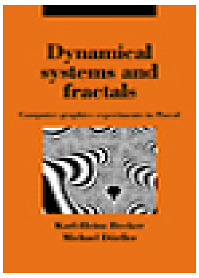

11 Building Blocks for Graphical Experiments

11.1 The Fundamental Algorithms

In the first eight chapters we have explained to you some interesting problems and formulated a large number of exercises, to stimulate your taste for computer graphics experiments. In the chapters following Chapter 11 we provide solutions to the exercises, sometimes as complete programs or else as fragments. The solution, the complete Pascal program, can be obtained by combining the ready-made component fragments, after which you can perform a great number of graphical experiments.

Structured programs are programs in which the founder of the programming language Pascal - the Swiss computer scientist Niklaus Wirth - has always taken a great interest. Such programs are composed only of procedures and functions. The procedures are at most a page long. Naturally such programs are well documented. Any user who reads these programs can understand what they do. The variable names, for example, convey their meaning and are commented at length when necessary. In particular the programs are written in a structured fashion, in which the 'indentation rules' and 'style rules' for Pascal are strictly adhered to. We hope that you too will write structured programs, and we would like to offer some advice.

Perhaps you have read the entire book systematically up to Chapter 11, to get a good survey of the formulation of the problem; perhaps you have also written the first few programs. By analysing our program description you have surely noticed that the structure of our programs is always the same. The reason is that we have always tried to write structured programs.

These are, in particular:

- portable, so that they can easily run on other computers ('machine-independent');
- clearly structured ('small procedures');
- well commented ('use of comments');
- devoid of arbitrary names ('description of variables and procedures').

Exceptions are limited and removed in a tolerable time.

New programs can quickly be assembled from the building blocks. You can get a clear idea of the structure of our programs from Example 11.1-1:

Reference Program 11.1-1 (cf. Program 2.1-1)

```
PROGRAM EmptyApplicationShell; (* for computer XYZ       *)
   (* library declarations where applicable              *)

  CONST
     Xscreen = 320;    (* e.g. 320 points in x-direction  *)
     Yscreen = 200;    (* e.g. 200 points in y-direction  *)

  VAR
     PictureName : string;
```

```
        Left, Right, Top, Bottom : real;
        (* include additional global variables here *)
        Feedback : real;
        Visible, Invisible : integer;

(* -----------------------UTILITY----------------------------*)
(* BEGIN:  Useful Subroutines *)
    PROCEDURE ReadReal (information : STRING; VAR value
                       : real);
    BEGIN
       write (information);
       readln (value);
    END;

    PROCEDURE ReadInteger (information : STRING; VAR value
                       : integer);
    BEGIN
       write (information);
       readln (value);
    END;

    PROCEDURE ReadString (information : STRING; VAR value
                       : string);
    BEGIN
       write (information);
       readln (value);
    END;

    PROCEDURE InfoOutput (information : STRING);
    BEGIN
       writeln (information);
       writeln;
    END;

    PROCEDURE CarryOn (information : STRING);
    BEGIN
       write (information, ' hit <RETURN>');
       readln
    END;

    PROCEDURE CarryOnIfKey;
    BEGIN
       REPEAT
       UNTIL KeyPressed (* DANGER - machine-dependent!       *)
    END;

    PROCEDURE Newlines (n : integer);
       VAR
          i : integer;
    BEGIN
       FOR i := 1 TO n DO
          writeln;
    END;

(* END: Useful Subroutines *)
(* -------------------------UTILITY--------------------------*)

(* ------------------------GRAPHICS--------------------------*)
(* BEGIN:  Graphics Procedures *)

    PROCEDURE SetPoint (xs, ys : integer);
    BEGIN
       (*  Insert machine-specific graphics commands here    *)
    END;

    PROCEDURE SetUniversalPoint (xw, yw: real);
       VAR xs, ys : real;
    BEGIN
       xs := (xw - Left) * Xscreen / (Right - Left);
       ys := (yw - Bottom) * Yscreen / (Top - Bottom);
       SetPoint (round(xs), round(ys));
    END;

    PROCEDURE GoToPoint (xs, ys : integer);
    BEGIN
       (* move without drawing *)
       (*  Insert machine-specific graphics commands here    *)
    END;

    PROCEDURE DrawLine (xs, ys : integer);
    BEGIN
       (*  Insert machine-specific graphics commands here    *)
    END;
    PROCEDURE DrawUniversalLine (xw, yw : real);
      VAR xs, ys : real;
    BEGIN
      xs := (xw - Left) * Xscreen/(Right - Left);
      ys := (yw - Bottom) * Yscreen / (Top - Bottom);
      DrawLine (round(xs), round(ys));
    END;

    PROCEDURE TextMode;
    BEGIN
      (* switch on text-representation *)
      (*  Insert machine-specific graphics commands here     *)
    END;

    PROCEDURE GraphicsMode;
    BEGIN
      (* switch on graphics-representation *)
      (*  Insert machine-specific graphics commands here     *)
    END;

    PROCEDURE EnterGraphics;
    BEGIN
      writeln ('To end drawing hit <RETURN> ');
      write ('now hit <RETURN> '); readln;
      GraphicsMode;
    END;

    PROCEDURE ExitGraphics
    BEGIN
     (* machine-specific actions to exit from Graphics Mode   *)
      TextMode;
    END;

(END: Graphics Procedures *)
(--------------------------GRAPHICS--------------------------*)

(------------------------APPLICATION-------------------------*)
(BEGIN: Problem-specific procedures *)
(* useful functions for the given application problem        *)

    FUNCTION f(p, k : real) : real;
```

```
   BEGIN f := p + k * p * (1-p);
   END;

   PROCEDURE FeigenbaumIteration;
      VAR
         range, i: integer;
         population, deltaxPerPixel : real;
      BEGIN
         deltaxPerPixel := (Right - Left) / Xscreen;
         FOR range := 0 TO Xscreen DO
            BEGIN
               Feedback := Left + range * deltaxPerPixel;
               population := 0.3;
               FOR i := 0 TO invisible DO
                  population := f(population, Feedback);
               FOR i := 0 TO visible DO
                  BEGIN
                     SetUniversalPoint (Feedback, population);
                     population := f(population, feedback);
                  END;
            END;
      END;

(* END: Problem-specific procedures                           *)
(*------------------------APPLICATION------------------------*)
(*---------------------------MAIN----------------------------*)
(* BEGIN: Procedures of Main Program                          *)

    PROCEDURE Hello;
    BEGIN
       TextMode;
       InfoOutput ('Calculation of                        ');
       InfoOutput ('-------------------------');
       Newlines (2);
       CarryOn ('Start :');
       Newlines (2);
    END;

    PROCEDURE Goodbye;
    BEGIN
       CarryOn ('To stop : ');
    END;
```

```
PROCEDURE Initialise;
    BEGIN
      ReadReal ('Left                           >', Left);
      ReadReal ('Right                          >', Right);
      ReadReal ('Top                            >', Top);
      ReadReal ('Bottom                         >', Bottom);
      ReadInteger ('Invisible                   >', invisible);
      ReadInteger ('Visible                     >', visible);
         (* possibly further inputs *)
      ReadString ('Name of Picture              >',PictureName);
    END;

    PROCEDURE ComputeAndDisplay;
    BEGIN
      EnterGraphics;
      FeigenbaumIteration;
      ExitGraphics;
    END;

(* END: Procedures of Main Program                            *)
(* ----------------------------MAIN------------------------- *)
    BEGIN (* Main Program *)
      Hello;
      Initialise;
      ComputeAndDisplay;
      Goodbye;
    END.
```

1 ***Structure of Pascal Programs***

All program examples are constructed according to the following scheme:

```
    PROGRAM ProgramName;
    |  Here follows the declaration part including
    |  •  Library declarations (if applicable)
    |  •  Constant declarations
    |  •  Type declarations
    |  •  Declaration of global variables
(*---------------------------------------------------------- *)
(* BEGIN: Useful Subroutines                                  *)
    |  •  Here follows the definition of the useful subroutines
(* END: Useful Subroutines                                    *)
```

```
(*---------------------------------------------------------- *)
(* BEGIN: Graphics Procedures                                 *)
   | • Here follows the definition of the graphics pProcedures
(* END: Graphics Procedures                                   *)
(*---------------------------------------------------------- *)
(* BEGIN: Problem-specific Procedures                         *)
   | • Here follows the definition of the problem-specific procedures
(* END: Problem-specific Procedures                           *)
(*---------------------------------------------------------- *)
(* BEGIN: Procedures of Main Program                          *)
   | • Here follows the definition of the
   | • Procedures of the main program:
   | • Hello,Goodbye,Initialise,ComputeAndDisplay
(* END: Procedures of Main Program                            *)
(*---------------------------------------------------------- *)
   BEGIN (* Main Program *)
      Hello;
      Initialise;
      ComputeAndDisplay;
      Goodbye;
   END.
```

2 *Layout of Pascal Programs*

All Pascal programs have a unified appearance.

- Global symbols begin with a capital letter. These are the name of the main program, global variables, and global procedures.
- Local symbols begin with a lower-case letter. These are names of local procedures and local variables.
- Keywords in Pascal are written in capitals or printed boldface.

3 *Machine-Independence of Pascal Programs*

By observing a few simple rules, all Pascal programs can be used on different machines. Of course today's computers unfortunately still differ widely from one another. For this reason we have set up the basic structure of our reference program (see overleaf), so that the machine-dependent parts can quickly be transported to other machines. Model programs and reference programs for different makes of machine and programming languages can be found in Chapter 12.

We now describe in more detail the overall structure of Pascal programs.

Global Variables

The definitions of global constants, types, and variables are identified with bold capital headings for each new global quantity. Although not all of them are used in the same program, a typical declaration for a Julia or a Gingerbread Man program might look like this:

Program Fragment 11.1-2

```
    CONST
    Xscreen = 320;             (*screen width in pixels      *)
    Yscreen = 200;             (*screen height in pixels     *)
    WindowBorder = 20;         (*0 for a Macintosh           *)
    Limit = 100;               (*test for end of iteration   *)
    Pi = 3.141592653589;       (*implemented in many dialects*)
    Parts = 64;                (* for fractal landscapes     *)
    PartsPlus2 = 66:           (*=Parts + 2, for landscapes  *)

    TYPE
    IntFile = FILE OF integer; (*machine-independent         *)
    CharFile = Text;           (*storage of picture data     *)

    VAR
    Visible, Invisible,        (*drawing limits for          *)
                               (*Feigenbaum diagrams         *)
    MaximalIteration, Bound,   (*drawing limits for          *)
                               (*Julia and Mandelbrot sets   *)
    Turtleangle, Turtlex, Turtley,
    Startx, Starty, Direction,
    Depth, Side,               (*turtle graphics             *)
    Xcentre, Ycentre, Radius,  (*screen parameters of        *)
                               (*Riemann sphere              *)
    Quantity, Colour, Loaded,  (*Data values for screen-     *)
                               (*independent picture storage *)
    Initial, Factor,           (*fractal mountains           *)
    D3factor, D3xstep, D3ystep,     (*3D-specialities        *)
                   :integer;
Ch                                  :char;
PictureName, FileName  :STRING;
Left, Right, Top, Bottom,      (*screen window limits        *)
Feedback, Population,          (*parameters for Feigenbaum   *)
N1, N2, N3, StartValue         (*parameters for Newton       *)
Creal, Cimaginary,             (*components of c             *)
```

```
FixValue1, FixValue2             (*tomogram parameters         *)
Width, Length,                   (*Riemann sphere coordinates  *)
Power,                           (*for generalised Gingerbread *)
HalfPi,                          (* used for arc sine, = Pi/2  *)
                        :real;
F, In, Out              : IntFile;
InText, OutText         : CharFile;
                                 (* for screen-independent     *)
                                 (* data storage               *)
Value                   : ARRAY [0..Parts, 0..PartsPlus2] OF
                                 integer;
                                 (* fractal landscapes         *)
CharTable               :ARRAY [0..63] OF Char;
                                 (* look-up tables             *)
IntTable                : ARRAY ['0'..'z'] OF integer;
D3max                   : D3maxType;
                                 (* maximal values for 3D      *)
```

Graphics Procedures

With graphics procedures the problem of machine-dependence becomes prominent. The main procedures for transforming from universal to picture coordinates are self-explanatory.

Problem-specific Procedures

Despite the name, there are no problems with problem-specific procedures. On any given occasion the appropriate procedures and functions are inserted in the declaration when called from the main program.

