenten_Datenverarbeitung;
const  . . .
type   . . .
       Studenten =  record
                      Matrikel_Nr :  integer;
                      Name        :  Namen;
                      Adresse     :  Adressen;
                      Geb_Datum   :  Datum;
                      Fach        :  Faecher;
                    end;
var  TU_Studentin, Stipendiat     : Studenten;

procedure Umzug(var Alt,Neu: Studenten);
  { Adressenaenderung bei Umzug }
  begin
    Alt.Adresse := Neu.Adresse
  end;

begin  { Hauptprogramm }
  . . .
  Umzug(Stipendiat,TU_Studentin);
  . . .
end.
```

Frage: Wer zieht zu wem?

- Komponententyp = Feldtyp: Zugriff auf das Gesamtfeld oder - mittels des Komponentenselektors - auf eine Feldkomponente ist möglich. Die Typverträglichkeit ist dabei besonders zu beachten. Beispiele:

```
Stipendiat.Fach        { Typ -> Faecher }
Stipendiat.Fach[2]     { Typ -> string[30] }
Stipendiat.Fach[j][1]  { Typ -> char }
Stipendiat.Fach[2] := 'Algorithmen und Prog.';
for j := 1 to Fachzahl do
    writeln( TU_Studentin.Fach[j]);
TU_Studentin.Fach := Stipendiat.Fach;
```

- Komponententyp = Recordtyp: Das Komponentenfeld ist strukturiert; damit können mehrere Komponentenselektoren aufeinanderfolgen ⇒ mehrstufige Bezeichner. Beispiele:

```
TU_Studentin.Name.Vorname := 'Franzi';
Stipendiat.Adresse.Ort    := 'Goslar';
Stipendiat.Name.Zuname := TU_Studentin.Name.Zuname;
write( Stipendiat.Geb_Datum.Jahr );
```

With-Anweisung

Sie dient der Vermeidung langer mehrstufiger Bezeichner. Sie eröffnet einen Record, so daß die Bezeichner der Datenfelder wie einfache Bezeichner benutzt werden dürfen. Beispiele:

```
with TU_Studentin do
begin
  writeln (Matrikel_Nr);
  Name.Vorname := 'Franzi';
  Name.Zuname  := 'Schubert'
end;
with TU_Studentin.Name do
  writeln (Vormane, Zuname);
```

Syntax der with-Anweisung:

with ⟨Liste von Recordvariablen⟩ do ⟨Anweisung⟩

Semantik:

- Die Recordvariablen-Liste besteht aus ein- oder mehrstufigen Recordvariablen. Die Anweisung

  ```
  with v1,v2,...,vn do <Anweisung>;
  ```

  ist äquivalent zu

  ```
  with v1 do
    with v2 do
      . . . . . .
        with vn do <Anweisung>;
  ```

- Durch die with-Anweisung wird ein Gültigkeitsbereich für die Datenfeldbezeichner des Record-Typs eingeführt; dieser Gültigkeitsbereich ist genau die hinter do stehende Anweisung!

  Beispiele:

```
with TU_Studentin do
begin
  writeln(Matrikel_Nr);
  with Name do
    writeln(Vorname + ' ' + Zuname);
  with Geb_Datum do
    writeln(Tag, Monat, Jahr);
  for j := 1 to Fachzahl do
    writeln(Fach[j])
end;  { of with }
```

6.1.2 Anwendungen von Records

Records sind nicht nur in der Personendaten-Verarbeitung von Bedeutung, sondern werden auch in naturwissenschaftlich-technischen Berechnungen häufig benutzt.

Beispiel 1 **(s.o.) Studentendaten**

Für n Studierende $(n \leq 50)$ sind die Daten zu Matrikel-Nr., Name und Adresse einzugeben. Alle Studierenden mit Adresse in Ilmenau sind zu suchen und mit Namen und Matrikel auszugeben.

```
Program Studenten_Datenverarbeitung;
const Fachzahl  =  25;  { max. Anzahl von Fächern }
      Studzahl  =  50; { max. Anzahl Studierender }

type  Faecher   =  array[1..Fachzahl] of string[30];
      Namen     =  record
                      Vorname     :  string[20];
                      Zuname      :  string[40];
                   end;
      Datum     =  record
                      Tag         :  1..31;
                      Monat       :  1..12;
                      Jahr        :  integer;
                   end;
      Adressen  =  record
                      PLZ         :  0..99999;
                      Ort         :  string[20];
                      Strasse     :  string[40];
                      Nr          :  1..999;
                   end;
```

```
      Studenten =  record
                      Matrikel_Nr :  integer;
                      Name        :  Namen;
                      Adresse     :  Adressen;
                      Geb_Datum   :  Datum;
                      Fach        :  Faecher;
                   end;
type  Stud_array = array[1..Studzahl] of Studenten;

var   n, i       : integer;
      Element    : Stud_array;

procedure Dateneingabe (var Student : Studenten);
  { Eingabe der Daten eines Studierenden }
  begin
    with Student do begin
      writeln;
      write('Matrikel_Nr.: '); readln(Matrikel_nr);
      with Name do begin
         write('Vorname     : '); readln(Vorname);
         write('Zuname      : '); readln(Zuname)
      end;
      with Adresse do begin
         write('Postleitzahl: '); readln(PLZ);
         write('Ort         : '); readln(Ort);
         write('Strasse     : '); readln(Strasse);
         write('Hausnummer  : '); readln(Nr)
      end    { . . .  usw. . . }
    end      { of with Student }
  end;       { of Dateneingabe }
```

```
procedure Datenausgabe (var Student : Studenten);
  { Ausgabe der Daten eines Studierenden }
  begin
    with Student do begin
      writeln;
      writeln('Matrikel_Nr.: ', Matrikel_nr);
      with Name do
         writeln('Name      : ', Vorname+' '+Zuname);
      with Adresse do begin
         writeln('PLZ       : ', PLZ);
         writeln('Ort       : ', Ort);
         writeln('Strasse   : ', Strasse);
         writeln('Hausnummer: ', Nr)
      end    { . . .  usw. . . }
    end      { of with Student }
  end;       { of Datenausgabe }

procedure Lineare_Suche (n : integer;
             var Element : Stud_array);
{ Lineare Suche aller Studierenden aus Ilmenau }

  var   i : integer;
  begin
    for i:=1 to n do begin
      if (Element[i].Adresse.Ort = 'Ilmenau') or
         (Element[i].Adresse.PLZ = 98693) then
           Datenausgabe(Element[i])
    end;
  end;     { of Lineare_Suche }
```

```
begin  { Hauptprogramm }
  write('Anzahl der Studierenden: ');
  readln(n); writeln;
  for i:=1 to n do Dateneingabe (Element[i]);
  readln; writeln;
  writeln('Studierende aus Ilmenau:');
  Lineare_Suche (n, Element);
  readln
end.
```

Beispiel 2 Intervall-Arithmetik

Gesucht ist eine einfache Nullstelle eines reellen Polynoms n-ten Grades

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n = \sum_{k=0}^{n} a_k x^k$$

in einem vorgegebenen Intervall. Zusätzlich soll eine Genauigkeitsaussage zum Ergebnis getroffen werden.

- Regeln der Intervall-Arithmetik:

  $[a,b] + [c,d] := [a+c, b+d]$
  $[a,b] - [c,d] := [a-d, b-c]$
  $[a,b] * [c,d] := [\min(ac,ad,bc,bd), \max(ac,ad,bc,bd)]$
  $[a,b]/[c,d] := [a,b] * [1/d, 1/c], \quad$ falls $0 \notin [c,d]$
  $[a,a] := a$

- Datentyp „Interval“: Seien m der berechnete Näherungswert, u die untere und o die obere Intervallgrenze.

```
type  Interval = record
                   u,m,o : real
                 end;
```

- Beachtung der Rundungsfehler:

  Rechnung mit s-stelliger Mantisse (z.B. s = 10) liefert bei jeder Grundoperation einen (maximalen) relativen Rundungsfehler

$$\boldsymbol{RelFeh = 0.5 \cdot 10^{-s} = 5 \cdot 10^{-11}.}$$

  Die erhaltenen Ergebnisintervalle $[u, o]$ werden deshalb nach folgenden Regeln (nach außen hin) korrigiert:

$$u \begin{cases} < 0 & , \text{ so } \quad u := u + u * RelFeh \\ = 0 & , \text{ so } \quad u := -RelFeh \\ > 0 & , \text{ so } \quad u := u \end{cases}$$

$$o \begin{cases} > 0 & , \text{ so } \quad o := o + o * RelFeh \\ = 0 & , \text{ so } \quad o := +RelFeh \\ < 0 & , \text{ so } \quad o := o \end{cases}$$

```
const  nmax     = 10;
       eps      = 1E-8;
type   interval = record
                    o,m,u : real
                  end;
       koeff    = array[0..nmax] of interval;

procedure Fehler(var u,o:real);
  const  relfeh = 5E-11;
         rho    = 1.00000000005;
  begin
    if u<0 then u := u*rho else if u=0 then u := -relfeh;
    if o>0 then o := o*rho else if o=0 then o := relfeh
  end;

procedure iadd(o1,o2:interval; var e:interval);
  begin  with e do
    begin   u := o1.u + o2.u;  o := o1.o + o2.o;
            m := o1.m + o2.m;  fehler(u,o)    end
  end;
```

```
procedure isub(o1,o2:interval; var e:interval);
  begin  with e do
    begin   u := o1.u - o2.o;  o := o1.o - o2.u;
            m := o1.m - o2.m;  fehler(u,o)    end
  end;

procedure imul(o1,o2:interval; var e:interval);
  var  z : real;
  begin  with e do
    begin  u := o1.u * o2.u;  o := u;  z := o1.o * o2.u;
           if z<u then u:=z else if z>o then o:=z;
           z := o1.u * o2.o;
           if z<u then u:=z else if z>o then o:=z;
           z := o1.o * o2.o;
           if z<u then u:=z else if z>o then o:=z;
           m := o1.m * o2.m;  fehler(u,o)       end
  end;

procedure idiv(o1,o2:interval; var e:interval);
  var  z : real;
  begin  with e do
    begin  if o2.u * o2.o <= 0 then
           begin  writeln('Division durch Null !');
                  readln; Halt  end;
           u := o1.u / o2.u;  o := o1.o / o2.o;
           if u>o then  begin z:=u; u:=o; o:=z end;
           z := o1.u / o2.o;
           if z>o then o:=z else if z<u then u:=z;
           z := o1.o / o2.u;
           if z>o then o:=z else if z<u then u:=z;
           m := o1.m / o2.m;  fehler(u,o)       end
  end;

procedure setinterval(r:real; var x:interval);
  begin  with x do
    begin   u := r; m := u; o := u  end
  end;
```

- Berechnung von $p(x)$ und $p'(x)$:

$$\begin{aligned} p(x) &= a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 \\ p'(x) &= d_n x^n + d_{n-1}x^{n-1} + \ldots + d_1 x + d_0 \quad \text{mit} \\ & \qquad d_n := 0, \quad d_{i-1} := a_i \cdot i \text{ für } i = 1(1)n. \end{aligned}$$

Anwendung des Hornerschemas zur Bestimmung von

(i) $p(x0)$ für $x0 \in$ real

(ii) $p'(x)$ für $x \in$ interval

für $a_i \in$ interval :

```
{ Bereitstellung der Koeffizienten d[i] }
for i:=1 to n do
  begin  del := i; setinterval(del,x0);
         imul(a[i],x0,d[i-1])
  end;
setinterval(0,d[n]);          { d[n] := 0 }
p := a[n]; p1 := d[n];    { Anfangswerte }
setinterval(x.m,x0);      { Mittelwert m }
for i:=n-1 downto 0 do { Hornerschemata }
begin
    imul(x0,p,p);  iadd(a[i],p,p);
    imul(x,p1,p1); iadd(d[i],p1,p1)
end;
{ Werte von p und p1 }
```

- Newton-Verfahren für Intervallzahlen:
Sei $x = [u, o]$ ein Intervall mit Mittelwert m, das eine einfache Nullstelle von $p(x)$ enthält. Mit $x0 := [m, m]$ lautet das Intervall-Newton-Verfahren:

$$x := \left(x0 - \frac{p(x0)}{p'(x)} \right) \cup x$$

```
procedure Nullstelle(var a:koeff; var x:interval);
 var  del, del1    : real;  i : integer;
      p,p1,x0,x1,z : interval;
      d            : koeff;
 begin
    for i:=1 to n do  begin
       del := i; setinterval(del,x0);
       imul(a[i],x0,d[i-1])
    end;  setinterval(0,d[n]);
    repeat
       del := x.o -x.u;
       p := a[n]; p1 := d[n];
       setinterval(x.m,x0);
       for i:=n-1 downto 0 do begin  { Hornerschema }
          imul(x0,p,p);  iadd(a[i],p,p);
          imul(x,p1,p1); iadd(d[i],p1,p1)
       end;
       if p1.u * p1.o <= 0 then begin
          writeln('Ableitung = 0  geworden !');
          readln; Halt
       end;
       idiv(p,p1,z); isub(x0,z,x1); { Newtonschritt }
       if x1.u > x.u  then x.u := x1.u;
       if x1.o < x.o  then x.o := x1.o;
       with x do
          if (x1.m<u)or(x1.m>o) then m := (u + o)/2.0
                                else m := x1.m;
       del1 := x.o - x.u;
       writeln('   x.m  = ',x.m)
   until  (del1 < eps) or (del1 = del)
end;                             { of Nullstelle }
```

- Hauptprogramm:

```
Program Newton;
uses   CRT;
const  nmax     = 10;
       eps      = 1E-8;
type   interval = record
                    o,m,u : real
                  end;
       koeff    = array[0..nmax] of interval;
var    n,i      : integer;
       x        : real;
       a        : koeff;
       x0       : interval;

{ ***  Deklaration der Prozeduren  fehler, iadd, isub
       imul, idiv, setinterval, Nullstelle   ***  }

begin  { Hauptprogramm }
  clrscr; write('Polynomgrad :  '); readln(n);
  for i:=n downto 0 do
  begin  write('   Koeffizient bei x^',i,' : ');
    readln(x);
    setinterval(x,a[i])
  end;
  writeln;  writeln('Startwert');
  write('   obere  Grenze : '); readln(x0.o);
  write('   untere Grenze : '); readln(x0.u);
  x0.m := (x0.o + x0.u)/2.0;
  writeln; writeln('Zwischenwerte ');
  Nullstelle(a,x0);
```

```
  writeln; writeln('Loesung :');
  writeln('   obere  Grenze :   ',x0.o);
  writeln('   Mittelwert    :   ',x0.m);
  writeln('   untere Grenze :   ',x0.u);
  readln
end.
```

- Beispiel: $\boldsymbol{p_4(x) = x^4 - 16\,, \quad [-2.7, -1.8]}$

```
Polynomgrad :  4
   Koeffizient bei x^4 : 1
   Koeffizient bei x^3 : 0
   Koeffizient bei x^2 : 0
   Koeffizient bei x^1 : 0
   Koeffizient bei x^0 : -16
Startwert
   obere  Grenze :  -1.8
   untere Grenze :  -2.7
Zwischenwerte
   x.m  = -2.0386659808E+00
   x.m  = -2.0010861837E+00
   x.m  = -2.0000008840E+00
   x.m  = -2.0000000000E+00
Loesung :
   obere  Grenze :  -1.9999999998E+00
   Mittelwert    :  -2.0000000000E+00
   untere Grenze :  -2.0000000001E+00
```

6.2 Algorithmen auf Dateien (files)

Eine Datei (file) ist ein Datentyp, der aus Komponenten des gleichen Typs besteht und insbesondere der dauerhaften Speicherung großer Datenmengen außerhalb des Arbeitsspeichers dient. Die Anzahl der Komponenten (die Länge der Datei) wird nicht durch die Definition festgelegt. Pascal unterscheidet

- Typisierte Binärdateien
- Nicht-typisierte Binärdateien
- Textdateien.

6.2.1 Bearbeitung typisierter Dateien

Aufbau von Dateien

- Der Zugriff zu einer Komponente geschieht durch einen Datei-Zeiger (pointer). Nach jedem Lese- oder Schreibvorgang rückt der Zeiger um eine Komponente weiter $\Rightarrow$ sequentieller Zugriff (sequential access files).
- Die Position einer beliebigen Komponente kann berechnet werden und der Zeiger damit auf eine beliebige Komponente gesetzt werden $\Rightarrow$ wahlfreier Zugriff (random access files).
- Das Dateiende wird durch ein spezielles Dateiende-Kennzeichen (End-of-file, eof) signalisiert, das nicht überschritten werden darf.
- Eine physisch vorhandene Datei mit 0 Komponenten heißt leere Datei.
- Schematische Darstellung:

Typ- und Variablendeklaration

Syntax der Typdeklaration: ⟨Typname⟩ = file of ⟨Typ⟩;

Semantik:

- Der Komponententyp kann beliebig sein - also auch strukturiert (string, array, record etc.), darf aber keinen Dateityp enthalten.
- Dateivariablen können wie üblich deklariert werden. Sie können durch Dateiprozeduren und -funktionen bearbeitet werden und dürfen selbst als Parameter in Unterprogrammen fungieren.

Beispiel 1: Real- und Stringdateien

```
type  Zahlenfile = file of single;
      Namenfile  = file of string[20];
var   Verfahren          : Namenfile;
      Eingabe, Resultate : Zahlenfile;
```

Beispiel 2: Dateien von Records

```
type  Name       =  record
                      Vorname     :  string[20];
                      Zuname      :  string[40]
                    end;
      Person     =  record
                      Lfd_Nr      :  integer;
                      Pers_Name   :  Name;
                      Alter       :  integer
                    end;
      Personfile =  file of Person;

var  Studrec     : Person;     { Record-Variable }
     Studfile    : Personfile;   { File-Variable }
     Namenfile   : file of Name;
```

Standard-Prozeduren zur Dateiverarbeitung

Dateiarbeit geschieht durchweg mittels Prozeduren/Funktionen. Nachfolgend steht `filvar` für eine beliebige Filevariable und `str` für einen Stringausdruck.

- Zuweisung eines Dateinamens

  Format : `assign (filvar, str);`

  Wirkung : Zuweisung des (physischen) Dateinamens str zur Dateivariablen filvar.

  Beispiel :
  ```
  assign( Namenfile , 'Liste1.dat');
  assign( Studfile , 'D:\daten\liste2.dat');
  ```

- Schaffung (Eröffnung) einer neuen Datei

  Format : `rewrite (filvar);`

  Wirkung : Schaffung einer (physischen) Datei mit dem durch assign zugewiesenen Namen; die Datei ist leer und der Zeiger steht auf Komponente 0 !

- Zurücksetzen des Zeigers auf den Dateianfang

  Format : `reset (filvar);`

  Wirkung : Die (physischen) Datei mit dem durch assign zugewiesenen Namen wird zur Bearbeitung freigegeben. Der Zeiger wird auf Komponente 0 gesetzt!

  Hinweis : filvar muß eine existierende Datei bezeichnen.

- Lesen aus einer Datei

  Format : `read (filvar, var_1, var_2, ..., var_n);`

  Wirkung : Lesen der nächsten Dateikomponente und Zuweisung anvar_i sowie Weitersetzen des Dateizeigers.

  Hinweis : var_i muß vom Komponententyp von filvar sein!

- Schreiben in eine Datei

  Format : `write (filvar,var_1,var_2,...,var_n);`

  Wirkung : Schreiben des Wertes von var_i in die nächste Dateikomponente und Weitersetzen des Zeigers.

  Hinweis : var_i muß vom Komponententyp von filvar sein! var_i wird binär verscchlüsselt abgespeichert.

- Setzen des Dateizeigers

  Format : `seek (filvar, n);`

  Wirkung : Setzen des Zeigers auf die n-te Komponente für wahlfreien Zugriff. n ist ein integer-Ausdruck.

  Hinweis : Die Position der ersten Komponente ist 0 !

  Beispiel : `seek( Namenfile, filesize(Namenfile));` setzt den Zeiger hinter die letzte Komponente.

- Schließen einer Datei

  Format : `close (filvar);`

  Wirkung : Schließen einer (physischen) Datei mit dem durch assign zugewiesenen Namen; die Datei wird auf den neuesten Stand gebracht (Informationen).

  Weitere Prozeduren dienen dem physischen Löschen (erase) sowie dem Umbenennen (rename) einer Datei.

Standard-Funktionen zur Dateiverarbeitung

- Dateiende (End-of-file)

  Format : `eof (filvar)`

  Wirkung : Boolean-Funktion mit Wert true, falls der Zeiger hinter der letzten Komponente steht; ansonsten ist der Funktionswert false.

- Aktuelle Zeigerposition

  Format : `filepos (filvar)`

  Wirkung : integer-Funktion, die die aktuelle Position des Zeigers liefert (Erste Komponente ergibt 0).

- Aktuelle Länge der Datei

  Format : `filesize (filvar)`

  Wirkung : integer-Funktion, die die Anzahl der Komponenten liefert (leere Datei ergibt 0).

Beispiel 3 Vereinfachte Studierendendatei

Eine Studierendendatei des obigen Typs „Personendatei“ liege unter dem Namen 'stud.dat' im aktuellen Verzeichnis vor. Sämtliche Datensätze der Datei sollen auf dem Bildschirm angezeigt werden.

```
{ Deklarationen des Hauptprogramms }
type  Name     =  record
                      Vorname     :  string[20];
                      Zuname      :  string[40]
                  end;
      Person   =  record
                      Lfd_Nr      :  integer;
                      Pers_Name   :  Name;
                      Alter       :  integer
                   end;

      Personfile = file of Person;
```

```
procedure Auflisten ( var studfile : Personfile);

 { *** Auflisten aller Records einer
       Studentendatei auf dem Bildschirm *** }

var studrec  : Person;   { Variable fuer 1 Record }
begin
  assign (studfile, 'stud.dat');      { Zuweisung }
  reset (studfile);                 { Ruecksetzen }

  clrscr;
  writeln('Ausgabe der Datei STUD.DAT');

  while not eof(studfile) do begin

    read (studfile, studrec);      { Lesen Record }

    with studrec do begin
      writeln('Lfd_Nr.   : ', Lfd_Nr);
      writeln('Name      : ',
        Pers_Name.Zuname, ' ', Pers_Name.Vorname);
      writeln('Alter     : ', Alter, ' Jahre');
      writeln; readln
    end
  end;
  writeln ('>>> Anzahl der Personen : ',
    filesize(studfile));
  close(studfile)
end;  { of Auflisten }
```

Behandlung von Ein- und Ausgabefehlern

Dateiarbeit betrifft meist große Datenmengen, so daß eine Sicherheit der Bearbeitung eine wesentliche Forderung darstellt. Unkontrollierte Abstürze eröffneter Dateien, deren Bearbeitungszustand dann oft nicht bekannt ist, bedeuten nicht nur den Verlust der in Arbeit befindlichen Dateien, sondern auch einen erheblichen Arbeitszeitausfall. Deshalb gebührt der Vermeidung von IO-Fehlern besondere Bedeutung.

- Compilerbefehl (Compilerschalter) `{$I+}` `{$I+}`

  Format : `{$I+}` oder `{$I+}`

  Wirkung : Der lokale Schalter legt fest, ob Ein- und Ausgaben automatisch auf Fehler geprüft werden sollen (+) oder nicht (-). Standard ist `{$I+}`.

  Beispiele : Versuch eines reset bei nicht-existenter Datei, Anssprechen einer Datei, die nicht durch reset oder rewrite eröffnet wurde.

- Standardfunktion IOResult

  Format : `function IOResult : integer;`

  Wirkung : Ist `{$I-}` geschaltet, so liefert IOResult den Fehlercode 0, falls die IO-Operation fehlerfrei war, ansonsten einen von 0 verschiedenen Wert.

  Bemerkung : Durch den Aufruf wird der Fehlerzustand zurückgesetzt, und weitere Ein- und Ausgaben sind möglich.

Beispiel 4 — Kontrolliertes Eröffnen einer neuen Datei

Nach Einlesen eines Dateinamens ist zu überprüfen, ob diese Datei bereits existiert. Dann ist zu entscheiden, ob sie überschrieben oder ein anderer Dateiname gewählt werden soll.

Beispiel 4 Kontrolliertes Eröffnen einer neuen Datei

Nach Einlesen eines Dateinamens ist zu überprüfen, ob diese Datei bereits existiert. Dann ist zu entscheiden, ob sie überschrieben oder ein anderer Dateiname gewählt werden soll.

