R E K U R S

Rekursive Algorithmen

© Herbert Paukert

[01] Rekursive Strukturen (- 02 -)
[02] Rekursives Scannen von Verzeichnissen (- 09 -)
[03] Rekursives Kopieren von Verzeichnissen (- 12 -)
[04] Iterative und rekursive Grafiken (- 19 -)
[05] Ein Generatorprogramm für Fraktale (- 27 -)
[06] Mandelbrot- und Julia-Mengen (- 28 -)

[01] Rekursive Strukturen

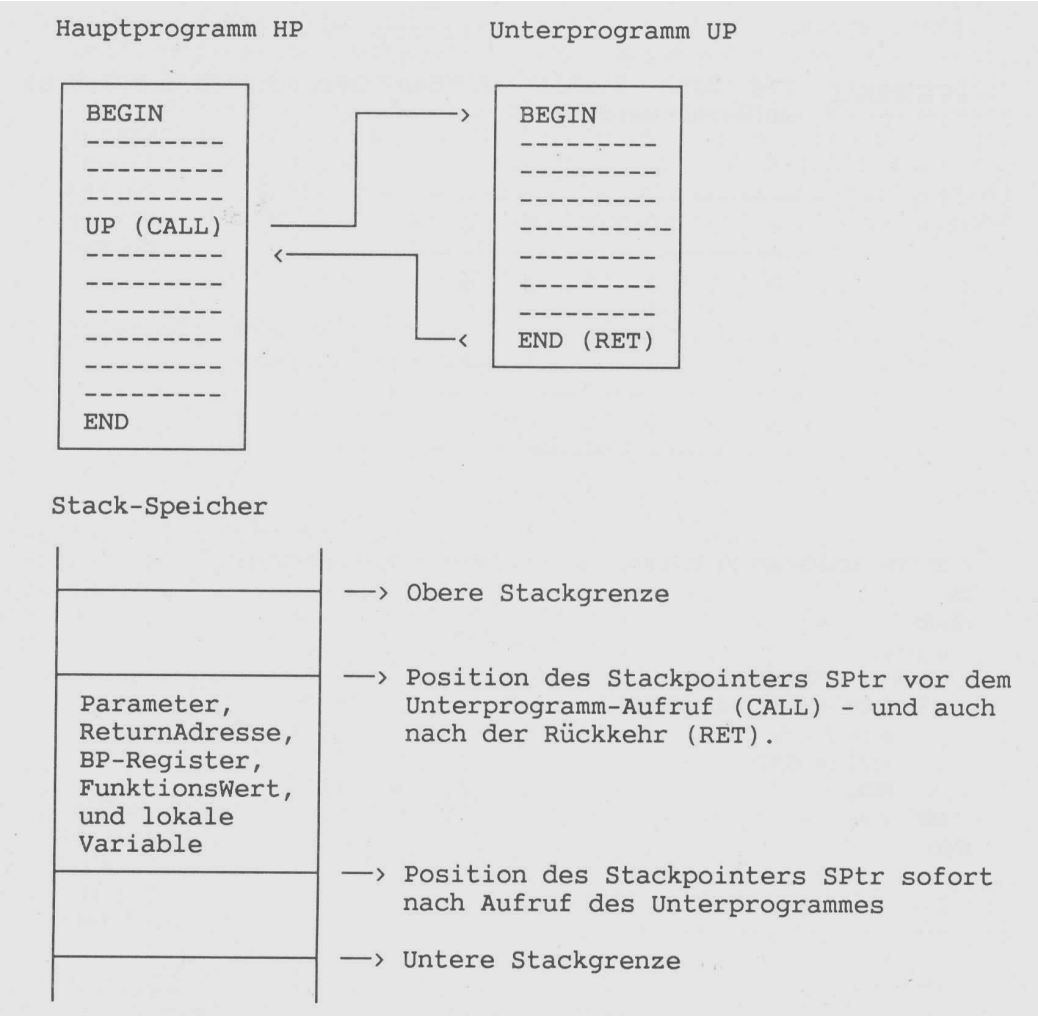
Die Arbeitsweise von Unterprogrammen

In diesem Kapitel soll die Arbeitsweise von Unterprogrammen (Prozeduren, Funktionen), insbesonders von rekursiven Unterprogrammen erläutert werden.

Bei jedem Unterprogramm-Aufruf (CALL) werden auf einem eigenen Stapelspeicher (Stack) alle Parameterwerte, die Rücksprungadresse (RET), der Inhalt des BP-Prozessor-Registers, ein etwaiger Funktionswert und alle lokal definierten Variablen zwischengespeichert. Diesen systeminternen Vorgang nennt man PUSH. Der Stackspeicher stapelt die Daten nach dem LIFO-Prinzip (Last in, First out), d.h., die zuletzt abgelegten Daten werden als erste wieder abgehoben. Das Abheben der Daten nennt man POP. Der Stapel wächst und schrumpft daher dynamisch. Das Wachstum des Stacks erfolgt dabei immer von oben nach unten. Die Speicheradresse der jeweiligen Stackspitze wird in einer eigenen, internen Zeigervariablen SPtr gemerkt.

Bei der Rückkehr zum aufrufenden Prozess (Hauptprogramm) werden die beim Aufruf abgelegten Daten wieder vom Stack abgehoben. Durch das Abheben der gestapelten Rücksprungadresse, kann zu jenem Programmpunkt gesprungen werden, der dem Unterprogramm-Aufruf nachfolgt. Ein etwaig abgelegter Funktionswert kann ebenfalls abgehoben und dem Hauptprogramm zur Verfügung gestellt werden. Alle anderen vom Stack entfernten Daten sind unwiederbringlich verloren.

In der unten abgebildeten schematischen Darstellung ruft ein Hauptprogramm HP ein einfaches Unterprogramm UP auf.



Rekursive Unterprogramme

Die **Rekursion** ist ein Verfahren zur schrittweisen Entwicklung einer Datenstruktur (Gebilde). Vor dem ersten Entwicklungsschritt muss eine Anfangsstruktur S(0) gegeben sein (Rekursionsanfang). Dann ist eine Rekursionsvorschrift gegeben, welche erklärt, wie man die n-te Entwicklungsstufe S(n) aus der vorangehenden (n-1)-ten Entwicklungsstufe S(n-1) erzeugt. Natürlich muss auch angegeben sein, wann die Entwicklung beendet ist (Abbruchbedingung).

Beispiel 1: Die Summe SUM(10) der ersten zehn Zahlen soll rekursiv gebildet werden.

Rekursionsanfang: Anfang mit SUM(0) := 0;
Rekursionsvorschrift: SUM(n) := SUM(n-1) + n;
Rekursionsabbruch: Ende bei n = 10;

Beispiel 2: Das Produkt FAK(10) der ersten zehn Zahlen soll rekursiv gebildet werden.

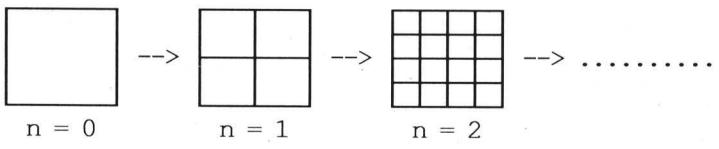
Rekursionsanfang: Anfang mit FAK(1) := 1;
Rekursionsvorschrift: FAK(n) := FAK(n-1) * n;
Rekursionsabbruch: Ende bei n = 10;

Beispiel 3: Die 10-te Potenz POT(x,10) der Zahl x soll rekursiv berechnet werden.

Rekursionsanfang: Anfang mit POT(x,0) := 1;
Rekursionsvorschrift: POT(x,n) := POT(x,n-1) * x;
Rekursionsabbruch: Ende bei n = 10;

Beispiel 4: Ein schachbrettartiger Raster mit 8 x 8 Feldern soll rekursiv gebildet werden.

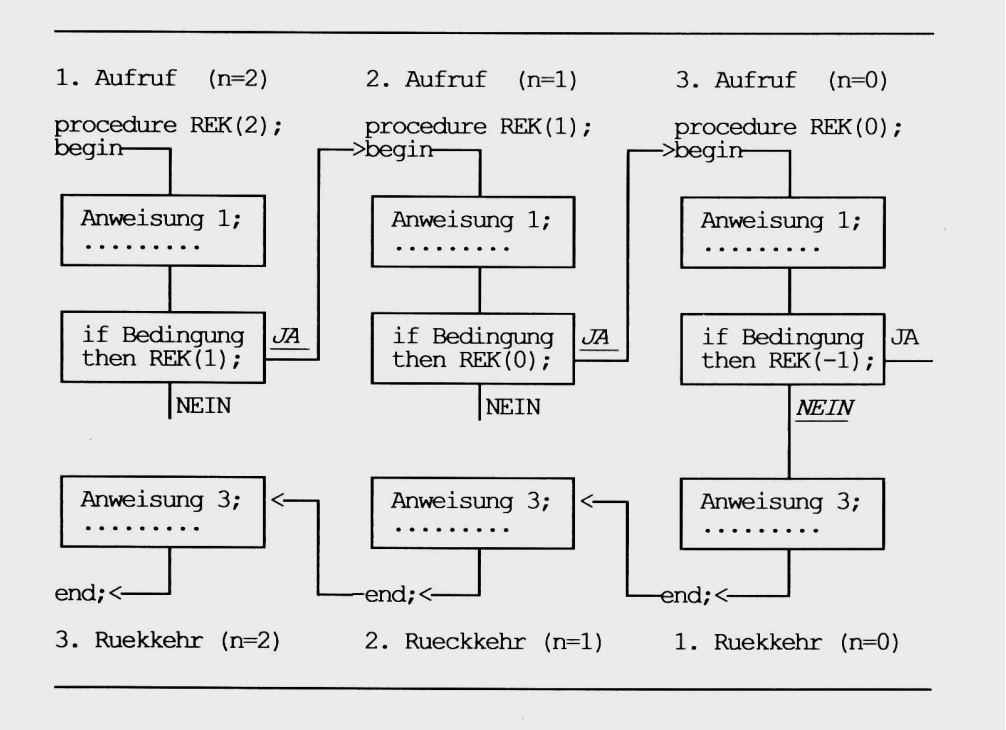
Rekursionsanfang: Anfang mit einem Quadrat der festen Seitenlänge a;
Rekursionsvorschrift: Zerlege das Quadrat in vier kongruente Teilquadrate und führe diese Zerlegung in jedem Teilquadrat durch;
Rekursionsabbruch: Ende nach 3 Rekursionsstufen (weil $2^3 = 8$);



Eine solche Rekursion wird programmtechnisch dadurch realisiert, dass ein Unterprogramm innerhalb seines Codes sich selbst aufruft. Dabei werden alle entsprechenden Daten (Parameter, Rücksprungadresse, lokale Variable) auf einen internen Stapelspeicher (Stack) abgelegt. Ein Parameter sollte bei jedem Aufruf in gewissem Sinne verändert (z.B. immer um Eins verringert) werden. Wird der andauernde Selbstaufruf des Unterprogrammes an eine logische Bedingung in Bezug auf diesen Parameter geknüpft, dann kann ein Abbruch erzwungen werden (z.B. wenn der Parameter Null wird) - andernfalls führen die andauernd aufgestapelten Daten zu einem Stacküberlauf. Wird eine Abbruchbedingung des rekursiven Selbstaufrufes in der Prozedur gesetzt, dann werden bei der Rückkehr ins Hauptprogramm die gestapelten Daten stufenweise so lange vom Stack abgehoben, bis der allerletzte Rücksprung in das Hauptprogramm zurückführt. Eine rekursive Prozedur enthält somit zwei Ablaufwege, nämlich die Anweisungen VOR und die Anweisungen NACH der Abbruchbedingung (rekursiver Einstieg - rekursive Rückkehr).