Useful Subroutines

In example program 11.1-1 we first formulate all the useful subroutines required for the program fragments of previous chapters; but these are not described in detail. This would probably be superfluous, since anyone with a basic knowldedge of Pascal or other programming languages can see immediately what they do.

Most of these procedures rely upon reading data from the keyboard, whose input is accompanied by the output of some prompt. In this way any user who has not written the program knows which input is required. The input of data can still lead to problems if this simple procedure is not carried out in the manner that the programmer intended. If the user types in a letter instead of a digit, in many dialects Pascal stops with an unfriendly error message. We recommend that you add to the basic algorithms procedures to protect against input errors. You can find examples in many books on Pascal.

We can now finish the survey of the structure and basic algorithms of our reference program, and in the next chapter we lay out solutions to problems that were not fully explained earlier.

Since the program structure is always the same, to provide solutions we need only give the following parts of the program:

- The problem-specific part.
- The input procedure.

From the input procedure you can read off without difficulty which global variables must be declared.

11.2 Fractals Revisited

Now we begin the discussion of the solutions to the exercises, supplementing the individual chapters. Occasionally we make a more systematic study of things that are explained in the first eight chapters. Why not begin at the end, which is likely to be still fresh in your mind?

Do you remember the 'fractal computer graphics' from Chapter 8? Possible partial solutions for the exercises listed there are here given as program fragments. In the main program the appropriate procedures must be called in the places signified. This type of sketch is very quick to set up.

Program Fragment 11.2-1 (for Chapter 8)

```
   . . .
     VAR
     (* insert further global variables here *)
        Turtleangle, Turtlex, Turtley : integer;
        Startx, Starty, Direction, Depth, Side : integer;
   . . .
(*-----------------------APPLICATION-----------------------*)
(* BEGIN: Problem-specific procedures *)
(*Here follow the functions needed in the application program*)

   PROCEDURE Forward (step : integer);
     VAR
        xStep, yStep : real;
   BEGIN
     xStep := step * cos (Turtleangle * Pi) / 180.0);
     yStep := step * sin (Turtleangle * Pi) / 180.0);
     Turtlex := Turtlex + trunc (xStep);
     Turtley := Turtley + trunc (yStep);
     DrawLine (Turtlex, Turtley);
   END;
   PROCEDURE Backward (step : integer);
   BEGIN
     Forward (- step);
   END;

   PROCEDURE Turn (alpha : integer);
   BEGIN
     Turtleangle := (Turtleangle + alpha) MOD 360;
   END;

   PROCEDURE StartTurtle;
   BEGIN
     Turtleangle := 90; Turtlex := Startx; Turtely := Starty;
     SetPoint (Startx, Starty);
   END;

   PROCEDURE dragon (Depth, Side : integer);
   BEGIN
     IF Depth = 0 THEN
       Forward (Side)
     ELSE IF Depth > 0 THEN
       BEGIN
         dragon (Depth - 1, trunc (Side));
         Turn (90);
         dragon (-(Depth - 1), trunc (Side));
       END
     ELSE
       BEGIN
         dragon (-(Depth + 1), trunc (Side));
         Turn (270);
         dragon (Depth + 1, trunc (Side));
       END;
   END;

(* END: Problem-specific procedures *)
(*------------------------APPLICATION--------------------- *)

   ...
   PROCEDURE Initialise;
   BEGIN
     ReadInteger ('Startx    >', Startx);
     ReadInteger ('Starty    >', Starty);
```

```
      ReadInteger ('Direction  >', Direction);
      ReadInteger ('Depth      >', Depth);
      ReadInteger ('Side       >', Side);
   END;

   PROCEDURE ComputeAndDisplay;
   BEGIN
      EnterGraphics;
      StartTurtle;
      dragon (Depth, Side);
      ExitGraphics;
   END;
```

The declaration part of the main program always stays the same.

Please take note of this solution, because it will run in this form on any computer, provided you insert your machine-specific commands in the appropriate places of the graphics part. The relevant global variables for all exercises are given in full. In the graphics part you must always insert your commands in the procedures `SetPoint`, `DrawLine`, `TextMode`, `GraphicsMode`, and `ExitGraphics` (see §11.1 and Chapter 12). We have included the implementation of our own turtle graphics in the problem-specific procedures. If your computer has its own turtle graphics system (UCSD systems, Turbo Pascal systems) you can easily omit our version. But do not forget to include the global variables of your turtle version. That applies also to the correct initialisation of the turtle, which we simulate with the aid of the procedure `StartTurtle`. Read the hints in Chapter 12.

All procedures which stay the same will here and in future be omitted to save space.

In Chapter 8, Figure 8.2-4 we showed a fractal landscape with mountains and seas. You must have wondered how to produce this graphic. Here you will find an essentially complete solution. Only the seas are absent.

Program 11.2-2 (for Figure 8.2-4)

```
    PROGRAM fractalLandscapes;
       CONST
          Parts = 64;
          PartsPlus2 = 66;  {= Parts + 2}
       VAR
          Initial, Picsize : integer;
          Value : ARRAY [0..Parts, 0..PartsPlus2] OF integer;
       Mini, Maxi, Factor, Left, Right, Top, Bottom : real;
    ...
 (* Insert the global procedures (see 11.1-1) here           *)
```

```
(* BEGIN: Problem-specific procedures                              *)
     FUNCTION RandomChoice (a, b : integer) : integer;
(* Function with a side-effect on Maxi and Mini                    *)
       VAR zw : integer;
     BEGIN
       zw := (a + b) DIV 2 + Random MOD Picsize - Initial;
       IF zw < Mini THEN Mini := zw;
       IF zw > Maxi THEN Maxi := zw;
       RandomChoice := zw;
     END;  (* of RandomChoice *)

     PROCEDURE Fill;
       VAR i, j : integer;

       PROCEDURE  full;
         VAR xko, yko : integer;
       BEGIN
         yko := 0;
         REPEAT
           xko := Initial;
           REPEAT
             Value [xko, yko] :=
               RandomChoice (Value [xko - Initial, yko],
                             Value [xko + Initial, yko]);
             Value [yko, yko] :=
               RandomChoice (Value [yko, xko - Initial ],
                            Value [yko, xko + Initial ]);
             Value [xko, Parts - xko - yko] :=
               RandomChoice (
                 Value [xko-Initial,
                       Parts-xko-yko+Initial ],
                 Value [xko+Initial,
                       Parts-xko-yko-Initial ]);
             xko := xko + Picsize;
             UNTIL xko > (Parts - yko);
             yko := yko + Picsize;
           UNTIL yko >= Parts;
         END;  (* of full *)

      BEGIN (* of Fill *)
        FOR i := 0 TO Parts DO
          FOR j := 0 TO PartsPlus2 DO
          Value [i, j] := 0;
      Mini := 0; Maxi := 0;
      Picsize := Parts;
      Initial := Picsize DIV 2;
      REPEAT
        full;
        Picsize := Initial;
        Initial := Initial DIV 2;
      UNTIL Initial := Picsize;
      Value [0, Parts + 1] := Mini;
      Value [1, Parts + 1] := Maxi;
      Value [2, Parts + 1] := Picsize;
      Value [3, Parts + 1] := Initial;
    END (* of Fill *)

    PROCEDURE Draw;
      VAR xko, yko : integer;

      PROCEDURE slant;
        VAR xko : integer;
      BEGIN (* of slant *)
        SetUniversalPoint (yko, yko+Value[0,yko]*Factor);
        FOR xko := 0 TO Parts - yko DO
          DrawUniversalLine (xko+yko,
                    yko + Value[xko,yko]*Factor);
        FOR xko := Parts - yko TO Parts DO
          DrawUniversalLine (xko+yko,
                    yko + Value[Parts-yko,
                            Parts-xko]*Factor);
      END;  (* of slant *)

      PROCEDURE along;
        VAR xko : integer;
      BEGIN
        SetUniversalPoint (xko, Value[xko,0]*Factor);
        FOR yko := 0 TO Parts -xko DO
          DrawUniversalLine (xko+yko,
                    yko+Value[xko,yko]*Factor);
        FOR yko := Parts - xko TO Parts DO
          DrawUniversalLine (xko+yko,
                    xko+Value[Parts-yko, Parts-xko]*Factor);
      END;  (* of along *)

    BEGIN (* of Draw *)
      FOR yko := 0 TO Parts DO
        slant;
      FOR xko := 0 TO Parts DO
        along;
      END; (* of Draw *)

(* END: Problem-specific Procedures *)
    PROCEDURE Initalise;
    BEGIN
      ReadReal (' Left      >, Left);
      ReadReal (' Right     >', Right);
      ReadReal (' Bottom    >', Bottom);
      ReadReal (' Top       >', Top);
      ReadReal (' Factor    >', Factor);
      Newlines (2);
      InfoOutput ('wait 20 seconds ');
      Newlines (2);
    END;

    PROCEDURE ComputeAndDisplay;
    BEGIN
      Fill;
      EnterGraphics;
      Draw;
      ExitGraphics;
    END;

  BEGIN (* Main Program *)
    Hello;
    Initialise;
    ComputeAndDisplay;
    Goodbye;
  END.
```

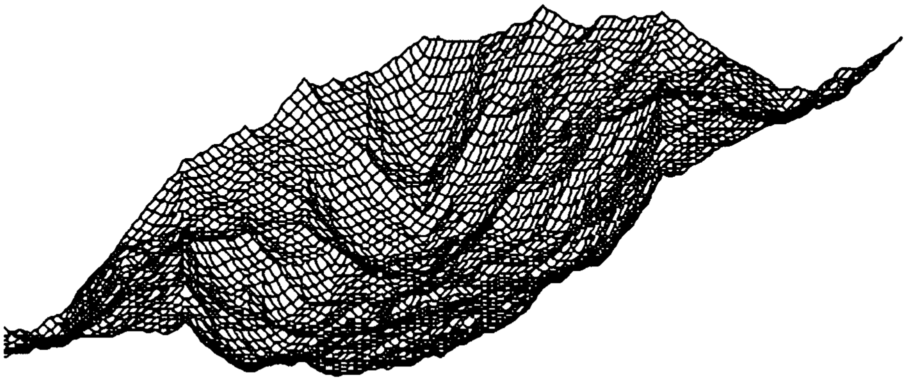

Figure 11.2-1 Fractal mountains.

The theory of graftals is certainly not easy to understand. For those who have not developed their own solution, we here list the appropriate procedure:

Program 11.2-3 (for §8.3)

```
PROCEDURE Graftal;  (* Following Estvanik [1986] *)
  TYPE
    byte = 0..255;
    byteArray = ARRAY[0..15000] OF byte;
    codeArray = ARRAY[0..7, 0..20] OF byte;
    realArray = ARRAY[0..15] OF real;
    stringArray = ARRAY[0..7] OF STRING[20];
  VAR
    code : codeArray;
    graftal : byteArray;
    angle : realArray;
    start : stringArray;
    graftalLength, counter, generationNo, angleNo
                 : integer;
    ready : boolean;

  FUNCTION bitAND (a, b : integer ) : boolean;
    VAR x, y :    RECORD CASE boolean OF
                  False : (counter : integer);
                  True : (seq : SET OF 0..15)
               END;
  BEGIN
```

```
   x.number := a; y.number := b;
   x.seq : x.seq * y.seq;
   bitAND := x.number <> 0;
END;  (* of bitAND *)

PROCEDURE codeInput
                  (VAR generationNo : integer;
                  VAR code : codeArray;
                  VAR angle : realArray;
                  VAR angleNo : integer;
                  VAR start : stringArray);
   VAR rule : STRING[20];

   PROCEDURE inputGenerationNo;
   BEGIN
      write (' Generation Number       > ');
      readln (generationNo);
      IF generationNo > 25 THEN generationNo := 25;
   END;

   PROCEDURE inputRule;
      VAR ruleNo, alphabet : integer;
   BEGIN
      FOR ruleNo := 0 TO 7 DO
      BEGIN
         write(' Input of ', ruleNo+1, '. Rule   > ');
         readln (rule);
         IF rule = ' ' THEN
            rule := '0';
         code [ruleNo, 0] := length (rule);
         start [ruleNo] := rule;
         FOR alphabet := 1 TO code [ruleNo, 0] DO
         BEGIN
            CASE rule[alphabet] OF
            '0' : code [ruleNo, alphabet] := 0;
            '1' : code [ruleNo, alphabet] := 1;
            '[' : code [ruleNo, alphabet] := 128;
            ']' : code [ruleNo, alphabet] := 64;
            END;
         END;
      END;
   END;
```

```
  PROCEDURE inputAngleNo;
    VAR k, i : integer;
  BEGIN
    write (' Number for angle        > ');
    readln (angleNo);
    IF angleNo > 15 THEN
      angleNo := 15;
    FOR k := 1 TO angleNo DO
    BEGIN
      write ('Input of ', k:2, '.Angle (Degrees) > ');
      readln (i);
      angle [k-1] := i*3.14159265 / 180.0;
    END;
  END;

  PROCEDURE controlOutput;
    VAR alphabet : integer;
  BEGIN
    writeln;
    writeln;
    writeln;
    writeln (' Control Output for input of code ' );
    writeln (' ---------------------------- ');
    FOR alphabet := 0 TO 7 DO
      writeln (alphabet+1 : 4, start [alphabet] : 20);
  END;
BEGIN
  Textmode;
  inputGenerationNo;
  inputRule;
  inputAngleNo;
  controlOutput;
  CarryOn ('Continue : ');
END;  (* Input of code *)

FUNCTION FindNext (p : integer;
             VAR source : byteArray;
             sourceLength : integer) : integer;
  VAR
    found: boolean;
    depth : integer;
  BEGIN
    depth := 0;
    found := False;
    WHILE (p < sourceLength) AND NOT found DO
    BEGIN
      p := p+1;
      IF (depth = 0) AND (source[p] < 2) THEN
      BEGIN
        findNext := source[p];
        found := True;
      END
      ELSE
      IF (depth = 0) AND (bitAND (source[p], 64)) THEN
        BEGIN
          findNext := 1;
          found := True;
        END
      ELSE IF bitAND (source[p], 128) THEN
        BEGIN
          depth := depth + 1
        END
      ELSE IF bitAND (source[p], 64) THEN
        BEGIN
          depth := depth - 1;
        END
      END;
    IF NOT found THEN
      findNext := 1;
END;    (* of findNext *)

PROCEDURE newAddition (b2, b1, b0 : integer;
                VAR row : byteArray;
                VAR code : codeArray;
                VAR rowLength : integer;
                angleNo ; intger;
  VAR ruleNo, i : integer;
BEGIN
  ruleNo := b2 * 4 + b1 * 2 + b0;
  FOR i := 1 TO code [ruleNo, 0] DO
    BEGIN
      rowLength := rowLength + 1;
```

```
            IF (code[ruleNo, i] >= 0) AND
                               (code[ruleNo, i] <= 63) THEN
              row[rowLength] := code[ruleNo, i];
            IF (code[ruleNo, i] = 64) THEN
              row[rowLength] := 64;
            IF (code[ruleNo, i] = 128) THEN
              row[rowLength] := 128 + Random MOD angleNo;
        END;
    END;  (* of newAddition *)

    PROCEDURE generation (VAR source : byteArray;
                          VAR sourceLength : integer;
                          VAR code : codeArray);
      VAR
        depth, rowLength, alphabet, k : integer;
        b0, b1, b2 : byte;
        stack : ARRAY[0..200] OF integer;
        row : byteArray;
    BEGIN
      depth := 0;
      rowLength := 0;
      b2 := 1;
      b1 := 1;
      FOR alphabet := 1 TO sourceLength DO
      BEGIN
        IF source[alphabet] < 2 THEN
        BEGIN
          b2 := b1;
          b1 := source[alphabet];
          b0 := findNext (alphabet, source, sourceLength);
          newAddition (b2, b1, bo, row, code,
                                sourceLength, angleNo);
        END
      ELSE IF bitAND (source[alphabet], 128) THEN
        BEGIN
          rowLength := rowLength + 1;
          row[rowLength] := source[alphabet];
          depth := depth + 1;
          stack[de[th] := b1;
        END
      ELSE IF bitAND( source[alphabet], 64) THEN
```

```
            BEGIN
               rowLength := rowLength + 1;
               row[rowLength] := source[alphabet];
               b1 := stack[depth];
               depth := depth - 1;
            END;
         END;
         FOR k := 1 TO rowLength DO source[k] := row[k];
         sourceLength := rowLength;
      END;  (* Of generation *)

      PROCEDURE drawGeneration (VAR graftal : byteArray;
                               VAR graftalLength : integer;
                               VAR angle : realArray;
                               VAR counter : integer);
         VAR
            arrayra, arrayxp, arrayyp : ARRAY[0..50] OF real;
            ra, dx, dy, xp, yp, length : real;
            alphabet, depth : integer;
      BEGIN
         xp := Xscreen / 2; yp := 0; ra := 0;
         depth := 0; length := 5;
         dx := 0; dy := 0;
         FOR alphabet := 1 TO graftalLength DO
         BEGIN
            IF graftal[alphabet] < 2 THEN
            BEGIN
               GoToPoint (round(xp), round(yp));
               DrawLine (round(xp+dx), round(yp+dy));
               xp := xp+dx;
               yp := yp+dy;
            END;
            IF bitAND (graftal[alphabet], 128) THEN
            BEGIN
               depth := depth + 1;
               arrayra[depth] := ra;
               arrayxp[depth] := xp;
               arrayyp[depth] := yp;
               ra := ra+angle[graftal[alphabet] MOD 16];
               dx := sin(ra) * length;
               dy := cos(ra) * length;
```

```
          END;
          IF bitAND (graftal[alphabet], 64) THEN
          BEGIN
             ra := arrayra[depth];
             xp := arrayxp[depth];
             yp := arrayyp[depth];
             depth := depth - 1;
             dx := sin(ra) * length;
             dy := cos(ra) * length;
          END;
       END;
       CarryOn ('  ');
    END  (* Of drawGeneration *)

    PROCEDURE printGeneration (VAR graftal: byteArray;
                               VAR graftalLength : integer);
       VAR p : integer;
    BEGIN
       writeln ('Graftal Length : ', graftalLength : 6);
       FOR p := 1 TO graftalLength DO
       BEGIN
          IF graftal[p] < 2 THEN write(graftal[p] : 1);
          IF bitAND (graftal[p], 128) THEN write ('[');
          IF bitAND (graftal[p], 164) THEN write (']');
       END;
       writeln;
    END;   (* Of printGeneration *)

 BEGIN
    inputCode (generationNo, code, angle, angleNo, start);
    graftalLength := 1;
    counter := 1;
    graftal[graftalLength] := 1;
    REPEAT
       generation (graftal, graftalLength, code);
       GraphicsMode;
       drawGeneration (graftal, graftalLength, angle,
                               counter);
       (* Save drawing ''Graftal'' *)
       TextMode;
       printGeneration (graftal, graftalLength);
       writeln ('There were ', counter, ' generations');
```

```
        CarryOn (' More? ');
        counter := counter + 1;
        ready := (graftalLength > 8000) OR
                      ( counter > generationNo)
                      OR button;   (* e.g. Keypressed *)
      UNTIL ready;
    END;
(* END: Problem-specific Procedures *)
```

Insert these procedures into the reference program in the appropriate places.

In the declaration part of the procedure, inside the REPEAT loop, two procedures are called which are enclosed in comment brackets. The procedure SaveDrawing provides a facility for automatically saving the picture to disk. Check in your technical manual to see whether this is possible for your machine. You can also write such a procedure yourself. The procedure printGeneration, if required, prints out the graftal as a string on the screen. It uses the form explained in Chapter 8, with the alphabet {0, 1, [,]}.

The input precedure, which we usually use to read in the data for the screen window, is absent this time. Instead, the required data are read in the procedure codeInput, which is part of the procedure Graftal. After starting the program you will get, e.g., the following dialogue:

```
Generation Number                 >10
Input of 1 .Rule  >  0
Input of 2 .Rule  >  1
Input of 3 .Rule  >  0
Input of 4 .Rule  >  1[01]
Input of 5 .Rule  >  0
Input of 6 .Rule  >  00[01]
Input of 7 .Rule  >  0
Input of 8 .Rule  >  0
Number for angle                  >4
Input of 1 .Angle (Degrees)       >-40
Input of 2 .Angle (Degrees)       >40
Input of 3 .Angle (Degrees)       >-30
Input of 4 .Angle (Degrees)       >30
```

```
Control Output for input of code
----------------------------
   1          0
   2          1
   3          0
   4          1[01]
   5          0
   6          00[01]
   7          0
   8          0
Continue: hit <RETURN>
```

Figure 11.2-2 Input dialogue for `Graftal`.

What else is there left to say about Chapter 8? In §8.4 on repetitive patterns a series of program fragments were given, but we think you can embed them in the reference program yourself without any difficulty.

11.3 Ready, Steady, Go!

'Ready, steady, go!' After fractals and graftals we return to the beginning of our computer graphics experiments on Feigenbaum diagrams, in particular landscapes, and the remarkable appearance of the Hénon attractor.

The first exercises that should have stimulated you to experiment were given in the form of the following program fragments:

- 2.1-1 `MeaslesValue`, numerical calculation
- 2.1.1-1 `MeaslesIteration`, graphical representation
- 2.1.2-1 `ParabolaAndDiagonal`, graphical iteration
- 2.2-2 `DisplayFeedback`, output of the feedback constant.

You can complete these without difficulty.

In §2.2.1, which dealt with the bifurcation scenario, we made an attempt to treat the k_i-values of the bifurcation points logarithmically, to estimate the Feigenbaum number (see Exercise 2.2.1-2). Because this exercise involves some difficulty, we give a solution here.

Program 11.3-1 (For Exercise 2.2.1-2)

```
PROCEDURE FeigenbaumIteration;
   VAR
      range, i, iDiv, iMod : integer;
      epsilon, kInfinity, population : real;
      delatxPerPixel : real;
BEGIN
```

```
      epsilon  := (ln (10.0) / 100);
      kInfinity := 2.57;
      Left := 0.0; Right := 400;
      FOR range := 0 TO Xscreen DO
         BEGIN
            iDiv := 1 + range DIV 100;
            iMod := range MOD 100;
            Feedback := kInfinity -
                 exp((100-iMod)*epsilon)*exp(-iDiv*ln(10.0));
            population := 0.3;
            IF Feedback > 0 THEN BEGIN
               FOR i := 0 to Invisible DO
                  population := f(population, Feedback);
               FOR i:= 0 TO Visible DO
               BEGIN
                  SetUniversalPoint (range, population);
                  population := f(population, Feedback);
               END;
         END;
      END;
    END;
```

In addition, the discussion of how one generates a Feigenbaum landscape may have been a little too brief.

Program 11.3-2 (Feigenbaum landscape)

```
(* BEGIN: Problem-specific procedures *)
    FUNCTION f (p, k : real) : real;
    BEGIN
       f := p + k * p * (1-p);
    END;

    PROCEDURE FeigenbaumLandscape;
       CONST
          lineNo := 100;
          garbageLine = 2;
       TYPE
          box = ARRAY[0..Xscreen] OF integer;
       VAR
          pixel, maximalValue : box;
          i, pixelNo : integer;
```

```
  p, k, real;

PROCEDURE initialisePicture;
  VAR
       j : integer;
BEGIN
(* For the Macintosh you must use:       *)
(* pixelNo = Xscreen-lineNo-WindowBorder *)
(* instead of                            *)
( * pixelNo = Xscreen-lineNo             *)
pixelNo = Xscreen-lineNo
FOR j := 0 TO Xscreen DO
  maximalValue[j] := WindowBorder;
              (*obliterate everything*)
END;

PROCEDURE initialiseRows (i : integer);
  VAR
     j : integer;
BEGIN
  FOR j := 1 TO pixelNo DO
     pixel[j] := 0;  (* clear pixels *)
  k := Right - j*(Right - Left)/ lineNo;
  p := 0.3;  (* leave start value the same *)
END;

PROCEDURE fill (p : real);
  VAR
     j : integer;
BEGIN
  j := trunc((p-Bottom) * pixelNo / (Top - Bottom));
  IF (j >= 0) AND )j <= pixelNo) THEN
     pixel[j] := pixel[j]+1;
END;

PROCEDURE iterate;
  VAR
     j : integer;
BEGIN
  FOR j := 1 TO Invisible DO
     p := f (p,k);
  FOR j := 1 TO Visible DO
```

```
      BEGIN
        fill (p);
        p := f (p,k);
      END;
    END;

    PROCEDURE sort (i : integer);
      VAR
        j, height : integer;
    BEGIN
      FOR j := 1 TO pixelNo DO
      BEGIN
        height := WindowBorder +
          garbageLines * i + factor * pixel[j];
        IF maximalValue[j+i] < height THEN
          maximalValue[j+i] := height;
      END;
    END;

    PROCEDURE draw (i : integer);
      VAR
        j : integer;
    BEGIN
      SetUniversalPoint (0,0);
      FOR j := 1 TO pixelNo +i DO
        DrawUniversalLine (j, maximalValue[j]);
    END;

  BEGIN
    initialisePicture;
    FOR i := 1 TO lineNo DO
      BEGIN
        initialiseRows (i);
        iterate;
        sort (i);
        draw (i);
      END;  (* for i *)
  END;
(* END : Problem-specific procedures *)
```

In Chapter 3 the Program fragments were given so clearly in Pascal notation that they can easily be incorporated into a complete program. Because some of you may perhaps not have seen the picture of the Rössler attractor, we give it here, together with the corresponding program.