```
procedure Neueroeffnung(var filevar : filetyp);
  var  io     : integer;
       ch     : char;
       name   : string[80];
begin
  repeat
    clrscr;
    write('Name der neuen Datei : '); readln(name);
    assign(filevar, name);

    {$I-} reset(filevar); {$I+}
    io := IOResult;

    if io = 0 then
    begin
      writeln('Achtung: Datei existiert bereits !');
      write('<U>eberschreiben oder ');
      write('<N>euer Dateiname?');
      readln(ch);
      if Upcase(ch) = 'U' then  io := 1
    end
  until  io <> 0 ;
  rewrite(filevar)   { Nun wird ueberschrieben... }
end;
```

Beispiel 5 Kontrolliertes Rücksetzen einer Datei

Nach Einlesen eines Dateinamens ist zu überprüfen, ob diese Datei bereits existiert. Erst dann ist sie mit reset zu eröffnen.

```
procedure Ruecksetzen(var filevar: filetyp);
  var  io    : integer;
       name  : string[80];
  begin
    repeat
      writeln;
      write('Name der Datei : '); readln(name);
      assign(filevar, name);

      {$I-}   reset(filevar);   {$I+}
      io := IOResult;

      if  io <> 0  then
      begin
        writeln('Datei ',name,' existiert nicht !');
        delay(3000)
      end
    until  io = 0
  end;
```

6.2.2 Grundaufgaben der Dateiarbeit

Wesentliche Aufgaben bei der Bearbeitung großer Dateien sind:

1. Verarbeitung der Gesamtdatei
   - Anlegen und Erstbeschreibung einer Datei
   - Lesen und Auflisten einer Datei
   - Löschen, Kopieren bzw. Umbenennen einer Datei
2. Verarbeitung einzelner Dateikomponenten
   - Suchen in einer Datei nach Suchmerkmalen
   - Korrigieren einzelner Komponenten
   - Erweitern einer Datei (am Ende)
   - Kopieren von Komponenten in eine zweite (neue) Datei
   - Löschen einzelner Komponenten usw.

Beispiel 3 **Vereinfachte Studierendendatei (s.o.)**

Beispielhaft sollen folgende 3 Aufgaben gelöst werden:

- Erweitern der Datei STUD.DAT (am Ende) durch neue Studierende
- Ausgabe der Namen aller Studierenden mit einem Suchmerkmal
- Erstellen einer neuen Datei dieser Studierenden

```
{ *** Deklarationen des Hauptprogramms:
      Typen Person, Personfile usw. *** }

procedure Erweitern ( var studfile : Personfile);
 { *** Erweitern einer Studentendatei um beliebig
       viele weitere Datensaetze *** }
var studrec  : Person;   { Variable fuer 1 Record }
    nr       : integer;
    ch       : char;
```

```
begin
  assign (studfile, 'stud.dat');      { Zuweisung }
  reset (studfile);                 { Ruecksetzen }
  clrscr;
  writeln('Erweitern der Datei STUD.DAT');
  nr := filesize(studfile);
  seek(studfile, nr);  { .. hinter letzten Record }
  repeat
    clrscr;
    nr := nr+1;  { Zaehler aktualisieren }
    ch := 'J';   { Standard setzen }
    writeln('Lfd_Nr.  : ',nr);
    with studrec do begin
      Lfd_Nr := nr;
      write('Zuname   : ');
        readln(Pers_Name.Zuname);
      write('Vorname  : ');
        readln(Pers_Name.Vorname);
      write('Alter    : ');
        readln(Alter);
      writeln
    end;

    write(studfile, studrec);  { Schreiben Record }

    write('>> Eingabe fortsetzen ((J)/N)? ');
    readln(ch)
  until upcase(ch) = 'N';
  writeln('Dateilaenge: ',filesize(studfile));
  close(studfile);  readln
end;  { of Erweitern }
```

Die Namen aller Studierenden aus STUD.DAT mit einem Alter von mindestens 21 Jahren sind auszugeben.

```
procedure Lineares_Suchen
    ( var studfile: Personfile;
      var studrec : Person );
  begin
    assign(studfile,'STUD.DAT');
    reset(studfile); clrscr;
    writeln('Mindestalter von 21 Jahren :');
    while not EOF(studfile) do
    begin
      read(studfile, studrec);
      with studrec do
        if alter >= 21 then
          writeln(Pers_Name.Vorname,' ',
                  Pers_Name.Zuname,
                  ' (Alter : ',Alter,')')
    end;
    close(studfile)
  end;
```

Für alle Studierenden aus STUD.DAT mit einem Alter von mindestens 21 Jahren ist eine neue Datei STUD21.DAT zu erstellen.

```
procedure Kopieren_in_Datei
          (var studfile1,studfile2: Personfile);
  var  studrec : Person;
  begin
    assign(studfile1,'A:STUD.DAT');
    reset(studfile1);
    assign(studfile2,'B:STUD21.DAT');
    rewrite(studfile2);

    while  not EOF(studfile1) do
    begin
      read(studfile1,studrec);
      if studrec.Alter >= 21 then
        write(studfile2,studrec)
    end;
    close(studfile1); close(studfile2)
  end;
```

Hinweis:

Das Löschen einzelner Komponenten kann nur dadurch geschehen, daß andere Komponenten an ihre Stelle gespeichert werden (z.B. das letzte Element der Datei). Soll die Anordnung jedoch erhalten bleiben, so empfiehlt sich das Löschen bzw. Verändern von Komponenten durch Erstellen einer 2. Datei („Tochterdatei“) nach dem Muster des Beispiels. Anschließendes Umbenennung in die „Mutterdatei“ liefert das gewünschte Ergebnis (Vorsicht: Datenverlust möglich!).

6.2.3 Textdateien

Textdateien werden im ASCII-Format abgespeichert und können deshalb – im Gegensatz zu binären Dateien – mit Texteditoren gelesen und bearbeitet werden. Die Ein- bzw. Ausgabe über Tastatur, Bildschirm und Drucker geschieht ebenfalls in Form von Textdateien. Nachteilig ist der i.allg. beträchtliche Speicheraufwand: Die double-Zahl
-3.1415926535897 93E+0100 belegt binär gespeichert 8 Bytes, wogegen die Zeichenkettenspeicherung 24 Bytes benötigt!

Deklaration und Aufbau

- Textdateien werden mittels des Standardtyps text deklariert.
  Interpretation: `text = file of char;`

  Beispiel:

```
var  Dokumentation : text;
     Ausgabe       : array [1..3] of text;
```

- Aufbau und Speicherung: Textdateien sind in Zeilen untergliedert. Die Zeilen werden durch ein Zeilenende-Kennzeichen (End-of-Line, eofln) markiert.
  Skizze:

- 2 Zugriffsarten für Dateien:
  - Dateien mit wahlfreiem Zugriff (random access files)
  - Dateien mit sequentiellem Zugriff (sequential access files)

  Typisierte Dateien besitzen wahlfreien Zugriff; dagegen sind Textdateien nur sequentiell ansprechbar. Die Position einer bestimmten Zeile (und damit eines bestimten Zeichens daraus) kann hier i.allg. nicht berechnet werden.

- Öffnen von Textdateien:
  - Rewrite wird benutzt, um eine neue Textdatei zu schaffen (Eröffnung zum Schreiben). Danach darf nur sequentiell geschrieben werden.
  - Reset öffnet eine bereits exisiterende Datei zum Schreiben. Es ist nur sequentielles lesen erlaubt.
  - Append läßt in BP das Öffnen einer existierenden Datei zum Schreiben zu. Der Zeiger steht hinter der letzten Komponente.

Standardroutinen für Textdateien

Zusätzlich zu den Dateiprozeduren und -funktionen sind folgende Routinen möglich:

- Zeilenende (End-of-line)

  Format : `eoln (filvar)`

  Wirkung : Boolean-Funktion mit Wert true, falls der Zeiger auf einer Zeilenende-Markierung steht; ansonsten ist der Wert false.

- Schreiben einer Zeilenende-Markierung

  Format : `writeln (filvar);`

  Wirkung : Prozedur, die eine Zeilenende-Markierung schreibt.

- Einstellen des Dateizeigers

  Format : `readln (filvar);`

  Wirkung : Prozedur, die den Dateizeiger auf den Anfang der nächsten Zeile rückt.

- Leseprozeduren

  Format : `read (filvar,var_1,var_2,...,var_n);`
  `readln (filvar,var_1,var_2,...,var_n);`

  Wirkung : identisch mit dem bekannten read bzw. readln, bezogen auf die Datei filvar.

- Schreibprozeduren

  Format : `write (filvar,Wpar_1,Wpar_2,...,Wpar_n);`
  `writeln (filvar,Wpar_1,Wpar_2,...,Wpar_n);`

  Wirkung : identisch mit dem bekannten write bzw. writeln, bezogen auf die Datei filvar. Wpar_i sind Schreibparameter (ggf. mit Formatangabe).

Beispiel 6 **Anlegen/Erweitern einer Protokolldatei**

Die Ergebnisse der Berechnung einer Stabmasse (vgl. Beispiel 8 aus Abschnitt 3.3.2) sollen in einer Protokolldatei in der Form

```
*****************************
Masse m eines Stabes
====================
Stablaenge       L = 4.6
Intervallanzahl  n = 100
Gesamtmasse      m = 11.979221
*****************************
Masse m eines Stabes
====================
Stablaenge       L = 4.6
Intervallanzahl  n = 200
Gesamtmasse      m = 11.978402
```

abgespeichert werden. Dabei sind folgende 3 Fälle vorzusehen:
– Anlegen einer neuen Protokolldatei
– Überschreiben einer alten Protokolldatei
– Anhängen an die bisherige Protokolldatei.

```
Program Stabmasse;
  uses Crt;
  { vgl. Programm in Abschnitt 3.3.2 }

procedure Eroeffne_Protokoll (var filevar : text);
 { ** Sichere Eroeffnung einer Protokolldatei ** }
  var  io     : integer;
       ch     : char;
       name   : string[80];
  begin
    repeat
      writeln; write('Name der Protokolldatei: ');
      readln(name);
      assign(filevar, name);
      {$I-}   reset(filevar);   {$I+}
      io := IOResult;
      if io <> 0          then
        rewrite(filevar)  else
      begin
        writeln('Achtung:Datei existiert bereits!');
        write('<U>eberschreiben,<A>nhaengen ');
        write('oder <N>euer Dateiname? ');
        readln(ch);
        case Upcase(ch) of
         'A': begin append(filevar);  io := 1 end;
         'U': begin rewrite(filevar); io := 1 end;
        end
      end
    until  io <> 0 ;
  end;  { of Eroeffne_Protokoll }
```

```
procedure Erstelle_Protokoll
          ( L : double; n : integer; m : double );
 { *** Erstellung einer Textdatei mit den
       Programmergebnissen: Protokolldatei *** }
  var  i         : byte;
       Protokoll : text;
  begin
    Eroeffne_Protokoll(Protokoll);
    for i:=1 to 30 do write(Protokoll,'*');
    writeln(Protokoll);
    writeln(Protokoll,'Masse m eines Stabes');
    writeln(Protokoll,'====================');
    writeln(Protokoll,'Stablaenge    L  =',L:12:6);
    writeln(Protokoll,'Intervallzahl n  =   ', n);
    writeln(Protokoll,'Gesamtmasse   m  =',m:12:6);
    writeln(Protokoll);
    close(Protokoll)
  end;   { of Erstelle_Protokoll }

begin    { Hauptprogramm }
  writeln('Masse m eines Stabes');
  writeln('====================');
  repeat  writeln;
    write  ('Stablaenge    L  = '); readln(L);
    write  ('Intervallzahl n  = '); readln(n);
    Trapez (0,L,n,m);
    writeln('Gesamtmasse   m  = ',m:12:6);
    Erstelle_Protokoll (L, n, m);
    write  ('Wiederholung(J/N)? '); readln(z)
  until  upcase(z) = 'N'
end.
```

6.2.4 Anwendung zur Ergebnis-Visualisierung

Mathematische Visualisierungsprogramme verarbeiten grafische Daten (z.B. Kurven, Flächen), indem die berechneten Kurven- oder Flächenpunkte $\boldsymbol{x = (x_1, x_2, ..., x_n) \in \mathbb{R}^n}$ als Textdateien vorliegen müssen. Typisierte Ergebnisdateien sind deshalb in Textdateien zu transferieren.

Beispiel 7 **Abspeicherung grafischer Daten**

Gegeben seien 2 reelle Funktionen

$$\boldsymbol{x = f(t), \quad y = g(t), \quad t_0 \le t \le t_e\ ,}$$

die die Parameterdarstellung einer ebenen Kurve beschreiben.

Aufgaben:

(a) Zu Parametergrenzen $\boldsymbol{t_0, t_e}$ und der Zahl $\boldsymbol{n}$ der darzustellenden Kurvenpunkte sind die Wertepaare $\boldsymbol{(x_i, y_i) = (f(t_i), g(t_i))}$ an den Parameterwerten $\boldsymbol{t_i = t_0 + (t_e - t_0) \cdot i/n}$ zu berechnen und in einer typisierten Datei <Name>.dat im Datenformat `single` abzuspeichern.

(b) (fakultativ) Nach Lesen der Bildpunkte $\boldsymbol{(x_i, y_i),\ i = 1(1)n}$ und Bestimmung der Extremalwerte $\boldsymbol{x_{min}, x_{max}, y_{min}, y_{max}}$ ist die Kurve punktweise darzustellen.

(c) Die typisierte Datei ist in eine Textdatei <Name>.txt mit möglichst geringem Speicherbedarf umzuwandeln und abzuspeichern. Beispiel:

```
#  2D-Kurvendatei  bsp1.txt
3.0000E+00  0.0000E+00
2.5043E+00  9.1209E-01
1.4667E+00  1.5228E+00
1.8265E-01  1.7171E+00
-1.0221E+00  1.4889E+00
```

(d) Die Textdatei <Name>.txt ist mit Maple einzulesen und mittels `pointplot` darzustellen.

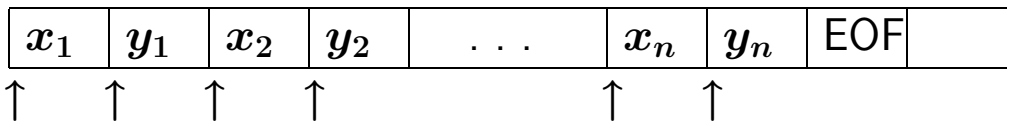
- Definition der Kurve: Darzustellen ist eine logarithmische Spirale

$$x = f(t) = 3e^{-0.1t}\cos t, \quad y = g(t) = 2e^{-0.1t}\sin t.$$

Die Rechnung kann im single-Format ausgeführt werden.

```
Program Kurven_in_2D;
uses  CRT, GRAPH;
type  singledatei = file of single;
var   ch     : char;
      Punkte : singledatei;

{ *** Definition der 2 Funktionen  f und g *** }
procedure Kurve (t:single; var x,y:single);
  begin
    x := 3.0*exp(-0.1*t)*cos(t);
    y := 2.0*exp(-0.1*t)*sin(t)
  end;
```

- Berechnung der Kurvenpunkte:
Die Wertepaare $(x_i, y_i) = (f(t_i), g(t_i))$ an den Parameterwerten $t_i = t_0 + (t_e - t_0) \cdot i/n$ sind zu berechnen und in einer typisierten Datei <Name>.dat abzuspeichern:




```
procedure Schreiben (var Punkte: singledatei);
var  t,t0,t1,dt,x,y  : single;
     n               : integer;
     Name            : string[30];
```

```
begin      { of Schreiben einer typisierten Datei }
  writeln;
  writeln('Eingabe der Intervallgrenzen :');
  write('t0 = '); readln(t0);
  write('t1 = '); readln(t1);
  write('Anzahl der Kurvenpunkte : '); readln(n);
  write('Name der zu erstellenden Datei ');
  write('<ohne .Typ>: ');
  readln(Name); Name:=Name+'.dat';
  writeln('Kompletter Dateiname ist  : ',Name);

  write('>> Abspeicherung von ',Name,' laeuft -->');
  dt := (t1-t0)/n;  t := t0;
  t1 := t1+ dt*0.01;      { Sicherheitsschranke }

  assign(Punkte,Name);
  rewrite(Punkte);  { Anlegen einer neuen Datei }
  while  t <= t1 do begin
      Kurve (t,x,y);
      write (Punkte,x,y);       { Abspeicherung }
      t := t + dt
  end;
  close(Punkte);          { Schliessen der Datei }
  writeln(' fertig !'); readln
end;
```

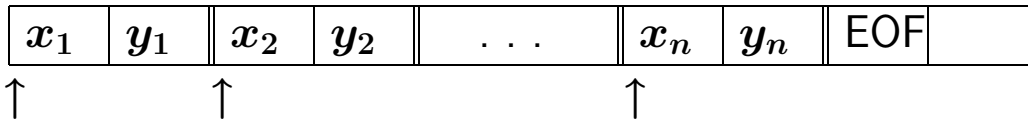
- Darstellung der Kurvenpunkte (fakultativ):
  Nach Lesen der Bildpunkte $(x_i, y_i),\ i = 1(1)n$ und Bestimmung der Extremalwerte $x_{min}, x_{max}, y_{min}, y_{max}$ ist die Kurve punktweise darzustellen (Bem.: aufwendig, plattform-abhängig, nicht zu empfehlen).

```
procedure Lesen_und_Zeichnen (var Punkte : singledatei);
  var   x,y,xmin,xmax,ymin,ymax,dx,dy  : single;
        i,io,l2,xx,yy                  : integer;
        Name                           : string[30];

  procedure Graphicinit;     { Initialisierung Graphik }
    var  gd,gm,err: integer;
    begin
      gd:=VGA; gm:=VGAhi;
      InitGraph(gd,gm,'O:\BP\BGI');  { ist anzupassen! }
      err:=graphresult;
      if err<>0 then
      begin
        write  ('Grafik konnte nicht initialisiert ');
        writeln('werden! (Fehler: ',err,')');
        readln;  Halt;
      end;
      setcolor(Yellow);
      setbkcolor(Blue);
    end;   { of Graphicinit }

  begin      { of Lesen_und_Zeichnen }
    repeat
      writeln;
      write('Name der zu lesenden Datei <ohne .Typ>: ');
      readln(Name);
      Name:=Name+'.dat';
      writeln('Kompletter Dateiname ist  : ',Name);
      assign(Punkte,Name);
      {$I-} reset(Punkte); {$I+}
      io := IOResult;
      if io<>0 then
        writeln('Datei exist. nicht od. falscher Name!')
    until io=0;

    l2 := filesize(Punkte) div 2;   {halbe Dateilaenge }
```

```
  if l2=0 then
  begin
    writeln('*** Datei enthaelt keine Punkte ! ***');
    delay(5000);
    close(Punkte)
  end;

  { ** Bestimmung der Extremwerte ** }
  read(Punkte,x,y);
  xmin:=x; xmax:=x; ymin:=y; ymax:=y;
  for i:=2 to l2 do
  begin
    read(Punkte,x,y);
    if x<xmin then xmin := x;
    if x>xmax then xmax := x;
    if y<ymin then ymin := y;
    if y>ymax then ymax := y
  end;   { Extremwerte gefunden }

  Graphicinit;    { VGA-Graphik : 640 X 480 Punkte }
  dx := (xmax-xmin)/620;
  dy := (ymax-ymin)/470;
  reset(Punkte);
  rectangle(0,0,639,479);
  for i:=1 to l2 do
  begin
    read(Punkte,x,y);
    { ** Transformation (x,y) --> (xx,yy) ** }
    xx := trunc((x-xmin)/dx + 10);
    yy := trunc(475 - (y-ymin)/dy);
    if (xx>=9) and (xx <= 629) and (yy >= 4) and
       (yy <= 476) then putpixel(xx,yy,LightMagenta)
  end;
  close(Punkte);
  readln;     { Graphikschirm anhalten! }
  closegraph
end;          { of Lesen_und_Zeichnen }
```

- Erzeugung einer Textdatei:

  Die typisierte Datei <Name>.dat ist in eine Textdatei <Name>.txt umzuwandeln und abzuspeichern. Jeder Kurvenpunkt (x_i, y_i) repräsentiere ein Feld des Typs kpunkt, so daß die gegebene Datei <Name>.dat vom Typ `file of kpunkt` ist:




```
{ ** Kopieren von typisierten Dateien <Name>.dat,
     die mit Schreiben(Punkte) erzeugt wurden,
     in Textdateien <Name>.txt zwecks weiterer
     Verarbeitung mit GNUPLOT, MAPLE und anderen
     Visualisierungsprogrammen ** }

procedure Abspeichern_als_Textdatei;
const  dim     = 2;
type   kpunkt  = array[1..dim] of single;
var    fsingle          : file of kpunkt;
       ftext            : Text;
       i,io,k           : integer;
       v                : kpunkt;
       s                : string[13];
       name,kname,gname : string[30];
begin
  writeln;
  repeat
    write('Name der Datei <ohne .Typ>: ');
    readln(name);
    kname:=name+'.dat';
    writeln('Kompletter Dateiname ist  : ',kname);
```