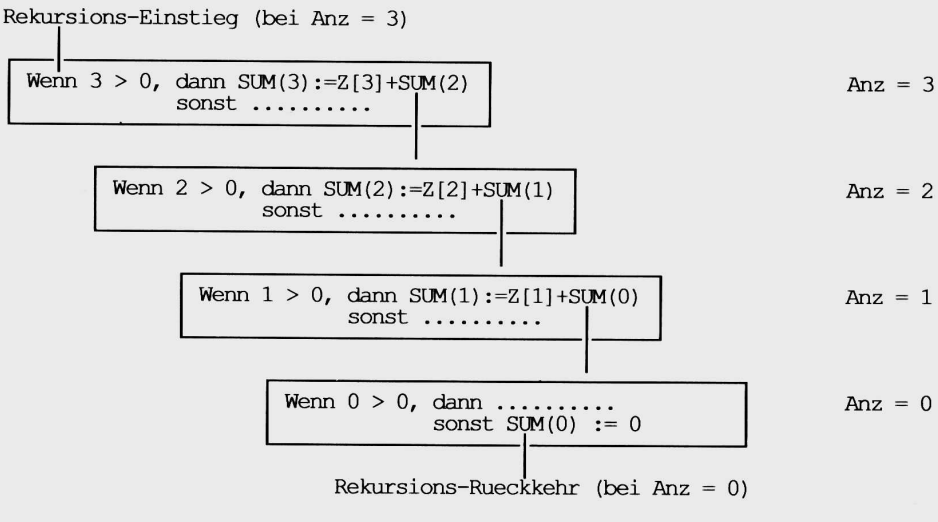
In dem nachfolgenden Beispiel wird bei jedem Aufruf der Prozedur ***REK*** der Werteparameter *n* um Eins verringert. Er wird zur Abbruchbedingung verwendet und steuert somit die Rekursionstiefe. Bei einem Start der Prozedur mit *n = 2* werden am Bildschirm folgende Werte für n ausgegeben: *2 – 1 – 0 – 0 – 1 – 2*. Die Ausgaben der ersten drei Werte erfolgen immer beim Einstieg, die restlichen bei der Rückkehr.

```
procedure REK(n: Integer);
begin
  Form1.Canvas.TextOut(100+20*n,100,IntToStr(n));  // Ausgabe beim Einstieg (1)
  if  n > 0  then  REK(n-1);                       // Abbruchbedingung (2)
  Form1.Canvas.TextOut(100+20*n,200,IntToStr(n));  // Ausgabe bei Rückkehr (3)
end;
```



Nachfolgende Funktion ***SUM*** berechnet rekursiv die Summe aller Zahlen eines Bereiches *Z*. Der erste Aufruf erfolgt im Hauptprogramm durch die Anweisung *N := SUM(Z,Anz).* Der Parameter *Anz* wird bei jedem Aufruf um Eins verringert und gibt somit die erreichte Rekursionstiefe an. Die Rekursion wird dann abgebrochen, wenn *Anz* den Wert Null erreicht. Das eingerahmte Diagramm zeigt schematisch den Ablauf für *Anz* := 3.

```
function SUM(var Z: Bereich; Anz : Integer): Integer;
{ Rekursive Summenbildung }
begin
  if Anz > 0 then SUM := Z[Anz] + SUM(Z,Anz-1)
             else SUM := 0;
end;
```



Zum tieferen Verständnis rekursiver Technik sollen vier weitere Beispiele besprochen werden. Alle besprochenen rekursiven Programmstrukturen und weitere rekursive Routinen befinden sich im Programm "***reku***". Zur Entwicklung rekursiver Grafikgebilde sind im Abschnitt [03] mehrere Programme ausführlich beschrieben.

1. Beispiel: Der größte gemeinsame Teiler zweier Zahlen.

Der größte gemeinsame Teiler *GGT* zweier ganzer Zahlen *a* und *b* soll ermittelt werden. Dabei findet der so genannte Euklidische Algorithmus seine Anwendung: Zuerst wird die Zahl *a* durch die Zahl *b* dividiert und der Rest c bestimmt (*c = a mod b*). Dann wird der Divisor durch den Rest dividiert und der neue Rest berechnet. Dieses Verfahren wird schrittweise so lange fortgesetzt, bis der Rest Null wird. Das muss nach endlich vielen Schritten eintreten, weil Dividend und Divisor immer kleiner werden. Jener letzte Rest, der noch größer als Null ist, ist dann der gesuchte *GGT*, weil er alle vorher erzeugten Dividenden und Divisoren teilt und jeder andere gemeinsame Teiler in ihm enthalten ist.

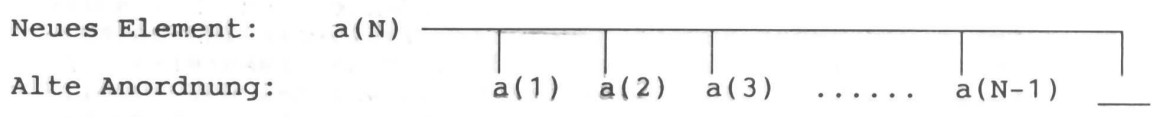
Der beschriebene Algorithmus wird durch die rekursive Funktion ***ggT*** verwirklicht, an welche der jeweilige Dividend und Divisor als Werteparameter übergeben werden.

Beispielsweise gilt: GGT(18,10) := GGT(10,8) := GGT(8,2) := GGT(2,0) := 2

```
function ggT(a,b: Integer): Integer;
{ Rekursive Berechnung des GGT }
begin
    if b > 0 then ggT := ggT(b,(a mod b))
             else ggT := a;
end;
```

2.Beispiel: Die Anordnungen (Permutationen) von n Elementen.

Wenn von (n-1) Elementen alle verschiedenen Anordnungen vorliegen, dann erhält man alle Anordnungen von n Elementen folgendermaßen: Zuerst wird zu jeder vorliegenden Anordnung der (n-1) Elemente das neue Element als letztes hinzugefügt und danach wird es mit jedem Element der jeweiligen Anordnung vertauscht. Damit ergeben sich n verschiedene Lagemöglichkeiten für das neue Element innerhalb einer alten Anordnung.



Wenn P(n-1) die Anzahl aller Anordnungen von (n-1) Elementen ist, so erhält man die Anzahl P(n) der Anordnungen von n Elementen mittels nachfolgender Formel:

P(n) = n * P(n-1) = 1 * 2 * 3 * ..... * (n-1) * n

In der Prozedur ***permut*** wird die Anzahl der Elemente n als Parameter übergeben. Dann werden n aufeinander folgende Buchstaben a, b, c ..... als Array-Elemente gebildet. Diese Elemente sollen nun permutiert und in einer *RichEdit*-Komponente ausgegeben werden. Dazu vertauscht man hintereinander das letzte Element der Ausgangsanordnung mit dem ersten, zweiten, dritten ..... Element. Vor jeder Vertauschung muss durch Rücktausch die Ausgangsanordnung wiederhergestellt werden. Nach jeder Vertauschung wird das letzte Element einer Anordnung festgehalten und die restlichen (n-1) Elemente permutiert. Beispielsweise sollen drei Elemente permutiert werden:

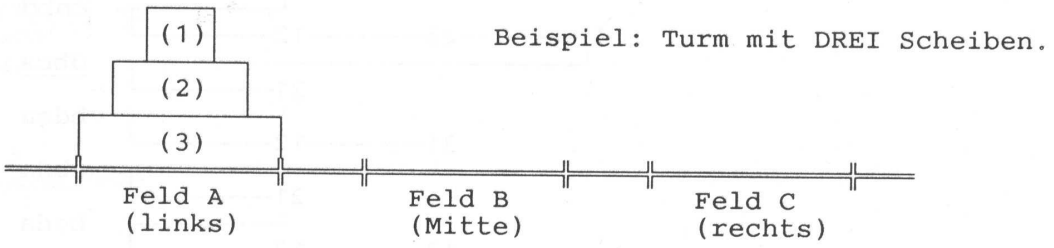
(abc) (abc) (abc)
abc, bac, cba, bca, acb, cab

```
procedure permut(RE: TRichEdit; n: Integer);
// Ermittlung aller Permutationen von n Elementen

type Feld = array[1..255] of Char;
var  el   : Feld;
     Anz,i: Integer;

     function PermAnz(k: Integer): Integer;
     // Anzahl der Permutationen
     begin
       if k = 1 then PermAnz := 1
                else PermAnz := k * Permanz(k-1);
     end;

     procedure Eingabe(n: Integer);
     // Eingabe der n Feldelemente
     var i : Integer;
     begin
       for i := 1 to n do el[i] := Chr(96+i);
     end;

     procedure Ausgabe(n: Integer);
     // Ausgabe einer Permutation
     var s : String;
         i : Integer;
     begin
       s := '';
       for i := 1 to n do s := s + el[i];
       RE.Lines.Add(#9+s);
     end;

     procedure Tausch(var x,y: Char);
     // Platztausch zweier Elemente
     var z: Char;
     begin
       z := x; x := y; y := z;
     end;

     procedure Perm(k: Integer);
     // Ermittelt alle Permutationen von n Elementen
     var i : Integer;
     begin
       if k = 1 then Ausgabe(n)
       else begin
         Perm(k-1);
         For i := 1 to k-1 do begin
           Tausch(el[i],el[k]);
           Perm(k-1);
           Tausch(el[i],el[k]);
         end;
       end;
     end;


// Hauptprogramm der Permutationen
begin
  Eingabe(n);
  Anz := PermAnz(n);
  with RE do begin
    Visible := True;
    Clear;
    Lines.Add(#32);
    Lines.Add(' Permutationen von '+IntToStr(n)+' Elementen = '+IntToStr(Anz));
    Lines.Add(#32);
  end;
  Perm(n);
end;
```

3. Beispiel: Die Türme von Hanoi.

Gegeben sind drei Felder A (Links), B (Mitte) und C (Rechts) und n verschieden große Scheiben. Diese Scheiben liegen am Anfang auf Feld A nach zunehmender Größe gestapelt. Die Aufgabe besteht nun darin, den Scheibenturm von Feld A nach Feld C so zu transportieren, dass seine Anordnung erhalten bleibt. Für den Scheibentransport gelten folgende Regeln:

[1] Es darf immer nur EINE Scheibe transportiert werden.
[2] Es darf immer nur eine KLEINERE auf einer GRÖSSEREN Scheibe liegen.
[3] Das Feld B darf zur Zwischenspeicherung von Scheiben benutzt werden.



Zur Lösung der Aufgabe wird der Turmaufbau rekursiv durchgeführt, indem ein Turm immer aus einer größten Scheibe an der Basis und einem darüber liegenden Turm mit (n-1) Scheiben besteht. Offenkundig richtet sich der Scheibentransport nach der jeweiligen Anzahl von Turmscheiben. Ist diese ungerade, dann muss die oberste Scheibe direkt auf das Zielfeld gelegt werden. Ist die Anzahl hingegen gerade, so muss der Scheibentransport auf der Zwischenablage beginnen. Diese Überlegungen müssen nun rekursiv für alle Teiltürme oberhalb der Basis realisiert werden, bis jeder Teilturm auf eine Scheibe geschrumpft ist.