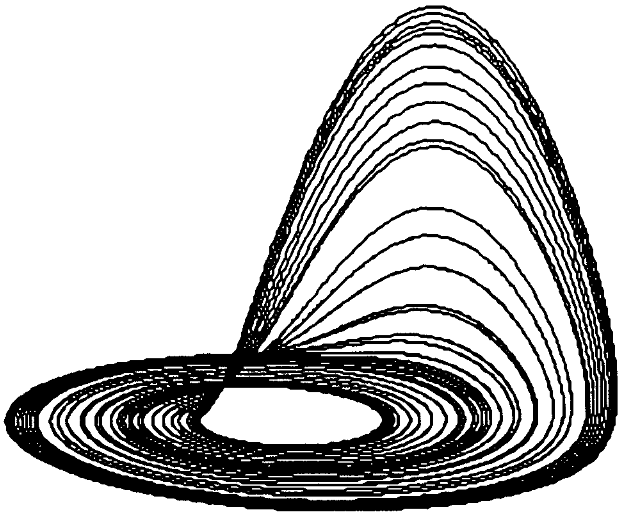

Figure 11.3-1 Rössler attractor.

You obtain this figure if you incorporate the following procedure into your program. Compute 1000 steps without drawing anything, so that we can be sure that the iteration sequence has reached the attractor. Then let the program draw, until we stop it.

Give the variables the following values:

```
Left : = -15; Right := 15;  Bottom := -15; Top := 60;
A := 0.2; B := 0.2;  C := 5.7;
```

Program 11.3-3 (Rössler attractor, see Program Fragment 3.3-1)

```
PROCEDURE Roessler;
   VAR
      i : integer;

   PROCEDURE f;
      CONST
         delta = 0.005;
      VAR
         DX, DY, DZ : REAL;
   BEGIN
      dx := - (y + z);
      dy := x + y * A;
      dz := B + z * (x - C);
      x := x + delta * dx;
      y := y + delta * dy;
      z := z + delta * dz;
    END;
  BEGIN  (* Roessler *)
    x := -10;
    y := -1;
    z := -1;
    f;
    REPEAT
      f;
      i := i + 1;
    UNTIL i = 1000;
    SetUniversalPoint (x, x+ z + z);
    REPEAT
      f;
      DrawUniversalLine (x, y + z + z);
    UNTIL Button;
  END;  (* Roessler *)
```

'Ready, steady, go' – in this section the speed is often a sprint. This happens because the problems, compared with the Julia and Mandelbrot sets, are very simple and do not require intensive computation. Nevertheless, before we devote our next section to this problem, we will give a few hints for Chapter 4.

The sketches that explain Newton's method must of course be provided with a drawing program, whose central part we now show you:

Program Fragment 11.3-4 (Newton demonstration)

```
VAR
  N1, N2, N3, StartValue, Left, Right, Top, Bottom : real;
...
FUNCTION f (x : real) : real;
BEGIN
  f := (x - N1) * (x - N2) * (x - N3);
END;

PROCEDURE drawCurve;
  VAR
    i : integer;
```

```
      deltaX : real;
BEGIN
(* First draw coordinate axes *)
   SetUniversalPoint (Left, 0);
   DrawUniversalLine (Right, 0);
   SetUniversalPoint (0, Bottom);
   DrawUniversalLine (0, Top);
(* Then draw curve *)
   SetUniversalPoint (Left, f(Left));
   i := 0;
   deltaX := (Right - Left) / Xscreen;
   WHILE i <= Xscreen DO
      BEGIN
         DrawUniversalLine (Left+i*deltaX,f(Left+i*deltaX));
         i := i+3;
      END;
END;  (* drawCurve *)

PROCEDURE approximation (x : real);
   CONST
      dx = 0.001;
   VAR
      oldx, fx, fslope : real;
BEGIN (* approximation *)
   REPEAT
      oldx := x;
      fx := f(x);
      fslope := (f(x+dx)-f(x-dx))/(dx+dx)
      IF fslope <> 0.0 THEN
         x := x - fx / fslope
      SetUniversalPoint (oldx, 0);
      DrawUniversalLine (oldx, fx);
      DrawUniversalLine (x, 0);
   UNTIL (ABS(fx) < 1.0E-5);
END;  (* approximation *)

PROCEDURE Initialise;
BEGIN
   ReadReal ('Left          >', Left);
   ReadReal ('Right         >', Right);
   ReadReal ('Bottom        >', Bottom);
   ReadReal ('Top           >', Top);
```

```
      ReadReal ('Root 1        >', N1);
      ReadReal ('Root 2        >', N2);
      ReadReal ('Root 3        >', N3);
      ReadReal ('Start Value   >', StartValue);
   END;

   PROCEDURE ComputeAndDisplay;
   BEGIN
      EnterGraphics;
      drawCurve;
      approximation (StartValue);
      ExitGraphics;
   END;
```

In the standard example of Chapter 4 the roots had the values

```
      N1 := -1; N2 := 0; N3 := 1;
```

Of course, that does not prevent you experimenting with other numbers.

11.4 The Loneliness of the Long-distance Reckoner

All our experiments with Julia and Mandelbrot sets have a distressing feature. They take a long time.

In this section we give you some hints for alleviating or exacerbating the loneliness of your long-distance computer.

In Chapter 5 we discussed the representation of Julia sets in the form of a detailed program fragment. We will now provide a full representation as a complete program, in which you see all of the individual procedures combined into a single whole. But again we limit ourselves, as usual, to giving only the problem-specific part and the input procedure.

Program Fragment 11.4-1

```
   PROCEDURE Mapping;
      CONST
         epsq = 0.0025;
      VAR
         xRange, yRange : integer;
         x, y, deltaxPerPixel, deltayPerPixel : real;

      FUNCTION  belongsToZa (x, y : real) : boolean;
         CONST
            xa = 1.0;
      ya = 0.0;
BEGIN
   belongsToZa := (sqr(x-xa)+sqr(y-ya) <= epsq);
END; (*  belongsToZa)

FUNCTION  belongsToZb (x, y : real) : boolean;
   CONST
      xb = -0.5;
      yb = 0.8660254;
BEGIN
   belongsToZb := (sqr(x-xb)+sqr(y-yb) <= epsq);
END; (*  belongsToZb)

FUNCTION  belongsToZc (x, y : real) : boolean;
   CONST
      xc = -0.5;
      yc = -0.8660254;
BEGIN
   belongsToZc := (sqr(x-xc)+sqr(y-yc) <= epsq);
END; (*  belongsToZc)

FUNCTION JuliaNewtonComputeAndTest (x, y : real)
                  : boolean;
   VAR
      iterationNo : integer;
      finished : boolean;
      xSq, ySq, xTimesy, denominator : real;
      distanceSq, distanceFourth : real;

   PROCEDURE startVariableInitialisation;
   BEGIN
      finished := false;
      iterationNo := 0;
      xSq := sqr(x);
      ySq := sqr(y);
      distanceSq := xSq + ySq;
   END (* startVariableInitialisation *)

   PROCEDURE compute;
   BEGIN
      iterationNo := iterationNo + 1;
      xTimesy := x*y;
      distanceFourth := sqr(distanceSq);
      denominator := distanceFourth+distanceFourth
                        +distanceFourth;
      x     := 0.666666666*x + (xSq-ySq)/denominator;
      y     := 0.666666666*y -
                  (xTimesy+xTimesy)/denominator;
      xSq      := sqr(x);
      ySq      := sqr(y);
      distanceSq := xSq + ySq;
   END;

   PROCEDURE test;
   BEGIN
      finished := (distanceSq < 1.0E-18)
         OR (distanceSq > 1.0E18)
            OR belongsToZa (x,y);
      IF NOT finished THEN finished := belongsToZb (x,y);
      IF NOT finished THEN finished := belongsToZc (x,y);
   END;

   PROCEDURE distinguish;
   BEGIN
   (* Choose one of the statements *)
   (* and delete all the others *)
      JuliaNewtonComputeAndTest :=
         iterationNo = maximalIteration;
      JuliaNewtonComputeAndTest := belongsToZc (x,y)
      JuliaNewtonComputeAndTest :=
         (iterationNo < maximalIteration) AND
                  odd (iterationNo);
      JuliaNewtonComputeAndTest :=
         (iterationNo<maximalIteration) AND
                  (iterationNo MOD 3 = 0);
   END;

BEGIN
   startVariableInitialisation;
   REPEAT
      compute;
      test;
   UNTIL (iterationNo = maxIteration) OR finished;
```

```
         distinguish;
      END; (* JuliaNewtonComputeAndTest *)

  BEGIN
     deltaxPerPixel := (Right - Left ) / Xscreen;
     deltayPerPixel := (Top - Bottom ) / Yscreen;
     y := Bottom;
     FOR yRange := 0 TO yScreen DO
     BEGIN
        x := Left;
        FOR xRange := 0 TO xScreen DO
        BEGIN
           IF JuliaNewtonComputeAndTest (x,y)
              THEN SetUniversalPoint (xRange, yRange);
           x := x + deltaxPerPixel;
        END;
        y := y + deltayPerPixel;
     END;
  END; (* Mapping *)

  PROCEDURE Initialise;
  BEGIN
     ReadReal (' Left         >', Left);
     ReadReal (' Right        >', Right);
     ReadReal (' Bottom       >', Bottom);
     ReadReal (' Top          >', Top);
     ReadInteger ('Maximal Iteration   >', MaximalIteration);
  END;
```

And now we give the version for Julia sets using quadratic iteration:

Program Fragment 11.4-2

```
  PROCEDURE Mapping;
     VAR
        xRange, yRange : integer;
        x, y, deltaxPerPixel, delayPerPixel : real;

        FUNCTION JuliaComputeAndTest (x, y : real) : boolean;
           VAR
              iterationNo : integer;
              xSq, ySq, distanceSq   : real;
        finished : boolean;
      PROCEDURE startVariableInitialisation;
      BEGIN
        finished := false;
        iterationNo := 0;
        xSq := sqr(x); ySq := sqr(y);
        distanceSq := xSq + ySq;
      END;  (* startVariableInitialisation *)

      PROCEDURE compute;
      BEGIN
        iterationNo := iterationNo  + 1;
        y := x * y;
        y : = y + y - cImaginary;
        x := xSq - ySq - cReal;
        xSq := sqr(x); ysQ := sqr(y);
        distanceSq := xSq + ySq;
      END;  (* compute *)

      PROCEDURE test;
      BEGIN
        finished := (distanceSq > 100.0);
      END;  (* test *)

      PROCEDURE distinguish;
      BEGIN  (* See also Program Fragment 11.4-1*)
        JuliaComputeAndTest :=
          iterationNo = maximalIteration;
      END;  (* distinguish *)

  BEGIN (* JuliaComputeAndTest *)
    startVariableInitialisation;
    REPEAT
      compute;
      test;
    UNTIL (iterationNo = maximalIteration) OR finished;
    distinguish;
  END; (* JuliaComputeAndTest *)

BEGIN
  deltaxPerPixel := (Right - Left ) / Xscreen;
  deltayPerPixel := (Top - Bottom ) / Yscreen;
```

```
      y := Bottom;
      FOR yRange := 0 TO yScreen DO
      BEGIN
         x := Left;
         FOR xRange := 0 TO xScreen DO
         BEGIN
            IF JuliaComputeAndTest (x,y)
               THEN SetUniversalPoint (xRange, yRange);
            x := x + deltaxPerPixel;
         END;
         y := y + deltayPerPixel;
      END;
   END; (* Mapping *)

   PROCEDURE Initialise;
   BEGIN
      ReadReal ('Left                    >', Left);
      ReadReal ('Right                   >', Right);
      ReadReal ('Bottom                  >', Bottom);
      ReadReal ('Top                     >', Top);
      ReadReal ('cReal                   >', cReal);
      ReadReal ('cImaginary              >', cImaginary);
      ReadInteger ('Maximal Iteration    >', MaximalIteration);
   END;
```

We have already explained in Chapter 5 that the wrong choice of *c*-value can considerably extend the 'loneliness of the long-distance computer' and leave you sittting in front of an empty screen for several hours. To get a quick preview and to shorten the time taken looking for interesting regions, we used the method of backwards iteration. We consider Program Fragments 5.2-3 and 5.2-4 to be so clear that they do not need further elaboration here.