```
  assign(fsingle,kname);
  {$I-} reset(fsingle); {$I+}
  io:=ioresult;
  if io<>0 then
  begin
    writeln;
    write  ('Achtung: Datei kann nicht ');
    writeln('geöffnet werden !!!');
  end
until io=0;

gname:=name+'.txt';
writeln('Name der Textdatei wird: ',gname);
writeln;
assign(ftext,gname);
rewrite(ftext);
writeln(ftext,'#  2D-Kurvendatei  '+ gname);

write('>> Umspeicherung von ',kname,' laeuft -->');
for k:=1 to filesize(fsingle)-2 do
begin
  read(fsingle,v);
  for i:=1 to dim do begin
    str(v[i]:13,s);      { Elimination von  }
    delete(s,10,2);      { Mantissenstellen }
    write(ftext,s,' ')
  end;
  writeln(ftext)
end;  {of k}
writeln(' fertig !');
close(ftext);
```

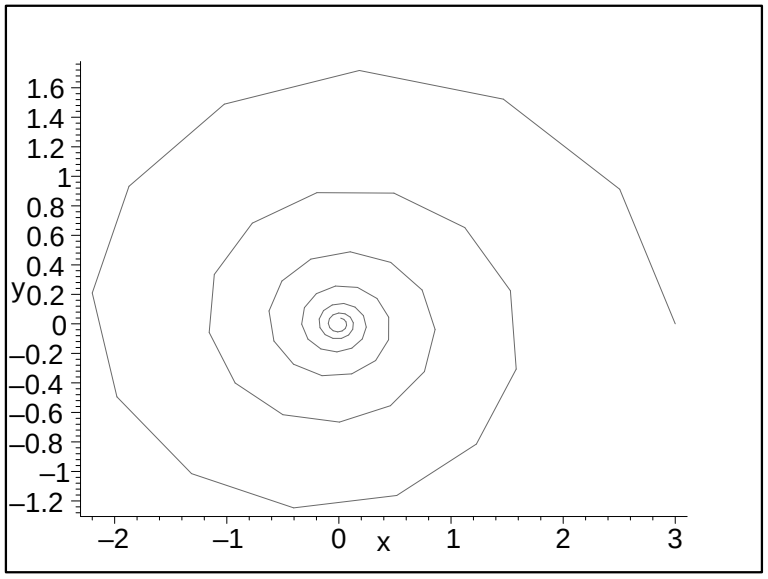
```
    writeln;
    writeln('Neue Datei ',gname,' ist erstellt !');
    writeln('Anzahl der Kurvenpunkte: ',
            filesize(fsingle)-2);
    close(fsingle);  readln
  end;  { of Abspeichern_als_Textdatei }


  begin  { *** Fortsetzung des Hauptprogrammes *** }
    repeat
      clrscr;
      writeln('Zeichnen von Kurven');
      writeln('====================');
      writeln;
      writeln('<S>chreiben einer Datei ?');
      writeln('<L>esen und Zeichnen einer Datei ?');
      writeln('<A>bspeichern als Textdatei ?');
      writeln('<E>nde des Programmes ?');
      writeln; write('Auswahl : '); readln(ch);
      ch := upcase(ch);
      case ch of
       'S' : Schreiben (Punkte);
       'L' : Lesen_und_Zeichnen (Punkte);
       'A' : Abspeichern_als_Textdatei
      end
    until ch = 'E'
  end.
```

Darstellung der Kurve mit Maple:

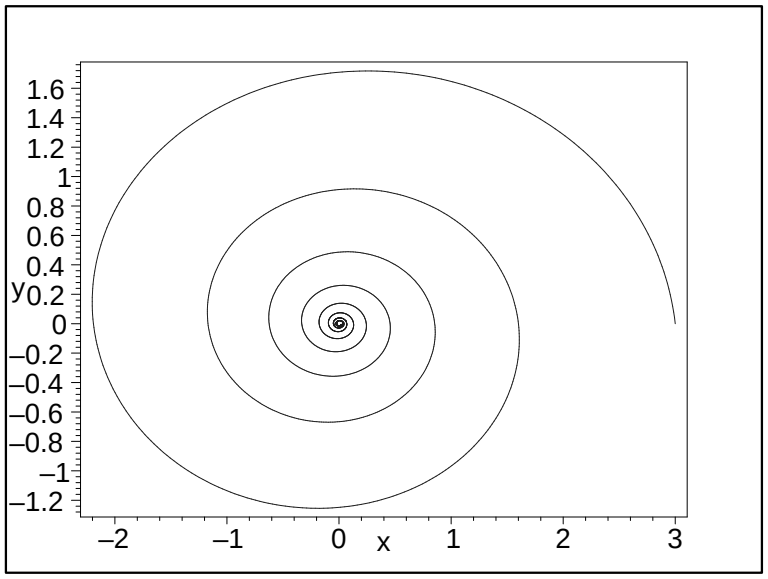
Die Textdatei `<Name>.txt` wird mittels `readdata` in die Variable `bsp1` eingelesen. Damit kann `pointplot` ein Grafikobjekt generieren, das mittels `display` dargestellt wird:

```
>  restart:
>  with(plots): # Laden des Grafik-Packages
>  # Lesen der Kurvendaten aus Textdatei
```

```
> bsp1 := readdata("Bsp1.txt",2):
> # Erzeugung des Grafikobjektes
> bsp1:=pointplot(bsp1,color=magenta, axes=FRAMED,
> style=LINE, thickness= 1, labels=[x,y]):
> display(bsp1);
```




```
> bsp2 := readdata("Bsp2.txt",2):
> bsp2:=pointplot(bsp2,color=blue, axes=BOXED, style=LINE,
> thickness= 2, labels=[x,y]):
> display(bsp2);
```



Kapitel 7

Modulare Programmierung

Die Entwicklung modularer Programmstrukturen erhöht die Universalität von Programmen (Nutzung allgemeiner Grundprinzipien, leichte Anwendbarkeit, Lauffähigkeit auf verschiedenen Computern, Einsetzbarkeit in andere Programme). Zur Bildung und Verarbeitung von Moduln wurden in Programmiersprachen syntaktische Einheiten geschaffen:

```
function, procedure, subroutine
module, unit, library, object, package
```

Ziel der Modularisierung: Eine weitgehende Unabhängigkeit des Algorithmus vom ausführenden Programm soll erreicht werden. Damit erhöhen sich Veränderbarkeit und Erweiterbarkeit des Gesamtprojektes.

Mittel der Modularisierung: Die Eingangs- und Ausgangsgrößen jedes Moduls sind genau zu definieren; globale Parameter sind zu vermeiden. Das methodische Prinzip der *Datenkapselung* sollte konsequent angewandt werden.

Definition 1 : Modul

Ein Modul ist ein unabhängiger Teilalgorithmus, bestehend aus

- einer (öffentlichen) Benutzerschnittstelle (interface) und
- einer (nicht-öffentlichen, privaten) Realisierung (implementation)

eines Algorithmus. Die Schnittstelle enthält

- eine Liste der bereitgestellten Objekte (die Exportliste) und evtl.
- eine Liste der zum Betrieb des Moduls vorausgesetzten anderen Moduln (die Importliste).

7.1 Deklaration von Moduln

In BP wurde das Konzept der Units und der dynamischen Linkbibliotheken (DLL) entwickelt. Eine **Unit** ist ein Modul, bestehend aus Interface und Implementation.

1. Im Interface können Deklarationen (Konstanten, Datentypen, Variablen, Prozeduren und Funktionen) exportiert sowie andere Units importiert werden.
2. Im Implementationsteil werden Algorithmen als Prozedur- und Funktionsblöcke definiert und ggf. Initialisierungen vorgenommen.
3. Units können separat compiliert, abgespeichert, bei Bedarf in ein Programm eingebunden und dort wie standardmäßige Programmobjekte benutzt werden.

Beispiele:

- Ansteuerung des Textbildschirmes
- Grafikfunktionen, -konstanten, -variablen und -typen
- Funktionen zur komplexen Arithmetik
- Funktionen zur Integralberechnung
- Funktionen zur Zeichenkettenverarbeitung usw.

4 Teile einer Unit:

- Unitkopf
- Interface
- Implementation
- Initialisierung

Beispiel 1 Operationen mit komplexen Zahlen

Für komplexe Zahlen $z = re + i * im$ mit Realteil re und Imaginärteil im sind die Addition AKZ und Multiplikation MKZ bereitzustellen.

```
Unit COMPL;    { Datei-Name : COMPL.PAS }

Interface
    type   complex = record
                       re,im : real;
                     end;
    procedure AKZ(x,y:complex; var z:complex);
    procedure MKZ(x,y:complex; var z:complex);

Implementation
    procedure AKZ;
      begin
        z.re := x.re + y.re;
        z.im := x.im + y.im
      end;
    procedure MKZ;   { 3 Multiplikationen ! }
      var h:real;
      begin
        h    := (x.re + x.im)*y.re;
        z.im := h - x.re*(y.re - y.im);
        z.re := h - x.im*(y.re + y.im)
      end;

end.
```

Unitkopf:

Die syntaktische Form ist `unit <Name>;` Der Dateiname muß mit dem Unitnamen übereinstimmen.

Interface-Teil:

1. Der Aufbau geschieht als Folge von Deklarationen; Abhängigkeiten sind dabei zu beachten.
2. Routinendeklarationen (Prozeduren, Funktionen) benötigen nur den Routinenkopf (Routinenblock erst im Implementationsteil!).
3. Routinendeklarationen wirken wie forward-Deklarationen, d.h. im Implementationsteil ist Bezug auf jede Routine möglich (rekursive Algorithmen!).
4. Gültigkeitsbereich der deklarierten Objekte ist der gesamte aufrufende Block (unter Beachtung der Sichtbarkeitsregel).
5. Das Interface kann selbst Units importieren - mittels uses-Anweisung zu Beginn (Importliste).
6. Das Interface kann leer sein.

Implementations-Teil:

1. Die Blöcke aller im Interface deklarierten Routinen (Prozeduren, Funktionen) sind anzugeben; die Parameterlisten können dagegen fehlen (s.Beispiel 1).
2. Im „privaten“ Teil der Unit sind Deklarationen möglich, die lokal, also für das aufrufende Programm unsichtbar sind.
3. Die Reihenfolge der Routinenimplementation ist beliebig (rekursive Algorithmen!).
4. Die Implementation - und damit die Unit - endet mit `end.`
5. Die Implementation kann leer sein.

Initialisierungen:

1. Im Implementationsteil darf nach den (privaten) Deklarationen auch ein Anweisungsteil `begin ... end` stehen. Damit können (öffentliche und private) Variablen initialisiert werden.
2. Ausführung stets nach den Deklarationen und vor den Anweisungen des aufrufenden Programmes.

Beispiel 2 Initialisierung von 3 Variablen

Eine Unit START soll die skalaren Variablen x (double) und c (char) sowie einen Vektor m (array) initialisieren.

```
Unit START;   { Dateiname : START.PAS }
Interface
    var  x : double;
         c : char;
         m : array[1..50]of integer;
Implementation
    var  i:integer;
    begin
      x := 1.223;
      c := '*';
      for i:=1 to 50 do  m[i] := i*i;
    end.
```

```
Program H_START;  { Dateiname : HSTART.PAS }
  uses START;
  var  j:integer;
  begin
    writeln;
    writeln('x  =  ',x);
    writeln('c  =  ',c);
    writeln;
    for j:=1 to 25 do writeln(m[j]);
    readln
  end.
```

Enthält eine Unit keine Routinen-Deklarationen, so kann der Implementationsteil leer sein; es entsteht eine reine „Deklarations-Unit“.

Beispiel 3 **Studenten-Datei**

Eine Unit STUD soll die benötigten Konstanten, Recordtypen und -variablen bereitstellen.

```
Unit STUD;                  { Dateiname : STUD.PAS }
Interface
  const  Fachzahl  =  25;   { max. Anzahl von Fächern }
  type   Faecher   =  array[1..Fachzahl] of string[30];
         Namen     =  record  Vorname     :  string[20];
                              Zuname      :  string[40];
                    end;
         Datum     =  record  Tag         :  1..31;
                              Monat       :  1..12;
                              Jahr        :  integer;
                    end;
         Adressen  =  record  PLZ         :  0..99999;
                              Ort         :  string[20];
                              Strasse     :  string[40];
                              Nr          :  1..999;
                    end;
         Studenten =  record  Matrikel_Nr :  integer;
                              Name        :  Namen;
                              Adresse     :  Adressen;
                              Geb_Datum   :  Datum;
                              Fach        :  Faecher;
                    end;

  var    MB_Student, TH_Studentin, Stipendiat : Studenten;
         Immatrikulation, Exmatrikulation     : Datum;
         Hauptwohnung, Nebenwohnung           : Adressen;

Implementation
End.
```

7.2 Aktivieren und Einbinden von Moduln

Das Aktivieren (Aufruf) von Units muß unmittelbar nach dem Programmkopf bzw. Unitkopf mittels einer uses-Anweisung erfolgen.

Syntax: ⟨Uses-Anweisung⟩ **::=** uses ⟨Bezeichner⟩**{,** ⟨Bezeichner⟩**}** **;**

Beispiele:
```
uses COMPL;
uses Crt, graph, STUD;
```

Semantik:

- Nutzt eine Unit B Bestandteile einer Unit A, so muß die Reihenfolge stets | uses A,B; | lauten.
- Im Interface einer Unit dürfen weitere Units mittels uses-Anweisung importiert werden („geschachtelte Units“).

Beispiel 1 **(s.o.) Operationen mit komplexen Zahlen**

Import der Unit COMPL in ein Hauptprogramm: Der Typ `complex` und die Prozeduren `AKZ, MKZ` stehen nun als „Standardobjekte“ zur Verfügung.

```
Unit COMPL;    { Datei-Name : COMPL.PAS }
Interface
    type   complex = record
                       re,im : real;
                     end;
    procedure AKZ(x,y:complex; var z:complex);
    procedure MKZ(x,y:complex; var z:complex);
Implementation
    . . . .
end.
```

```
Program H_COMPL;   { Datei-Name : HCOMPL.PAS }

  uses Crt, COMPL;

  var a,b,c:complex;
  begin
    clrscr;
    readln (a.re,a.im);
    readln( b.re,b.im);
    AKZ (a,b,c);
    writeln ('Summe   = ',c.re,'+',c.im,'*i');
    MKZ (a,b,c);
    writeln ('Produkt = ',c.re,'+',c.im,'*i');
  end.
```

Bedeutung der Reihenfolge bei abhängigen Units

Trifft der Compiler auf eine uses-Anweisung, so

- trägt er die Interface-Deklarationen in seine Symboltabelle ein und
- stellt dem Programmverbinder (Linker) den Maschinencode des Implementationsteils zur Verfügung.

Units werden in derjenigen Reihenfolge in die Symboltabelle eingefügt, in der sie im Quelltext angegeben sind (Reihenfolge beachten!).

Beispiel 4 **Geschachtelte Units**

Die Gleichung $f(x) = e^{-x} - x = 0$ ist mit dem Newton-Verfahren

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}\,, \quad i = 0, 1, 2, 3, \ldots$$

zu lösen. Einzugeben sind der Startwert x_0 , die gewünschte Genauigkeit $\varepsilon > 0$ und die maximale Iterationszahl i_{max} .

Für diese und andere mathematische Verfahren ist folgende Hierarchie von Units üblich:

- **Unit A:** beschreibt Funktionen, z.B. $\boldsymbol{y = f(x)}$ **,** die sich häufig ändern können
- **Unit B:** beschreibt mathematische Verfahren (Gleichungslösung, Optimierung, Integration,...), in der die Funktion $\boldsymbol{f(x)}$ benutzt wird $\Longrightarrow$ Import der Unit A
- **Programm:** beschreibt die Ein- und Ausgabe (mit Fehlerkontrolle) und ruft das Verfahren auf $\Longrightarrow$ Import der Unit B

```
Unit FU;                    { Datei-Name : FU.PAS }
Interface
    function f (x:double):double;
    function f1(x:double):double;
Implementation
  function f;
    begin
      f := exp(-x) - x;
    end;
  function f1;
    var  delta : double;
    begin
    { Analytisch exakte 1. Ableitung:
      f1:= -exp(-x) - 1.0; }
    { Differenzenapproximation: }
     delta := 1E-5 * (abs(x) + 1.0);
     f1 := (f(x+delta) - f(x))/ delta;
    end;
end.   { of Unit FU }
```

```
Unit NEWT;                { Datei-Name : NEWT.PAS }
Interface

  uses FU;

  function Zero (x0,eps : double;
           itmax : integer) : double;

Implementation
  function Zero;
    var x, d : double;
        it   : integer;
    begin
      x:=x0; it:=0;
      repeat
        inc(it);
        if it>itmax then begin
        writeln('Iteration konvergiert nicht nach',
                    itmax, ' Schritten !');
          readln; Halt
        end;
        d := f(x)/f1(x);
        x := x - d
      until abs(d)/(abs(x)+1.0) < eps;
      Zero := x
    end;
end.   { of Unit NEWT }
```

```
Program Newton_Test;          { Datei: H_NEWT.PAS }

  uses CRT, NEWT;

  var  eps, x0, y : double;
     itmax, i   : integer;
  begin
    clrscr;
    writeln('Newton-Test-Programm');
    writeln('====================');
    write('Startwert   x0    = ');
    readln(x0);
    repeat
      write('Genauigkeit eps   = ');
      readln(eps);
      write('itmax (0 -> Ende) = ');
      readln(itmax);
      y := Zero(x0,eps,itmax);
      writeln('Ergebnis  y = ',y);
      readln
    until itmax <=0;
  end.  { of Program  H_NEWT }
```

Abhängigkeiten der Units:

Fazit:

FU wird von Unit NEWT *direkt vorausgesetzt* und muß deshalb per uses-Anweisung importiert werden; analoges gilt für NEWT und das Hauptprogramm. Die Beziehung zwischen FU und H_NEWT dagegen ist *indirekt* - sie wird beim Übersetzen vom Linker automatisch berücksichtigt!

Einbinden von Moduln

Ablauf der Programmübersetzung *ohne Unit*:

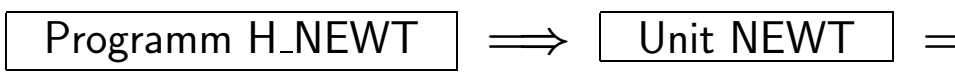
0. Ausgangs-Dateien: Programm-Quelltext (PAS-Datei)
1. Übersetzen (Compilieren) liefert das Compilat (relokativer Code, nicht-ausführbare Datei)
2. Verbinden (Linken) liefert den Maschinencode (Objektcode, ausführbare Datei, EXE-Datei)
3. Ausführen (Run) der EXE-Datei möglich

Ablauf der Programmübersetzung *mit Units*:

0. Ausgangs-Dateien: Programm-, Unit-Quelltexte (PAS-Dateien)
1. Übersetzen (Compilieren)
   - speichert den Objektcode der Units unter <Name>.TPU als TPU-Dateien ab
   - stellt die Deklarationen des Interface den aufrufenden Programmen/Units zur Verfügung
   - liefert das Compilat (relokativer Code, nicht-ausführbare Datei)
2. Verbinden (Linken) fügt die benötigten Teile aus den übersetzten Units <Name>.TPU in das Compilat ein und liefert den Maschinencode (Objektcode, ausführbare Datei, EXE-Datei) des Hauptprogrammes
3. Ausführen (Run) der (vollständig verbundenen) EXE-Datei möglich

Die MAKE–Funktion des Compilers

Bestehen zwischen den Units eines Projektes *Abhängigkeiten*, z.B.



so werden diese bei der Übersetzungs-Reihenfolge berücksichtigt. BP stellt 2 Optionen für das Compilieren bereit:

1. Projekt neu erstellen (Build-Funktion, FP: Rebuild all)

   Der im Editor befindliche Quelltext wird übersetzt. Importierte Units werden *prinzipiell neu übersetzt* und automatish eingebunden, falls eine Quelltext-Version existiert (Andernfalls wird nach der TPU-Datei gesucht und diese sofort eingebunden). Anwendung empfiehlt sich, wenn eine komplette Neuübersetzung erforderlich wird.

2. Projekt aktualisieren (Make-Funktion, FP: Compile)

   Der im Editor befindliche Quelltext wird übersetzt. Importierte Units werden *unter Berücksichtigung* der Quelltext-Datei `<Unitname>.PAS` und der compilierten Unit `<Unitname>.TPU` eingebunden:

   (a) Wird nur `<Unitname>.TPU` gefunden, so wird diese sofort eingebunden.

   (b) Wird nur `<Unitname>.PAS` gefunden, so wird diese automatisch in `<Unitname>.TPU` übersetzt, abgespeichert und eingebunden.

   (c) Sind sowohl `<Unitname>.PAS` als auch `<Unitname>.TPU` vorhanden, so wird das Veränderungsdatum und die Verändereungszeit der beiden Dateien verglichen und die neuere von beiden gemäß der Punkte (a) bzw. (b) verarbeitet (Bei gleicher Erstellungszeit gemäß (a)).

Auswirkung der Veränderung einer Unit:

Bei der Übersetzung mit MAKE werden Abhängigkeiten zwischen den verwendeten Units erkannt. Damit vermeidet man zeitaufwendige Neuübersetzungen des gesamten Projektes.

Beispiel 4 (s.o.) Newton-Verfahren

Nimmt man eine Veränderung an der Funktion $y = f(x)$ in Unit FU vor, so möchte man die Neuübersetzung gern auf diese Unit beschränken. Man muß dazu 2 Fälle unterscheiden:

1. Veränderung des Implementations- und Initialisierungsteils von FU.PAS:

   Sie kann isoliert betrachtet werden, d.h MAKE generiert lediglich (automatisch) eine neue Version von FU.TPU. Dagegen bleibt NEWT.TPU unverändert.

2. Veränderung des Interfaceteils von FU.PAS:

   Danach müssen auch alle diejenigen Units neu übersetzt werden, die FU *direkt benutzen*, d.h. NEWT.PAS wird ebenfalls neu übersetzt.

Fazit:

- Häufig zu verändernde Projektteile sollten in separaten Units (vgl. FU) notiert werden.
- Gut erprobte umfangreiche (mathematische) Algorithmen sollten in Units (vgl. NEWT) zusammengefaßt werden, um den Übersetzungsaufwand minimal zu halten.
- Das Hauptprogramm sollte sich auf ein absolutes Minimum an Deklarationen und Funktions-/Prozeduraufrufen beschränken.

7.3 Standard-Moduln

Für die Gliederung von DV-Projekten in Moduln (Modularisierung) mittels Units und/oder DLLs (dynamische Linkbibliotheken) spechen insbesondere folgende Gründe:

1. Verringerung der Übersetzungszeiten umfangreicher Projekte:

   Leichte Änderbarkeit einzelner Programmteile, ohne den Zwang, das Gesamtprojekt neu zu übersetzen

2. Mögliche Gliederung umfangreicher Prokekte:

   (a) Separate Übersetzung der Moduln bietet die Möglichkeit der separaten Testung auf syntaktische Richtigkeit von Moduln

   (b) Zusammenfassung inhaltlich zusammengehörender Routinen in Moduln (Bibliotheken) dienen dem modularen Programmaufbau

   (c) Nach Definition der Schnittstellen der Moduln ist relativ unabhängige Programmentwicklung möglich (Teamarbeit)

3. Übergabe von Objektcode:

   Selbst entwickelte Verfahren können an Nutzer im Objektcode übergeben werden (Copyright!); lediglich die Schnittstellen-Beschreibung ist im Quelltext bereitzustellen.

4. Leistungsfähige vorgefertigte Units:

   Standard-Units werden in einer Bibliothek (TPL-Dateien) bereitgestellt und erweitern somit beträchtlich den Sprachumfang.

Einige Standard-Units

- System (keine Einbindung mittels **uses**-Anweisung nötig)
- Dos (BP: WinDos)
- Crt (BP: WinCrt)
- Graph

Nachfolgende Beispiele demonstrieren die Anwendung der vordefinierten Units **Dos** und (in BP) **WinCrt**.

Beispiel 5 Auflistung von Dateinamen eines Types

Die Namen aller Dateien des aktuellen Verzeichnisses mit dem Typ PAS sollen aufgelistet werden.

Die Unit **Dos** enthält u.a. folgende Programmobjekte:

- FindFirst : Prozedur zum Feststellen des ersten Auftretens einer Datei im Verzeichnis
- FindNext : Prozedur zum Feststellen des nächsten Auftretens einer Datei im Verzeichnis nach erfolgtem `FindFirst`
- SearchRec : Record mit Informationen zu einer Datei im Ergebnis von `FindFirst` bzw. `FindNext` mit folgender Struktur:

```
SearchRec = record
  Fill       : array [1..2] of byte;
  Attr       : byte;
  Time, Size : longint;
  Name       : string[12];
end;
```

- AnyFile : Standardkonstante für Dateiattribute, wenn unter allen möglichen Dateien gesucht werden soll;
  weitere Standardkonstanten: ReadOnly, SysFile, Archive,...
- DosError : Variable, die nach jedem FindFirst bzw. FindNext einen Fehlercode erhält (0 = Suche war erfolgreich, sonst $\neq \mathbf{0}$)

```
Program PAS_DAT;
  uses Crt,Dos;                { Standard-Units }
  var  Such_Ergebnis : SearchRec;
       i             : integer;
  begin
    clrscr; i := 0;    { Zaehler der Dateien }
    writeln('PAS-Dateien der akt. Directory');
    writeln('==============================');
    FindFirst ('*.PAS',AnyFile,Such_Ergebnis);
    while DosError = 0 do
    begin
      inc(i);
      writeln(Such_Ergebnis.Name);
      FindNext(Such_Ergebnis)
    end;
    writeln('--------------');
    writeln('Anzahl der Pas-Dateien : ',i);
    readln
  end.
```