Die unten stehende Prozedur ***hanoi*** zeigt die rekursive Lösung des Turmproblems von Hanoi.

```
procedure hanoi(RE: TRichEdit; n: Integer);
// Ermittlung der Türme von Hanoi mit n Turmscheiben

type Platz = String[6];
var  Anz: Integer;

     procedure Anzeigen(Scheibe: Integer; Anfa,Ende: Platz);
     // Anzeigen der Umlegung einer Scheibe von Anfa auf Ende
     var S: String;
     begin
       Anz := Anz + 1;
       Str(Anz:3,S);
       S := ' [' + S + ']   ' + IntToStr(Scheibe) + '.Scheibe:   '
            + Anfa + '  -->  ' + Ende;
       RE.Lines.Add(S);
     end;

     procedure Turmbauen(Scheibe: Integer; Start,Ablage,Ziel: Platz);
     // Rekursiver Aufbau des Turmes
     begin
       if Scheibe = 1 then Anzeigen(Scheibe,Start,Ziel)
       else begin
         Turmbauen(Scheibe-1,Start,Ziel,Ablage);
         Anzeigen(Scheibe,Start,Ziel);
         Turmbauen(Scheibe-1,Ablage,Start,Ziel);
       end;
     end;
```

```
// Hauptprogramm der Türme von Hanoi
begin
  with RE do begin
    Visible := True;
    Clear;
    Lines.Add(#32);
    Lines.Add(' Türme von Hanoi ');
    Lines.Add(#32);
    Lines.Add(' Anzahl der Turmscheiben: ' + IntToStr(n));
    Lines.Add(#32);
  end;

  Turmbauen(n,'Links ','Mitte ','Rechts');

  with RE do begin
    Lines.Add(#32);
    Lines.Add(' Anzahl der Umlegungen: ' + IntToStr(Anz));
    Lines.Add(#32);
  end;
end;
```

Nachfolgend wird das Ergebnis des Turmproblems mit vier Scheiben angezeigt:

```
Türme von Hanoi
Anzahl der Turmscheiben: 4
Anzahl der Umlegungen  : 15

[  1]   1.Scheibe:   Links   -->  Mitte
[  2]   2.Scheibe:   Links   -->  Rechts
[  3]   1.Scheibe:   Mitte   -->  Rechts
[  4]   3.Scheibe:   Links   -->  Mitte
[  5]   1.Scheibe:   Rechts  -->  Links
[  6]   2.Scheibe:   Rechts  -->  Mitte
[  7]   1.Scheibe:   Links   -->  Mitte
[  8]   4.Scheibe:   Links   -->  Rechts
[  9]   1.Scheibe:   Mitte   -->  Rechts
[ 10]   2.Scheibe:   Mitte   -->  Links
[ 11]   1.Scheibe:   Rechts  -->  Links
[ 12]   3.Scheibe:   Mitte   -->  Rechts
[ 13]   1.Scheibe:   Links   -->  Mitte
[ 14]   2.Scheibe:   Links   -->  Rechts
[ 15]   1.Scheibe:   Mitte   -->  Rechts
```

4. Beispiel: Konvertierung und Rekonvertierung von Zahlen.

Die Prozedur ***Konvert*** zeigt eine elegante Anwendung der Rekursion zur Umwandlung einer ganzen Dezimalzahl in ein anderes Ziffernsystem. Dabei wird die ZAHL fortlaufend durch die BASIS dividiert und der jeweilige REST der Division bestimmt. Dieser wird in das entsprechende ZEICHEN im ASCII-Code umgewandelt und ergibt das Ziffernzeichen an der jeweiligen Stelle.

```
procedure Konvert(Zahl: Integer; var Basis: Integer; var Zeichen: String);
var Rest: Byte;
begin
  Zeichen := '';
  if (Basis < 2) or (Basis > 35) then Exit;
  if Zahl > 0 then Konvert(Zahl div Basis,Basis,Zeichen);
  Rest := Zahl Mod Basis;
  if Rest < 10 then Zeichen := Zeichen + chr(48 + Rest)
               else Zeichen := Zeichen + chr(55 + Rest);
end;
```

Die Funktion ***Rekonvert*** ist das Gegenstück zur Prozedur ***Konvert***. Es wandelt eine Zahl aus einem anderen Ziffernsystem wieder in eine Dezimalzahl um. Dabei wird aber nicht rekursiv, sondern iterativ vorgegangen.

```
function ReKonvert(Zeichen: String; var Basis: Integer): Integer;
const alfa = ['0'..'9','A'..'Z'];
var   Zahl: Integer;
      I,X : Integer;
      CH: Char;
begin
  Result := 0;
  if (Basis < 2) or (Basis > 35) then Exit;
  Zahl := 0;
  For I := 1 to Length(Zeichen) do begin
    CH := UpCase(Zeichen[I]);
    if (CH in ALFA) then begin
       if CH >= 'A' then X := ord(CH) - 55
                    else X := ord(CH) - 48;
       Zahl := (Zahl * Basis) + X;
    end
    else Exit;
  end;
  Result := Zahl;
end;
```

[02] Rekursives Scannen von Verzeichnissen

Im Programm "***dscan***" wird zuerst ein bestimmtes Verzeichnis mit Hilfe einer *DirectoryBox* ausgewählt und daraus ein entsprechender Verzeichnispfad (*Pfad*) gebildet. Als Nächstes wird eine bestimmte Erweiterung (*FExt*) eines Dateinamens oder einfach nur ein Stern (*) eingegeben, was jede beliebige Datei bedeuten soll.

Dann erfolgt der Aufruf der Routine *DirScan1(Pfad,FExt).* Diese rekursive Prozedur bewirkt ein rekursives Durchlaufen des gesamten Verzeichnisbaumes, der unterhalb von *Pfad* liegt. In jedem dabei erreichten Unterverzeichnis werden von jeder Datei mit der Namenserweiterung *FExt* ihr Name, ihre Größe und ihr Dateiattribut gelesen und diese Kenndaten fortlaufend in der Stringliste *DatList* gespeichert. Außerdem werden alle Dateien (files) und Verzeichnisse (directories) gezählt und deren Anzahlen in den Variablen *af* und *ad* festgehalten.

Um eine Datei zu suchen, wird ihr Namen oder ein Teil davon (*Such*) eingegeben und die Routine *FileSearch(Such)* aufgerufen. Dabei wird die Stringliste *DatLis*t sequentiell nach dem Suchbegriff *Such* durchsucht. Bei Erfolg werden von der gefundenen Datei deren Kenndaten fortlaufend in der Stringliste *FoundList* gespeichert. Außerdem werden alle Fundstellen gezählt (*ff*). Zusätzlich werden die beiden Stringlisten *DatList* und *FoundList* in zwei externe Textdateien *scanfile.txt* und *scanfound.txt* abgespeichert und können von dort in eine *RichEdit*-Komponente geladen und angezeigt werden.

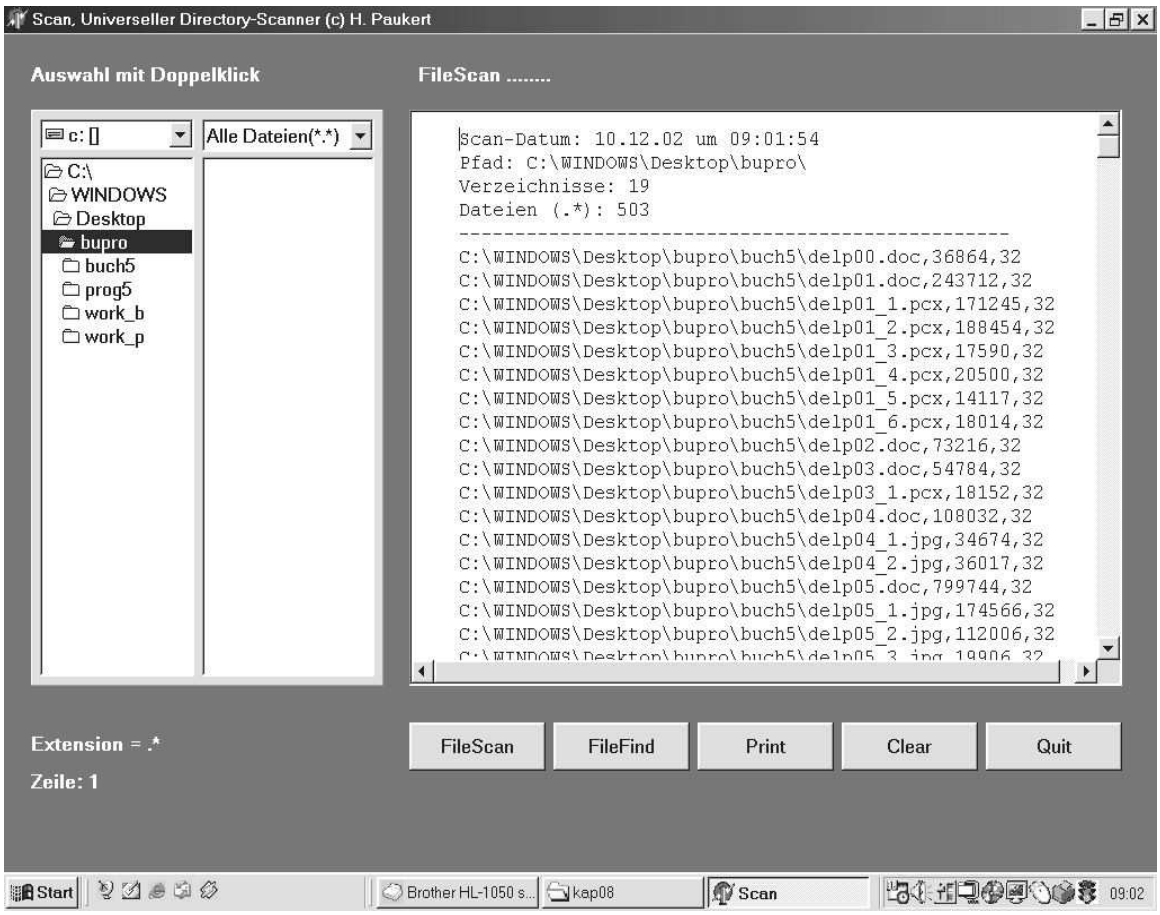
Sehr wichtig ist der vordefinierte Objekttyp *TSearchRec*, welcher alle Kennwerte einer Datei aufnimmt (Name, Datum, Größe, Attribut usw.). Für die einzelnen Dateiattribute (*TSearchRec.Attr*) besitzt DELPHI bereits vordefinierte Konstante (*faAnyFile*, *faDirectory* usw.).

Die Funktion *FindFirst(Pfad,Attribut,SearchRec)* sucht in einem angegebenen Verzeichnis (*Pfad*) nach Dateien mit einem bestimmten Attribut. Bei Erfolg wird der Wert Null zurückgeliefert, andernfalls ein Fehlercode. Um im Verzeichnis weiter zu suchen, verwendet man die Funktion *FindNext(SearchRec).* Ist die Suche beendet, wird die Prozedur *FindClose(SearchRec)* aufgerufen, um belegte Speicherbereiche frei zu geben.