Proceeding from Julia sets, we finally made our 'encounter with the Gingerbread Man'. Again we will collect together the important parts of the program here.

Program Fragment 11.4-3 (see amplifying remarks in Chapter 6)

```
   PROCEDURE Mapping;
      VAR
         xRange, yRange : integer;
         x, y, x0, y0, deltaxPerPixel, deltayPerPixel : real;
```

```
FUNCTION MandelbrotComputeAndTest (cReal, cImaginary
                 : real)
           : boolean;
  VAR
    iterationNo : integer;
    x, y, xSq, ySq, distanceSq : real;
    finished: boolean;

  PROCEDURE  StartVariableInitialisation;
  BEGIN
    finished := false;
    iterationNo := 0;
    x := x0;
    y := y0;
    xSq := sqr(x);
    ySq := sqr(y);
    distanceSq := xSq + ySq;
  END; (* StartVariableInitialisation *)

  PROCEDURE compute;
  BEGIN
    iterationNo := iterationNo + 1;
    y := x*y;
    y := y+y-cImaginary;
    x := xSq - ySq -cReal;
    xSq := sqr(x);
    ySq := sqr(y);
    distanceSq := xSq + ySq;
  END; (* compute *)

  PROCEDURE test;
  BEGIN
    finished := (distanceSq > 100.0);
  END; (* test *)

  PROCEDURE distinguish;
  BEGIN  (* See also Program Fragment 11.4-1 *)
    MandelbrotComputeAndTest : =
      iterationNo = maximalIteration;
  END;  (* distinguish *)
```

```
    BEGIN (* MandelbrotComputeAndTest *)
      StartVariableInitialisation;
      REPEAT
        compute;
        test;
      UNTIL (iterationNo = maximalIteration) OR finished;
      distinguish;
    END;  (* MandelbrotComputeAndTest *)

  BEGIN
    deltaxPerPixel := (Right - Left) / Xscreen;
    deltayPerPixel := (Top - Bottom) / Yscreen;
    x0 := 0.0; y0 := 0.0;
    y := Bottom;
    FOR yRange := 0 TO Yscreen DO
    BEGIN
      x:= Left;
      FOR xRange := 0 TO Xscreen DO
        BEGIN
          IF MandelbrotComputeAndTest (x, y)
            THEN SetPoint (xRange, yRange);
          x := x + deltaxPerPixel;
        END;
      y := y + deltayPerPixel;
    END;
  END; (*Mapping)

  PROCEDURE Initialise;
  BEGIN
    ReadReal ('Left                > '; Left);
    ReadReal ('Right               > '; Right);
    ReadReal ('Bottom              > '; Bottom);
    ReadReal ('Top                 > '; Top);
    ReadReal ('MaximalIteration    > '; MaximalIteration );
  END;
```

In addition, the five different methods by which we represented the basins of attraction in §6.2 should briefly be mentioned.

The simplest is Case 1. We have already dealt with this in Program Fragment 11.4-3 without saying so. If you give the starting values `x0` and `y0` in `Mapping` another value, the computation can explode.

In order that the other four cases can be investigated with the fewest possible program changes, we modify the procedure `Mapping` only slightly. Before calling `MandelbrotComputeAndTest` we insert one from a block of four program segments, which ensure that the right variables change and the others stay constant. The two global variables `FixedValue1` and `FixedValue2` must be read from the keyboard.

Program Fragment 11.4-4 (Cases 2 to 5)

```
PROCEDURE Mapping;
   VAR
      xRange, yRange : integer;
      x, y, x0, y0, deltaxPerPixel, deltayPerPixel : real;
...

BEGIN
   deltaxPerPixel := (Right - Left) / Xscreen;
   deltayPerPixel := (Top - Bottom) / Yscreen;
   y := Bottom;
   FOR yRange := 0 TO Yscreen DO
   BEGIN
      x:= Left;
      FOR xRange := 0 TO Xscreen DO
         BEGIN
            (* Case 2 *)
            x0 := FixedValue1;
            y0 := y;
            cReal := FixedValue2;
            cImaginary := x;
            IF MandelbrotComputeAndTest (x, y)
               THEN SetPoint (xRange, yRange);
            x := x + deltaxPerPixel;
         END;
      y := y + deltayPerPixel;
   END;
END; (*Mapping)
```

```
| (* Case 3 *)                    | (* Case 4 *)              |
| x0 := FixedValue1;              | x0 := y;                  |
| y0 := y;                        | y0 := FixedValue1;        |
| cReal := x;                     | cReal := FixedValue2;     |
| cImaginary := FixedValue2;      | cImaginary := x;          |
| (* Case 5 *)                  | (* Case 1, alternative *) |
| x0 := y;                      | x0 := FixedValue1;        |
| y0 := FixedValue1;            | y0 := FixedValue2;        |
| cReal := x;                   | cReal := x;               |
| cImaginary := FixedValue2;    | cImaginary := y;          |
```

Select whichever version is best for your problem.

```
BEGIN
   ReadReal ('Left              > '; Left);
   ReadReal ('Right             > '; Right);
   ReadReal ('Bottom            > '; Bottom);
   ReadReal ('Top               > '; Top);
   ReadReal ('FixedValue1       > '; FixedValue1 );
   ReadReal ('FixedValue2       > '; FixedValue2 );
   ReadReal ('MaximalIteration  > '; MaximalIteration );
END;
```

The central procedure of the program, with which we generated Figures 6.3-4 to 6.3-6, is given in the next program fragment. The drawing, which almost always takes place within `Mapping`, is here done inside the procedure `computeAndDraw`.

Program Fragment 11.4-5 (Quasi-Feigenbaum diagram)

```
PROCEDURE Mapping;
   VAR
      xRange : integer;
      x1, y1, x2, y2, deltaXPerPixel : real;
      dummy : boolean;

   FUNCTION ComputeAndTest (cReal, cIamginary : real)
                  : boolean;
      VAR
         IterationNo : integer;
         x, y, xSq, ySq, distanceSq : real;
         finished : boolean;

      PROCEDURE StartVariableInitialisation;
      BEGIN
         x := 0.0; y := 0.0;
         finished := false;
         iterationNo := 0;
         xSq := sqr(x); ySq := sqr(y);
         distanceSq := xSq + ySq;
      END; (* StartVariableInitialisation *)

      PROCEDURE computeAndDraw;
      BEGIN
         iterationNo := iterationNo + 1;
         y := x*y;
         y := y+y-cImaginary;
         x := xSq - ySq - cReal;
         xSq := sqr(x); ySq := sqr(y);
         distanceSq := xSq + ySq;
         IF (iterationNo > bound) THEN
                    SetUniversalPoint (cReal,x);
      END; (* ComputeAndTest *)

      PROCEDURE test;
      BEGIN
         finished := (distanceSq > 100.0);
      END; (* test *)

   BEGIN (* ComputeAndTest *)
      StartVariableInitialisation;
      REPEAT
         computeAndDraw;
         test;
      UNTIL (iterationNo = maximalIteration) OR finished;
   END (* ComputeAndTest *)

BEGIN
   x1 := 0.1255; y1 := 0. 6503;
   x2 := 0.1098; y2 := 0.882;
   FOR xRange := 0 TO Xscreen DO
      ComputeAndTest
         (x1-(x2-x1)/6+xRange*(x2-x1)/300,
         y1-(y2-y1)/6+xRange*(y2-y1)/300);
END;  (* Mapping *)
```

For §6.4, 'Metamorphoses', in which we dealt with higher powers of complex numbers, we will show you only the procedure `Compute`. It uses a local procedure `compPow`. Everything else remains as you have seen it in Program Fragment 11.4-3. Do not forget to give `power` a reasonable value.

Program Fragment 11.4-6 (High-powered Gingerbread Man)

```
PROCEDURE Compute
  VAR
    t1, t2 : real;

  PROCEDURE compPow (in1r,in1i, power: real;
                    VAR outr, outi: real);
    CONST
      halfpi := 1.570796327;
    VAR
      alpha, r : real;
  BEGIN
    r := sqrt (in1r*in1r + in1i * in1i);
    IF  r > 0.0 then r := exp (power * ln(r));
    IF ABS(in1r) < 1.0E-9 THEN
      BEGIN
        IF in1i > 0.0 THEN alpha := halfpi;
                  ELSE alpha := halfpi + Pi;
      END ELSE BEGIN
        IF in1r > 0.0 THEN alpha := arctan (in1i/in1r)
                  ELSE alpha := arctan (in1i/in1r) + Pi;
      END;
    IF alpha < 0.0 THEN alpha := alpha + 2.0*Pi;
    alpha := alpha * power;
    outr := r * cos(alpha);
    outi := r * sin(alpha);
  END;  (* compPow *)

BEGIN (* Compute *)
  compPow (x, y, power, t1, t2);
  x := t1 - cReal;
  y := t2 - cImaginary;
  xSq := sqr (x);
  ySq := sqr (y);
  iterationNo := iterationNo + 1;
END;  (* Compute *)
```

From Chapter 7 we display only the pseudo-3D representation. This time it was a Julia set that we drew. The remaining program fragments are so clearly listed that it will give you no trouble to include them.

Program Fragment 11.4-7 (Pseudo-3D graphics)

```
    TYPE
      D3maxtype = ARRAY[0..Xscreen] OF integer;
    VAR
      D3max : D3maxtype;
      Left, Right, Top, Bottom,
      D3factor, CReal, CImaginary : real;
      D3xstep, D3ystep, MaximalIteration, Bound : integer;
      PictureName : STRING;

  PROCEDURE D3mapping;
    VAR
      dummy : boolean;
      xRange, yRange : integer;
      x, y, deltaxPerPixel, deltayPerPixel : real;
    FUNCTION D3ComputeAndTest (x, y : real; xRange, yRange
                        : integer)
                        : boolean;
      VAR
        iterationNo : integer;
        xSq, ySq, distanceSq : real;
        finished: boolean;
      PROCEDURE StartVariableInitialisation;
      BEGIN
        finished := false;
        iterationNo := 0;
        xSq := sqr (x);
        ySq := sqr (y);
        distanceSq := xSq + ySq;
      END; (* StartVariableInitialisation *)

      PROCEDURE Compute;  (* Julia-set *)
      BEGIN
        iterationNo := iterationNo + 1;
        y := x * y;
        y := y + y - CImaginary;
        x := xSq - ySq - CReal;
        xSq := sqr (x);
        ySq := sqr (y);
        distanceSq := xSq + ySq;
      END;  (*Compute *)

      PROCEDURE Test;
      BEGIN
         finished := (distanceSq > 100.0);
      END;   (* Test *)

      PROCEDURE D3set (VAR D3max : D3maxType;
                   column, row, height : integer);
         VAR
            cell, content : integer;
      BEGIN
         cell := column + row - (yScreen -100) DIV 2;
         IF (cell >= 0) AND (cell <= xScreen) THEN
         BEGIN
            content := height * D3factor + row;
            IF content > D3max[cell] THEN
               D3max[cell] := content;
         END;
      END;  (* D3set *)

   BEGIN (* D3ComputeAndTest *)
      D3ComputeAndTest := true;
      StartVariableInitialisation;
      REPEAT
         Compute;
         Test;
      UNTIL (iterationNo = maximalIteration) OR finished;
      D3set (D3max, xRange, yRange, iterationNo);
   END (* D3ComputeAndTest *)

   PROCEDURE D3draw (D3max: D3maxType);
      VAR
         cell, coordinate : integer;
   BEGIN
      setUniversalPoint (Left, Bottom);
      FOR cell := 0 TO xScreen DO
         IF (cell MOD D3xstep = 0) THEN
         BEGIN

(* Warning!  The procedure pensize used below is Macintosh- *)
(* specific and cannot be implemented easily in other       *)
(* Pascal dialects.  If it does not exist on your computer, *)
(* omit the next few lines of code                         *)
(* from here ----------------------------------------------- *)
                IF cell > 1 THEN
                  IF (D3max[cell] = 100+yRange) AND
                     (D3max[cell-D3xstep]=100+yRange)
                  THEN
                    pensize (1, D3ystep)
                  ELSE
                    pensize (1, 1)
(*---------------------------------------------- to here  *)

                coordinate := D3max[cell];
                IF coordinate >0 THEN DrawLine
                      (cell, coordinate);
              END;
          END;  (* D3draw *)

      BEGIN
        FOR xRange := 0 TO xScreen DO
          D3max[xRange] := 0;
        deltaxPerPixel := (Right - Left) / (xScreen - 100);
        deltayPerPixel := (Top - Bottom) / (yScreen - 100);
        y := Bottom;
        FOR yRange := 0 to (yScreen - 100) DO
          BEGIN
            x := Left;
            FOR xRange := 0 TO (xScreen - 100) DO
              BEGIN
                IF (xRange MOD d3ystep = 0) THEN
                  dummy:= D3ComputeAndTest
                             (x, y, xRange, yRange);
                x := x + deltaxPerPixel;
              END;
            D3Draw (D3max);
            y := y + deltayPerPixel;
          END;
      END; (* Mapping *)
      (* END: Problem-specific procedures *)

      PROCEDURE Initialise;
```

```
BEGIN
   ReadReal ('Left                  >', Left);
   ReadReal ('Right                 >', Right);
   ReadReal ('Top                   >', Top);
   ReadReal ('Bottom                >', Bottom);
   ReadReal ('c-real                >', CReal);
   ReadReal ('c-imaginary           >', CImaginary);
   ReadInteger ('Max. iteration No  >', MaximalIteration);
   ReadInteger ('3D factor          >', D3factor);
   ReadInteger ('3D step - x        >', D3xstep);
   ReadInteger ('3D step - y        >', D3ystep);
END;
```

11.5 What You See Is What You Get

Now we will eliminate another minor disadvantage in our graphics. Pictures on the screen may appear quite beautiful - but it is of course better to print them out, solving the problem of Christmas and birthday presents, or to record the data directly on floppy disk and reload them into the computer at will.