```
**********************************
PAS-Dateien der aktuellen Directory
==================================
FU.PAS
H_NEWT.PAS
NEWT.PAS
PAS_DAT.PAS
--------------
Anzahl der Pas-Dateien : 4
```

7.4 Dynamische Linkbibliotheken (DLL)

Moderne Betriebssysteme gestatten die gleichzeitige Verarbeitung mehrerer Programme, das Multitasking. Dabei tritt häufig der Fall ein, daß eine Unit in mehreren Programmen gleichzeitig benutzt wird. Da Units bereits beim Linken in die EXE-Datei gebunden werden, befindet sich folglich deren Maschinencode zu ein und demselben Zeitpunkt *mehrfach im Speicher* (Skizze).

Dynamische Linkbibliotheken

(DLL, Dynamic Link Libraries, load-time linking)

- sind *übersetzte und ausführbare* Moduln,
- die aus Daten (Konstanten, Variablen, Typen) sowie Routinen (Funktionen, Prozeduren) und weiteren Ressourcen (Icons, Menüs, Dialoge etc.) bestehen können.
- Die Routinen können exportiert und zur Laufzeit in Applikationen eingebunden werden
- und damit gleichzeitig von mehreren Programmen zur Laufzeit - also dynamisch - genutzt werden (Skizze).

Vorteile von DLL's gegenüber Units:

1. Routinen sind nicht Bestandteil der EXE-Dateien $\Rightarrow$ wesentlich kürzerer Programmcode
2. Routinen/Ressourcen stehen nur einmal im Speicher (auch bei mehreren Programm-Instanzen) $\Rightarrow$ Speicherökonomie !
3. Änderungen in einer DLL ohne Veränderung der Aufrufparameter der Routinen $\Rightarrow$ keine Neuübersetzung der Hauptprogramme nötig
4. DLL's können in Programme importiert werden, die in einer anderen Programmiersprache erstellt wurden $\Rightarrow$ gemischtsprachige Projekte möglich (Kompatibilität der Datentypen beachten!)

Beispiel einer DLL: Die Grafiktreiber von Windows sind eine DLL, die von allen Windows-Programmen als Dienstleistung benutzt werden kann. So existiert nur eine Kopie von GDI.EXE im Speicher.

Aufbau einer DLL

1. DLL-Kopf: `library <DLL-Name>;`
   Der Dateiname muß mit dem DLL-Namen übereinstimmen.
2. Uses-Anweisungen: nur bei importierten Units
3. Definition der Routinen und evtl. der Daten; zu exportierende Routinen sind mit `export` zu übersetzen
4. Export-Liste (exports): Liste der zu exportierenden Routinen

Beispiel 7 **DLL für mathematische Funktionen (FP)**

```
{Beispielcode fuer die DLL Mathedll }
library Mathedll;

  function tan(alpha : double):double; export;
  begin   { Tangensfunktion }
    tan := sin(alpha) / cos(alpha);
  end;

  function sh(alpha : double):double; export;
  begin   { Hyperbolische Sinusfunktion }
    sh := 0.5*(exp(alpha)- exp(-alpha));
  end;
```

```
  function power (x,p : double):double; export;
    {  Potenzfunktion  fuer  x hoch p }
    var  pint : integer;
         phil : double;
    function frac (zahl : double):double;
      begin
        frac := zahl - (trunc(zahl));
      end;
    begin  { of power }
      if x=0 then power := 0;
      if x>0 then power := exp(p*ln(x));
      if (x<0) and (frac(p)=0) then begin
        pint := trunc(p);
        phil := exp(pint * ln(abs(x)));
        if odd(pint) then power := (-1)*phil
                     else power := phil;
      end;
    end;  { of power }

exports
  tan     Index 1,
  power   Index 2,
  sh      Index 3;

begin
end.
```

Exports-Liste: Sie kann an beliebiger Stelle der DLL stehen. Die Angabe eines Index (schnellste Zugriffsweise!) ist nicht erforderlich. Fehlt der Index, so wird automatisch ein entsprechender Integer-Wert zugewiesen: `exports tan,power,sh;`). Statt Indizes können auch Namen (String-Konstanten) zugewiesen werden (langsamere Variante wegen des Suchprozesses).

Import von DLL-Routinen

Damit eine Routine aus einer DLL aufgerufen (importiert) werden kann, muß sie mittels einer `external`-Deklaration im aufrufenden Modul (Programm, Unit) bekanntgemacht werden. Das kann über den Namen, einen (neuen) Alias-Namen oder über einen Index geschehen. Beim Import über den Index (s.o.) muß nicht in der *name table* gesucht werden $\Rightarrow$ schnellster Zugriff.

Interface-Unit:
Für den Import empfiehlt sich eine Unit, die alle später benötigten Routinen der DLL importiert und damit für die Anwendungen einfach aufrufbar macht (Skizze). Im Interface stehen die Routinenköpfe, während im Implementationsteil diese Routinen mit den Routinen der DLL über den Index verbunden werden („Interface-Unit“). Die External-Anweisung signalisiert dem Compiler, daß die Definition der Routine in einer externen Bibliothek vorliegt.

Beispiel 7 **(s.o.) Unit zur Mathedll**

```
{ Unit fuer die Mathedll }
Unit Mathe;
Interface
  function power (x,p : double):double;
  function tan (alpha : double):double;
Implementation
  function tan;    external 'Mathedll' index 1;
  function power;  external 'Mathedll' index 2;
begin
end.
```

Beispiel 7 (s.o.) Applikation zur DLL

```
{ Beispielprogramm fuer die Mathedll }
Program  Mathedll_Demo;
uses  Mathe;
Var   x,p : double;
begin
  repeat
    write('Basis     x =  '); readln(x);
    write('Exponent  p =  '); readln(p);
    writeln('Potenz   z  =  ', power(x,p):14:8);
    writeln('Tangens  t  =  ', tan(x):15);
  until (x=0) and (p=0);
end.
```

Import in Applikationen (in BP)

Die Windows-Ressourcen stellen zahlreiche Routinen und Datentypen zur Verwaltung von DLL's bereit. Wesentlichste Beispiele sind:

- THandle : Zeigertyp („Griff“) auf einen Modul (Unit, DLL)
- LoadLibrary : Funktion zum Laden eines Bibliotheksmoduls
- FreeLibrary : Freigabe des Speichers eines Bibliotheksmoduls

7.5 Mathematische Modul-Bibliotheken

Die Lösung natur– und ingenieurwissenschaftlicher Probleme, z.B.die
– Lösung linearer und nichtlinearer Gleichungssysteme,
– Berechnung einfacher und mehrfacher Integrale,
– Lösung gewöhnlicher und partieller Differentialgleichungen,
erfordert *mathematische Näherungsverfahren*. Diese liegen in Form von Modul-Bibliotheken (Pakete, Toolboxen) vor.

Prinzipielle Struktur

1. Programme $\Rightarrow$ eingeschränkte Datenkommunikation
2. Unterprogramme (Subroutinen, Funktionen, Prozeduren)
   $\Rightarrow$ gute Datenkommunikation, aber Vielzahl von UPs
3. Moduln (Units, Modules, Klassen, DLLs, Packages etc.)
   $\Rightarrow$ Modularität und Überschaubarkeit

Klassifikation von Modul-Bibliotheken

1. Textbuch-Bibliotheken (Programmsammlungen):
   - Auflistung unabhängiger Grundverfahren im Quelltext
   - als Begleit-CD zum Buch bzw. per FTP erhältlich
   - Begleitmaterial (Companion Websites)

2. „Klassische“ Universalbibliotheken in FORTRAN und C:
   - NAG – Numerical Algorithms Group
   - IMSL – International Mathematical Software Library

3. Strukturierte Modul-Bibliotheken („Toolboxen“):
   Preiswerte Sammlung gegliederter Moduln;
   Sprachen: FORTRAN, PASCAL, ANSI-C, C++

Kapitel 8

Rekursive Algorithmen

Rekursion stellt eines der grundlegendsten algorithmischen Konzepte dar. Sie tritt deshalb in der Programmierung häufig auf, wobei 2 Erscheinungsformen zu unterscheiden sind:

- Rekursive Algorithmen: rekursive Funktionen, rekursive Prozeduren
- Rekursive Datenstrukturen: Listen (Ketten, Stapel etc.), Bäume, Graphen (Geflechte)

Der induktive Aufbau der natürlichen Zahlen $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ durch die PEANOschen Axiome ermöglicht außer dem Beweisprinzip der vollständigen Induktion auch rekursive Definitionen von Abbildungen.

Rekursive Definition

Eine Abbildung $f : \mathbb{N} \longrightarrow W$ wird *rekursiv definiert*, indem man

(i) $f(0)$ explizit festlegt und

(ii) $f(n)$ für beliebiges $n \in \mathbb{N}_{\geq 1}$ in Abhängigkeit von $f(n-1)$ definiert, d.h. auf $f(n-1)$ zurückführt.

Damit ist die Abbildung f für alle $n \in \mathbb{N}$ definiert.

Beispiel 1 **Fakultät** (Hier ist $f : \mathbb{N} \longrightarrow \mathbb{N}$)

$$0! = 1\,, \quad n! = n \cdot (n-1)! \quad \text{für } n \geq 1,\ n \in \mathbb{N}$$

Beispiel 2 **Unbestimmtes Integral** ($f : \mathbb{N} \to C^\infty(\mathbb{R})$)

$$\int x^n e^{ax}\,dx = \begin{cases} \frac{1}{a}e^{ax} & \text{falls} \quad n = 0 \\ \frac{1}{a}x^n e^{ax} - \frac{n}{a}\int x^{n-1}e^{ax}\,dx & \text{falls} \quad n \geq 1 \end{cases}$$

Verallgemeinerte Rekursion

Außer in der Grundform kommen rekursive Definitionen in zahlreichen Varianten vor: Häufig wird $f(n)$ nicht nur auf $f(n-1)$ zurückgeführt, sondern auf $f(n')$ mit beliebigem $n' < n$. Mitunter wird $f(n)$ auf mehrere Vorgänger zurückgeführt, z.B. auf $f(n_1),\ f(n_2)$ mit $n_1 < n,\ n_2 < n$.

Beispiel 3 **Fibonacci-Zahlen**

$$u_n = \begin{cases} 1 & \text{falls} \quad n = 1,\ 2 \\ u_{n-1} + u_{n-2} & \text{falls} \quad n \geq 3 \end{cases}$$

Die nicht-rekursive Definition ist bedeutend schwieriger:

$$u_n = \frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right\}, \quad n \geq 1$$

Beispiel 4 **Größter gemeinsamer Teiler**

Sei $m, n \in \mathbb{N}$ mit $m \geq n$. $GGT(m,n)$ ist die größte ganze Zahl $l \in \mathbb{N}$ mit $l|m$ und $l|n$. Grundlage des Euklidischen Algorithmus ist die folgende rekursive Definition (Begründung später):

$$GGT(m,n) = \begin{cases} m & \text{falls} \quad n = 0 \\ GGT(n, m \bmod n) & \text{falls} \quad n \geq 1 \end{cases}$$

8.1 Rekursive Funktionen

Rekursiver Aufruf

(i) Ein *rekursiver Aufruf* einer Funktion oder Prozedur $\boldsymbol{A}$ liegt vor, wenn das Unterprogramm vor Beendigung der Abarbeitung seines Anweisungsteils erneut aufgerufen wird.

(ii) *Direkte Rekursivität:* Das Unterprogramm $\boldsymbol{A}$ ruft sich während seiner Abarbeitung selbst (direkt) auf.

(iii) *Indirekte Rekursivität:* $\boldsymbol{A}$ ruft sich über mindestens ein weiteres Unterprogramm $\boldsymbol{B}$ auf.

Direkte Rekursivität

Beispiel 1 **Fakultät als PASCAL-Funktion**

```
function FAK(n : longint) : longint;
  { Bestimmung von  n! fuer  n >=0 }
  begin
    if n = 0 then  FAK := 1
             else  FAK := n*FAK(n - 1)
  end;
```

Beispiel 2 **Unbestimmtes Integral als MAPLE-Funktion**

```
integral := proc(n::nonnegint)
  if n=0 then 1/a * exp(a*x) else
     1/a*x^n * exp(a*x) - n/a* integral(n-1)
  fi;
  simplify(%) + C     # Integrationskonstante
end:
```

Beispiel 3 Fibonacci-Zahlen als PASCAL-Funktion

```
function FIB(n : integer) : integer;
  { Fibonacci - Zahlen ;  n >= 1 }
  begin
    if (n = 1) or (n = 2) then FIB := 1 else
       FIB := FIB(n-1) + FIB(n-2)
  end;
```

Beispiel 4 GGT als PASCAL-Funktion

```
function GGT(m,n : integer) : integer;
  { Groesster gemeinsamer Teiler von m und n ;
      m > 0  und  n > 0 }
  begin
    if n=0 then GGT := m
           else GGT := GGT(n,m mod n)
  end;
```

Beispiel 4 GCD als MAPLE-Funktion

```
# GCD zweier natürlicher Zahlen m und n
GCD := proc(m,n)
  if n=0 then m
         else GCD(n,irem(m,n))
  fi              # irem - Interger remainder
end:
```

```
# GCD zweier Polynome m(x)  und  n(x)
PolynomGCD := proc(m::polynom,n::polynom,x::name)
    if n=0 then m
           else PolynomGCD(n,rem(m,n,x),x)
    fi            # rem - Polynomial Remainder
end:
```

Frage: Können diese Vereinbarungen nicht nur als (mathematische) Definitionen, sondern tatsächlich als (ausführbare) Algorithmen angesehen werden? Wie verhält sich der Computer bei der Abarbeitung von `FIB(15)` oder `FAK(2)` ?

Beispiel 1 **Fakultät $n!$**

```
Program H_FAK;  {$N+}
  var  m : longint;

function FAK(n : longint) : longint;
  { Bestimmung von  n! fuer  n >=0 }
  begin
    if n = 0 then  FAK := 1
             else  FAK := n*FAK(n - 1)
  end;

begin
  repeat
    write('m = '); readln(m);
    writeln('f = ',fak(m)); writeln
  until m=0; readln
end.
```

Abarbeitungsprinzip rekursiver Funktionen

Bei der Abarbeitung von FAK(2) werden die Parameter wie üblich behandelt, d.h. Wertparameter und lokale Parameter werden im Stack abgelegt (Stapelung, Kellerung), Variablenparameter und globale Parameter dagegen nicht. Das heißt insbesondere:

1. Mit jedem Aufruf von FAK wird eine neue *Version (Inkarnation)* von FAK erzeugt.

   $$\text{FAK(2)} \longrightarrow \text{FAK(1)} \longrightarrow \text{FAK(0)}$$

   sind die Inkarnationen zu $\boldsymbol{n = 2}$ **.**

2. Zu jeder Inkarnation können Wert- und lokale Parameter gehören, deren *Gültigkeitsbereich* genau der Block dieser Inkarnation ist. Bei erneutem Aufruf von FAK werden sie dann unsichtbar.

3. Diese Parameter $\boldsymbol{n, n-1, n-2, \ldots, 2, 1, 0}$ **,** werden im Stack abgelegt, bis die Rekursion *abbricht (terminiert)* und erstmalig der Wert FAK $\boldsymbol{:= 1}$ berechnet wird.

4. Die Inkarnationen werden nun sukzessive in umgekehrter Reihenfolge abgeschlossen:

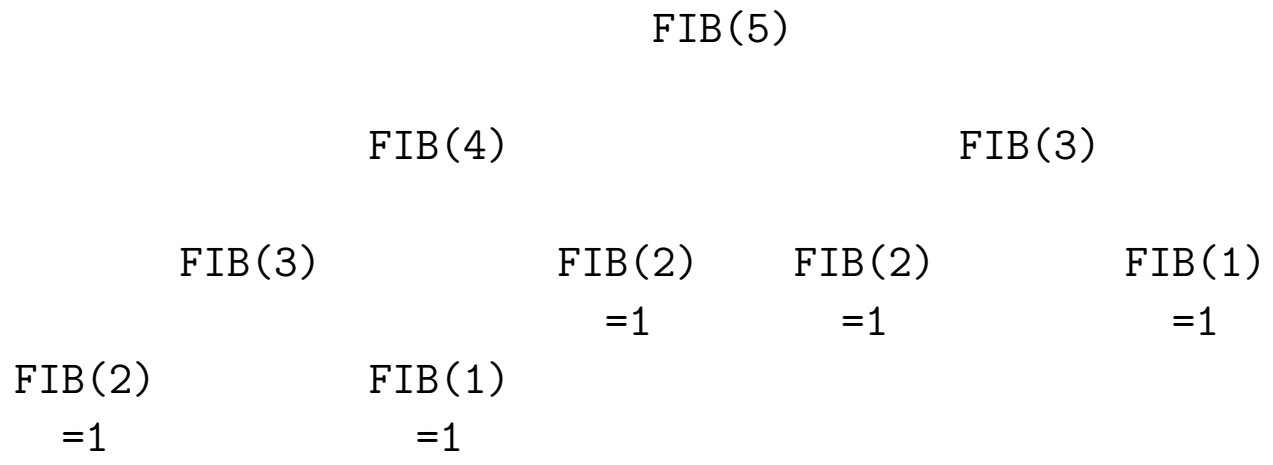
   $$\text{FAK(0)} \longrightarrow \text{FAK(1)} \longrightarrow \text{FAK(2)},$$

   wobei die gesperrten (gekellerten) Werte sukzessive sichtbar und verarbeitet werden.

Fazit: Die Ablage und Verarbeitung auf dem Stack erfolgt nach dem automatischen Speicherprinzip (vgl. Speicherklassen und Blockstruktur).

Beispiel 3 Fibonacci-Zahlen

Ist die Anweisung `m := FIB(5)` abzuarbeiten, so lauten die Inkarnationen:



Die rekursive Funktion FIB ist äußerst ineffektiv, da Inkarnationen mehrfach (besser: vielfach!) auftreten, und der Rechenaufwand exponentiell mit n zunimmt.

Lineare und nichtlineare Rekursion

(i) Ist $\boldsymbol{F}$ eine Funktion und $\boldsymbol{A}$ eine Anweisung, so heißt $\boldsymbol{A}$ *linear in* $\boldsymbol{F}$, wenn gilt:

1. Ist $\boldsymbol{A}$ keine bedingte Anweisung, so enthält $\boldsymbol{A}$ höchstens einen Aufruf von $\boldsymbol{F}$.
2. Ist $\boldsymbol{A}$ eine bedingte Anweisung der Form

```
if B1 then A1 else
  if B2 then A2 else
    . . . . . .
      if B{n} then A{n} else A{n+1};
```

oder eine äquivalente case-Anweisung, so sind alle Anweisungen `A1,A2,...,A{n+1}` linear in $\boldsymbol{F}$.

(ii) Eine rekursive Funktion $\boldsymbol{F}$ mit Anweisungsteil $\boldsymbol{A}$ heißt *linear-rekursiv*, wenn $\boldsymbol{A}$ linear in $\boldsymbol{F}$ ist. Andernfalls heißt sie *nichtlinear-rekursiv*.

Beispiele: Die Funktionen FAK (Beispiel 1) und GGT (Beispiel 4) sind linear-rekursiv, wogegen FIB (Beispiel 3) nichtlinear-rekursiv ist.

Binärer Baum (Bintree)

$\boldsymbol{M}$ sei eine Menge (die Knotenmenge). Dann wird die Menge $\boldsymbol{B_2(M)}$ der *Binärbäume über* $\boldsymbol{M}$ rekursiv definiert:

(i) $\varepsilon \in \boldsymbol{B_2(M)}$ mit ε - leerer Baum

(ii) Ist $\boldsymbol{a \in M}$ und $\boldsymbol{x, y \in B_2(M)}$, so ist auch $\boldsymbol{(a, x, y) \in B_2(M)}$.

Der Knoten $\boldsymbol{a}$ heißt Wurzel, $\boldsymbol{x}$ linker Teilbaum (Unterbaum), $\boldsymbol{y}$ rechter Teilbaum (Unterbaum) des Binärbaumes $\boldsymbol{(a, x, y)}$. Ein Binärbaum $\boldsymbol{(a, \varepsilon, \varepsilon)}$ heißt Blatt.

Folgerung:
Der zu einem Aufruf der Funktion $\boldsymbol{F}$ aus der Menge $\boldsymbol{M}$ aller möglichen Inkarnationen durch die Rekursionsvorschrift erzeugte binäre Baum heißt *Rekursionsbaum*.

Gehen von einer Wurzel $\boldsymbol{a}$ genau $\boldsymbol{p}$ Unterbäume aus, so spricht man von p-adischen Bäumen (z.B. triadische Bäume bei $\boldsymbol{p = 3}$). Damit verallgemeinert man die Definition:

p-adischer Baum

$\boldsymbol{M}$ sei eine Menge (die Knotenmenge). Dann wird die Menge $\boldsymbol{B_p(M)}$ der *p-adischen Bäume über* $\boldsymbol{M}$ rekursiv definiert:

(i) $\varepsilon \in \boldsymbol{B_p(M)}$ mit ε - leerer Baum

(ii) Ist $\boldsymbol{a \in M}$ und $\boldsymbol{x_1, x_2, \ldots, x_p \in B_p(M)}$, so auch $\boldsymbol{(a, x_1, x_2, \ldots, x_p) \in B_p(M)}$.

$\boldsymbol{x_k}$ heißt k-ter Teilbaum (Unterbaum) des Baumes $\boldsymbol{(a, x_1, x_2, \ldots, x_p)}$. Ein Baum $\boldsymbol{(a, \varepsilon, \ldots, \varepsilon)}$ heißt Blatt.

Folgerungen:

1. Binärbäume sind die dyadischen Bäume über M.
2. Seien $p_1, p_2, \ldots, p_r$ die Anzahlen von Unterbäumen, die in einem beliebigen Baum auftreten und $p := \max_{1=1(1)r} p_i$. Ergänzt man die Zahl der Unterbäume in jeder Wurzel durch leere Bäume zur Zahl p, so ist nun der Baum p-adisch.
3. Für $p = 1$ ergibt sich der Spezialfall der Folge (Sequenz).
4. Jede rekursive Funktion F generiert einen p-adischen Rekursionsbaum. F ist genau dann linear-rekursiv, falls $p = 1$ ist; andernfalls ist F nichtlinear-rekursiv.
5. Als *Rekursionstiefe* einer Inkarnation a bezeichnet man die Anzahl der Teilbäume, in denen a enthalten ist.

Terminiertheit rekursiver Funktionen

Eine Funktion F *terminiert* für eine gegebene Parameterbelegung V, wenn die Bestimmung von $F(V)$ nach endlich vielen Aufrufen (Inkarnationen) von F einen definierten Wert ergibt.

Beispiel 1 **Fakultät n!**

`FAK(n)` terminiert für jedes $n \in \mathbb{N}$, denn die Inkarnationen lauten

`FAK(n), FAK(n-1),...,FAK(0),`

womit wegen der Terminierungsbedingung `FAK(0) := 1` die Aufrufe beendet werden. Dagegen terminiert offenbar die Funktion

```
function FAK(n : longint) : longint;
  begin
    if n = 0 then  FAK := 1
             else  FAK := n*FAK(n + 1)
  end;
```

nicht für $n \geq 1$, da die Terminierungsbedingung nicht erreicht wird.

Beispiel 5 Collatz-Funktion

Für die Funktion

$$\boldsymbol{WAS}(n) = \begin{cases} \mathbf{1} & \text{falls} \quad \boldsymbol{n} = \mathbf{0}, \mathbf{1} \\ \boldsymbol{WAS}(\mathbf{3}\boldsymbol{n} + \mathbf{1}) & \text{falls} \quad \boldsymbol{n} \geq \mathbf{2},\ \boldsymbol{n} \text{ ungerade} \\ \boldsymbol{WAS}(\boldsymbol{n}/\mathbf{2}) & \text{falls} \quad \boldsymbol{n} \geq \mathbf{2},\ \boldsymbol{n} \text{ gerade} \end{cases}$$

ist die Terminierungsfrage nichttrivial!