Zum besseren Verständnis wird zunächst der Quellcode der Routine ***DirScan*** aufgelistet, welche nichts anderes tut, als einen gegebenen Verzeichnisbaum rekursiv zu durchlaufen und alle vorhandenen Verzeichnisse und Dateien zu zählen und sie zusätzlich in einer *RichEdit*-Komponente auszugeben.

```
procedure DirScan(Pfad: String);
// Einfacher Directory-Scanner mit Anzeige aller Dateien
// in allen Unterverzeichnissen von "Pfad"
var SRec: TSearchRec;
    R: Integer;
begin
  R := FindFirst(Pfad + '*.*',faAnyFile,SRec);
  While R = 0 do begin
    if Copy(SRec.Name,1,1) <> '.' then begin
       if (SRec.Attr and faDirectory) > 0 then begin
          DirScan(Pfad + SRec.Name + '\');
          Inc(ad);
       end
       else begin
          Form1.RichEdit1.Lines.Add(Pfad + SRec.Name);
          Inc(af);
       end;
    end;
    R := FindNext(SRec);
  end;
  FindClose(SRec);
end;
```

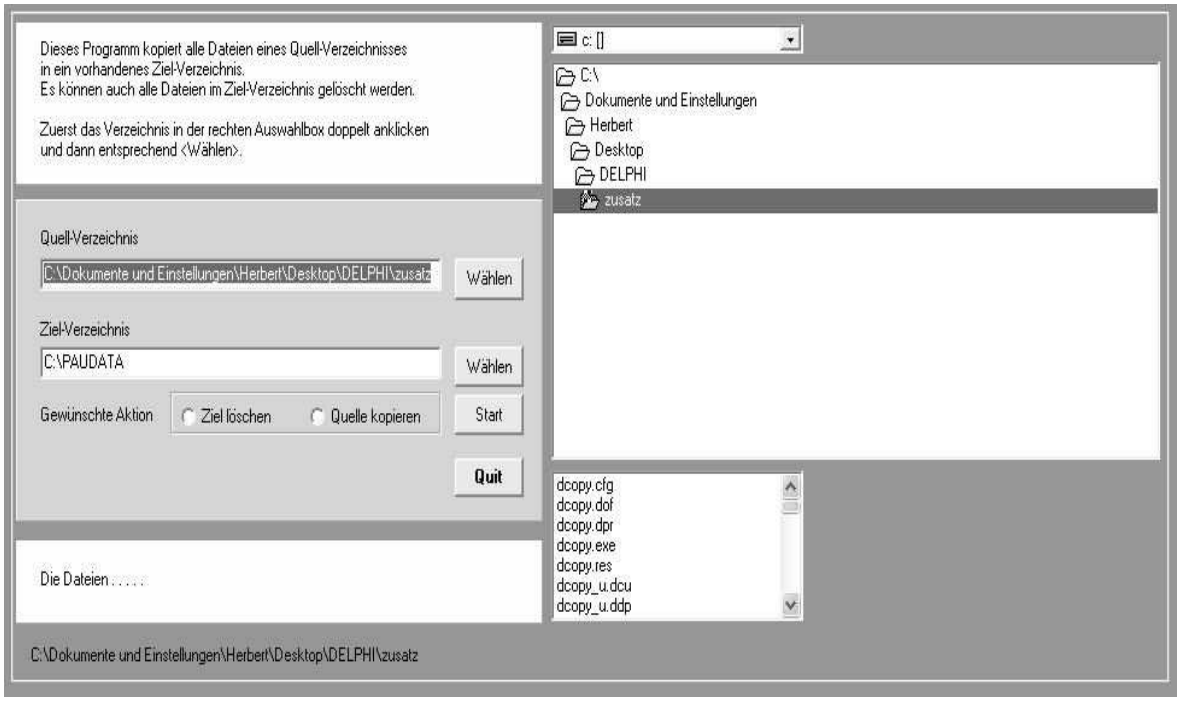
Nachfolgend ist vom Programm"***dscan***" das Formular abgebildet. Dann folgen alle Variablen-Deklarationen und die Auflistung der eigentlichen Kernroutine ***DirScan1*** und auch der Suchroutine ***SearchFile***, welche oben bereits ausfühlich besprochen wurden.



```
const dat1 = 'scanfile.txt';     // Alle gescannten Dateien
      dat2 = 'scanfound.txt'     // Alle gefundenen Dateien

var   DatList : TStringList;     // Stringliste aller gescannten Dateien
      FoundList : TStringList;   // Stringliste der gefundenen Dateien

      f1,f2: TextFile;           // Dateilisten
      pfad:  string;             // Laufwerk bzw. Verzeichnis
      D1: string;                // Scan-Datum

      ad,af,ff: Word;            // Anzahlen: Verzeichnisse, Dateien, Funde

      Verz : String;             // Aktuelles Verzeichnis
      FExt : String;             // Dateiextension


procedure DirScan1(Pfad,FExt:string);
// Erweiterter Directory-Scanner mit Speicherung der Datei-Kennwerte
// in einer Stringliste "DatList"
var SRec: TSearchRec;
    R   : Integer;
    Ext,s0,s1,s2: string;
begin
  R := FindFirst(pfad + '*.*',faAnyFile,SRec);
  While R = 0 do begin
    if Copy(SRec.Name,1,1) <> '.' then begin
       if (SRec.Attr and faDirectory) > 0 then begin
          DirScan1(Pfad + SRec.Name + '\',FExt);
          inc(ad);
       end
       else begin
          s1 := IntToStr(SRec.Size);
          s2 := IntToStr(SRec.Attr);
          if FExt = '.*' then begin
             inc(af);
             s0 := pfad+SRec.Name+','+s1+','+s2;
             DatList.Add(s0);
          end
          else begin
             Ext := LowerCase(ExtractFileExt(SRec.Name));
             if Ext = FExt then begin
                inc(af);
                s0 := pfad+SRec.Name+','+s1+','+s2;
                DatList.Add(s0);
              end;
            end;
        end;
       end;
       R := FindNext(SRec);
  end;
  FindClose(SRec);
end;

procedure FileSearch(Such: String);
// Teil eines Dateinamens in der Stringliste "DatList" suchen und
// gefundene Dateien in einer zweiten Liste "FoundList" speichern
var S,T : String;
    i: Integer;
begin
  S := Trim(LowerCase(Such));
  ff := 0;
  For i := 0 to DatList.Count-1 do begin
      T := Trim(LowerCase(DatList[i]));
      if Pos(S,T) > 0 then begin
         Inc(ff);
         FoundList.Add(T);
      end;
  end;
end;
```

[03] Rekursives Kopieren von Verzeichnissen

Das Programm „dcopy“ ermöglicht erstens das Kopieren und zweitens das Löschen eines ganzen Verzeichnisbaumes.




```
unit dcopy_u;
// Kopieren und Löschen von Dateien und Ordnern (c) Herbert Paukert

interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls,
     Forms, Dialogs, StdCtrls, ExtCtrls, FileCtrl;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
    Panel3: TPanel;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Bevel1: TBevel;
    RadioGroup1: TRadioGroup;
    DriveComboBox1: TDriveComboBox;
    DirectoryListBox1: TDirectoryListBox;
    FileListBox1: TFileListBox;
```

```
    procedure FormCreate(Sender: TObject);
    procedure FileListBox1DblClick(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
  private { Private-Deklarationen }
  public  { Public-Deklarationen }
end;

type EInvalidDest = class(EStreamError);
     EFCantMove   = class(EStreamError);

var  Form1: TForm1;

implementation
{$R *.DFM}

var Verz1,Verz2,S1,S2: String;
    Actio: Integer;

const SInvalidDest  = 'Destination %s does not exist';
      SFCantMove    = 'Cannot move file %s';
      SFOpenError   = 'Cannot open this file';
      SFCreateError = 'Cannot create this file';

function HasAttr(const FileName: string; Attr: Word): Boolean;
// Überprüft das Attribut einer Datei
begin
  Result := (FileGetAttr(FileName) and Attr) = Attr;
end;

function GetFileSize(const FileName: string): LongInt;
// Ermittelt Größe einer Datei
var SearchRec: TSearchRec;
begin
  if FindFirst(ExpandFileName(FileName), faAnyFile, SearchRec) = 0 then
     Result := SearchRec.Size
  else Result := -1;
end;

procedure CopyFile(const FileName, DestName: string);
// Kopiert die Datei "FileName" in ein Verzeichnis "DestName"
const ChunkSize   : Longint = 65536;
var   CopyBuffer  : Pointer;
      BytesCopied : Longint;
      Source, Dest: Integer;
      Destination : TFileName;
begin
  Destination := ExpandFileName(DestName);
  if HasAttr(Destination, faDirectory) then
     if Destination[Length(Destination)] = '\' then
        Destination := Destination + ExtractFileName(FileName)
     else
        Destination := Destination + '\' + ExtractFileName(FileName);
  GetMem(CopyBuffer, ChunkSize);
```

```
  try
    Source := FileOpen(FileName, fmShareDenyWrite);
    if Source < 0 then
       raise EFOpenError.CreateFmt(SFOpenError, [FileName]);
    try
      Dest := FileCreate(Destination);
      if Dest < 0 then
         raise EFCreateError.CreateFmt(SFCreateError, [Destination]);
      try
        repeat
          BytesCopied := FileRead(Source, CopyBuffer^, ChunkSize);
          if BytesCopied > 0 then FileWrite(Dest, CopyBuffer^, BytesCopied);
        until BytesCopied < ChunkSize;
      finally
        FileClose(Dest);
      end;
    finally
      FileClose(Source);
    end;
  finally
    FreeMem(CopyBuffer, ChunkSize);
  end;
end;

procedure CopyAllFiles(SourceName, DestName: string);
// Kopiert alle Dateien aus einem Quell- in ein Ziel-Verzeichnis
var SRec: TSearchRec;
    N,R : Integer;
    DName,DName1: String;
begin
  if (SourceName[Length(SourceName)] = '\') then
     SourceName := Copy(SourceName,1,Length(SourceName)-1);
  if DestName[Length(DestName)] = '\' then
     DestName := Copy(DestName,1,Length(DestName)-1);
  try
    MkDir(DestName);
  except
  end;
  N := 0;
  R := FindFirst(SourceName + '\*.*',faAnyFile,SRec);
  While R = 0 do begin
    DName := SourceName + '\' + SRec.Name;
    if NOT HasAttr(DName,faDirectory) then begin
       Inc(N);
       CopyFile(DName,DestName);
       DName1 := DestName + '\' + SRec.Name;
       if (SRec.Attr and faReadOnly) > 0 then
          SetFileAttributes(PChar(DName1),FILE_ATTRIBUTE_NORMAL);
    end;
    R := FindNext(SRec);
  end;
  FindClose(SRec);
end;
```

```
procedure CopyAll(SourceName, DestName: string);
// Kopiert einen kompletten Verzeichnisbaum in ein Zielverzeichnis

procedure CopyAllDir(SourceName, Source, Destination: string);
// Kopiert ALLES aus einem Quell- in ein Ziel-Verzeichnis
// Die Parameter "SourceName" und "Source" sind IDENTISCH
var SRec: TSearchRec;
    R: Integer;
    DName0,DName1: String;
begin
  if (SourceName[Length(SourceName)] = '\') then
     SourceName := Copy(SourceName,1,Length(SourceName)-1);
  if Destination[Length(Destination)] = '\' then
     Destination := Copy(Destination,1,Length(Destination)-1);
  if SourceName = Source then begin
     CopyAllFiles(Source,Destination);
  end;
  R := FindFirst(SourceName + '\*.*',faAnyFile,SRec);
  While R = 0 do begin
    DName0 := SourceName + '\' + SRec.Name;
    if HasAttr(DName0,faDirectory) then begin
       if Copy(SRec.Name,1,1) <> '.' then begin
          DName1 := DName0;
          Delete(DName1,1,Length(Source));
          DName1 := Destination + DName1;
          try
            MkDir(DName1);
          except
          end;
          CopyAllFiles(DName0,DName1);
          CopyAllDir(DName0,Source,Destination);
       end;
    end;
    R := FindNext(SRec);
  end;
  FindClose(SRec);
end;

begin
  try
    MkDir(DestName);
  except
  end;
  CopyAllDir(SourceName,SourceName,DestName);
end;

procedure DelAllFiles(SourceName: string);
// Löscht alle Dateien aus einem Quell-Verzeichnis
var SRec: TSearchRec;
    N,R : Integer;
    DName: String;
begin
  if SourceName[Length(SourceName)] = '\' then
     SourceName := Copy(SourceName,1,Length(SourceName)-1);
  N := 0;
  R := FindFirst(SourceName + '\*.*',faAnyFile,SRec);
```