To make ourselves absolutely clear: we will not tell you here how to produce a screen-dump, so-called *hard copy*, on your computer and with your particular printer. The possible combinations are innumerable, and all the time you find new tricks and tips in the computer magazines. Instead we take the point of view that you have a knob somewhere on your computer which causes whatever is on the screen to be printed out on paper, or that you own a graphics utility program that collects the pictures you have produced in memory or on disk, pretties them up, and prints them.

In this chapter we prefer to go into the problem of *soft copy*, that is, into machine-independent methods for storing information generated by computations. A computer does not have to be capable of graphics under all circumstances: it need only generate the data. Admittedly, it is often possible to draw the corresponding pictures on the same machine. The data obtained in this way can be sent to other chaos researchers, and to other types of computer. From there the output can be processed further, for example as coloured pictures.

And in contrast to all drawing methods that we have explained previously, not a single bit of information is lost, so that we can always distinguish between 'black and white'.

We will explain three methods for storing graphical data. The reason is that speed of operation and compactness of storage often conflict. It is left up to you to select which method to use, if you implement one of the following methods in your programs, or whether you develop an entirely different storage concept. But remember, it must be comprehensible to the people (and their computers) with whom you work. In Figure 11.5-1 we show the three methods together.

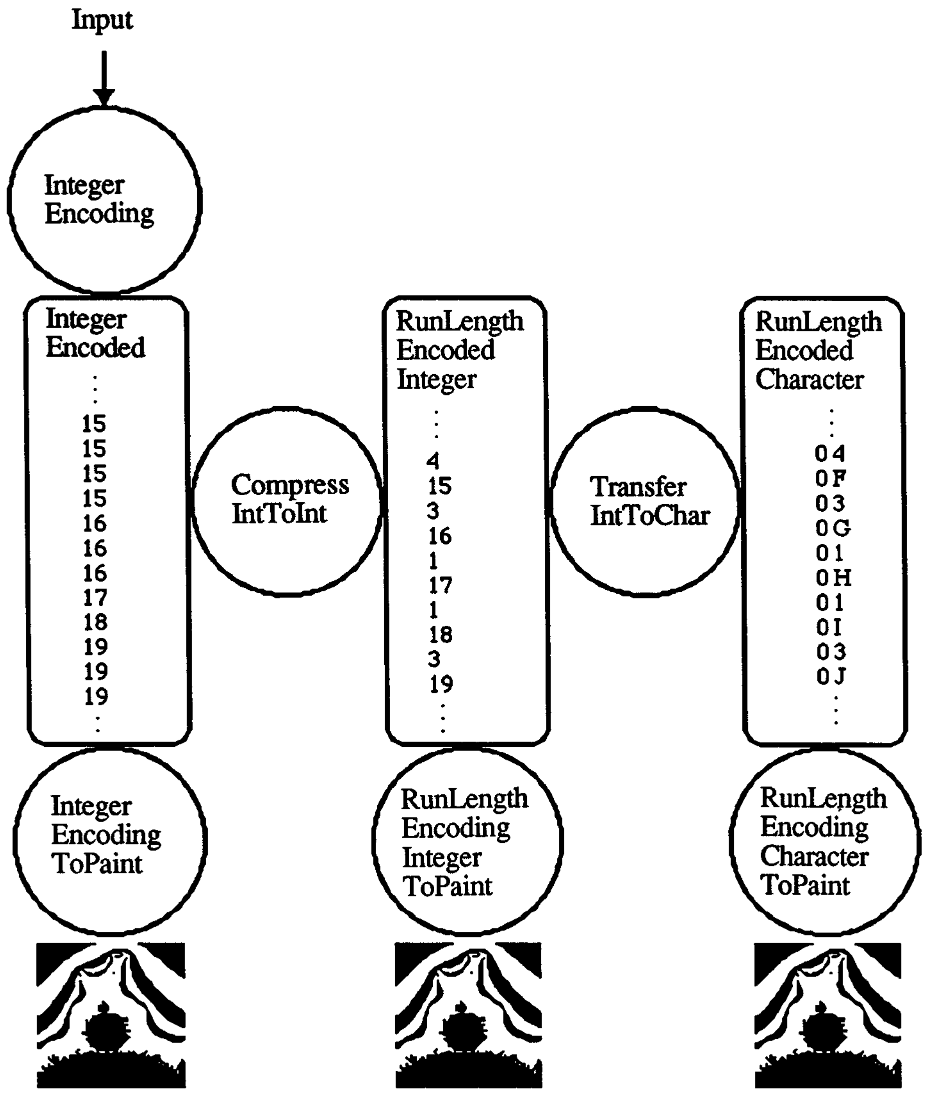

Figure 11.5-1 Three methods of data compression.

The circles represent six programs, which perform the different transformations. Others are certainly possible, and you can develop your own. In the upper part of the picture you can see the different states of the data in our conception. We start with just one idea, in which we communicate with the computer in the form of keyboard input. Instead of producing a graphic (lower picture) straight away, as we have usually done so far, we record the results of the computation as data (long rectangle) on a floppy disk. In the Mapping program this is given by the variable iterationNo, whose current value represents the result of the computation. In order to build in a few possibilities for error detection and further processing, we always write a '0' (which, like negative numbers, cannot occur) when a screen row has been calculated. This number acts as an 'end of line marker'.

The corresponding program (for example the Gingerbread Man) on the whole differs only a little from the version we have seen already. But at the start, instead of the graphics commands, now superfluous, is a variable of type file, and three procedures which make use of it are introduced. These are Store, EnterWriteFile, and ExitWriteFile, which are called in place of Mapping or ComputeAndDisplay. The procedures reset and rewrite are standard Pascal. In some dialects however these may be omitted. This is also true for the implementation of read and write or put and get. If so, you must make special changes to the relevant procedures. Find out about this from your manuals.[1]

The functional procedure MandelbrotComputeAndTest uses, in contrast to the direct drawing program, an integer value, which can immediately be transcribed to disk. We call this method of data representation *integer encoding*.

Program 11.5-1 (Integer encoding)

```
    program integerencoding; (* berkeley pascal on sun or vax *)
       const
          stringlength = 8;
          xscreen = 320;   (* e.g. 320 pixels in x-direction *)
          yscreen = 200;   (* e.g. 200 pixels in y-direction *)
       type
          string8 = packed array[1..stringlength] of char;
          intfile = file of integer;
       var
          f : intfile;
          dataname = string8;
          left, right, top, bottom : real;
     (* include further global variables here *)
          maximal iteration : integer;
(* ---------------------- utility -------------------------*)
(* begin: useful subroutines *)
     procedure readstring (information : string8; var value
                                : string8);
```

[1]TurboPascal 3.0 running under MS-DOS departs from the language standard. Here reset and rewrite must be used in conjuntion with the assign command. In many Pascal implementations the combination of read and write is defined only for text files. In TurboPascal and Berkeley Pascal, however, it is defined more generally.

```
    begin
      write  (information); value := 'pictxxxx';
(* a part for interactive input of the data name in a      *)
(* packed array [...] of char, which we do not give        *)
(* here.  Pascal I/O is often extremely machine-           *)
(* specific, because no data type 'string' is provided.    *)
(*The name here is fixed: pictxxxx (8 letters)             *)
    end;

(* end: useful subroutines *)
(* ---------------------- utility ------------------------- *)

(* ----------------------- file -------------------------  *)
(* begin: file procedures *)
    procedure store (var f: intfile; number : integer);
    begin
      write (f, number);
    end;

    procedure enterwritefile(var f : intfile; filename
                          : string8);
    begin
      rewrite (f, filename);
    end;

    procedure exitwritefile (var f : intfile);
    (* if necessary close(f); *)
    begin;
    end;

(* end: file procedures *)
(* ----------------------- file -------------------------  *)

(* --------------------- application -------------------  *)
(* begin: problem-specific procedures *)

    procedure mapping;
      var
        xrange, yrange : integer;
        x, y, xo, y0, deltaxperpixel
      function mandelbrotcomputeandtest (creal, cimaginary
                          : real)
```

```
            : integer;
   var
      iterationno : integer;
      x, y, xsq, ysq, distancesq : real;
      finished: boolean;

   procedure  startvariableinitialisation;
   begin
      finished := false;
      iterationno := 0;
      x := x0;
      y := y0;
      xsq := sqr(x);
      ysq := sqr(y);
      distancesq := xsq + ysq;
   end; (* startvariableinitialisation *)

   procedure compute;
   begin
      iterationno := iterationno + 1;
      y := x*y;
      y := y+y-cimaginary;
      x := xsq - ysq -creal;
      xsq := sqr(x);
      ysq := sqr(y);
      distancesq := xsq + ysq;
   end; (* compute *)

   procedure test;
   begin
      finished := (distancesq > 100.0);
   end; (* test *)

   procedure distinguish;
   begin  (* see also program fragment 11.4-1 *)
      mandelbrotcomputeandtest : =iterationno;
   end;  (* distinguish *)

begin (* mandelbrotcomputeandtest *)
   startvariableinitialisation;
   repeat
```

```
          compute;
          test;
        until (iterationno = maximaliteration) or finished;
        distinguish;
      end;  (* mandelbrotcomputeandtest *)

    begin  (* mapping *)
      deltaxperpixel := (right - left) / xscreen;
      deltayperpixel := (top - bottom) / yscreen;
      x0 := 0.0; y0 := 0.0;
      y := bottom;
      for yrange := 0 to yscreen do
      begin
        x:= left;
        for xrange := 0 to xscreen do
          begin
            store(f,mandelbrotcomputeandtest(x,  y));
            x := x + deltaxperpixel;
          end;
          store  (f,  0);     { at the end of each row }
        y := y + deltayperpixel;
      end;
    end; (*mapping)

(* end: problem-specific procedures *)
(* ----------------------- application -------------------- *)

(* -------------------------- main ----------------------- *)
(* begin: procedures of main program *)
    procedure hello;
    begin
      writeln;
      writeln ('computation of picture data ');
      writeln ('----------------------- ');
      writeln; writeln;
    end;

    procedure Initialise;
    begin
      readreal ('left              > '; left);
      readreal ('right             > '; right);
      readreal ('bottom            > '; bottom);
      readreal ('top                  > '; top);
      readreal ('maximaliteration     > '; maximaliteration );
   end;

   procedure computeandstore;
   begin
      enterwritefile (f, dataname);
      mapping;
      exitwritefile (f);
   end;

(* end: procedures of main program *)
(* ------------------------- main ------------------------- *)
   begin  (* main program *)
      hello;
      initialise;
      computeandstore;
   end.
```

You must be wondering why - in contrast to our own rules of style - everything is written in lower case in this program. The reasons are quite straightforward:

- The program is written in Berkeley Pascal, which is mainly found on UNIX systems with the operating system 4.3BSD. This Pascal compiler accepts only lower case (cf. hints in Chapter 12).
- It is an example to show that you can generate data on any machine in standard Pascal.