```
function WAS(n : longint) : longint;
  begin
    if (n = 0) or (n = 1) then  WAS :=1  else
      if (n > 1) and (n mod 2 = 1) then
        WAS := WAS(3 * n + 1)  else
        WAS := WAS(n div 2)
  end;
```

Beispiel 4 Größter gemeinsamer Teiler

```
function GGT(m,n : integer) : integer;
  begin
    if n=0 then GGT := m
           else GGT := GGT(n,m mod n)
  end;
```

Diese Funktion ist für jedes $\boldsymbol{m}, \boldsymbol{n} \in \mathbb{N}$ terminiert (Übungsaufgabe).

Korrektheit rekursiver Funktionen

Trotz nachgewiesener Terminiertheit einer rekursiven Funktion $\boldsymbol{F}$ ist damit i. allg. noch nicht verifiziert, daß $\boldsymbol{F}$ gerade die gewünschte Abbildung darstellt.

Beispiele 1,3 und 4

- `FAK(n)`
  liefert offenbar das Produkt $n(n-1)\cdots 2\cdot 1 = n!$, damit ist die Korrektheit leicht nachweisbar.
- `FIB(n)`
  gibt die übliche mathematische Definition der Fibonacci-Zahlen an.
- `GGT(m,n)`
  gibt nicht die algebraische Definition des größten gemeinsamen Teilers an (s.o.). Dennoch gilt: Die Funktion `GGT(m,n)` ist korrekt, d.h. sie liefert den größten gemeinsamen Teiler von m und n (Beweis als Übungsaufgabe).

8.2 Entwurfsprinzipien rekursiver Prozeduren

Die für rekursive Funktionen eingeführten Begriffe (direkte und indirekte Rekursivität, Inkarnation, Rekursionsbaum, lineare und nichtlineare Rekursion, Terminiertheit, Korrektheit) treffen auch auf Prozeduren zu.

Beispiel 6 **n-fache Laufanweisung**

Gegeben seien die m Mengen ganzer Zahlen ($m \leq 100$)

$$M_i = \{k \in \mathbb{Z} \mid a_i \leq k \leq b_i\}, \quad i = 1(1)m$$

mit $a_i, b_i \in \mathbb{Z}$. Die Menge $M = M_1 \times M_2 \times \cdots \times M_m$ aller m-Tupel ist zu erzeugen und auszugeben.

- Beispiel: Ist $a_i = 1$, $b_i = n$, $i = 1(1)m$, so liegen Variationen von n Elementen der Klasse m mit Wiederholung vor.
- Globale Objekte: Sei die maximale Elementezahl `Max = 100 >= m`. Globale Felder sind `a[1..Max]`, `b[1..Max]`, `c[1..Max]`. Die Prozedur `N_FOR(n)` liefert den Vektor `C` mit `C[i]` $\in M_i$.

```
Program Variationen;
Uses   Crt;
Const  Max = 100;            { max. Elementezahl }
Var  m,i   : Integer;
     a,b,c : Array[1..max] of Integer; { global }

Procedure Ausgabe;
  Var i:Integer;
  Begin
    For i:= 1 to m do Write(c[i]:5);
    Writeln
  End;   { of Ausgabe }

Procedure N_FOR(n : Integer);
  { n-fach geschachtelte Laufanweisung,  n >= 1 }
  Var  i : Integer;           { lokale Variable }
  Begin
    If n=0 then Ausgabe          { Terminierung }
    else
      For i:=a[n] to b[n] do begin
        c[n]:=i; N_FOR(n-1)
      end
  End;   { of N_FOR }

Begin    { Hauptprogramm }
  ClrScr; Write('m = '); ReadLn(m); Writeln;
  For i:=1 to m do begin a[i]:=1; b[i]:=2 end;
  N_FOR(m); ReadLn
End.
```

Bemerkungen zur Abarbeitung:

1. Wegen $a_i = 1,\ b_i = 2$ ist der Rekursionsbaum ein ausbalancierter Binärbaum (Skizze). Für den Aufruf `N_FOR(4);` ist die maximale Rekursionstiefe 5, denn die Terminierung erfolgt erst für die Inkarnation `N_FOR(0)`.
2. Durch Änderungen und Ergänzungen können mit `N_FOR` auch Permutationen und Kombinationen gebildet werden (Übung).
3. Nur die Variablen n (Wertparameter) und i (lokale Variable) werden gestapelt, die globalen Variablen m,i,a,b,c dagegen nicht!
4. Nur im Falle $a[i] = b[i],\ i = 1(1)m$ ist `N_FOR` linear-rekursiv; andernfalls ist `N_FOR` stets nichtlinear-rekursiv.

Entwurfsprinzipien rekursiver Algorithmen

Während sich bei rekursiven Funktionen die Gestalt der PASCAL-function unmitelbar ableiten läßt (vgl. FAK, FIB, GGT, WAS), ist für umfangreiche Algorithmen oft erheblicher gedanklicher Aufwand nötig, um einen korrekten – und möglichst kurzen – rekursiven Algorithmus zu entwerfen. Folgende allgemeine *Entwurfsprinzipien* liegen jedoch den meisten der rekursiven Algorithmen zugrunde:

- Einbettungsprinzip
- Prinzip „Teile und herrsche“ (Divide et impera)
- Backtracking-Prinzip (Trial and error)

Prinzip der Einbettung

Die Größe, nach der die Rekursion verläuft, kommt im Algorithmus selbst häufig nicht vor, so daß die gestellte Aufgabe $\boldsymbol{A}$ nicht rekursiv angebbar ist.

Idee: Eine allgemeinere Aufgabe $\boldsymbol{A' = A(n)}$, die $\boldsymbol{A}$ als Spezialfall enthält, läßt sich aber oft rekursiv lösen, wenn ein zusätzlicher (Rekursions-)Parameter $\boldsymbol{n}$ eingeführt wird, d.h $\boldsymbol{A}$ wird in $\boldsymbol{A'}$ „eingebettet".

Beispiel 7 Türme von HANOI

Als der Tempel von Hanoi gebaut wurde, wurden 3 große Stangen in den Boden gerammt und 64 Scheiben abnehmenden Durchmessers auf die erste Stange gesteckt - mit der kleinsten an der Spitze und der größten am Boden. Die Tempelmönche mußten die Scheiben ununterbrochen von Stange 1 zu Stange 3 unter Benutzung der Stange 2 und Beachtung der folgenden 2 Regeln bewegen:

1. Nur 1 Scheibe darf jedesmal bewegt werden.
2. Keine Scheibe darf auf eine kleinere Scheibe gelegt werden.

Laut Überlieferung wrid, wenn sie die Aufgabe bewältigt haben, das Ende der Welt eintreten.

Einbettungsprinzip:

- Einfacher Algorithmus $\boldsymbol{A}$: „Lege 64 Scheiben von 1 auf 3 mit Hilfsstab 2" $\Rightarrow$ keine Parameter
- Verallgemeinerter Algorithmus $\boldsymbol{A'}$: „Lege n Scheiben von x auf y mit Hilfsstab z" $\Rightarrow$ Parameter sind nun x,y,z,n; n ist der „Rekursionsparameter"

Grundform des Algorithmus `HANOI(x,y,z,n)`

1. Lege n-1 Scheiben von x auf z mit Hilfe von y !
2. Lege 1 Scheibe von x auf y (Ausgabe) !
3. Lege n-1 Scheiben von z auf y mit Hilfe von x !

Terminierungsbedingung:

„Falls keine Scheibe auf x liegt, so fertig!"

```
Program Tuerme_von_Hanoi;
  uses CRT;
  var  n : integer;

  procedure HANOI(x,y,z,n : integer);
  { Umlegen eines Turmes mit n Scheiben; n>=1 }
    begin
      if n > 0 then begin
        HANOI(x,z,y,n-1);
        Writeln('Lege von ',x,' auf ',y,' !');
        HANOI(z,y,x,n-1)
      end
    end;       { of HANOI }

  begin
    ClrScr;
    Writeln('Tuerme von Hanoi');
    Write('Scheibenzahl  n = '); Readln(n);
    Writeln;
    HANOI(1,3,2,n);
    Writeln('Fertig !!!'); Readln
  End.
```

Anzahl der Umlegungen:

Offenbar ist $\boldsymbol{T(0) = 0}$ und

$$\boldsymbol{T(n) = T(n-1) + 1 + T(n-1),\ n \geq 1\ .}$$

Induktiv zeigt man daraus $\boldsymbol{T(n) = 2^n - 1}$ (Übung).

Prinzip „Teile und herrsche“

Die Entwurfsstrategie empfiehlt

1. die Zerlegung eines Problems $\boldsymbol{P_0(n)}$ der Größe $\boldsymbol{n}$ in mehrere Teilprobleme geringerer Größe $\boldsymbol{P_1(n'),\ n' < n,}$
2. die jeweils gelöst werden
3. und anschließend zu einer Lösung des Grundproblems zusammengesetzt werden.

Mit jedem Teilproblem wird analog verfahren, bis man zu Teilproblemen gelangt, die klein genug sind, um ohne weitere Teilung gelöst zu werden. Häufigster Spezialfall: Zerlegung von $\boldsymbol{P_0(n)}$ in *2 Teilprobleme* $\boldsymbol{P_1(n_1)}$ und $\boldsymbol{P_2(n_2)}$ mit $\boldsymbol{n_1 + n_2 = n}$ **.**

Typische Anwendungen:

- Schnelle Suchverfahren, Sortierverfahren
- Schnelle Fouriertransformation (FFT)
- Adaptive Integrationsverfahren
- Parallelisierte Verfahren (Numerik, Visualisierung) etc.

Die entstandenen Verfahren gehören zu den effektivsten Algorithmen!

8.3 Komplexität und Sortierverfahren

Der Zeitbedarf für die Abarbeitung eines (korrekten) Algorithmus ist ein wesentliches Kriterium für seine Güte, denn ein Algorithmus mit kürzerer Laufzeit ist stets vorzuziehen. Dabei ist man an Aussagen interessiert, die unabhängig von der Geschwindigkeit des ausführenden Computers sind. Gesucht ist die sog. *Zeitkomplexität* des Algorithmus.

Grundoperationen:

Es wird angenommen, daß alle Grundoperationen von derselben Komplexität sind, d.h. sie sind mit demselben Zeitaufwand durchzuführen. „Grundoperationen“ können z.B. sein:

- Arithmetische Operationen (Festpunkt, Gleitpunkt)
- Vegleichsoperationen (Zahlen, Zeichen, -ketten)
- Lese- und Speicheroperationen (Records, Dateien).

Laufzeit und Problemgröße:

Da mit wachsender Problemgröße die Zeitkomplexität zunimmt, ist man an einer Darstellung in Abhängigkeit von dieser Problemgröße interessiert. Man nimmt an, daß die Problemgröße durch eine einzige Zahl $\boldsymbol{n}$ beschrieben werden kann. Die Laufzeit $\boldsymbol{T(n)}$ ist dann die Gesamtzahl von Operationen bei gegebener Problemgröße.

Dominanter Term:

Um die Komplexität zu erhalten, geht man von großen Problemen aus und bestimmt die für $\boldsymbol{n \longrightarrow \infty}$ dominierenden Terme. Falls $\boldsymbol{T(n)}$ additiv aufgebaut ist, so ist dies i.allg. ein einzelner Term $\boldsymbol{D(n)}$.

Beispiele:

- Fakultät (Bsp. 1):
  Grundoperationen sind longint-Multiplikationen, womit sich $\boldsymbol{T(n) = n-1}$ ergibt. Der dominante Term ist $\boldsymbol{D(n) = n}$.
- Gaußscher Algorithmus zur Lösung eines linearen Gleichungssystems mit n Unbekannten (Bsp. s.o.):
  Grundoperationen sind die 4 Gleitpunkt-Operationen, womit

$$\boldsymbol{T(n) = \frac{2n^3}{3} + \frac{3n^2}{2} - \frac{7n}{6} \quad \Longrightarrow \quad D(n) = \frac{2}{3}n^3}$$

Ordnungssymbol und Zeitkomplexität

(i) Eine Funktion $\boldsymbol{T : \mathbb{N} \longrightarrow \mathbb{R}}$ ist von der *Ordnung* $\boldsymbol{\mathcal{O}(f(n))}$, falls ein $\boldsymbol{n_0 \in \mathbb{N}}$ und ein $\boldsymbol{C > 0}$ existieren, so daß für alle $\boldsymbol{n \in \mathbb{N},\ n \geq n_0}$ gilt:

$$|T(n)| \ \leq \ C \cdot |f(n)| \ .$$

Die Schreibweise mit Landau-Symbol lautet dann

$$T(n) = \mathcal{O}(f(n)) \ .$$

(ii) Gilt für den Algorithmus $\boldsymbol{A}$ die Laufzeit $\boldsymbol{T(n) = \mathcal{O}(f(n))}$, so gibt $\boldsymbol{f(n)}$ die *Zeitkomplexität des Algorithmus* $\boldsymbol{A}$ an.

(iii) Ist $\boldsymbol{f(n) = n^p,\ p \in \mathbb{N}}$, so heißt der Algorithmus *polynomial beschränkt*; ist aber $\boldsymbol{f(n) = a^n,\ a \in \mathbb{R}_{>0}}$, so ist $\boldsymbol{A}$ *exponentiell beschränkt*.

Beispiele:

- Fakultät (Bsp. 1): Wegen $\boldsymbol{T_1(n) = n - 1 = \mathcal{O}(n)}$ ist der Algorithmus polynomial beschränkt.
- Gaußscher Algorithmus (Bsp. s.o.):

$$T_2(n) = \frac{2n^3}{3} + \frac{3n^2}{2} - \frac{7n}{6} \quad = \mathcal{O}(n^3) \ .$$

- Türme von Hanoi (Bsp.7): Wegen $\boldsymbol{T_3(n) = 2^n - 1 = \mathcal{O}(2^n)}$ ist der Algorithmus nicht polynomial beschränkt.
- Potenzierung $\boldsymbol{x^n,\ n \in \mathbb{N}}$: Bei einfacher Aufmultiplikation ist $\boldsymbol{T(n) = n - 1}$. Jedoch ist eine Zeitkomplexität

$$T_4(n) = \mathcal{O}(\log_2 n)$$

möglich (Übungsaufgabe).

Komplexität von Sortierverfahren

Das Sortieren großer Datenbestände ist eine häufige Aufgabe der Informationsverarbeitung. Dafür wurden schnelle Verfahren entwickelt. Grundoperation ist dabei die Vergleichsoperation.
Ein Feld $A = \{a_1, a_2, a_3, \ldots, a_n\}$ von n Elementen eines Datentyps (Elementtypes) M heißt *sortiert (geordnet)*, falls

$$a_1 \leq a_2 \leq a_3 \leq \ldots \leq a_n$$

mit einer Ordnungsrelation $\leq$ in M gilt.

Beispiele für Datentypen M :

- $M_1 =$ reelle Zahlen mit der $\leq$-Relation
- $M_2 =$ Zeichen mit der $\leq$-Relation gemäß dem ASCII-Code
- $M_2 =$ Zeichenketten mit der $\leq$-Relation gemäß der lexikografischen Ordnung

1.Verfahren Sortieren durch Einfügen (insertion)

A' entsteht aus A in folgenden Schritten:

(i) a_1 wird als erstes Element in A' aufgenommen.
(ii) Für $i = 2(1)n$ wird a_i so in A' eingefügt, daß A' ein sortiertes Feld der Länge i ist.

Laufzeit und Wertung: Die Maximalzahl der Vergleiche beträgt

$$T_1(n) = 1 + 2 + 3 + \ldots + (n-1) = \frac{1}{2}n(n-1) \approx n^2/2$$

("worst case"). Durchschnittlich sind - bei Annahme einer Gleichverteilung der a_i nur $T_1(n) \approx n^2/4$ ("average case") Operationen nötig. Die Zeitkomplexität ist damit in jedem Falle $T_1(n) = \mathcal{O}(n^2)$. Allerding sind viele Umspeicherungen erforderlich (Programm als Übung).

2.Verfahren Sortieren durch Auswahl (selection)

A' entsteht aus A, indem n-mal das kleinste Element in A bestimmt wird, aus A entfernt und zum Feld A' als letztes Element hinzugefügt wird.

Laufzeit und Wertung: Die Zahl der Vergleiche beträgt in jedem Falle

$$T_1(n) = 1 + 2 + 3 + \ldots + (n-1) = \frac{1}{2}n(n-1) \approx n^2/2 = \mathcal{O}(n^2)$$

(im "worst case"und im "average case"). Allerdings muß nun kein Hilfsfeld benutzt werden, es kann mit weniger Umspeicherungen „auf Platz" sortiert werden.

3.Verfahren Sortieren durch Mischen (Mergesort)

A' entsteht aus A indem

(i) A in n Folgen $A_1 = \{a_1\}, A_2 = \{a_2\}, \ldots, A_n = \{a_n\}$ der Länge 1 zerlegt und

(ii) so oft ein Zusammenfügen (Mischen) wiederholt wird, bis nur eine Folge der Länge n übrigbleibt.

Rekursiver Algorithmus:

1. Nach dem Teile-und-herrsche-Prinzip wird das Ausgangsfeld `a[l],a[l+1],...,a[r]` in zwei möglichst gleichlange Felder *geteilt:*

   `a[l],a[l+1],...,a[m]` und `a[m+1],a[m+2],...,a[r]` .

2. Diese Felder werden getrennt sortiert - nach demselben Verfahren.

3. Anschließend werden sie gemischt (zusammengefügt), d.h. in sortierter Reihenfolge auf ein zweites Feld kopiert.

4. Die Rekursion *terminiert*, wenn $l = m$ und $m + 1 = r$ wird.

```
Program Sortieren_durch_Mischen;
uses CRT;
const  n_max   = 100000;    { Maximale Elementezahl  }
type   element = integer; { Daten des Typs Element }
       vektor  = array[1..n_max] of element;
var    i,n : integer;
       a   : vektor;

procedure Mergesort(var a:vektor; l,r:integer);
  { Sortieren der Feldelemente a[l],a[l+1],...,a[r]
    in aufsteigender Anordnung }
  var  b : vektor;   { Hilfsvektor }

  procedure SORTIERE(l,r:integer; var x,y:vektor);
    var  i,j,k,m : integer;  { lokale Parameter }
    begin
      { Divide ! ... }
      m := (l+r) div 2;
      if l < m   then  SORTIERE(l,m,y,x);
      if m+1 < r then  SORTIERE(m+1,r,y,x);
      { ... et impera !  --> Zusammenfuegen }
      i := l; j := m+1; k := l;
      while (i <= m) and (j <= r) do
        if x[i] < x[j] then begin
                              y[k] := x[i];
                              inc(k); inc(i)
                            end
                       else begin
                              y[k] := x[j];
                              inc(k); inc(j)
                            end;
```

```
      while  i <= m do begin
                          y[k] := x[i];
                          inc(k); inc(i)
                        end;
      while  j <= r do begin
                          y[k] := x[j];
                          inc(k); inc(j)
                        end
    end;            { of SORTIERE }

  begin
    b := a;
    SORTIERE(l,r,b,a);
  end;               { of Mergesort }

begin    { Hauptprogramm zum Rechenzeittest }
  Clrscr;
  writeln('Sortieren mit Mergesort');
  writeln('======================');
  repeat
    writeln;
    write('Anzahl n = '); readln(n);
    for i := 1 to n do
      a[i] := random(1000);
    write('Start des Sortierens --> <CR> druecken !');
    readln;
    Mergesort(a,1,n);
    writeln('Sortieren beendet !!!')
  until n <= 1 ;
  readln
end.
```

Laufzeit und Wertung: Die Zeitkomplexität beträgt im "average case"

$$T_3(n) = \mathcal{O}(n \log_2 n) \, .$$

Allerdings sind viele Umspeicherungen erforderlich (Übung).

4.Verfahren Sortieren durch Partitionieren (Quicksort)

A' entsteht aus A folgendermaßen:

(i) Für $n \leq 2$ wird höchstens 1 Vertauschung ausgeführt.

(ii) Für $n > 2$ wird A so zu einer Elementefolge

$$A_1, \{a^*\}, A_2 \qquad \text{mit} \quad a^* \in A$$

umgeordnet (geteilt), daß gilt:

(1) $a_i \leq a^*$ für $a_i \in A_1$,

(2) $a_i > a^*$ für $a_i \in A_2$.

Anschließend werden A_1 und A_2 entsprechend sortiert.

Beispiel 8 Schnelles Sortieren von Namen

Nach kompletter Eingabe von maximal 1000 Namen sind diese lexikografisch zu sortieren und anschließend in sortierter Reihenfolge auszugeben.

```
Program Sortierung_von_Namen_durch_Partitionierung;
  uses  crt;
  const maxlaenge = 25;        { max. Namenslaenge }
        maxanzahl = 1000;      { max. Namensanzahl }
  type  element   = string[maxlaenge];
        vektor    = array[1..maxanzahl] of element;
  var   i,n : integer;   a : vektor;
```

```
  procedure QUICKSORT(var a:vektor; l,r:integer);
    { Sortieren der Elemente  a[l],a[l+1],...,a[r]
      in aufsteigender Anordnung }
    var  i,j : integer;   g : element;
    begin
      i := l;  j := r;  g := a[i];
      { Divide ! ...  -> Teilen des Feldes }
      repeat
        while (a[j] >= g) and (i < j) do  dec(j);
        a[i] := a[j];
        while (a[i] <= g) and (i < j) do  inc(i);
        a[j] := a[i]
      until i = j;
      a[i] := g;
      { ... et impera !  -> Sortieren der Teile }
      if l < i-1  then  QUICKSORT(a,l,i-1);
      if i+1 < r  then  QUICKSORT(a,i+1,r)
    end;          { of QUICKSORT }

begin         { Hauptprogramm }
  ClrScr;
  writeln('Sortieren mit Quicksort');
  writeln('======================');
  write('Anzahl  = '); readln(n); writeln;
  writeln('Unsortierte Folge :');
  for i := 1 to n do
  begin write('Name[',i,']  =  '); readln(a[i]) end; writeln;

  QUICKSORT(a,1,n);

  writeln('Sortierte Folge :');
  for i := 1 to n do writeln('Name[',i,']  =  ',a[i]);
  readln
end.
```

Laufzeit und Wertung: Quicksort benötigt zum vollständigen Sortieren

- im ungünstigsten Fall ("worst case")

$$T_4(n) = \frac{1}{2}n(n-1) \approx n^2/2$$

- im günstigsten Fall nur $T_4(n) = n \log_2 n$
- durchschnittlich - bei Annahme einer Gleichverteilung der a_i im "average case" - jedoch auch nur $T_4(n) = 1.4n \log_2 n$ Operationen.

Satz über den Sortieraufwand:
Jedes Verfahren, das vollständig sortiert, benötigt im Mittel bei Gleichverteilungsannahme (im "average case") mindestens

$$T(n) \geq C \cdot n \log_2 n\,, \qquad C = \text{ const}$$

Vergleiche. □

Weitere Sortierverfahren sind *Shell-Sort, Heap-Sort, Quicker-Sort, Clever-Quicksort* etc. (vgl. Literatur).

8.4 Rekursion im wissenschaftlichen Rechnen

Zur näerungsweisen Berechnung einfacher und mehrfacher bestimmter Integrale bis auf eine vorgegebene Genauigkeit $\varepsilon > 0$ (adaptive Integration) ist die Notation rekursiver Prozeduren von Vorteil.

Beispiel 9 **Adaptive Trapezformel**

Das Integral $s = \int_a^b f(x)\,dx$ soll bis auf eine vorgegebene Genauigkeit $\varepsilon > 0$ bestimmt werden.

Problemanalyse:

1. Um das Integral über $[x_1, x_2]$ zu bestimmen, stehen zur Verfügung:
   - Werte $x_1, f_1 = f(x_1), x_2, f_2 = f(x_2), h$
   - Trapezwert $I = \frac{1}{2}h(f_1 + f_2)$
2. Zuerst wird $[x_1, x_2]$ halbiert („teile...“) und x_m, f_m sowie : $h := h/2$ bestimmt. Damit können die 2 Teilwerte
$$I_1 = \frac{1}{2}h(f_1 + f_m) \quad \text{und} \quad I_2 = \frac{1}{2}h(f_m + f_2)$$
ermittelt werden sowie der „bessere“ Integralwert $I_{neu} = I_1 + I_2$.
3. Terminierungsbedingung: Ist $|I - I_{neu}| < \varepsilon * (|I| + \varepsilon)$ (Absolut-Relativfehler), so kann I_{neu} akzeptiert und zum bisherigen Integralwert $s(x_1) = \int_a^{x_1} f(x)dx$ addiert werden.
4. Andernfalls stehen für
   - $[x_1, x_m]$ die Werte f_1, f_m, h, I_1 und für
   - $[x_m, x_2]$ die Werte f_m, f_2, h, I_2

   zur Verfügung, um rekursiv den Algorithmus für jedes der beiden Teilintervalle ausführen zu können.