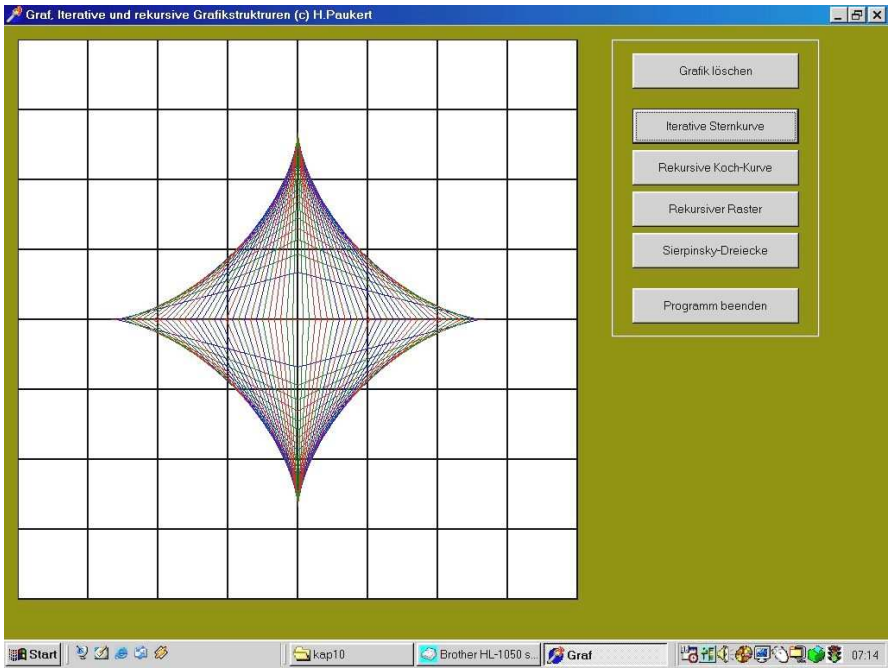
```
 While R = 0 do begin
    DName  := SourceName + '\' + SRec.Name;
    if NOT HasAttr(DName,faDirectory) then begin
       if SRec.Name <> 'worte.txt' then begin
          Inc(N);
          DeleteFile(PChar(DName));
       end;
    end;
    R := FindNext(SRec);
  end;
  FindClose(SRec);
end;

procedure DelAll(SourceName: string);
// Löscht einen kompletten Verzeichnisbaum "SourceName"

procedure DelAllDir(SourceName, Source: string);
// Löscht ALLES aus einem Quell-Verzeichnis
// Die Parameter "SourceName" und "Source" sind IDENTISCH
var SRec: TSearchRec;
    R: Integer;
    DName0: String;
begin
  if (SourceName[Length(SourceName)] = '\') then
     SourceName := Copy(SourceName,1,Length(SourceName)-1);
  if SourceName = Source then begin
     DelAllFiles(Source);
  end;
  R := FindFirst(SourceName + '\*.*',faAnyFile,SRec);
  While R = 0 do begin
    DName0 := SourceName + '\' + SRec.Name;
    if HasAttr(DName0,faDirectory) then begin
       if Copy(SRec.Name,1,1) <> '.' then begin
          DelAllFiles(DName0);
          DelAllDir(DName0,Source);
          RemoveDir(DName0);
       end;
    end;
    R := FindNext(SRec);
  end;
  FindClose(SRec);
end;

begin
  DelAllDir(SourceName,SourceName);
  RemoveDir(SourceName);
end;

function RemoveDirSlash(V: String): String;
// Entfernt von einem Verzeichnisnamen V den lezten Backslash
begin
  if (V[Length(V)] = '\') then Result := Copy(V,1,Length(V)-1)
                          else Result := V;
end;
```

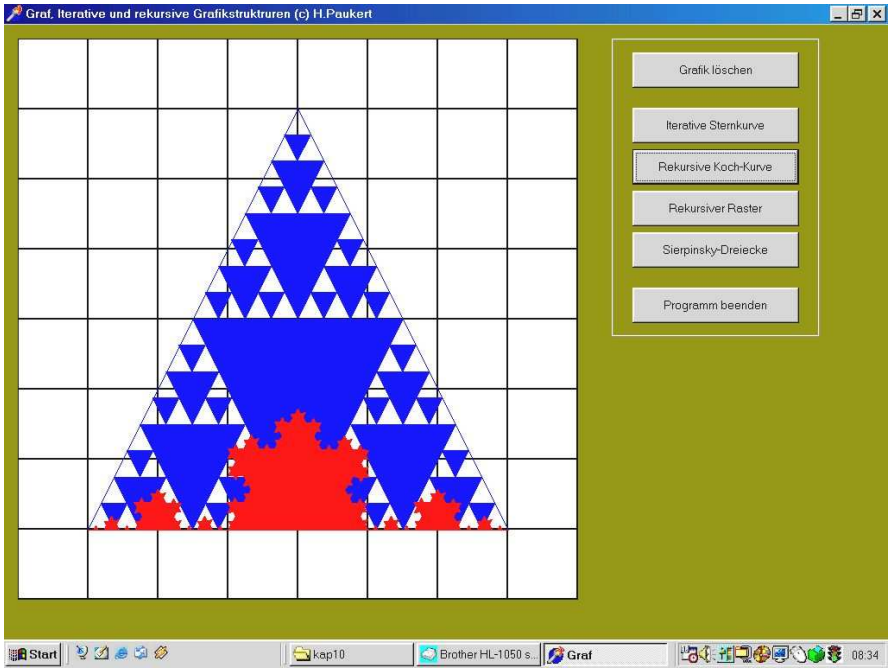
```
function RemoveBlank(S: String): String;
// Entfernt alle Blanks aus einem String;
begin
  while Pos(#31,S) > 0 do Delete(S,Pos(#32,S),1);
  Result := S;
end;

procedure TForm1.FormCreate(Sender: TObject);
// Initialisierungen
begin
  Form1.Color := RGB(140,150,170);
  GetDir(0,Verz1);
  Verz1 := RemoveDirSlash(Verz1);
  Verz2 := 'C:\PAUDATA';
  S1 := Verz1;
  S2 := Verz2;
  Edit1.Text := S1;
  Edit2.Text := S2;
  Label2.Caption := 'Dieses Programm kopiert ein Quell-Verzeichnis '  + #13 +
                    'in ein vorhandenes Ziel-Verzeichnis.'+#13+
                    'Es kann das Ziel-Verzeichnis gelöscht werden.' + #13 + #13 +
                    'Zuerst Verzeichnis in der Box doppelt anklicken' + #13 +
                    'und dann entsprechend <Wählen>.';
  RadioGroup1.ItemIndex := -1;
end;

procedure TForm1.Button1Click(Sender: TObject);
var Error : Boolean;
    A: Integer;
begin
  Label5.Caption := 'Die Dateien.....';
  S1 := RemoveDirSlash(RemoveBlank(LowerCase(Edit1.Text)));
  S2 := RemoveDirSlash(RemoveBlank(LowerCase(Edit2.Text)));
  Actio := RadioGroup1.ItemIndex;
  if Actio = -1 then begin
     ShowMessage('Bitte gewünschte Aktion auswählen !');
     Exit;
  end;
  A := MessageBox(0,'Aktion wirklich durchführen ?','Frage',36);
  if A <> 6 then begin
     ShowMessage('Aktion abgebrochen !');
     Exit;
  end;
  if Not DirectoryExists(S1) then begin
     Error := True;
     ShowMessage(S1 + ' NICHT gefunden !');
     Exit;
  end;
  if Not DirectoryExists(S2) then begin
     Error := True;
     ShowMessage(S2 + ' NICHT gefunden !');
     Exit;
  end;
  if Actio = 0 then Label5.Caption := 'Die Dateien werden gelöscht . . . . .';
  if Actio = 1 then Label5.Caption := 'Die Dateien werden kopiert . . . . .';
  AppliCation.ProcessMessages;
  Screen.Cursor := crHourGlass;
  Error := False;
```

```
  try
    if Actio = 0 then begin
       try
         RemoveDir(S2);
       except
       end;
       DelAll(S2);
    end;
    if Actio = 1 then begin
       CopyAll(S1,S2);
    end;
  except
    Error := True;
    ShowMessage('File-Error');
  end;
  Screen.Cursor := crDefault;
  if Actio = 0 then Label5.Caption := 'ALLE Dateien gelöscht !';
  if Actio = 1 then Label5.Caption := 'ALLE Dateien kopiert !';
  if Error then Label5.Caption := 'KEINE Datei gelöscht oder kopiert !';
  DirectoryListBox1.Update;
  FileListBox1.Update;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Application.Terminate;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  Edit1.Text := Label6.Caption;
  Label5.Caption := 'Die Dateien.....';
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  Edit2.Text := Label6.Caption;
  Label5.Caption := 'Die Dateien.....';
end;

procedure TForm1.FileListBox1DblClick(Sender: TObject);
var S: String;
    L: LongInt;
begin
  S := '';
  L := 0;
  S := Label6.Caption + '\' + Edit3.Text;
  L := Round(GetFileSize(S)/1000);
  Label5.Caption := 'Die Dateien.....';
  ShowMessage(S + ' = ' + IntToStr(L) + ' KB');
end;

end.
```

[04] Iterative und rekursive Grafiken

Das Programm "***rekugraf***" demonstriert die Erzeugung von iterativen und rekursiven grafischen Gebilden. Die Abbildungen zeigen das Formular mit seinen verschiedenen Komponenten.