This program also runs on a large computer such as a VAX, SUN, or your PC. Only devotees of Turbo Pascal must undertake a small mofidication to the data procedures[2].

A further hint: in standard Pascal the data-type 'string' is not implemented, so that the programmer must work in a very involved manner with the type

```
    packed array [..] of char
```

(cf. procedure `readString`).

A few other new procedures include a drawing program, which reads the files thus generated and produces graphics from them. The data are opened for reading with `reset` instead of with `rewrite`, and `Store` is replaced by `ReadIn`. Otherwise `MandelbrotComputeAndTest` remains the same as before, except that distinguishing what must be drawn occurs within the procedure `Mapping`.

[2]Read the information on the `assign` command.

Program Fragment 11.5-2 (Integer coding to paint)

```
...
PROCEDURE ReadIn (VAR F : IntFile; VAR number : integer);
BEGIN
   read(F, number);
END;

PROCEDURE EnterReadFile (VAR F: IntFile; fileName
                        : STRING);
BEGIN
   reset (F, fileName);
END;

PROCEDURE ExitReadFile (VAR F: IntFile);
BEGIN
   close (F);
END;

PROCEDURE Mapping;
   VAR
      xRange, yRange, number : intger;
BEGIN
   yRange := 0;
   WHILE NOT EOF (F) DO
   BEGIN
      xRange := 0;
      ReadIn (F, number);
      WHILE NOT (number = 0) DO
      BEGIN
         IF (number = MaximalIteration)
            OR ((number < Bound) AND odd (number)) THEN
               SetPoint (xRange, yRange);
         ReadIn (F, number);
         xRange := xRange + 1;
      END;
      yRange := yRange + 1;
   END;
END;
...
PROCEDURE ComputeAndDisplay;
BEGIN
   EnterGraphics;
```

```
      EnterReadFile  (F,  fileName);
      Mapping;
      ExitReadFile  (F,  fileName);
      ExitGraphics;
   END;
```

With these two programs we can arrange that what takes a few hours to compute can be drawn in a few minutes. And not only that: if the drawing does not appeal to us, because the contour lines are too close or a detail goes awry, we can quickly produce further drawings from the same data. For this all we need do is change the central `IF` condition in `Mapping`. By using

```
    IF (number = MaximalIteration) THEN ...
```

we draw only the central figure of the Mandelbrot set; but with

```
    IF ((number>Bound) AND (number<maximalIteration)) THEN ...
```

we draw a thin neighbourhood of it.

This is also the place where we can bring colour into the picture. Depending on the value input for `number`, we can employ different colours from those available. You will find the necessary codes in your computer manual.

A short mental calculation reveals the disadvantages of integer coding. A standard screen with 320 × 200 = 64 000 pixels requires roughly 128 kilobytes on the disk, because in most Pascal implementations an integer number takes up two bytes of memory. And a larger picture, perhaps in DIN-A 4-format, can easily clog up a hard disk. But the special structure of our pictures gives us a way out. In many regions, nearby points are all coloured the same, having the same iteration depth. Many equal numbers in sequence can be combined into a pair of numbers, in which the first gives the length of the sequence, and the second the colour information. In Figure 11.5-1 you can see this: for example from 15, 15, 15, 15 we get the pair 4, 15. This method is called *run length encoding* and leads to a drastic reduction in storage requirement, to around 20%. That lets us speak of *data compression.*

Because very many disk movements are necessary for this transformation, to do the work on your PC we will use the silent pseudo-disk.[3] This is very quick and more considerate, especially for those family members who are not reminded of the music of the spheres by the singing of a disk-drive motor. And every 'freak' knows that 'it's quicker by RAMdisk'.

A suitable program `CompressToInt` is given here in standard Pascal (Berkeley Pascal). It uses the file procedures descibed above.

[3]Find out how to set up a RAMdisk for your computer.

Program 11.5-3

```
program compresstoint; (* standard pascal *)
  const
    stringlength = 8;
  type
    intfile = file of integer;
    string8 = packed array[1..stringlength] of char;
  var
    dataname : string8;
    in, out: intfile;
    quantity, colour, done : integer;

  procedure readin (var f : intfile; var number
                  : integer);
  begin
    read(f, number);
  end;

  procedure enterreadfile (var f: intfile; filename
                  : string8);
  begin
    reset (f, filename);
  end;

  procedure store (var f : intfile; number : integer);
  begin
    write (f, number);
  end;

  procedure enterwritefile (var f : intflie; filename
                  : string8);
  begin
    rewrite (f, filename);
  end;

  procedure exitreadfile (var f: intfile);
    (* if necessary close (f) *)
  begin end;

  procedure exitwritefile (var f: intfile);
    (* if necessary coose (f) *)
  begin end;
```

```
begin
  enterreadfile (in, 'intcoded');
  enterwritefile (out, 'rlintdat');
  while not eof (in) do
    begin
      quantity := 1;
      readin (in, colour);
      repeat
        readin (in, done);
        if (done <> 0) then
          if (done = colour) then
            quantity := quantity + 1
          else
            begin
              store (out, quantity);
              store (out, colour);
              colour := done;
              quantity := 1;
            end;
      until (done := 0) or eof (in);
      store (out, quantity);
      store (out, number);
      store (out, 0);
    end;
  exitreadfile (in); exitwritefile (out);
end.
```

In the first program, 11.5-1, you can use the procedure `readstring` to read in an arbitrary data name of length 10. The program requires an input procedure with the name `intcoded`. The resulting compressed data is always called `RLIntDat`.

Before the compression run, name your data file as `intcoded`. Note that standard Pascal requires certain conditions. Always naming the data the same way is, however, not too great a disadvantage. In that way you can automate the entire transformation process - which is useful if you have a time-sharing system, which can process your picture data overnight when the computer is free. You will find hints for this in Chapter 12.

We can also draw the compressed data `RLIntDat`[4]. For this data type too we show you, in Program Fragment 11.5-4, how to produce drawings. Because we no longer have to specify every individual point, there is a small gain in drawing speed. From 11.5-2 we must change only the procedure `Mapping`. Whether or not we wish

[4]`RLIntDat` stands for **Run Length Integer Data.**

to represent colour on the screen, we use `DrawLine` to draw a straight line on the screen of the appropriate length, or use `GotoPoint` to go to the appropriate place.

Program Fragment 11.5-4

```
PROCEDURE Mapping;
   VAR
      xRange, yRange, quantity, colour : integer;
BEGIN
   yRange := 0;
   WHILE NOT eof (F) DO
   BEGIN
      xRange := 0;
      GotoPoint (xRange, yRange);
      ReadIn (F, quantity);
      WHILE NOT (quantity = 0) DO
      BEGIN
         xRange := xRange + quantity;
         ReadIn (F, colour);
         IF (colour = MaximalIteration) OR
            ((colour < Bound) AND odd (colour)) THEN
               DrawLine (xRange -1, yRange)
         ELSE
            GotoPoint (xRange, yRange);
         ReadIn (F, quantity);
      END;
      yRange := yRange + 1;
   END;
END;
```

The third approach, known as the *run length encoded character* method, has advantages and disadvantages compared with the above (RLInt) method. It is more complicated, because it must be encoded before storage and decoded before drawing. However, it uses a genuine Pascal text file. We can change this with special programs (editors) and - what is often more important - transmit it by electronic mail.

In text data there are 256 possible symbols. Other symbols, such as accents, are not uniquely defined in ASCII, so we omit them as far as possible. In fact, we restrict ourselves to just 64 symbols,[5] namely the digits, capital and lower-case letters, and also '>' and '?'.

By combining any pair of symbols we can produce an integer code[6] between 0

[5]The method goes back to an idea of Dr Georg Heygster of the regional computer centre of the University of Bremen.

[6]In this way we can represent 4096 different colours. If you need even more, just combine three of the symbols together, to get 262 144 colours. Good enough?

and 4095 (= 64*64-1). This range of values should not be exceeded by the length information (maximal value is the row length) nor by the colour information (maximal value is the iteration depth).

Program 11.5-5 (Transfer int to char[7])

```
program transferinttochar;
   const stringlength = 8;
   type
      intfile = file of integer;
      charfile = text;
      string8 = packed array[1..stringlength] of char;
   var
      in : intfile;
      outtext : charfile;
      quantity, colour : integer;
      chartable : array[0..63] of char;
      dataname : string8;

   procedure readin (var f : intfile; var quantity
                        : integer);
   begin
      readin (f, quantity);
   end;

   procedure enterreadfile (var f : intfile; filename
                        : string8);
   begin
      reset (f, filename);
   end;

   procedure enterwritefile (var f : charfile; filename
                        : string8);
   begin
      rewrite (f, filename);
   end;

   procedure exitreadfile (var f : intfile);
   begin
      (* if necessary close (f) *)
   end;
```

[7]We repeat that in some Pascal compilers `read` and `write` must be replaced by `put` and `get`, and assignments must be specified for the 'window variable' `datavariable`.

```
procedure exitwritefile (var f : charfile);
begin
   (* if necessary close (f) *)
end;

procedure store (var outtext : charfile; number
                  : integer);
begin
   if number = 0 then writeln (outtext)
   else
   begin
      write (outtext, chartable[number div 64]);
      write (outtext, chartable[number mod 64]);
   end;
end;

procedure inittable;
   var i : integer;
begin
   for i = 0 to 63 do
   begin
      if i < 10 then
         chartable[i] := chr(ord('0') + i)
      else if i < 36 then
         chartable[i] := chr(ord('0') + i + 7)
      else if i < 62 then
         chartable[i] := chr(ord('0') + i + 13)
      else if i = 62 then
         chartable[i] := '>'
      else if i = 63 then
         chartable[i] := '?';
   end;
end;

begin
   inittable;
   enterreadfile (in, 'rlintdat');
   enterwritefile (outtext, 'rlchardat');
   while not eof (in) do
   begin
```

```
        readin (in, quantity);
        if quantity = 0 then
           store (outtext, 0)
        else
        begin
           store (outtext, quantity);
           readin (in, colour);
           store (outtext, colour);
        end;
      end;
      exitreadfile (in); exitwritefile (outtext);
    end.
```

The coding happens in the two combined programs, 11.5-5 and 11.5-6, by means of a *look-up table*. That is, a table that determines how to encode and decode the appropriate information. These tables are number fields, initialised once and for all at the start of the program. They can then be used for further computations in new runs.

The first of the two programs, 11.5-5, converts the run length encoded integers into characters, and is listed here. Note the line markers, which we insert with `writeln (outtext)` when a row of the picture has been completed. They make editing easier. We can imagine using them to help carry out particular changes to a picture.

In the final program of this section the drawing will be done. The table contains integer numbers, whose actual characters will be used as an index. The procedure `Mapping` conforms to the version of 11.5-4: only `InitTable` must be called at the beginning. The main changes occur in the procedure `ReadIn`.

Program Fragment 11.5-6 (Run length encoding char to paint)

```
...
TYPE
   CharFile : text;
VAR
   InText : CharFile;
   IntTable : ARRAY['0'..'z'] OF integer;
...
PROCEDURE InitTable;
   VAR
      ch : char;
BEGIN
   FOR ch := '0' TO 'z' DO
      BEGIN
         IF ch IN ['0'..'9'] THEN
            IntTable[ch] := ord (ch) - ord ('0')
```

```
          ELSE IF ch IN ['A'..'Z'] THEN
             IntTable[ch] := ord (ch) - ord ('0') - 7
          ELSE IF ch IN ['a'..'z'] THEN
             IntTable[ch] := ord (ch) - ord ('0') - 13
          ELSE IF ch = '>' THEN
             IntTable[ch] := 62
          ELSE IF ch = '?' THEN
             IntTable[ch] := 63
          ELSE
             IntTable[ch] := 0;
  END;
END;

PROCEDURE ReadIn (VAR InText : CharFile; VAR number
                  : integer);
  VAR ch1, ch2 : char;
BEGIN
  IF eoln (InText) THEN

  BEGIN
     readln (InText);
     number := 0;
  END
  ELSE
  BEGIN
     read (InText, ch1);
     read (InText, ch2);
     number := (64*IntTable[ch1]+IntTable[ch2]);
  END;
END;

PROCEDURE Mapping;
  VAR xRange, yRange, quantity, colour : integer;
BEGIN
  yRange := 0;
  WHILE NOT eof (InText) DO
  BEGIN
     xRange := 0;
     GotoPoint (xRange, yRange);
     ReadIn (InText, quantity);
     WHILE NOT (quantity = 0) DO
```

```
        BEGIN
           xRange := xRange + quantity;
           ReadIn (InText, colour);
           IF (colour >= MaximalIteration) OR
              ((colour < Bound) AND odd (colour)) THEN
                 DrawLine (xRange - 1, yRange)
           ELSE
              GotoPoint (xRange, yRange);
           ReadIn (InText, quantity);
        END;
        yRange := yRange + 1;
     END;
  END;
```

The character files can take up the same space as the compressed integer data - in the above example around 28 kilobytes. On some computers, the storage of an integer number can take up more space than that of a character, implying some saving.