```
procedure INTEGRAL(x1,x2,f1,f2,I,h : double);
{ Bestimmtes Integral mit adaptiver Trapezregel }
  var  I1,I2,I12,xm,fm : double;
  begin
    xm  := (x1 + x2) * 0.5;
    fm  := f(xm);  h := 0.5 * h;
    I1  := 0.5 * h * (f1 + fm);
    I2  := 0.5 * h * (fm + f2);
    I12 := I1 + I2;
    if abs(I - I12) < eps*(abs(I) + eps)
       then  s := s + I12
       else begin
              INTEGRAL(x1,xm,f1,fm,I1,h);
              INTEGRAL(xm,x2,fm,f2,I2,h)
            end
  end;    { of INTEGRAL }
```

Wertung:

- Beim Verfahren INTEGRAL wird - im Gegensatz zum schrittweisen Vorgehen - kein Funktionswert doppelt berechnet.
- Anfangsintervall der Rekursion ist nachfolgend $[a, b]$; günstig ist mitunter eine kleinere Anfangsschrittweite $h0$.

Anwendung: Man bestimme das Integral

$$s = \int_{-100}^{100} \frac{dx}{10^{-4} + x^2} = 314.13926535904599051253 1...$$

bis auf eine Genauigkeit $\varepsilon = 10^{-4}$.

```
Program Bestimmtes_Integral;
  uses CRT;
  var  a,b,eps,fa,fb,h,s,I : double;

  function f(x:double) : double;
    { Definition des Integranden }
    begin
      f := 1.0/(1E-4 + sqr(x));
    end;

  procedure INTEGRAL(x1,x2,f1,f2,I,h:double);
  { Bestimmtes Integral mit adaptiver Trapezregel }
```

```
var  I1,I2,I12,xm,fm : double;
begin
  xm  := (x1 + x2) * 0.5;
  fm  := f(xm);  h := 0.5 * h;
  I1  := 0.5 * h * (f1 + fm);
  I2  := 0.5 * h * (fm + f2);
  I12 := I1 + I2;
  if abs(I - I12) < eps*(abs(I) + eps)
      then  s := s + I12
      else begin
              INTEGRAL(x1,xm,f1,fm,I1,h);
              INTEGRAL(xm,x2,fm,f2,I2,h)
           end
end;    { of INTEGRAL }
```

```
begin   { Hauptprogramm }
  ClrScr;
  writeln('Bestimmtes Integral ueber f(x)');
  writeln('============================');
  writeln;
  write('Untere Grenze  a  = '); readln(a);
  write('Obere Grenze   b  = '); readln(b);
  write('Genauigkeit  eps  = '); readln(eps);
  writeln;
  { Vorbereitung des Prozeduraufrufes }
  fa := f(a); fb := f(b);
  h := b - a;  s := 0;
  I := 0.5 * h * (fa + fb);
  INTEGRAL(a,b,fa,fb,I,h);
  writeln('Integralwert  s  = ',s);
  readln
end.
```

Resultate: (FW - Funktionswerte, RT - Rekursionstiefe)

ε	FW	s	RT
1E-1	89	314.778 004	15
1E-2	297	314.376 312 9	18
1E-3	969	314.153 587 7	19
1E-4	3087	314.142 155 7	21
	$s =$	314.139 265...	

Weitere Anwendungen rekursiver Prozeduren

- Doppelintegrale über einem ebenen dreieckigen Gebiet $\boldsymbol{\Delta}$

$$\boldsymbol{S_2 = \int\int_{\Delta} f(x,y)\,dx\,dy}$$

können ähnlich wie in Beispiel 9 *adaptiv* berechnet werden (Übung). Mittels Triangulierung können damit Integrale über beliebigen polygonal berandeten Gebieten ermittelt werden. Analog dazu ist auch die Berechnung von Dreifachintegralen

$$\boldsymbol{S_3 = \int\int\int_{\Delta} f(x,y,z)\,dx\,dy\,dz}$$

über tetraederförmigen Gebieten möglich.

- Das Entwurfsprinzip "Backtracking" (Zurückverfolgung, trial and error) findet - neben den bekannten Labyrinth- und Schachbrett-Problemen - auch in der Mathematik Anwendung zum Zeichnen *selbstähnlicher Kurven* (Hilbert-Kurve, Koch-Kurve etc.) und bei Optimierungsverfahren.

- Indirekte Rekursivität tritt besonders bei der Programmierung von Formelparsern auf (vgl. Beispiel 7 in Kap. 4.3).

Beispiel 10 Symbolische Differentiation

Eine MAPLE-Funktion $\boldsymbol{deriv(f,x)}$ ist zu entwickeln, die algebraische Funktionen $\boldsymbol{y = f(x)}$ mit Summen- und Produktregel differenzieren kann. Andernfalls soll eine Fehlermeldung erfolgen.

Zusatzaufgabe: Man veranschauliche Rekursionsbäume und Stackbelegung anhand der angeführten Beispiele.

```
> deriv := proc(a::algebraic, x::name)
    local u,v;
    if not depends(a,x) then 0           # Konstante
    elif type(a,name) then               # Variable
      if a=x then 1 else 0 fi
    elif type(a,`+`) then map(deriv,a,x) # Summenregel
    elif type(a,`*`) then                # Produktregel
       u:=op(1,a); v:=a/u; deriv(u,x)*v+deriv(v,x)*u
    else ERROR(`: kann`,a,`nicht differenzieren!`)
    fi;
  end;
```

$$deriv := \mathbf{proc}(a::algebraic,\ x::name)$$
$$\quad \mathbf{local}\ u,\ v;$$
$$\quad\quad \mathbf{if}\ \mathbf{not}\ \mathrm{depends}(a,\ x)\ \mathbf{then}\ 0$$
$$\quad\quad \mathbf{elif}\ \mathrm{type}(a,\ name)\ \mathbf{then}\ \mathbf{if}\ a = x\ \mathbf{then}\ 1\ \mathbf{else}\ 0\ \mathbf{fi}$$
$$\quad\quad \mathbf{elif}\ \mathrm{type}(a,\ `+`)\ \mathbf{then}\ \mathrm{map}(deriv,\ a,\ x)$$
$$\quad\quad \mathbf{elif}\ \mathrm{type}(a,\ `*`)\ \mathbf{then}\ u := \mathrm{op}(1,\ a)\,;\ v := a/u\,;\ \mathrm{deriv}(u,\ x) \times v + \mathrm{deriv}(v,\ x) \times u$$
$$\quad\quad \mathbf{else}\ \mathrm{ERROR}(`:\ kann`,\ a,\ `nicht\ differenzieren!`)$$
$$\quad\quad \mathbf{fi}$$
$$\quad \mathbf{end}$$

```
> deriv(2-x,x);
                                -1
> deriv(x+y-2.3+Pi*y,y);
                              1 + Pi
> deriv(x*deriv(x*2,x)+3,x);
                                 2
> deriv(3*x-(x-4)*(6*x+1)*x*(x+4),x);
     3 - (6 x + 1) x (x + 4) - (6 x (x + 4) + (2 x + 4) (6 x + 1)) (x - 4)
> simplify(%);
                        19 - 24 x^3 - 3 x^2 + 192 x
> deriv(5*x^4+3,x);
Error, (in deriv) : kann, x^4, nicht differenzieren!
```

Kapitel 9

Algorithmen auf dynamischen Datenstrukturen

Die behandelten Datentypen `array, file` sind wenig geeignet, wenn folgende häufig auftretende Aufgabe zu lösen ist:

> „Gegeben ist eine sortierte Folge von Datenelementen. Weitere Elemente sollen einsortiert (eingefügt) werden!“

In beiden Fällen ist der Umspeicheraufwand beträchtlich. Nur wenige Operationen sind dagegen nötig, wenn eine neue Datenstruktur mit folgenden Eigenschaften implementiert wird:

- Jedes Element $\boldsymbol{E_i}$ enthält die Information und eine Adresse.
- Der Adreßteil wird mit der Speicheradresse des nachfolgenden Elementes gefüllt; sie „zeigt“ auf das Element $\boldsymbol{E_{i+1}}$.
- Auf $\boldsymbol{E_1}$ muß ein „Anfangs-Zeiger“ verweisen; im Adreßteil des letzten Elementes muß ein Endekennzeichen, eine „Null-Adresse“ eingetragen werden.

Zum Einfügen eines neuen Elementes $\boldsymbol{E_{neu}}$ ist nur dessen Zeiger auf Element $\boldsymbol{E_{i+1}}$ zu richten und der Zeiger des Elementes $\boldsymbol{E_i}$ auf $\boldsymbol{E_{neu}}$ zu „verbiegen“. Umspeicherungen der bisherigen Elemente sind nicht erforderlich!

9.1 Zeiger (Pointer, Verweise)

Sei T ein Datentyp. Dann ist

```
type  <zeiger_auf_T> = ^T;
```

eine Typdefinition für einen Zeigertyp auf T. Variablendeklarationen erfolgen auf die übliche Weise.

Beispiel 1 **Zeiger auf Namen**

```
type  Name   = string[30];
      Zeiger = ^Name;    { Zeiger auf Name }
var   a,b    : Zeiger;
```

Beispiel 2 **Zeiger auf Felder**

```
type  Vektor  = array [1..100] of real;
      PVektor = ^Vektor;  { Zeiger auf Vektor }
var   x,y,z   : PVektor;
```

Leerer Zeiger (Nullzeiger): Um das Ende einer Folge zu kennzeichnen, existiert die Zeigerkonstante **nil** (Null, not in list), die jeder Zeigervariablen zugewiesen werden kann.

Beispiel 1: `a := nil;  b := nil;`

Dynamische Variablen (Bezugsvariablen)

Für jede Zeigervariable **p** vom Zeigertyp `PT` mit der Deklaration

```
type PT = ^T;  var  p: PT;
```

ist eine dynamische Variable `p^` des Typs `T` *implizit deklariert*
Skizze:

Diese *Bezugsvariable* `p^` wird nicht explizit deklariert. Sie ist auch an die Arbeit mit Werten von p gebunden.

Erzeugung dynamischer Variablen

- Die Speicherplatzzuweisung an die Zeigervariable mit `var p:PT;` erfolgt während der Compilierung („Compilierungszeit-Objekt“)
  $\Rightarrow$ `p` gehört zur *automatischen Speicherklasse*.

- Der Variablen `p^` des Typs T wird jedoch zur Übersetzung kein Speicher zugewiesen. Die Speicher-Bereitstellung und -Freigabe erfolgt während der Laufzeit („Laufzeit-Objekt“)
  $\Rightarrow$ `p^` gehört zur *dynamischen Speicherklasse*.

- Erzeugung einer Variablen `p^` (Allozierung): `new(p);`
  1. Im Freispeicher (Heap) werden soviele Bytes bereitgestellt, wie eine Variable des Typs T erfordert.
  2. Die Anfangsadresse dieses Speicherbereiches wird der Zeigervariablen p zugewiesen (Beginn der Gültigkeit).
  3. Die Variable selbst bleibt anonym, ein Zugriff auf sie ist nur durch `p^` möglich.

- Freigabe einer Variablen `p^` (Disallozierung): `dispose(p);`
  1. Der durch `new(p);` reservierte Speicherbereich wird dem Heap wieder zur Verfügung gestellt.
  2. Die Bezugsvariable `p^`, auf die `p` verweist, ist nun nicht mehr verfügbar (Ende der Gültigkeit).
  3. Intern wird der Variablen `p` der Wert `nil` zugewiesen, d.h. `p^` wird „abgehängt“.

Organisation des Speichers

Die prinzipielle Aufteilung des verfügbaren Speichers beinhaltet insbesondere

- den *statischen Speicher* mit den Programmsegmenten, dem Datensegment (für statische Objekte), das Stack-Segment und das Overlay-Segment.
- Der *Stack (Kellerspeicher)* dient der Speicherung der *automatischen Objekte*. Er wächst in Richtung absteigender Adressen. Die aktuelle Belegung wird durch den *Stack-Pointer* SPtr gekennzeichnet.
- Als *Heap (Freispeicher)* wird der gesamte restliche Speicher bezeichnet, der nicht durch Betriebssystem und laufendes Programm belegt wird. Er nimmt die *dynamischen Objekte* auf.

Dynamische Verwaltung des Heap

1. Die *Startadresse* des Heap (Heap Origin) ist in der globalen Variablen `HeapOrg` gespeichert, die Startadresse des freien Heaps im Heap-Pointer `HeapPtr`.
2. *Dynamische Belegung* mittels Prozedur `new(p)` bewirkt:
   (a) Prüfung, ob oberhalb des Heap-Pointers genügend Speicher vorhanden ist
   (b) Erhöhung des Heap-Pointers um die Größe der dynamischen Variablen `p^`
   (c) Übergabe des alten Wertes von `HeapPtr` an die Zeigervariable `p` als Anfangsadresse (Leitadresse) des von `p^` belegten Bereiches.
   Nacheinander erzeugte dynamische Variablen werden aufeinander abgelegt.
3. *Disallozierung* mittels der Prozedur `dispose(p)` gibt genau den Speicherbereich auf dem Heap frei, der durch `p` adressiert und die Länge des `p^` zugeordneten Typs begrenzt wird. Damit entstehen Lücken im belegten Heap, die verwaltet werden. Hinweis: Bei erneutem `new(q);` wird versucht, `q^` in vorhandene Lücken einzupassen.

9.2 Lineare Listen (Ketten)

9.2.1 Definition und Aufbau

Die einfachste dynamische Datenstruktur ist die *einfach verkettete lineare Liste (Kette)*.

Lineare Liste (Kette)

$\boldsymbol{M}$ sei eine Menge von Knoten (die Knotenmenge). Dann wird die Menge $\boldsymbol{L(M)}$ der *linearen Listen über* $\boldsymbol{M}$ rekursiv definiert:

(i) Eine leere Menge von Knoten ist eine lineare Liste.

(ii) Sei $\boldsymbol{k \in M}$ ein Knoten und $\boldsymbol{l \in L(M)}$ eine lineare Liste. Dann ist das geordnete Paar $\boldsymbol{(k, l) \in L(M)}$ eine lineare Liste.

Der Knoten $\boldsymbol{k}$ heißt Listenkopf, $\boldsymbol{l}$ heißt Listenkörper.

Realisierung linearer Listen als rekursive Datenstruktur:

- Jeder *Knoten* $\boldsymbol{K_i}$ enthält einen Dateninhalt und einen Zeiger auf den Nachfolgeknoten $\boldsymbol{K_{i+1}}$.
- Den Listenanfang bildet ein spezieller Zeiger, die *Wurzel (der Anker)*. Er verweist auf den Listenkopf $\boldsymbol{K_1}$.
- Das Listenende wird durch die *Erdung (Nullung)* des letzten Knotens $\boldsymbol{K_n}$ mittels `nil` im Adreßteil markiert.

In PASCAL ist zuerst ein Zeigertyp `PElement` zu deklarieren. Anschließend muß der zugehörige Knotentyp `Element` als Record-Typ vereinbart werden. Dieser enthält eine Komponente des Typs `PElement`.

Beispiel 4 Deklaration einer Namensliste

```
type  string20  =  string[20];
      PElement  =  ^Element;
      Element   =  record
                     Name  :  string20;
                     Next  :  PElement
                   end;
var   Wurzel, z :  PElement;
```

Zum physischen Aufbau einer Kette sind die benötigten Knoten mittels new zu erzeugen und in die bisherige Kette „einzuhängen“. Es ist einfacher, die Liste in umgekehrter Anordnung $\boldsymbol{E_n, E_{n-1}, ..., E_2, E_1}$ aufzubauen (warum?).

Beispiel 4 Generieren einer Namensliste

```
procedure Generierung(var Wurzel: PElement);
  var  z : PElement;
  begin
    Wurzel := nil;        { Erdung der Liste }
    repeat
      new(z);             { Erzeugung eines neuen Knotens}
      readln(z^.Name);    { Einlesen des Namens }
      z^.Next := Wurzel;  { Verbinden mit naechstem Knoten}
      Wurzel := z         { Aktualisieren der Wurzel }
    until  z^.Name = ''   { Abbruch = leere Zeichenkette }
  end;                    { of Generierung }
```

9.2.2 Operationen auf linearen Listen

Außer der Listengenerierung sind folgende elementare Operationen von besonderer Bedeutung:
– Ausgabe aller Knoteninhalte (Listenausgabe)
– Einfügen eines Knotens
– Physisches Löschen eines Knotens
– Suchen eines Knotens mit gegebenem Suchmerkmal
– Suchen aller Knoten mit gegebenem Suchmerkmal
– Physisches Kopieren einer Liste
– Physisches Aneinanderhängen (Verketten) zweier Listen usw.
Einige Operationen werden nachfolgend anhand des Beispiels 4 vorgestellt; Anpassungen an andere Knoteninhalte sind unproblematisch.

Beispiel 5 **Listenausgabe:**
Ein Hilfszeiger z weist auf den Kopfknoten. Nach Ausgabe des Namens `z^.Name` wird z auf den nächsten Knoten weitergesetzt usw. bis `z = nil` ist.

```
Unit LISTE;
Interface
  type  string20  =  string[20];
        PElement  =  ^Element;
        Element   =  record
                       Name  :  string20;
                       Next  :  PElement
                     end;
  procedure Generierung(var Wurzel: PElement);
  procedure Ausgabe(var Wurzel: PElement);
```

```
Implementation

  procedure Generierung(var Wurzel: PElement);
    var  z : PElement;
    begin
      Wurzel := nil;        { Erdung der Liste }
      repeat
        new(z);             { Erzeugung eines neuen Knotens}
        readln(z^.Name);    { Einlesen des Namens }
        z^.Next := Wurzel;  { Verbinden mit naechstem Knoten}
        Wurzel := z         { Aktualisieren der Wurzel }
      until  z^.Name = ''
    end;                    { of Generierung }

  procedure Ausgabe(var Wurzel: PElement);
    var  z : PElement;
    begin
      z := Wurzel;          { Pointer auf Wurzel gesetzt }
      while  z <> nil do begin
        writeln(z^.Name);   { Ausgabe des Namens }
        z := z^.Next        { Pointer weiterruecken }
      end
    end;                    { of Ausgabe }
End.
```

Rekursive Listen-Algorithmen

Gemäß Defintion ist eine lineare Liste entweder leer oder ein Knoten, verknüpft mit einer linearen Liste. Damit lassen sich Listenalgorithmen oft bequemer *rekursiv* definieren.

Ausgabe der Liste (rekursive Prozedur)

Wenn die Liste nicht leer ist, so gebe man das Element aus, auf das der Listenzeiger verweist und rufe dieselbe Prozedur für den Nachfolger des Elementes auf.

```
procedure Ausgabe(var Listenzeiger: PElement);
  begin
    if Listenzeiger <> nil then
    begin
      writeln(Listenzeiger^.Name);
      Ausgabe(Listenzeiger^.Next)
    end
  end;                        { of Ausgabe }
```

Einfügen eines Knotens (rekursive Prozedur)

Gegeben sind ein Testname und eine Zeichenkette. Letztere ist in die Liste hinter Testname einzufügen.

```
procedure Einfuegen(var Listenzeiger: PElement;
                    var Kette,Testname: string20);
  var  hilf : PElement;
  begin
    if Listenzeiger <> nil then
      if Listenzeiger^.Name = Testname then
      begin
        new(hilf);
        hilf^.Name := Kette;
        hilf^.Next := Listenzeiger^.Next;
        Listenzeiger^.Next := hilf
      end  else
        Einfuegen(Listenzeiger^.Next,Kette,Testname)
  end;    { of Einfuegen }
```

Physisches Loeschen eines Knotens (rekursive Prozedur)

Ein gegebener Testname ist *physisch* zu löschen; d.h. der Speicherplatz ist freizugeben.