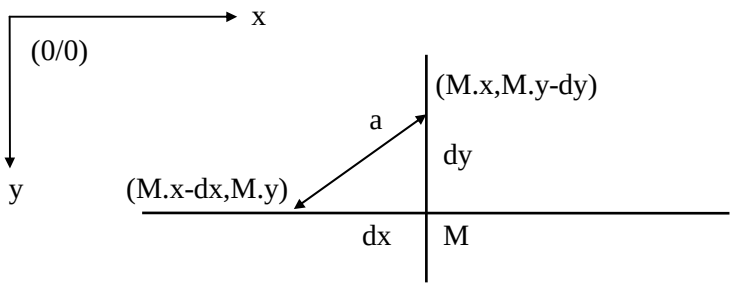


Am Programmanfang müssen folgende globale Variable deklariert und initialisiert werden:

```
var Xmax,Ymax: Integer;
    M: TPoint;

with Form1.Image1 do begin
  Xmax := Width;                // horizontale und vertikale Bildabmessungen
  Ymax := Height;
  M.X  := Round(Xmax/2);        // Mittelpunkt M des Bildes
  M.Y  := Round(Ymax/2);
end;
```

Iterative Erzeugung von grafischen Gebilden

Die Prozedur *GrafikDemo1*, die mit Hilfe eines entsprechenden Schaltknopfes aufgerufen werden kann, erzeugt in einer Wiederholungsschleife (**iterativ**) ein Grafikgebilde, das in oben stehender Abbildung dargestellt ist. Dabei wird in der Bildmitte *M(x,y)* ein Achsenkreuz gedacht und eine Strecke *a* mit konstanter Länge *(Round(Xmax/3))* gleitet mit ihren Endpunkten entlang der Achsen. Dieser Prozess findet in allen vier Quadranten statt. Die einhüllende Kurve nennt man Asteroide. Vorgegeben ist die *Anzahl* der Verschiebungen, woraus sich die Verschiebungsstücke *dx* und *dy* entlang der gedachten Achsen errechnen lassen. Wichtig ist, dass alle reellwertigen Koordinaten mittels der *Round*-Funktion in ganzzahlige umgewandelt werden, weil es nur ganzzahlige Pixelkoordinaten auf der Bildleinwand gibt.



Zusätzlich wird noch der Zufallsgenerator mit *Randomize* initialisiert, sodass dann mit Hilfe der Funktion *Random(N)* ganze Zufallszahlen von *0* bis *(N-1)* erzeugt werden können. In unserem Fall dienen sie zur zufälligen Auswahl der drei Zeichenfarben Rot, Grün und Blau.

Wenn *Xmax* (Image.Width) und *Ymax* (Image.Height) die Abmessungen der Image-Komponente sind, dann erhält man die Bildmitte *M* mittels *M.x := Xmax div 2* und *M.y := Ymax div 2*.

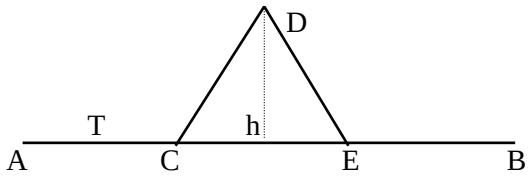
```
procedure GrafikDemo1;
{ Einfache Grafikroutine zur iterativen Erzeugung }
{ einer sternförmigen Figur - Hüllkurve Asteroide }
const Anzahl : Integer = 20;
var   a,d,i,e: Integer;
      dx,dy  : Integer;
      Farbe  : array[0..2] of Tcolor;
      s      : String;
begin
  s := InputBox('Sternkurve','Iterationsanzahl n = 1 bis 100','20');
  val(s,Anzahl,e);
  if (Anzahl < 1) or (Anzahl > 100) or (e > 0)  then Anzahl := 1;
  Farbe[0] := clRed;
  Farbe[1] := clGreen;
  Farbe[2] := clBlue;
  a := Round(Xmax/3);
  d := Round(a/Anzahl);
  Randomize;
  with Form1.Image1.Canvas do begin
    For i := 0 to Anzahl do begin
      Pen.Color := Farbe[Random(3)];
      dx := a-i*d;
      dy := Round(Sqrt(Abs(a*a-dx*dx)));
      MoveTo(M.X-dx,M.Y);
      LineTo(M.X,M.Y+dy);
      MoveTo(M.X,M.Y+dy);
      LineTo(M.X+dx,M.Y);
      MoveTo(M.X+dx,M.Y);
      LineTo(M.X,M.Y-dy);
      MoveTo(M.X,M.Y-dy);
      LineTo(M.X-dx,M.Y);
    end;
    Pen.Color  := clBlack;
  end;
end;
```

Rekursive Erzeugung von grafischen Gebilden

[A] Die Kochkurve

Die Prozedur *Grafikdemo2*, die mit Hilfe eines entsprechenden Schaltknopfes aufgerufen werden kann, erzeugt durch **rekursive Programmiertechnik** ein schneeflockenartiges Gebilde.

Im vorliegenden **Demonstrationsprogramm** soll eine gegebene Strecke AB in drei gleich lange Teilstrecken zerlegt werden. Über dem mittleren Teil ist ein gleichseitiges Dreieck zu errichten. Dieser Prozess wird dann rekursiv in jeder der so erzeugten vier gleich langen Strecken weitergeführt, bis eine vorgegebene Rekursionsstufe erreicht ist. Das Programm erzeugt also immer feiner werdende Streckenzüge, welche einem schneeflockenartigen Gebilde ähneln, das auch als KOCH-Kurve bezeichnet wird. Die Berechnung der Zwischenpunkte C, D, E aus den Endpunkten A, B einer Basisstrecke erfolgt mit den Hilfsmitteln der elementaren Vektorgeometrie:



Die Punkte A,B,C,D,E und der Vektor T sind allesamt vom vordefinierten Datentyp *TPoint*.

Der Hilfsvektor **T** sei ein Drittel des Basisvektors **AB**. Dann gilt: T.x := Round((B.x-A.x)/3) und T.y := Round((B.y-A.y)/3). Für den Normalvektor **N** gilt: N.x := T.y und N.y := -T.x. Die Höhe **h** im gleichseitigen Dreieck kann nach der Formel h = a/2*Sqrt(3) berechnet werden, wobei **a** die Dreiecksseite (also die Länge von T bzw. N) ist. Damit gelten folgende Vektorgleichungen:

```
C  :=  A + T;
D  :=  (A+B)/2 + N*Sqrt(3)/2;
E  :=  C + T;
```

Daraus können unschwer die entsprechenden Koordinatenformeln gewonnen werden. Die fünf Punkte A, C, D, E, B und noch einmal A werden in ein Punktarray *P[1..6]* abgespeichert, sodass der geschlossene Streckenzug ACDEBA mit *Polygon(P)* sofort gezeichnet werden kann.

Das gestellte Problem wird mit Hilfe der rekursiven Prozedur *Zeichnen(A,B,Stufe)* gelöst, welche in der Prozedur *GrafikDemo2* eingebettet ist. Dabei werden der Anfangspunkt *A*, der Endpunkt *B* einer Strecke und die Rekursionstiefe *Stufe* als Werteparameter übergeben. Zunächst erfolgt die Berechnung von allen benötigten Zwischenpunkten, welche in einem lokalen Array *P* abgelegt werden. Dann wird der entsprechende Streckenzug durch diese Punkte mittels *Polygon(P)* gezeichnet und der Parameter *Stufe* um *Eins* verringert. Wenn diese Rekursionsstufe *Null* ist, wird die Prozedur mit *Exit* abgebrochen. Andernfalls erfolgt ein viermaliger Selbstaufruf der Prozedur mit den entsprechenden Punkten und der verringerten Rekursionsstufe als Parameter.

In unserem Beispiel besteht der rekursive Einstieg bis zur Abbruchbedingung in der andauernden Erzeugung der Koordinaten der Zwischenpunkte der übergebenen Strecken AB. Diese Punkte werden sodann grafisch durch einen Streckenzug verbunden. Nach dem Abbruch der Prozedur mit *Exit* erfolgt die rekursive Rückkehr, bis der letzte Rücksprung ins aufrufende Hauptprogramm (hier die Prozedur *GrafikDemo2*) zurückführt.

```
procedure GrafikDemo2;
{ Einfache Grafikroutine zur rekursiven Erzeugung }
{ einer schneeflockenförmigen Figur - Kochkurve   }
var  A, B : TPoint;
     Stufe, e : Integer;
     s : String;

procedure Zeichnen(A,B: TPoint; Stufe: Integer);
{ Rekursives Unterprogramm }
var P : Array[1..6] of TPoint;
    T : TPoint;
    i : Integer;
begin
    T.X := Round((B.X -A.X)/3);
    T.Y := Round((B.Y -A.Y)/3);
    P[1]   := A;
    P[2].X := P[1].X + T.X;
    P[2].Y := P[1].Y + T.Y;
    P[3].X := P[2].X + Round(T.X / 2)  +  Round(T.Y * Sqrt(3)/2);
    P[3].Y := P[2].Y + Round(T.Y / 2)  -  Round(T.X * Sqrt(3)/2);
    P[4].X := P[2].X + T.X;
    P[4].Y := P[2].Y + T.Y;
    P[5]   := B;
    P[6]   := A;
    Form1.Image1.Canvas.Polygon(P);
    Stufe := Stufe - 1;
    if Stufe = 0  then Exit;
    For i := 1 to 4 do Zeichnen(P[i],P[i+1],Stufe);
end;

begin
    s := InputBox('Kochkurve','Rekursionstiefe t = 1 bis 8','4');
    val(s,Stufe,e);
    if (Stufe<1) or (Stufe>8) or (e>0) then Stufe := 1;
    With Form1.Image1.Canvas do begin
      Brush.Style := bsSolid;
      Brush.Color := clRed;
      Pen.Color   := clRed;
      A.X := Round(Xmax/8);
      A.Y := Ymax - Round(Ymax/8);
      B.X := Xmax - A.X;
      B.Y := A.Y;
      Zeichnen(A,B,Stufe);
      Brush.Style := bsClear;
      Brush.Color := clWhite;
      Pen.Color   := clBlack;
    end;
end;
```

[B] Ein Linienraster

Als ein zweites einfaches Übungsbeispiel zur rekursiven Grafikprogrammierung soll das im Projekt vorgegebene Bildrechteck (0,0,Xmax,Ymax) rekursiv durch fortgesetzte Seitenhalbierung in lauter gleich große Rechtecke zerlegt werden. N Halbierungen führen zu 2^N Streckenteilen und somit zu 2^{2N} Teilrechtecken, z.B. erzeugt N = 3 ein schachbrettartiges Muster mit 64 Rechtecken. In der rekursiven Prozedur ***Zeichnen*** werden der linke obere Eckpunkt ***P***, die Länge ***L***, die Breite ***B*** und die Rekursionstiefe ***Stufe*** als Parameter übergeben. Diese müssen vor dem ersten Aufruf im Hauptprogramm mit den Werten *P.x = 0, P.y = 0, L = Xmax, B = Ymax* und z.B. *Stufe = 3* belegt sein.

```
procedure GrafikDemo3;
{ Einfache Grafikroutine zur rekursiven Erzeugung }
{ schachbrettförmigen Musters }
var P : TPoint;
    L,B,Stufe : Integer;
    s : String;
    e : Integer;
```

```
procedure Zeichnen(P: TPoint; L,B,Stufe: Integer);
{ Rekursives Unterprogramm }
var Q: TPoint;
begin
  Form1.Image1.Canvas.Rectangle(P.X,P.Y,P.X+L,P.Y+B);
  if Stufe = 0  then Exit;
  Stufe := Stufe - 1;
  L := L div 2;
  B := B div 2;
  Q.X := P.X;   Q.Y := P.Y;   Zeichnen(Q,L,B,Stufe);
  Q.X := P.X;   Q.Y := P.Y+B; Zeichnen(Q,L,B,Stufe);
  Q.X := P.X+L; Q.Y := P.Y+B; Zeichnen(Q,L,B,Stufe);
  Q.X := P.X+L; Q.Y := P.Y;   Zeichnen(Q,L,B,Stufe);
end;

begin
    Form1.Image1.Canvas.Brush.Style := bsClear;
    s := InputBox('Raster mit 2^t Teilungen','Rekursionstiefe t = 1 bis 8','3');
    val(s,Stufe,e);
    if (Stufe<1) or (Stufe>8) or (e>0) then Stufe := 1;
    P.X := 0;  P.Y := 0;
    L := Xmax; B := Ymax;
    Zeichnen(P,L,B,Stufe);
end;
```

[C] Das Sierpinsky-Dreieck

Aufgabe: Verbindet man in einem gegebenen Dreieck die Seitenmittelpunkte, so erhält man vier Teildreiecke. Dieser Zerlegungs-Prozess wird nun in den drei Rand-Dreiecken rekursiv so lange weitergeführt, bis eine vorgegebene Rekursionstiefe erreicht ist. Das Programm erzeugt also immer feiner strukturierte Dreiecks-Zerlegungen.