If you transmit the character data using a 300 baud modem, it can take longer than 15 minutes, so you should only do this on local phone lines. We now explain one way to reduce the telephone bill. The text files produced with Program 11.5-5 contain a very variable frequency of individual characters. For example there are very few zeros. In this case the *Huffman method* of text compression (Streichert 1987) can lead to a saving in space of around 50%. The same applies to the telephone bill!

11.6 A Picture Takes a Trip

We can make use of the possibilities described in the previous chapter for screen- and machine-independent generation of data, when we want to send our pictures to a like-minded 'Gingerbread Man investigator'. For this purpose there are basically two methods:

- the normal mail service
- electronic mail ('e-mail').

There is no probem when two experimenters possess the same make of computer. Then it is quite straightforward to exchange programs and data on floppy disks using the normal mail. But we often encounter a situation where one of them has, say, a Macintosh or Atari, and the other has an MS-DOS machine.

In this case one possibility is to connect both computers by cable over the V24 interface, and to transmit programs and data using special software. Many such *file transfer programs* are available for many different computers. We recommend the popular 'Kermit' program, which is available for most machines.[8]

A much simpler, but more expensive, possibility is to send your programs and data

[8]Read the hints in §12.7 and the instructions in the Kermit documentation for your computer, to see how to install this.

by telephone, locally or worldwide.

If you live in the same town all you need next to your computer is an acoustic coupler or a modem. In this way you can transmit Pascal programs and picture data between different computers without moving your computer from the desk. And you do not have to mess about with the pin-connections of the V24 interface on different computers, because when you bought the modem for your machine you naturally also bought the appropriate connecting cable.

It is well known how to transmit data over the local telephone network by modem. Not so well known is the fact that worldwide communication networks exist, which can be used to send mail. Of course they are not free.

Basically, by paying the appropriate user fees, anyone can send a letter to - for instance - the USA. Standard communications networks, which can be used from Europe, are CompuServe and Delphi. Of course, the user who wishes to send a picture or a Pascal program must have access to the appropriate network. Another possibility is to send mail over the worldwide academic research network. Only universities and other research institutions can do this.

How do we send e-mail? Easy!

With an acoustic coupler or modem we call the data transmission service of the Post Office and dial the number for the computer to which we wish to send the mail.

First, you must call the DATEX node computer of the German Federal Post[9], and give your NUI[10].

```
DATEX-P: 44 4000 99132
nui dxyz1234
DATEX-P: Password
XXXXXX

DATEX-P: Usercode dxyz1234 active
set 2:0, 3:0, 4:4, 126:0
```

After setting the PAD parameters (so that the input is not echoed and the password remains invisible) then the telephone number of the computer with which we wish to communicate is entered (invisibly).

```
(001) (n, Usercode dxyz1234, packet-length: 128)

RZ Unix system 4.2 BSD
login: kalle
```

[9]*Translator's note*: This is the procedure in Germany. In other countries it is very similar, but the names of the services and their telephone numbers are different.

[10]Network User Identification. This consists of two parts: your visible identification and a secret password.

```
Password:
Last login Sat Jul 11 18:09:03 on ttyh3
4.2 BSD UNIX Release 3.07 #3 (root§FBinf) Wed Apr 29 18:12:35
EET 1987
You have mail.
TERM = (vt100)
From ABC007§PORTLAND.BITNET Sat Jul 11 18:53:31 1987
From ABC007§PORTLAND.BITNET Sun Jul 12 01:02:24 1987
From ABC007§PORTLAND.BITNET Sun Jul 12 07:14:31 1987
From ABC007§PORTLAND.BITNET Mon Jul 13 16:10:00 1987
From ABC007§PORTLAND.BITNET Tue Jul 14 03:38:24 1987
kalle§FBinf 1) mail
Mail version 2.18 5/19/83.  Type ? for help.
''/use/spool/mail/kalle'': 5 messages 5 new
>N   1  ABC007§PORTLAND.BITNET  Sat  Jul  11  18:53:31  15/534
''Saturday''
From  ABC007§PORTLAND.BITNET  Sun  Jul  12  01:02:24  31/1324
''request''
From ABC007§PORTLAND.BITNET Sun Jul 12 07:14:31 47/2548 ''FT''
From ABC007§PORTLAND.BITNET Mon Jul 13 16:10:00 22/807 ''Auto''
From ABC007§PORTLAND.BITNET Tue Jul 14 03:38:24 32/1362
&2 Message 2:
From ABC007§PORTLAND.BITNET Sun Jul 12 01:02:24 1987
Received: by FBinf.UUCP; Sun, 12 Jul 87 01:02:19 +0200; AA04029
Message-Id: <8707112302.AA04029§FBinf.UUCP>
Received:  by FBinf.BITNET from portland.bitnet(mailer) with bsmtp
Received:  by PORTLAND (Mailer X1.24) id 6622; Sat, 11 Jul 87
19:01:54 EDT
Subject: request
From: ABC007§PORTLAND.BITNET
To: KALLE§RZA01.BITNET
Date: Sat, 11 Jul 87 19:00:08 EDT
Status: R

Dear Karl-Heinz,
it is atrociously hot here this summer.  Instead of sitting on the
deck with a six-pack, i've got to spend the next few weeks putting
together a small survey of information technology.  It has to be
at an elementary level.  In particular I want to include aspects
that are relatively new - such as the connection between computer
graphics and experimental mathematics.
```

```
The simplest example would be a Feigenbaum diagram.
Please send me the Pascal program to gienerate the picture - and,
for safety, a copy of the picture itself - as soon as possible.
By e-mail, the ordinary mail is dreadful as usual and it takes
about 12 days.
Apart from that, there;s nothing much happening here.
When are you coming over next?
Best wishes
Otmar
   &q
Held 5 messages in usr/spool/mail/kalle
0.9u 1.2s 4:54 0% 8t+4d=13<18 19i+28o 38f+110r 0w
kalle§FBinf 2) logout
```

After reading all the news the connection is broken, and we use `logout` to leave the Unix system. The features of Unix and the details of e-mail will not be given here, because they can differ from one computer to the next. For more information, read your manual.

Three days later...

```
DATEX-P: 44 4000 49632
nui dxyz1234
DATEX-P: Password
XXXXXX

DATEX-P: Usercode dxyz1234 active
set 2:0, 3:0, 4:4, 126:0
(001) (n, Usercode dxyz1234, packet-length: 128)

RZ Unix system 4.2 BSD

login: kalle
Password:
Last login Tue Jul 14 07:05:47 on ttyh3
4.2 BSD UNIX Release 3.07 #3 root§FBinf) Wed Apr 29 18:12:35
EET 1987
You have mail.
TERM = (vt100)
kalle§FBinf 2) mail ABC007§Portland.bitnet
Subject Pascalprogram for Feigenbaum
Dear Otmar,
many thanks for your last letter.   Unfortunately I didn't have
```

```
time to reply until today.
Here is the required Turbopascal program for the Mac.
As usual start with the following inputs:
Left = 1.8
Right = 3.0
Bottom = 0
Top = 1.5
------------------------------cut here --------------------
PROGRAM EmptyApplicationShell; (* TurboPascal on Macintosh *)
   USES MemTypes, QuickDraw;
   CONST
      Xscreen = 320; (* e.g. 320 pixels in x-direction *)
      Yscreen = 200; (* e.g. 200 pixels in y-direction *)
         VAR
            PictureName : string;
            Left, Right, Top, Bottom : real;
         (* include other global variables here *)
```

And so on... we will not give the program at full length here: see §12.4 for the complete listing.

```
(* --------------------MAIN-----------------*)

BEGIN (*Main Program *)
   Hello;
   Initialise;
   ComputeAndDisplay;
   Goodbye;
END

That's it!   Good luck with the survey.
Best wishes Karl-Heinz.

EOT
ABC007$Portland.Bitnet... Connecting to portland.bitnet...
ABC007$Portland.Bitnet... Sent
File 2119 Enqueued on Link RZB23
Sent file 2119 on link RZB23 to PORTLAND ABC007
From RZB23: MDTNCM147I SENT FILE 1150 (2119)
ON LINK RZ0ZIB21 TO PORTLAND IXS2
From RZSTU1: MDTVMB147I SENT FILE 1841 (2119)
```

```
ON LINK DEARN TO PORTLAND ABC007
...
```

You can see on the screen how the letter is sent on from station to station. In a few minutes it has reached its destination. A few minutes later the picture is on the way too...

```
kalle§FBinf ) mail ABC007§Portland.Bitnet
Subject: Picture
Dear Otmar,
now I'm sending you the picture you wanted.
(This file must be converted with BinHex 4.0)

:#dpdE@&bFb''#D@N!&19%G338j8!*!%(L!!N!3EE!#3!`2rN!MGrhIrhIphrpeh
hhAIGGpehUP@U9DT9UP99reAr9Ip9rkU3#11GZhIZhEYhL*!)x6!$'pM!$)f!%!)
J!£K!''2q)N!2rL*!$ri#3!rm)N!1!!*!(J%!J!!)%#!##4$P%JJ'3!rKd)NH2&b
a9D''!3&8+''!3J8)L3''!8#[`#r[1#3''!#3#)!!#!#!!!J!L!!L!)J!)J#))SJ
))US!UJ#U!+S!r`$r!2m!r`!4)N5)%5*%L2m!N!2r!*!$!3)%#''!J3)#U!)!!L!
!!2q!N!F)(#,''J!%#'')J8)N')!+S!3+!!!!3+!!!$K%J`$!)''!B#!36i)#''6
!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N
!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N
!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N
!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N
```

We do not list the entire encoded picture data...

```
!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N
!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N
!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!Z3#j!,N!'H!!!!:

------------------
As usual, please cut out the bit between the two colons.
Best wishes, Karl-Heinz
```

A day later, back comes the answer.

```
Message 3:
From ABC007§PORTLAND.BITNET Thu Jul 16 03:19:30 1987
Received: by FBinf.UUCP; Thu 16 Jul 87 03:19:27 +0200; AA04597
Message-Id: <8707160119.AA04597§FBinf.UUCP>
Received:  by FBinf.BITNET from portland.bitnet(mailer) with bsmtp
Received:  by PORTLAND (Mailer X1.24) id 5914; Wed, 15 Jul 87
21:11:18 EDT
```

```
Subject: Picture?Pascalprogram
From: ABC007$PORTLAND.BITNET
To: KALLE$RZA01.BITNET
Date: Wed, 15 Jul 87 21:09:05 EDT
Status: R

Dear Kalle,
the picture has arrived perfectly.  I have converted it without
any problems using binhex.  You might try using packit next time.
The text file was larger than 10K - the resulting paint file was
only 7K!
More next time,
Otmar.
```

A few explanations to end with.

We have printed the dialogue in full to show you how

- to write your programs in such a form that they can be transferred between computers of different types and different disk formats;
- to consider how to set up communications within a single town or internationally;
- to think about the problem of data compression.

Our pen-pal Otmar has already spoken about this last problem. When two computers communicate directly, binary files, that is, pictures, can be transmitted directly. You can do this with Kermit. But it only makes sense to do it when both computers are of the same make, otherwise the picture cannot be displayed.

Between different computers, we can use the intermediate format set out in §11.5, which changes a picture to a text file. If a picture passes between several computers *en route* to its destination, and if the sending and receiving computers are of the same type, we recommend the use of programs that convert a binary file into a hex file (BinHex program). Such programs are available on any computer. There also exist programs that can take a hex file (e.g. generated from a picture) and compress it still further, to make the text file smaller ('Compress', 'PackIt', etc.).

A very well-known method of data compression is the so-called Huffman method. It achieves an astonishing degree of compression. Of course the person with whom you are communicating must also have such a program. Maybe this is an idea for a joint programming project. Consult the appropriate technical literature, in order to implement the algorithm: Huffmann (1952), Mann (1987), Streichert (1987).

We end the problem of picture transmission here; but the problem of different computers will continue to concern us in Chapter 12.