```
procedure Loeschen(var Listenzeiger: PElement;
                   var Testname: string20);
  var  hilf : PElement;
  begin
    if Listenzeiger <> nil then
      if Listenzeiger^.Name = Testname then
      begin
        hilf := Listenzeiger^.Next;
        Listenzeiger^ := hilf^ ;
        dispose(hilf)
      end  else
        Loeschen(Listenzeiger^.Next,Testname)
  end;    { of Loeschen }
```

Beispiel 6 **Kopieren und Verketten von 3D-Punktlisten:**

Bei der Interpolation oder Approximation von Raumkurven werden Raumpunkte $\boldsymbol{P} = (\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3})$ oft als Elemente linearer Listen gespeichert. Dann können bei Bedarf beliebig viele Punkte eingefügt (interpoliert) werden, um eine Approximation zu verbessern.
Skizze:

1. Entwicklung einer geeigneten Datenstruktur:

Die Unit PUNKTE besitze die folgende Schnittstelle

```
Unit PUNKTE;
Interface
  const Dimension =  3;           { Raumdimension }
  type  Punkt     =  array[1..Dimension] of double;
        PElement  =  ^Element;
        Element   =  record
                       Koor :  Punkt;
                       Next :  PElement
                     end;

  procedure Generierung (var Wurzel: PElement);
  procedure Ausgabe (var Wurzel: PElement);
  procedure Einfuegen (var Wurzel: PElement;
              var Neupunkt,Testpunkt: Punkt);
  procedure Loeschen (var Wurzel: PElement;
              var Testpunkt: Punkt);
  . . . . . .
  function Kopie (Wurzel: PElement): PElement;
  function Append (Wurzel_X,
             Wurzel_Y : PElement) : PElement;
```

2. Kopieren einer linearen Liste:

Eine physische Kopie der Liste ist zu schaffen. Ist `Wurzel <> nil`, so erzeugt man ein neues Element, überträgt die Punktkoordinaten und hängt die bereits kopierte Teilkette an. Implementation:

```
function Kopie (Wurzel: PElement): PElement;
  { Kopieren einer Liste (rekursive Funktion) }
  var  hilf : PElement;
```

```
  begin
    if Wurzel = nil then Kopie := nil  else
    begin
      new(hilf);
      Kopie      := hilf;
      hilf^.Koor := Wurzel^.Koor;
      hilf^.Next := Kopie(Wurzel^.Next)
    end
  end;   { of Kopie }
```

3. Verketten zweier linearer Listen:

Die Liste Y ist an X *physisch* anzuhängen, so daß die neue Liste $Z = X + Y$ entsteht. Implementation:

```
function Append (Wurzel_X,
                 Wurzel_Y : PElement) : PElement;
  { Verketten zweier Listen (rekursive Funktion) }

  var  hilf : PElement;
  begin
    if Wurzel_X = nil  then
      Append := Kopie(Wurzel_Y)
    else begin
      new(hilf);
      Append     := hilf;
      hilf^.Koor := Wurzel_X^.Koor;
      hilf^.Next := Append(Wurzel_X^.Next,Wurzel_Y)
    end
  end;    { of Append }
```

9.3 Anwendung dynamischer Datenstrukturen

Außer einfach verketteten Listen (Einfach-Ketten) nutzt man häufig folgende Listenstrukturen (Skizzen):

- Zyklische Liste (zirkulare Liste, Ring)
- Spezielle einfach verkettete Liste
  - Schlange (queue)
  - Stapel (Keller, stack)
- Doppelt verkettete Liste (Doppelkette)

```
{ Typdeklaration der Doppelkette }
type  Datentyp  =  <Typ>;
      PElement  =  ^Element;
      Element   =  record
                     Info :  Datentyp;
                     Vor  :  PElement;
                     Nach :  PElement
                   end;
var   Kopf_vor, Kopf_nach : PElement;
```

Nichtlineare dynamische Datenstrukturen:

Binäre Bäume werden mittels Zeigern dargestellt, indem der Informationsteil (info) mit einem Verweis auf den linken Unterbaum (left) und den rechten Unterbaum (right) zu einem Knoten (left,info,right) zusammengefaßt wird:

```
{ Typdeklaration des Binaerbaumes }
type  Datentyp  =  <Typ>;
      PBintree  =  ^Bintree;
      Bintree   =  record
                     left  :  PBintree;
                     Info  :  Datentyp;
                     right :  PBintree
                   end;
var   Root : PBintree;
```

Ähnlich lassen sich p-adische Bäume und Geflechte definieren. Nachfolgend wird eine Anwendung im wissenschaftlichen Rechnen vorgestellt.

Beispiel 7 Langzahl–Arithmetik:

Große Primzahlen, wie z.B. Mersennesche Primzahlen der Form $M_k = 2^k - 1$ besitzen große Bedeutung bei der Datenverschlüsselung und der Netzwerk-Sicherheit. Um z.B. die 1998 entdeckte 909526-stellige Primzahl $M_{3021377} = 2^{3021377} - 1$ zu berechnen, braucht man eine Arithmetik für ganze Zahlen beliebiger Länge. Nachfolgend wird eine Implementation mittels zyklischer Listen beschrieben.

Darstellung im Positionssystem:

- Sei $B \in \mathbb{N},\ B \geq 2$ die Zahlenbasis. Dann lautet die Darstellung einer natürlichen Zahl im Positionssystem

$$z = z_0 + z_1 B + z_2 B^2 + \ldots + z_n B^n\ ,\ 0 \leq z_i < B,\ i = 0(1)n.$$

- Idee: Wahl der internen Basis B so, daß jede Ziffer in eine Darstellung von `integer` oder `longint` paßt! Wegen der Produktbildung $z_i z_k$ und der Berücksichtigung des Vorzeichens wählt man oft $B = 10^4$ (Dezimalbasis) oder $B = 2^{15}$ (Dualbasis).
- Beispiel: Sei $B = 10^5 = 100000$.

$$\begin{aligned} z &= 459\ 76349\ 72106\ 98463 \\ &= (98463) + (72106) \cdot B + (76349) \cdot B^2 + (00459) \cdot B^3 \end{aligned}$$

Definition der Datenstruktur:

- Organisation der zyklischen Kette so, daß man von der niedrigsten Position $\boldsymbol{B^0}$ zur höchsten $\boldsymbol{B^n}$ gelangt (warum?)
- Markierung des Kettenendes durch die „unzulässige Ziffer“ $\boldsymbol{-1}$ (warum?)

```
Unit Bigints;
Interface
  const  Laenge  = 4;          { Ziffernlaenge}
         Basis   = 10000;      { Ziffernbasis }
  type   digit   = integer;    { Zifferntyp   }
         Bigint  = ^ktyp;
         ktyp    =  record
                      ziffer : digit;
                      next   : Bigint;
                    end;
{ Deklaration der Arithmetik-Routinen }
  procedure Readln_Big (var p:Bigint);
  procedure Write_Big  (p:Bigint);
  procedure Writeln_Big(p:Bigint);
  function  Add_Big    (p,q:Bigint):Bigint;
  procedure Free_Big   (p:Bigint);

  function  Size_big   (p:Bigint):integer;
  function  Mult_number(p:Bigint; z:digit):Bigint;
  procedure Shr_1      (var p:Bigint; l:integer);
  function  Mult_Big   (p,q:Bigint):Bigint;
Implementation
  . . .
End.
```

Ausgabe einer Bigint-Zahl p:

Gegeben eine Zahl $z = (z_n z_{n-1} \ldots z_1 z_0)$ als Kette. p sei der Wurzelzeiger, q ein interner Hilfszeiger, der auf den jeweils aktuellen Knoten zeigt. Die Knoteninhalte sind in umgekehrter Reihenfolge - mit führenden Nullen - auszugeben.

```
procedure Write_Big(p:Bigint);
  { Ausgabe der Zahl p ohne Zeilenwechsel }
  var  kette : string[Laenge];
  procedure check(q:Bigint);
    { Rekursive Ausgabe in umgekehrter Anordung }
    begin
      if q^.next = p  then write(q^.ziffer) else
      begin
        check(q^.next);
        str(q^.ziffer:Laenge,kette);
        for lv:=1 to Laenge do
          if kette[lv] = ' ' then kette[lv]:='0';
        write(kette)
      end
    end;  { of check }
  begin
    { Uebergehen des ersten Knotens }
    check(p^.next)
  end;    { of Write_Big }

procedure Writeln_Big(p:Bigint);
  { Ausgabe der Zahl p mit Zeilenwechsel }
  begin
    Write_Big(p);  writeln
  end;     { of Writeln_Big }
```

Addition zweier Bigint-Zahlen p und q:

Lösung der Aufgabe $\boldsymbol{s = p + q}$ in den Schritten:

1. Ziffernweise Addition an i-ter Stelle:

$$\text{gesamt} \;=\; \boldsymbol{p_i} + \boldsymbol{q_i} + \;\text{Übertrag}_{\boldsymbol{i-1}}$$

2. Summe und Übertrag:

$$\begin{aligned} \text{Summe} \;&=\; \boldsymbol{s_i = x} \bmod \boldsymbol{B}, \\ \text{Übertrag} \;=\; \text{carry} \;&=\; \boldsymbol{x} \operatorname{div} \boldsymbol{B} \end{aligned}$$

3. Generieren eines neuen Knotens in s für $\boldsymbol{s_i}$

```
function Add_Big(p,q:Bigint):Bigint;
  { p und q sind die Koepfe der beiden Ketten
    fuer die zu addierenden Operanden }
  var    s,ss,pp,qq,r,h : Bigint;
         carry          : 0..1;
         zahl, gesamt   : digit;
  begin
    { pp und qq zeigen auf die Knoten, die den
      Koepfen folgen; die Funktion liefert den
      Kopf der Summenkette. }
    pp := p^.next;
    qq := q^.next;
    { Es werde ein Kopf fuer die Summe erzeugt. }
    new(s);
    s^.ziffer := -1;
    s^.next   :=  s;
    ss := s;
    carry := 0; { Initialisierung des Uebertrages }
```

```
while (pp^.ziffer <> -1) and (qq^.ziffer <> -1) do
begin
  { Paralleles Durchmustern beider Ketten;
    Addieren der Ziffernteile zweier Knoten und
    des vorherigen Uebertrages. }
  gesamt := pp^.ziffer + qq^.ziffer + carry;
  zahl   := gesamt mod Basis;
  carry  := gesamt div Basis;  { neuer Uebertrag }
  new(h);
  ss^.next   := h;
  ss         := h;
  ss^.ziffer := zahl;
  ss^.next   := s;
  pp         := pp^.next;
  qq         := qq^.next
end; { of while }
{ Nun ist eine der beiden Ketten abgearbeitet -
  oder beide, wenn die Zahlen gleich lang sind.}
if pp^.ziffer <> -1 then  r := pp  else  r := qq;
while r^.ziffer <> -1 do begin
  { Durchmustern des Restes der Kette.}
  gesamt := r^.ziffer + carry;
  zahl   := gesamt mod Basis;
  carry  := gesamt div Basis;
  new(h);
  ss^.next   := h;
  ss         := h;
  ss^.ziffer := zahl;
  ss^.next   := s;
  r          := r^.next
end;  { of while }
```

```
    if carry = 1 then
    begin
      new(h);
      ss^.next   := h;
      ss         := h;
      ss^.ziffer := carry;
      ss^.next   := s
    end;
    Add_Big := s
  end;     { of Add_Big }
```

Routinen zur Multiplikation $\boldsymbol{p \cdot q}$:

Auszuführen ist die „klassische“ Multiplikation $\boldsymbol{p \cdot q_i}$ mit Verschiebung und Aufaddieren für $\boldsymbol{i = 0(1)n}$. Routinen (als Übung):

```
(1)  function Size_Big(p:Bigint):integer;
     { Laenge ( = Ziffernanzahl ) der Zahl p }

(2)  function Mult_number(p:Bigint; z:digit):Bigint;
     { Multiplikation der Zahl p mit der Ziffer z }

(3)  procedure Shr_l(var p:Bigint; l:integer);
     { Verschiebung der Zahl p um l Stellen nach rechts }

(4)  function Mult_Big(p,q:Bigint):Bigint;
     { p und q sind die Koepfe der beiden Operanden;
      die Funktion liefert den Kopf der Produktkette }

(5)  procedure Free_Big(p:Bigint);
     { Freigabe des durch die Kette p belegten Heaps }
```

Anwendungsaufgabe:

Man bestimme das Skalarprodukt s der 2 Vektoren $a = (a_1, a_2, \ldots, a_m)$, $b = (b_1, b_2, \ldots, b_m)$, mit Bigint-Zahlen a_j, b_j :

$$s = \sum_{j=1}^{m} a_j \cdot b_j \ , \quad a_j, b_j \text{ Bigint-Zahlen}$$

```
Program HBigint;
uses    Crt, Bigints;
const   nmax   = 100;
type    Vektor = array[1..nmax] of Bigint;
var     a,b    : Vektor;
        s      : Bigint;
        i,n    : integer;
begin
  Clrscr;
  writeln('Skalarprodukt mit Bigint-Arithmetik');
  writeln('==================================');
  write('Eingabe der Komponentenzahl n = ');
  readln(n);writeln;
  writeln('Eingabe der Vektoren  a und b :');
  for i:=1 to n do begin
    write('a[',i,'] = '); Readln_Big(a[i]);
    write('b[',i,'] = '); Readln_Big(b[i]);
    writeln
  end;  writeln;
  s:= Mult_Big(a[1],b[1]);     { Anfangswert }
  for i:=2 to n do
    s := Add_Big(Mult_Big(a[i],b[i]),s);
  writeln('Skalarprodukt  a*b :');
  Writeln_Big(s);
end.
```

Kapitel 10

Elemente objektorientierter Programmierung

Objektorientierte Programmierung (OOP) ist ein *Programmierstil*, der aus der Entwicklung von Programmiersprachen mit dem Ziel entstanden ist, große Software-Projekte besser managen zu können. Qualitätsnormen im Software-Engineering sind:

- *Korrektheit:* Erfüllung der Anforderungsspezifikation durch die entwickelte Software
- *Erweiterbarkeit:* Fähigkeit, Software an erweiterte Spezifikationen anzupassen
- *Wiederverwendbarkeit:* Möglichkeit, Software teilweise oder vollständig für neue Anwendungen wiederzuverwenden
- *Kompatibilität:* Fähigkeit, verschiedene Softwareprodukte miteinander zu kombinieren

Programmierstile

1. Prozedurale Programmierung (Pascal, C, FORTRAN):
   Programme bestehen aus Prozeduren und Funktionen, die mittels eines Algorithmus in geeigneter Weise angearbeitet werden.
   $\Longrightarrow$ Anwendung im wissenschaftlich-technischen Rechnen; Compilersprachen

2. Deklarative Programmierung (PROLOG):
   Programme bestehen aus Fakten und logischen Regeln. Mit Hilfe des Backtracking werden Lösungen gefunden, die die Regeln und die Wissensbasis erfüllen.
   $\Longrightarrow$ Anwendung zur Implementierung von Expertensystemen; Compiler und Interpreter

3. Funktionale Programmierung (LISP, Mathematica):
   Mit Hilfe von Rekursion (rekursive Algorithmen, rekursive Strukturen) werden nutzerdefinierte Funktionen auf Listenoperationen zurückgeführt.
   $\Longrightarrow$ Anwendung zu Symbolverarbeitung und Computeralgebra; Interpretersprachen

4. Objektorientierte Programmierung (SMALLTALK, C++, GPC):
   Algorithmus und Datenstruktur werden nicht mehr getrennt, sondern durch interagierende Objekte modelliert, die eingebaute Methoden benutzen.
   $\Longrightarrow$ Anwendung zur Nachbildung komplexer Modelle der Realität; umfangreiche Compiler

Historie:

- SIMULA 76 (1976) : Klassenkonzept
- ALGOL 68 (1968) : Operatorüberladung
- SMALLTALK (1980) : erste rein objektorientierte Sprache
- C++ (1986ff) : hybride Sprache mit ständigen Erweiterungen
- EIFFEL (1992) : rein objektorientierte Sprache
- Pascal-Erweiterungen : hybride Sprache
  - Borland Pascal (1991ff) : ohne Operatorüberladung
  - Pascal-XSC (1990ff) : mit Operatorüberladung
  - Free-Pascal (2000) : ohne funktionierende Op.-überladung
  - GNU-Pascal (2001) : mit Operatorüberladung

3 wesentliche Eigenschaften der OOP:

1. Datenkapselung (encapsulation, information hiding):

   Kombination von Daten mit Funktionen und Prozeduren in Objekten bzw. Klassen, Abschirmung eines Objektes und seiner Methoden nach außen und Schutz vor Zugriffen, Prinzip der schmalen Schnittstelle

2. Vererbung (inheritance):

   Eingliederung der Objekte in die Hierarchie eines Objektstammbaumes; jedes abstammende (abgeleitete) Objekt erbt Code und Daten der Vorfahren (Basisobjekte); Einfachvererbung und Mehrfachvererbung

3. Polymorphismus:

   Unterschiedliche Objekte reagieren auf dieselbe Methode mit unterschiedlichen Aktionen; jedes Objekt der Hierarchie implementiert diese Methode in der für dieses Objekt erforderlichen Art und Weise

Zielsetzung:

Definition neuer, spezialisierter, abstrakter Datentypen (ADT) durch Übernahme der Eigenschaften von einfacheren, allgemeineren Datentypen mittels Vererbung.

10.1 Überladen von Operatoren

Um arithmetische Operatoren (+,-,*,/, ...) auch auf komplexe Zahlen, Vektoren, Matrizen oder andere Operanden anwenden zu können, ist eine Definition ihrer neuen Bedeutung - eine Operatordefinition - erforderlich. Der entsprechende Operator wird überladen (operator overloading). Da ein- und derselbe Operator dann in mehreren Bedeutungen auftritt, führt dies zu einem Operator-Polymorphismus.

10.1.1 Operatordefinition in GNU-PASCAL

GNU-PASCAL (GPC) gestattet das Überladen der meisten unären und binären Operatoren. Die Operator-Definition legt (a) die Eingangsparameter und deren Typ, (b) den Ergebnistyp und (c) den Inhalt des Algorithmus fest. Der Ergebnistyp muß nun nicht skalar sein.

Syntax der Operator-Definition

Sie ähnelt der Funktions-Definition - mit folgenden 2 Abweichungen:

- Operatorkopf:

  ⟨Operatorkopf⟩ **::=** operator ⟨Operatorzeichen⟩
  (⟨Formalparameter⟩)
  ⟨Ergebnisbezeichner⟩ **:** ⟨Ergebnistyp⟩;

- Anweisungsteil:
  Mindestens 1 Anweisung muß dem Ergebnisbezeichner einen Wert zuweisen:

  ⟨Ergebnisbezeichner⟩ **::=** ⟨Ausdruck⟩;

  Der Ergebnisbezeichner darf nicht wie eine übliche Variable benutzt werden; er darf bei Zuweisungen nur links stehen.

Beispiel 1 Addition zweier Vektoren

Der Additionsoperator + soll für Vektoren mit 6 Komponenten des Typs `double` überladen werden.

```
type  vektor = array[1..6] of double;
operator + (a,b : vektor) resultat : vektor;
  var  sum : vektor;
       i   : integer;
  begin
    for i:= 1 to n do
      sum[i] := a[i] + b[i];
    resultat := sum;   { 1 Ergebniszuweisung! }
  end;
```

Semantik der Operatorüberladung:

1. Argumentanzahl und -reihenfolge müssen mit der des überladenen Standardoperators übereinstimmen.
2. Überladene Operatoren übernehmen Priorität und Assoziativität des Standardoperators.
3. Die Ergebnisübergabe an den Ergebnisbezeichner sollte in einer einzigen Anweisung erfolgen (Hilfsvariablen benutzen).
4. Operatoren können beliebig oft überladen werden. Dabei müssen sich die Parameterlisten (Anzahl, Typ) voneinander unterscheiden.

Beispiel 1 Vektorarithmetik mit Anwendung

Für Vektoren des Typs `double` sind folgende Operatoren zu definieren:

$+, -$: Addition $\boldsymbol{a} + \boldsymbol{b}$, Subtraktion $\boldsymbol{a} - \boldsymbol{b}$
$*$: Skalarprodukt $\boldsymbol{a} * \boldsymbol{b}$
$*$: Vektor $*$ Skalar bzw. Skalar $*$ Vektor
/ : Vektor / Skalar

```
Program vektari;  { Vektorarithmetik }
  const n = 6;    { Dimension der Vektoren }
  type  vektor = array[1..n] of double;
  operator + (a,b : vektor) resultat : vektor;
    { Summe a+b zweier Vektoren }
    var  sum : vektor;
         i   : integer;
    begin
      for i:= 1 to n do
        sum[i] := a[i] + b[i];
      resultat := sum;   { 1 Ergebniszuweisung! }
    end;
  operator * (a,b : vektor) resultat : double;
    { Skalarprodukt a*b zweier Vektoren }
    var  s : double;
         i : integer;
    begin
      s := 0;
      for i:= 1 to n do
        s := s + a[i] * b[i];
      resultat := s;
    end;
  operator * (a: double; b: vektor) resultat: vektor;
    { Skalar * Vektor }
    var  s   : vektor;
         i   : integer;
    begin
      for i:= 1 to n do
        s[i] := a * b[i];
      resultat := s;
    end;
```

```
operator * (a: vektor; b: double) resultat: vektor;
  { Vektor * Skalar }
  var  s   : vektor;
       i   : integer;
  begin
    for i:= 1 to n do
      s[i] := a[i] * b;
    resultat := s;
  end;
operator - (a,b : vektor) resultat : vektor;
  { Differenz  a-b  zweier Vektoren }
  begin
    resultat := a + (-1.0)*b;
  end;
operator / (a: vektor; b: double) resultat: vektor;
  { Vektor / Skalar }
  var  s   : vektor;
       i   : integer;
  begin
    for i:= 1 to n do
      s[i] := a[i] / b;
    resultat := s;
  end;

procedure vektorwriteln(a : vektor);
  var  i : integer;
  begin
    for i:= 1 to n do
      writeln('  Komponente[',i, '] = ', a[i]);
  end;
```

```
var   u,v,w   : vektor;
      s,t1,t2 : double;
begin
  writeln('Vektor-Arithmetik');
  writeln('==================');
  writeln('Dimension n = ',n); writeln;
  u[1] := 1.22;      v[1] := 1e-4;
  u[2] := -3.0;      v[2] := 2;
  u[3] := 5.80;      v[3] := -1.6e2;
  u[4] := 0.026;     v[4] := 7.0;
  u[5] := -3;        v[5] := 22.1;
  u[6] := -1e2;      v[6] := 9.15;

  w := u + v;
  writeln('Summenvektor  w :');
  vektorwriteln(w); readln;

  s := (u+v)*(u-v);
  writeln('Skalarprodukt s1 : ',s); readln;

  s := u * v * w * u;
  writeln('Skalarprodukt s2 : ',s); readln;

  t1 := s+3.6;  t2 := s-44.8;
  w  :=  (t2 * u * (s+t1) - (w - v) / sin(t2)) / s;
  writeln('Gesamtvektor w :');
  vektorwriteln(w); readln;

end.
```

```
****************************************
Vektor-Arithmetik
=================
Dimension n = 6
Summenvektor  w :
  Komponente[1] =  1.220100000000000e+00
  Komponente[2] = -1.000000000000000e+00
  Komponente[3] = -1.542000000000000e+02
  Komponente[4] =  7.026000000000000e+00
  Komponente[5] =  1.910000000000000e+01
  Komponente[6] = -9.084999999999999e+01

Skalarprodukt s1 : -1.617200342401000e+04
Skalarprodukt s2 : -1.558525073665400e+07

Gesamtvektor w :
  Komponente[1] = -3.802811671742304e+07
  Komponente[2] =  9.351176241989273e+07
  Komponente[3] = -1.807894073451259e+08
  Komponente[4] = -8.104352743057370e+05
  Komponente[5] =  9.351176241989273e+07
  Komponente[6] =  3.117058747329758e+09
****************************************
```

10.1.2 Abstrakte Datentypen

Ein abstrakter Datentyp (ADT) ist eine Kollektion von Daten und eine Menge von Operationen auf diesen Daten. Dabei ist die *interne Repräsentation* der Daten und der Operationen je nach Standpunkt nicht wichtig, nicht sichtbar oder nicht beeinflußbar. Werkzeuge zur Definition von ADT sind insbesondere Objektdefinitionen und Operatorüberladung in Units.

Beispiel 2 **Differentiationsarithmetik**

Ein ADT `diff` ist in einer Unit zu definieren, mit dessen Hilfe eine „automatische Differentiation“ beliebiger rationaler Funktionen erfolgen kann, z.B für

$$f(x) = x\frac{4+x}{3-x}\,.$$

- Sei $F = (f, f')$ das geordnete Paar mit Funktionswert $f = f(x)$ und Ableitungswert $f' = f'(x)$ an einer festen Stelle x. Gesucht ist z.B. für $x = 1.0$ das Paar $F = (2.5, 4.25)$, also $f(1.0) = 2.5,\ f'(1.0) = 4.25.$

- Differentiationsregeln für $F = (f, f'),\ G = (g, g')$:

$$\begin{aligned} F + G &= (f + g, f' + g') \\ F - G &= (f - g, f' - g') \\ F * G &= (f * g, f * g' + f' * g) \\ F/G &= (f/g, (f' - f * g'/g)/g) \end{aligned}$$

- *Konvertierungsregeln* für $F = (f, f')$:

$$\begin{aligned} f(x) = c &\Rightarrow F = (c, 0) \quad \text{Konstante} \\ f(x) = x &\Rightarrow F = (x, 1) \quad \text{Diff.-variable} \end{aligned}$$

```
Unit diffari; { Differentiations-Arithmetik }
Interface
  type diff = record
                f, df : double;
          end;

  { ** Infixoperatoren ** }
  operator + (a,b:diff) res:diff;
  operator - (a,b:diff) res:diff;
  operator * (a,b:diff) res:diff;
  operator / (a,b:diff) res:diff;

  { **  Konvertierung von Konstante und Variable }
  function diffconst (r : double) : diff;
  function diffvar (r : double) : diff;

  { **  Ausgabefunktion ** }
  procedure diffwrite(x : diff);

Implementation

{ ** Infixoperatoren ** }
operator + (a,b:diff) res:diff;
  begin  res.f  := a.f + b.f;
         res.df := a.df + b.df;
  end;
operator - (a,b:diff) res:diff;
  begin  res.f  := a.f - b.f;
         res.df := a.df - b.df;
  end;
```

```
operator * (a,b:diff) res:diff;
  begin  res.f  := a.f * b.f;
         res.df := a.f*b.df + a.df*b.f;
  end;
operator / (a,b:diff) res:diff;
  begin  res.f  := a.f / b.f;
         res.df := (a.df - a.f*b.df/b.f) / b.f;
  end;

{ **  Konvertierung von Konstante und Variable ** }
function diffconst (r : double) : diff;
  var  help : diff;
  begin  help.f  := r;
         help.df := 0.0;
     diffconst := help; { 1 Ergebniszuweisung! }
  end;
function diffvar (r : double) : diff;
  var  help : diff;
  begin  help.f  := r;
         help.df := 1.0;
     diffvar := help;  { 1 Ergebniszuweisung! }
  end;

{ **  Ausgabefunktion ** }
procedure diffwrite(x : diff);
  begin
    write('[', x.f, ',', x.df, ']');
  end;
end.
```

Anwendung: Für obige Funktion $f(x)$ sind Wert und 1. Ableitung mit Rechnergenauigkeit zu bestimmen, wobei $x = -4.0(0.5)2.0$ gilt.