Die gestellte Aufgabe wird mit Hilfe der rekursiven Prozedur *Zeichnen(A,B,C,Stufe)* gelöst. Dabei werden die Eckpunkte *A*, *B*, *C* des Dreiecks und *Stufe* als Werteparameter übergeben. Zunächst wird der Rekursionsparameter abgefragt. Wenn die Rekursionsstufe den Wert Null erreicht hat, dann wird abgebrochen. Andernfalls folgt die Berechnung von den drei Seitenmittelpunkten. Dann wird das Mittendreieck gezeichnet und die Rekursionstufe um Eins verringert. Daraufhin erfolgt in jedem der drei entstandenen Randdreiecke ein neuerlicher Aufruf von der Prozedur *Zeichnen*.

```
procedure GrafikDemo4;
{ Einfache Grafikroutine zur rekursiven Erzeugung }
{ von Sierpinsky-Dreiecken                         }
var  A,B,C : TPoint;
     Stufe, e : Integer;
     s : String;

procedure Zeichnen(A,B,C: TPoint; Stufe: Integer);
{ Rekursives Unterprogramm }
var M : Array[1..4] of TPoint;
    T : Tpoint;
    i : Integer;
begin
    M[1].X := (A.X + B.X) div 2;
    M[1].Y := (A.Y + B.Y) div 2;
    M[2].X := (A.X + C.X) div 2;
    M[2].Y := (A.Y + C.Y) div 2;
    M[3].X := (B.X + C.X) div 2;
    M[3].Y := (B.Y + C.Y) div 2;
    M[4]   := M[1];
    Form1.Image1.Canvas.Polygon(M);
    Stufe := Stufe - 1;
    if Stufe = 0  then Exit;
    Zeichnen(A,M[1],M[2],Stufe);
    Zeichnen(B,M[1],M[3],Stufe);
    Zeichnen(C,M[2],M[3],Stufe);
end;
```

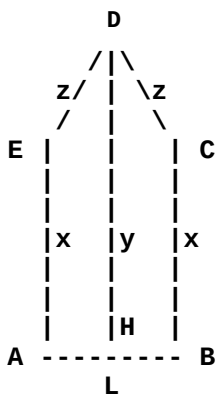
```
begin
    s := InputBox('Sierpinsky-Dreieck','Rekursionstiefe t = 1 bis 8','4');
    val(s,Stufe,e);
    if (Stufe<1) or (Stufe>8) or (e>0) then Stufe := 1;
    With Form1.Image1.Canvas do begin
      Brush.Style := bsSolid;
      Brush.Color := clBlue;
      Pen.Color   := clBlue;
      A.X := Round(Xmax/8);
      A.Y := Ymax - Round(Ymax/8);
      B.X := Xmax - A.X;
      B.Y := A.Y;
      C.X := Round(Xmax/2);
      C.Y := Round(Ymax/8);
      MoveTo(A.X,A.Y); LineTo(B.X,B.Y);
      MoveTo(B.X,B.Y); LineTo(C.X,C.Y);
      MoveTo(C.X,C.Y); LineTo(A.X,A.Y);

      Zeichnen(A,B,C,Stufe);

      Brush.Style := bsClear;
      Brush.Color := clWhite;
      Pen.Color   := clBlack;
    end;
end;
```

[D] Ein rekursiver Baum

Aufgabe: Das Grundmuster des Baumes (siehe unten stehende Skizze) ist ein Fünfeck mit der Grundsstrecke L und den zwei senkrechten Randstrecken x und der Mittelstrecke y. Dabei gelten die Verhältnisse v1 = x : L und v2 = y : L. Nun werden auf den beiden oberen, schrägen Seiten z des Fünfecks zum Grundmuster ähnliche Fünfecke errichtet. Dieser Aufbau-Prozess wird nun rekursiv so lange weitergeführt, bis eine vorgegebene Rekursionstiefe erreicht ist. Das Programm erzeugt so immer feinere Baumäste.

Die gestellte Aufgabe wird mit Hilfe der rekursiven Prozedur *Zeichnen(A,B,Stufe)* gelöst. Dabei werden die Eckpunkte *A, B* des Fünfecks und *Stufe* als Werteparameter übergeben. Zunächst werden die fehlenden Eckpunkte *C, D, E* berechnet und das Fünfeck gezeichnet. Dabei wird die Tatsache verwendet, dass die drei Vektoren *AE, HD* und *BC* auf den Basisvektor *AB* normal stehen. Dann wird der Rekursionsparameter abgefragt. Wenn die Rekursionsstufe den Wert Null erreicht hat, dann wird abgebrochen. Andernfalls erfolgt durch neuerliche Aufrufe der Prozedur *Zeichnen* der Aufbau von zwei neuen Fünfecken über den beiden oberen, schrägen Seiten des erzeugten Fünfecks.

Einfache Skizze des Fünfecks (ABCDE):




```
Grundstrecke  L = AB
Randstrecke   x = Round(v1 * L)
Mittelstrecke y = Round(v2 * L)
```

```
procedure Grafikdemo5;
// Ein rekursiver Grafikbaum

var XMax,YMax : Integer;         // Bildabmessungen
    Zeit  : Integer;             // Wartezeit
    Stop  : Boolean;             // Steuervariable

    Code  : Integer;             // Hilfsvariable
    S     : STring;              // Hilfsvariable

    L     : Integer;             // Grundstrecke des Fünfecks
    V1,V2 : Real;                // die zwei Verhältnisse
    STUFE : Integer;             // Rekursionsstufe
    A,B   : TPoint;              // Basiseckpunkte des Fünfecks


procedure FillImage(F: TColor);
// Füllt ein Image mit einer Farbe
begin
  with Form1.Image1 do begin
     Canvas.Brush.Color := F;
     Canvas.Brush.Style := bsSolid;
     Canvas.Pen.Color   := clWhite;
     Canvas.Rectangle(0,0,Width,Height);
     Canvas.Pen.Color   := Canvas.Brush.Color;
     Canvas.Brush.Style := bsClear;
  end;
end;


procedure Wait(Zeit: Integer);
// Warten mit einer Wartezeit in Millisekunden
var Zeit1 : Integer;
begin
  Zeit1 := GetTickCount;
  repeat
    Application.ProcessMessages;
  until (GetTickCount - Zeit1 > Zeit);
end;


procedure Zeichnen(A,B: TPoint; STUFE: Integer);
// Rekursives Unterprogramm
var  P : array [1..6] OF TPoint;
     XD,YD : Real;
begin
   if (GetAsyncKeyState(VK_ESCAPE) <> 0) then Stop := True;
   Wait(Zeit);
   if Stop then Exit;
   XD := B.X - A.X; YD := B.Y - A.Y;
   P[1] := A; P[5] := B; P[6] := A;
   P[2].X := P[1].X + Round(V1*YD);
   P[2].Y := P[1].Y - Round(V1*XD);
   P[3].X := P[1].X + Round(XD/2) + Round(V2*YD);
   P[3].Y := P[1].Y + Round(YD/2) - Round(V2*XD);
   P[4].X := P[5].X + Round(V1*YD);
   P[4].Y := P[5].Y - Round(V1*XD);
   Form1.Image1.Canvas.Polygon(P);
   STUFE := STUFE - 1;
   if (STUFE = 0) then Exit;
   Zeichnen(P[2],P[3],STUFE);
   Zeichnen(P[3],P[4],STUFE);
end;
```

```
begin
  FillImage(clBlack);
  S := InputBox('Input','Eingabe der Baumbreite (ca. 30)','30');
  val(S,L,Code);
  if (Code <> 0) or (L < 10) or (L > 100) then L := 30;
  S := InputBox('Input','Seitliche Höhenstreckung (ca. 4.0)','4.0');
  val(S,V1,Code);
  if (Code <> 0) or (V1 < 1) or (V1 > 10) then V1 := 4.0;
  S := InputBox('Input','Mittlere Höhenstreckung (ca. 4.5)','4.5');
  val(S,V2,Code);
  if (Code <> 0) or (V2 < 1) or (V2 > 10) then V2 := 4.5;
  S := InputBox('Input','Eingabe der Rekursionstiefe (1 bis 12)','8');
  val(S,Stufe,Code);
  if (Code <> 0) or (Stufe < 1) or (Stufe > 12) then Stufe := 1;
  With Form1.Image1 do begin
    XMax := Width;
    YMax := Height;
    Zeit := 10;
    A.x := Round(XMax/2);
    A.y := YMax - Round(YMax/6);
    B.x := A.x + L;
    B.y := A.y;
    Canvas.Brush.Style := bsSolid;
    Canvas.Brush.Color := clGreen;
    Canvas.Pen.Color   := clWhite;
    Stop := False;

    Zeichnen(A,B,Stufe);

    Stop := False;
  end;
end;
```

Erwähnenswert ist die Tatsache, dass mit der *Escape*-Taste jederzeit die rekursive Prozedur abgebrochen werden kann. Dazu dient die Steuervariable *Stop*. Außerdem wird vor jedem Aufruf der rekursiven Prozedur eine bestimmte Zeit gewartet, sodass der ganze Ablauf besser beobachtet werden kann. Dieses leistet das Unterprogramm *Wait*. Die Prozedur *FillImage* schließlich löscht das Image.

[05] Das Programm „FRAKTAL“

Der vorliegende FRAKTAL-GENERATOR mit Lindenmayer-Syntax stellt ein kreatives Werkzeug zur Erzeugung von fraktalen Objekten dar. FRAKTALE sind selbstähnlichen Gebilde. Das sind Gebilde, die ihrerseits aus selbstähnlichen Teilen bestehen und rekursiv (bzw. iterativ) entwickelt werden.

Die Entwicklung wird durch eine "Formel" gesteuert, die fünf Symbole enthalten darf: F, +, -, (,).

F ... Bewege den Grafikcursor um eine bestimmte Strecke
in eine bestimmte Richtung (Forward move).
\+ ... Führe eine Rechtsdrehung um einen Winkel aus.
\- ... Führe eine Linksdrehung um einen Winkel aus.
(... Speichere die aktuelle Cursorposition und Richtung.
) ... Springe zurück zur zuletzt gespeicherten Position.