```
Program Hdiffari (input,output);
  { Anwendung der Differentiations-Arithmetik }
uses diffari;

{ **  Anwendungs-Funktion ** }
function f(x : diff) : diff;
  var  drei, vier : diff;
  begin
    drei := diffconst(3.0);
    vier := diffconst(4.0);
    f    := x*(vier + x)/(drei - x);
  end;

var  x,y : diff;
     h   : double;
     i   : integer;
begin
  writeln('Tabellierung von f(x) und f''(x)');
  writeln('================================');
  h    := 0.5;
  writeln('x':10,'f(x)':27,'f''(x)':25);
  for i:=0 to 12 do begin
    x := diffvar(-4.0 + i*h);
    y   := f(x);
    writeln(x.f,'   ',y.f,'   ',y.df)
  end; readln
end.
```

```
Tabellierung von f(x) und f'(x)
===============================
       x                    f(x)                    f'(x)
-4.0000000e+00   0.000000000000000e+00   -5.714285714285714e-01
-3.5000000e+00  -2.692307692307692e-01   -5.029585798816568e-01
-3.0000000e+00  -5.000000000000000e-01   -4.166666666666667e-01
-2.5000000e+00  -6.818181818181818e-01   -3.057851239669421e-01
-2.0000000e+00  -8.000000000000000e-01   -1.600000000000000e-01
-1.5000000e+00  -8.333333333333334e-01    3.703703703703703e-02
-1.0000000e+00  -7.500000000000000e-01    3.125000000000000e-01
-5.0000000e-01  -5.000000000000000e-01    7.142857142857143e-01
 0.0000000e+00   0.000000000000000e+00    1.333333333333333e+00
 5.0000000e-01   9.000000000000000e-01    2.360000000000000e+00
 1.0000000e+00   2.500000000000000e+00    4.250000000000000e+00
 1.5000000e+00   5.500000000000000e+00    8.333333333333334e+00
 2.0000000e+00   1.200000000000000e+01    2.000000000000000e+01
```

Aufgaben:

1. Man erweitere die Unit `diffari` um die Differentiationsregeln für die bekanntesten Standardfunktionen (exp, ln, sin, cos, arctan,...) unter Benutzung der Kettenregel. Achtung: GPC gestattet (bisher) keine Überladung von Funktionen.
2. Man entwickle abstrakte Datentypen für lineare Listen, Vektoren und Matrizen.

Hinweis:
GPC ist der freie 32/64-bit Compiler der GNU Compiler-Sammlung. Die aktuelle Version 20010409 unterstützt

- ISO-7185 Standard Pascal
- große Teile von ISO-10206 Extended Pascal
- Borland Pascal 7.0 (mit einigen Delphi-Erweiterungen)
- Teile von Pascal-SC (PXSC).

Vgl. *The GNU Pascal Manual, Free Software Foundation.* Alle Informationen und Downloads unter

www.gnu-pascal.de

10.2 Objekte und Methoden

10.2.1 Objekte und abgeleitete Objekte

Darstellung eines Bildschirmpunktes mit Koordinaten (x,y) mittels Record:

```
type  Location = record
                   X,Y: Integer;
                 end;
var   ALocation : Location;
```

Beleuchtung des Bildschirmpunktes durch Hinzunahme der Komponente `Visible` ergibt den Datentyp `Point`:

```
type  Location = record
                   X,Y: Integer;
                 end;
      Point    = record
                   Pos   : Location;
                   Visible : Boolean;
                 end;
var   APoint : Point;
```

`Point` ist Nachkommenstyp von `Location` und erbt alle Eigenschaften seines Vorfahrenstyps.

Deklaration als Objekte:

```
type  Location = object
                   X,Y: Integer;
                 end;
      Point    = object (Location)
                   Visible : Boolean;
                 end;
```

Deklaration als Objekte:

```
type  Location = object
                   X,Y: Integer;
                 end;
      Point    = object (Location)
                   Visible : Boolean;
                 end;
```

Syntax der Objektdefinition:

Semantik:

1. `Location` ist das Basisobjekt, von welchem das Objekt `Point` abgeleitet ist. `Point` ist direkter Nachkomme von `Location` und erbt alle seine Eigenschaften.
2. Die Datenfelder X und Y von `Location` gehören auch zu `Point`, obwohl sie dort nicht explizit aufgeführt sind!
3. Ein Objekt kann nur 1 direkten Vorfahren, jedoch beliebig viele Nachkommen besitzen (Prinzip der Einfach-Vererbung).
4. Durch Ableitung von `Point` läßt sich eine Hierarchie, d.h ein Stammbaum der Objekte, aufbauen.

```
type  Location = object
                   X,Y: Integer;
                 end;

      Point    = object (Location)
                   Visible : Boolean;
                 end;

      Circle   = object (Point)
                   Radius: Integer;
                 end;
```

Instanzen von Objekttypen:

Variablen von Objekttyp („Instanzen“) können wie andere Variablen in Pascal deklariert werden, d.h. statisch (besser: automatisch) oder dynamisch. Beispiele:

```
type  PointPtr = ^Point;

var   APoint, P1, Q : Point;
      DynaPoint     : PointPtr;
      Koord         : Location;
      k1, MyCircle  : Circle;

begin  new(DynaPoint);
       . . . . . .
```

Zugriff auf die Felder eines Objektes:

Die Datenfelder können wie bei Records angesprochen werden. Das betrifft auch die ererbten Felder. Sie sind standardmäßig „öffentlich“ (public) und damit frei erreichbar. Beispiele:

```
APoint.Visible := false;
APoint.X       := 19;
with P1 do begin
  X := 341;
  Y := 38;
end;
P1.Visible := not APoint.Visible;

DynaPoint^.X := P1.X + 12;
DynaPoint^.Y := P1.Y - 6;

with Koord do begin
  X := APoint.X;
  Y := APoint.Y;
end;
```

Private Datenfelder:

Um Daten vor Zugriffen zu schützen und „schmale Schnittstellen“ von Objekten zu erreichen, können Felder eines Objektes mit `private` deklariert werden. Alle nach dem Schlüsselwort stehenden Felder sind dann von außen nicht direkt ansprechbar! Beispiele:

```
type  Location = object
                 private
                   X,Y: Integer;
                 end;
      Point    = object (Location)
                   Visible : Boolean;
                 end;
```

Ein Zugriff auf die Koordinaten X und Y ist nun nicht mehr möglich.

```
type  Name      =  record
                     Vorname     :  string[20];
                     Zuname      :  string[40];
                   end;
      Datum     =  record
                     Tag         :  1..31;
                     Monat       :  1..12;
                     Jahr        :  integer;
                   end;
      Adresse   =  record
                     PLZ         :  0..99999;
                     Ort         :  string[20];
                     Strasse     :  string[40];
                     Nr          :  1..999;
                   end;

      Student   =  object
                     Matrikel_Nr :  integer;
                     Stud_Name   :  Name;
                   private
                     Wohnung     :  Adresse;
                     Geb_Datum   :  Datum;
                   end;

var   TU_Student, MB_Student      : Student;
begin
  MB_Student.Matrikel_Nr := 23456;  { ausfuehrbar }
  writeln(TU_Student.Wohnung.Ort);  { unzulaessig }
```

Frage: Wie greift man auf private Datenfelder zu? Mittels (öffentlicher) Methoden!

10.2.2 Methoden und deren Gültigkeitsbereich

Eine *Methode* ist eine Prozedur oder Funktion, innerhalb derer man wie in einer with-Anweisung direkt auf alle – öffentlichen und privaten – Datenfelder des Objektes zugreifen kann. Sie gehört zu dem betreffenden Objekt.

Beispiel 3 **Initialisierung und Zugriff**

Eine Initialisierungsprozedur `Init` soll sämtliche Datenfelder von `Location` mit Anfangswerten initialisieren. Weiterhin sollen Zugriffsfunktionen `GetX` und `GetY` den Zugriff auf die Koordinaten ermöglichen.

```
type  Location = object
         X,Y: Integer;
         procedure Init(InitX, InitY: Integer);
         function GetX: Integer;
         function GetY: Integer;
      end;

procedure Location.Init(InitX, InitY: Integer);
begin
  X := InitX;
  Y := InitY;
end;
function Location.GetX: Integer;
begin
  GetX := X;
end;
function Location.GetY: Integer;
begin
  GetY := Y;
end;
```

Erläuterung:

1. Zur Objektdefinition gehört die *Deklaration* der Methode, d.h. des Methodenkopfes. Die vollständige *Definition* der Methode (die Implementation) erfolgt außerhalb und muß deshalb den Objektnamen zusätzlich erhalten, um den Bezug zum Objekt herzustellen:

   ```
   Location.Init, Location.GetX, Location.GetY
   ```

2. In BP müssen die Datenfelder vor den Methoden deklariert werden - sowohl im öffentlichen als auch im privaten Teil. In GNU-Pascal (GPC) ist die Reihenfolge beliebig.

3. Beim Aufruf einer Methode setzt sich deren Name aus dem Instanzbezeichner und dem Methodenname - wie bei Datenfeldern - zusammen. Beispiel:

   ```
   var  MyLocation : Location;
   . . .
   MyLocation.Init(13,56);
   writeln(MyLocation.GetX, MyLocation.GetY);
   ```

Gültigkeitsbereich von Methoden und Datenfeldern:

Der Gültigkeitsbereich (scope) ist der jeweilige Objekttyp - einschließlich der Methodendefinitionen. Konsequenzen:

- Alle Datenfelder (hier: X,Y) sind in den Methoden frei verfügbar; eine with-Anweisung ist deshalb nicht erforderlich.
- Formale Parameter von Methoden dürfen nicht identisch mit Datenfeldern sein (deshalb InitX, InitY und nicht x, y).
- Lokale und formale Parameter einer Methode liegen im selben Gültigkeitsbereich und dürfen deshalb nicht identisch sein.

Objektdefinition in Units

Um Objekte und Objekthierarchien allgemein zugängig zu machen, empfiehlt sich ihre Definition in Units. Damit unterliegen sie auch den Regeln, die für die Programmobjekte (type, var, const, procedure, function) in Units gelten. Zudem gilt:

1. *Öffentliche Objekte* sind im Interface zu definieren; die Methoden-Rümpfe dagegen liegen im Implementationsteil der Unit.
2. *Private Objekte* werden komplett im Implementationsteil der Unit definiert; sie sind von außen nicht ansprechbar.
3. Öffentliche Objekte dürfen private Nachkommen (d.h.im Implementationsteil) besitzen.
4. *2 Units:* Ist $\boldsymbol{O_A}$ öffentliches Objekt in Unit A und wird diese Unit in einer Unit B verwendet, so können in Unit B Nachkommen von $\boldsymbol{O_A}$ definiert werden, d.h $\boldsymbol{O_A}$ wird aus A exportiert).

Beispiel 4 **Unit „Points“**

Die Objekte `Location` und `Point` (s.o.) sollen in einer Unit `Points` definiert werden.
Methoden von `Point` sind:

Init	:	Initialisierung von X,Y, Visible
Show	:	Zeichnen eines Punktes
Hide	:	Löschen eines Punktes
IsVisible	:	Statusabfrage eines Punktes
MoveTo	:	Verschieben eines Punktes auf (NewX, NewY).

```
Unit Points;
Interface
uses Graph;

type  Location = object
        X,Y: Integer;
        procedure Init(InitX, InitY: Integer);
        function GetX: Integer;
        function GetY: Integer;
      end;

      Point = object(Location)
        Visible: Boolean;
        procedure Init(InitX, InitY: Integer);
        procedure Show;
        procedure Hide;
        function IsVisible: Boolean;
        procedure MoveTo(NewX, NewY: Integer);
      end;

Implementation

{ Location's method implementations: }

procedure Location.Init(InitX, InitY: Integer);
  begin
    X := InitX;
    Y := InitY;
  end;
```

```
function Location.GetX: Integer;
  begin
    GetX := X;
  end;
function Location.GetY: Integer;
  begin
    GetY := Y;
  end;

{ Points's method implementations: }

procedure Point.Init(InitX, InitY: Integer);
  begin
    Location.Init(InitX, InitY);
    Visible := False;
  end;
procedure Point.Show;
  begin
    Visible := True;
    PutPixel(X, Y, GetColor);
  end;
procedure Point.Hide;
  begin
    Visible := False;
    PutPixel(X, Y, GetBkColor);
  end;
function Point.IsVisible: Boolean;
  begin
    IsVisible := Visible;
  end;
```

```
procedure Point.MoveTo(NewX, NewY: Integer);
  begin
    Hide;
    Location.Init(NewX, NewY);
    Show;
  end;
end.
```

Anwendung der Unit Points:

Eine Instanz `APoint` soll erzeugt, initialisiert und angezeigt werden. Anschließend soll `APoint` auf eine neue Koordinate verschoben und gelöscht werden.

```
Program HPoints;
uses    Crt, Graph, Points;
const   PathToDrivers = 'C:\BP\BGI';
var     GraphDriver, GraphMode : Smallint;
var     APoint : Point;
begin
  GraphDriver := Detect;
  DetectGraph(GraphDriver, GraphMode);
  InitGraph(GraphDriver, GraphMode, PathToDrivers);
  . . .
  with APoint do begin
    Init(151,82);              { Anfangswerte fuer x,y }
    Show;  readln;             { APoint einschalten    }
    MoveTo(163,101);           { APoint verschieben    }
    Hide;                      { APoint ausschalten    }
  end;
  . . .
  CloseGraph;
end.
```

Merke:

Datenkapselung bedeutet die Verschmelzung von Daten (Datenfeldern, Attributen) und Code (Methoden) in Objekten. Pascal läßt als hybride Sprache zwar auch den direkten Zugriff auf die Daten einer Instanz zu

```
APoint.X := 151;  APoint.Y := 83;
```

OOP bedeutet jedoch Zugriff durch geeignet definierte Methoden, z.B.

```
APoint.Init(152,84);
```

10.3 Vererbung und Erweiterung von Objekten

Wenn ein Nachkomme eines Basisobjektes - als abgeleitetes Objekt - definiert wird, werden sämtliche Datenfelder und Methoden des Vorfahren übernommen (vererbt). Man beachte dabei:

- Vererbte Methoden können durch eigene Methoden des Nachkommen ersetzt werden (Re-definition). Dazu ist eine Methode des gleichen Namens zu definieren. Anweisungsteil und Formalparameter können anders sein (vgl. Bsp.2: `Init`).
- Vererbte Datenfelder dagegen können nicht ersetzt werden (Also darf der Nachkomme kein Datenfeld gleichen Namens definieren).
- Der Nachkomme darf zusätzlich eigene, neue Methoden und Datenfelder erhalten, die den Umfang des Basisobjektes erweitern (vgl. Bsp.2: `Visible, Show, Hide` usw.).
- Auf die Methoden und Datenfelder des Basisobjektes - mit Ausnahme der als `private` deklarierten - darf der Nachkomme direkt zugreifen. Private Daten sind dagegen nicht erreichbar – `private` schützt vor jeglichem äußeren Zugriff.

Beispiel 5 Objekt „Circle“

Ein Objekt `Circle` soll als Nachkomme von `Point` in der erweiterten Unit `Circles` definiert werden.

Datenfelder von `Circle` sind:

X,Y	:	Koordinaten X,Y (vererbt)
Visible	:	Sichtbarkeits-Zustand (vererbt)
Radius	:	Kreisradius (neu!)

Methoden von `Circle` sind:

GetX, GetY	:	Zugriff auf X,Y (vererbt)
IsVisible	:	Statusabfrage (vererbt)
Init	:	Initialisierung von X,Y, Visible, Radius (redefiniert)
Show	:	Zeichnen eines Kreises (redefiniert)
Hide	:	Löschen eines Kreises (redefiniert)
MoveTo	:	Verschieben eines Kreises (redefiniert)
Expand	:	Vergrößerung eines Kreises (neu!)
Contract	:	Verkleinerung eines Kreises (neu!).

Skizze der Objekte:

```
Unit Circles;
Interface
uses Graph;

type  Location = object
        X,Y: Integer;
        procedure Init(InitX, InitY: Integer);
        function GetX: Integer;
        function GetY: Integer;
      end;

      Point = object(Location)
        Visible: Boolean;
        procedure Init(InitX, InitY: Integer);
        procedure Show;
        procedure Hide;
        function IsVisible: Boolean;
        procedure MoveTo(NewX, NewY: Integer);
      end;

      Circle = object (Point)
        Radius: Integer;
        procedure Init(InitX, InitY: Integer;
                       InitRadius: Integer);
        procedure Show;
        procedure Hide;
        procedure Expand(ExpandBy: Integer);
        procedure Contract(ContractBy: Integer);
        procedure MoveTo(NewX, NewY: Integer);
      end;
```

```
Implementation

{ Locations's and Point's method implementations: }
. . . .

{ Circle's method implementations:                }
procedure  Circle.Init(InitX, InitY: Integer;
                       InitRadius: Integer);
  begin
    Point.Init(InitX, InitY);
    Radius := InitRadius;
  end;
procedure Circle.Show;
  begin
    Visible := True;
    Graph.Circle(X, Y, Radius);
  end;
procedure Circle.Hide;
  var TempColor: Word;
  begin
    TempColor := Graph.GetColor;
    Graph.SetColor(GetBkColor);
    Visible := False;
    Graph.Circle(X, Y, Radius);
    Graph.SetColor(TempColor);
  end;
procedure Circle.Expand(ExpandBy: Integer);
  begin
    Hide;
    Radius := Radius + ExpandBy;
    if Radius < 0 then Radius := 0;
    Show;
  end;
procedure Circle.Contract(ContractBy: Integer);
  begin
    Expand(-ContractBy);
  end;
```

```
procedure Circle.MoveTo(NewX, NewY: Integer);
  begin
    Hide;
    X := NewX;
    Y := NewY;
    Show;
  end;

end.
```

Erläuterungen:

1. Aufruf einer vererbten Methode im Nachkommen ist stets möglich mittels des kompletten Methodennamens <Vorfahr>.<Methode>. Dies betrifft alle Vorfahren in der Hierarchie. Beispiele innnerhalb von Circle:

```
Point.Show;           { aber: Show;              }
Point.Init(202,34);   { aber: Init(202,34,6.4);}
Location.Init(55,6);  { indirekter Vorfahre!   }
Location.GetX;        { einfacher: GetX;       }
Point.MoveTo(33,4);   { aber: MoveTo(33,4);    }
```

2. Aufruf von Routinen aus TPU-Dateien ist ebenfalls mittels des kompletten Methodennamens <Vorfahr>.<Methode> möglich. Beispiele in Circle:

```
Graph.Circle(X,Y, Radius);  { Kreis zeichnen }
Graph.SetColor(TempColor);  { Zeichenfarbe setzen }
```

3. Aufbau einer neuen Initialisierungsprozedur Init durch
   - Übernahme der Initialisierungen des Vorgängers
   - Hinzunahme zusätzlicher Initialisierungen.

   Damit wird gesichert, daß der Nachkomme stets den Funktionsumfang des Vorfahren übernimmt – unabhängig von späteren Änderungen am Objekt Point.

Statische und virtuelle Methoden

In Beispiel 3 sind die Methoden `Point.MoveTo` und `Circle.MoveTo` identisch. Weshalb ist letztere dann nötig?

Statische Methoden (statische, frühe Bindung):

- Alle bisher definierten Methoden sind *statisch*: Bei der Complierung wird die Startadresse anstelle des Aufrufs in die aufrufende Methode eingesetzt - und kann nun nicht mehr geändert werden (frühe Bindung). `Point.MoveTo` ruft damit stets die „eigenen“ Methoden `Show` und `Hide` auf.
- Aufrufmechanismus bei Abarbeitung der Folge:

```
var  ACircle: Circle;  n: integer;
. . .
ACircle.Expand(60);
ACircle.Show;
ACircle.IsVisible;
n := ACircle.GetX;
ACircle.MoveTo(121,70);
```

  Methoden werden stets im aktuellen Objekt gesucht, danach im unmittelbaren Vorfahren usw. bis zum Basisobjekt (ansonsten erfolgt eine Fehlermeldung), d.h.

  `Circle -> Point -> Location -> Fehlermeldung.`

- Fehlt `Circle.MoveTo`, so wird deshalb `Point.MoveTo` genommen, das aber voll übersetzt vorliegt. Es benutzt also „seine Methoden“ `Show` und `Hide`. Also wird nur ein Punkt verschoben, nicht jedoch der ganze Kreis!

Virtuelle Methoden (dynamische Bindung, späte Bindung):

- Durch das Schlüsselwort `virtual` wird eine Methode als „virtuell“ deklariert. Alle Methoden dieses Namens in den Nachkommen müssen dann ebenfalls mit `virtual` deklariert werden („Einmal virtuell - immer virtuell!“).
- Mittels einer *Virtuellen-Methoden-Tabelle (VMT)* wird die Verbindung zwischen aufrufender und aufgerufener Methode erst dann hergestellt, wenn ein Aufruf erfolgt, also zur Laufzeit (späte, dynamische Bindung).
- Ein sog. *Konstruktor* erledigt die Initialisierungen bei Vorliegen virtueller Methoden. Er muß vor der virtuellen Methode aufgerufen werden.

Beispiel 6 **Virtuelle Methoden in „Figures“**

Die Methoden `Show` und `Hide` werden virtuell definiert.

```
Unit Figures;
Interface
  uses Graph, Crt;

  type  Location = object
          X,Y: Integer;
          procedure Init(InitX, InitY: Integer);
          function GetX: Integer;
          function GetY: Integer;
        end;
```

```
  Point = object (Location)
    Visible: Boolean;
    constructor Init(InitX, InitY: Integer);
    destructor Done; virtual;
    procedure Show; virtual;
    procedure Hide; virtual;
    function IsVisible: Boolean;
    procedure MoveTo(NewX, NewY: Integer);
  end;

  Circle = object (Point)
    Radius: Integer;
    constructor Init(InitX, InitY: Integer;
                     InitRadius: Integer);
    procedure Show; virtual;
    procedure Hide; virtual;
    procedure Expand(ExpandBy: Integer); virtual;
    procedure Contract(ContractBy: Integer); virtual;
  end;

implementation
. . .
end.
```

Merke: In `Circle` muß nun keine Methode `MoveTo` deklariert werden. Denn `MoveTo` kann von `Point` vererbt werden. Beim Aufruf

```
ACircle.MoveTo(151,83);
```

greift `MoveTo` nun auf die Methoden `Show` und `Hide` von `Circle` zu – und nicht auf diejenigen von `Point` wie im statischen Fall!

********** Ende des Script-Teils 3 von 3 **********