Als Parameter werden am Anfang eingegeben: die Startposition, die Startstrecke, die Startrichtung (entweder vertikal oder horizontal), der Drehwinkel, die Zahl der Iterationen und ein Verkleinerungsfaktor. Mit diesem wird bei jeder Positionsspeicherung und auch bei jedem Rücksprung die Weglänge multipliziert. Dadurch werden unterschiedliche Weglängen erzeugt. Der Drehwinkel bezieht sich immer auf die horizontale Achse; durch diesen Winkel ist die Bewegungsrichtung bestimmt.

Zusätzlich kann noch der Radius der Knoten eingegeben werden. Wenn er größer als Null ist, dann wird am Ende jeder Strecke ein roter Kreis mit diesem Radius gezeichnet. Außerdem kann noch die Wartezeit an den Knoten eingegeben werden.

Bei einer zusätzlichen Iteration (> 1) wird in der ganzen Formel jeder Teil "F" wieder durch die ganze Formel ersetzt. So entstehen Teilgebilde, die zum ganzen Gebilde ähnlich sind. Die Endposition und Endrichtung einer Iteration werden zur Startposition und Startrichtung der nachfolgenden Iteration. Der Grafikcursor wird als Schildkröte (turtle) bezeichnet, die sich in der Ebene gemäß den Steuerbefehlen bewegt.

Einfaches Dreieck:	**Rekursive Koch-Kurve:**
Steuerformel = "F-F-F"	Steuerformel = "F-F++F-F"
Faktor = 1.00	Faktor = 1.00
Drehwinkel = 120°	Drehwinkel = 60°
Startlänge = 100 (horiz.)	Startlänge = 4 (horiz.)
Iterationen = 1	Iterationen = 3

Einfache Baumgabelung:	**Rekursive Baumgabelung:**
Steuerformel = "F(-F)+F-"	Steuerformel = "F(-F)+F-"
Faktor = 0.70	Faktor = 0.70
Drehwinkel = 20°	Drehwinkel = 20°
Startlänge = 50 (vert.)	Startlänge = 10 (vert.)
Iterationen = 1	Iterationen = 3

Bei der Entwicklung von FRAKTALEN sollte immer das Bildungsprinzip sprachlich formuliert werden. Für die "Koch-Kurve" lautet es: "Zerlege eine Strecke in drei gleichlange Teile und errichte über dem Mittleren ein gleichseitiges Dreieck. Führe diesen Prozess in jeder entstehenden Teilstrecke aus".

Nachdem alle Parameter eingetragen sind, wird die gewünschte Startposition mit einem Mausklick in der Grafik gesetzt. Die grafische Darstellung des Fraktals wird dann mit Schalter <Erzeugen> gestartet und kann mit <Esc> abgebrochen werden.

HINWEIS: Jede Grafik kann zusätzlich mit <F9> in eine Datei im JPEG-Format abgespeichert werden. Dieser sichtbare Hilfetext wird mittels <Drucken> ausgedruckt.

Bei der Formeleingabe können die Funktionstasten <F1> bis <F7> gedrückt werden. Dadurch werden bestimmte Formeln vorgegeben: Vier Bäume (F1,F2,F3,F4), Koch-Kurve (F5), Sierpinsky-Dreieck (F6) und Peano-Kurve (F7).

[06] Das Programm „MANDEL“

Das Programm MANDEL.EXE erzeugt Mandelbrot-Mengen und Julia-Mengen. Im Folgenden soll die mathematische Theorie dieser Mengen skizziert werden.

MANDELBROT-Mengen

Gegeben ist ein rechteckiger Ausschnitt der Gaußschen-Zahlenebene (xU/yU) - (xO/yO). Darin wird ein Punkt C (komplexe Zahl) gesetzt.

AUFPUNKT C(p/q), p = xC, Realteil
q = yC, Imaginärteil

Mit diesem gewählten Aufpunkt C wird eine Folge von Punkten (komplexen Zahlen Z(n)) entsprechend folgendem Bildungsgesetz rekursiv erzeugt:

$Z(n+1) = Z^2(n) + C$, mit dem Startwert Z(0) = (0/0) und dem Iterations-Zähler n = 0,1,2,.....

In Koordinaten (Real- und Imaginärteil):
$x(n+1) = x(n)^2 - y(n)^2 + p$
$y(n+1) = 2*x(n)*y(n) + q$

Die komplexe Zahlenfolge lautet dann:

Z(0) = (0/0)
Z(1) = C
Z(2) = C^2 + C
Z(3) = (C^2+C)^2 + C = C^4 + 2*C^3 + C^2 + C
.....
.....
Z(n) = ...

Diese rekursive Folge von komplexen Punkten liefert für Real- und Imaginärteil ein System von zwei reellen nicht-linearen Differenzengleichungen.

Die Zahlenfolge Z(n) erzeugt in der Ebene eine Bahn von Punkten. Je nach Wahl des Aufpunktes C und des Startwertes Z(0) laufen die Bahnen der Punkte in der Ebene völlig verschieden. In einigen Fällen führen sie ins Unendliche, in anderen wiederum bleiben sie innerhalb bestimmter Grenzen. Die zugehörigen Aufpunkte sollen dementsprechend als Ausbrecher oder Gefangene bezeichnet werden.

Für den Aufpunkt C = (0/0) gibt es grundsätzlich drei Möglichkeiten, je nach Startwert Z(0):

1) Ist Abs(Z(0)) < 1, dann konvergiert Z(n) gegen Null
2) Ist Abs(Z(0)) = 1, dann liegen alle Z(n) am Einheitskreis
3) Ist Abs(Z(0)) > 1, dann strebt Z(n) gegen Unendlich

Im Folgenden soll der Startwert Z(0) immer gleich (0/0) sein. Wird nun der Aufpunkt C(p/q) innerhalb des vorgegebenen Ebenen-Ausschnittes variiert, dann wird das Verhalten der zugehörigen Zahlenfolge Z(n) komplizierter.

Alle Aufpunkte C(p/q) im vorgegebenen Ebenen-Ausschnitt, für welche die komplexe Zahlenfolge Z(n) nicht gegen Unendlich strebt (Gefangene bzw. keine Ausbrecher) bilden die sog. MANDELBROT-Menge (so benannt nach dem Mathematiker Benoit Mandelbrot). Der Rand dieser Menge bildet ein Fraktal - er besteht aus selbstähnlichen Teilen.

Ein mathematischer Lehrsatz besagt: Wenn der Absolutbetrag von Z(n) jemals die Zahl ZWEI erreicht, dann entweicht die Bahn der Punkte auf Nimmerwiedersehen ins Unendliche. Damit ist ein brauchbares Kriterium zur Unterscheidung von Gefangenen und Ausbrechern gegeben.

Grafische Darstellung:

Das Programm <MANDEL> liefert Darstellungen der Mandelbrot-Menge. Anstelle der Zahl ZWEI wird zur besseren grafischen Darstellung eine größere Schranke DMAX für den Betrag von Z(n), also für die Entfernung der Bahnpunkte vom Nullpunkt festgesetzt (z.B. 50). Dann wird in einer Wiederholungsschleife jeder Punkt des vorgegebenen Ebenen-Ausschnittes als AUFPUNKT C(p/q) genommen. Zuletzt wird die zugehörige Zahlenfolge Z(n) solange iterativ berechnet, bis die Schranke DMAX überschritten wird. Tritt dieser Fall nicht ein, so wird zur Vermeidung einer Endlos-Schleife nach SMAX (50) Iterationsschritten abgebrochen. Ist dann nach dem Abbruch der Schrittzähler n < SMAX, so handelt es sich um einen Ausbrecher und der Aufpunkt wird WEISS gefärbt. Ist hingegen n = SMAX, so liegt ein Gefangener vor, welcher in SCHWARZ gezeichnet wird. Alle diese schwarzen Aufpunkte stellen somit die Mandelbrot-Menge dar.

Eine andere grafische Möglichkeit besteht darin, den Wert des Schrittzähler n beim Abbruch der Iteration direkt als Farbe des Aufpunktes C(p/q) zu verwenden. Anstelle von WEISS spiegelt so die Farbe die Fluchtgeschwindigkeit eines Ausbrechers wider. Dadurch ergeben sich sehr schöne grafische Muster.

Ein günstiger Ebenen-Ausschnitt zur Erzeugung der Mandelbrot-Menge ist durch den Bereich (-2.25/+1.00) - (-1.50/+1.50) gegeben. Verfeinert man die X- und Y-Schritte des AUFPUNKTES C(p/q) bei seinem Durchlauf durch den Bereich und verschiebt man überdies noch den linken, unteren Eckpunkt, so können aus der Mandelbrot-Menge beliebige Teilgebiete vergrößert dargestellt (gezoomt) werden.

Ein anderer Anfangswert Z(0) als (0/0) führt zu erheblichen Verzerrungen der Menge, bringt aber grundsätzlich nicht Neues.

Anmerkung: Natürlich muss durch eine geeignete Transformation der vorgegebene Ausschnitt der Gaußschen Zahlenebene auf ein Fenster des Grafik-Bildschirmes abgebildet werden.

Die Erzeugung von JULIA-Mengen

Wird in obigen Überlegungen der AUFPUNKT C(p/q) festgehalten und der Startwert Z(0) der Zahlenfolge variiert, dann erhält man eine JULIA-Menge (benannt nach dem Mathematiker Gaston Julia). Diese besteht nun aus allen STARTWERTEN Z(0) für welche die zugehörige Zahlenfolge Z(n) nicht gegen Unendlich strebt. Im urprünglichen Sinne wird nur der Rand dieser Menge als JULIA-Menge bezeichnet. Die Form der JULIA-Menge hängt vom fest gewählten Aufpunkt C(p/q) in der Mandelbrot-Menge ab. Alle diese JULIA-Mengen haben eine fraktale Randstruktur.

Ein interessanter, mathematischer Lehrsatz besagt: Nur wenn der AUFPUNKT C(p/q) aus dem Inneren der Mandelbrot-Menge stammt, ist die zugehörige Julia-Menge zusammenhängend. Daraus folgt, dass je weiter der Aufpunkt C sich von der inneren Mitte der Mandelbrot-Menge nach außen verlagert, umso mehr verästelt sich die Julia-menge bis sie schließlich zu Sternenstaub zerfällt.

Sehr schöne JULIA-Mengen liefern folgende Parameter:

(a) Aufpunkt: +0.40/+0.35 (im Innern der Hauptknospe)
Zoomfaktor: +10; linkes, unteres Zoom-Eck: 0.45/-0.90

(b) Aufpunkt: -0.51/+0.57 (im Innern einer Randknospe)
Zoomfaktor: +1;

Schließlich sei noch der Bezug dieser Mengen zum CHAOS aufgezeigt: Die rekursiv definierte, komplexe Folge $Z(n+1) = Z(n)^2 + C$ kann in zwei nicht lineare, reelle Differenzengleichungen für den Realteil (X) und den Imaginärteil (Y) aufgespaltet werden. Für bestimmte Parameterwerte können sich solche Funktionen chaotisch verhalten. Dieses Verhalten hängt bei den Julia-Mengen weitgehend vom jeweiligen Startwert der Z(n) ab.