Part II: Programming concepts: beyond notation

Thoughts on the role of programming notations

A programming language is the main interface between a programmer and the physical machine, and a novice programmer will tend to identify "programming" with "programming in the particular language she has learned". The realization that there is much to programming "beyond notation" (i.e. principles that transcend any one language) is a big step forward in a programmer's development.

Part II aims to help the reader take this step forward. We present examples that are best understood by focusing on abstract principles of algorithm design, and only later do we grope for suitable notations to turn this principle into an algorithm expressed in sufficient detail to become executable. In keeping with our predilection for graphic communication, the first informal expression of an algorithmic idea is often pictorial. We show by example how such representations, although they may be incomplete, can be turned into programs in a formal notation.

The literature on programming and languages. There are many books that present principles of programming and of programming languages from a higher level of abstraction. The principles highlighted differ from author to author, ranging from intuitive understanding to complete formality. The following textbooks provide an excellent sample from the broad spectrum of approaches: [ASS 84], [ASU 86], [Ben 82], [Ben 85], [Ben 88], [Dij 76], [DF 88], [Gri 81], and [Mey 90].

4. Algorithms and programs as literature: substance and form

Learning objectives:

- programming in the large versus programming in the small
- large flat programs versus small deep programs
- programs as literature
- fractal pictures: snowflakes and Hilbert's space-filling curve
- recursive definition of fractals by production or rewrite rules
- Pascal and programming notations

Programming in the large versus programming in the small

In studying and discussing the art of programming it is useful to distinguish between large programs and small programs, since these two types impose fundamentally different demands on the programmer.

Programming in the large

Large programs (e.g. operating systems, database systems, compilers, application packages) tax our *organizational ability*. The most important issues to be dealt with include requirements analysis, functional specification, compatibility with other systems, how to break a large program into modules of manageable size, documentation, adaptability to new systems and new requirements, how to organize the team of programmers, and how to test the software. These issues are the staple of software engineering. When compared to the daunting managerial and design challenges, the task of actual coding is relatively simple. Large programs are often *flat:* Most of the listing consists of comments, interface specifications, definitions, declarations, initializations, and a lot of code that is executed only rarely. Although the function of any single page of source code may be rather trivial when considered by itself, it is difficult to understand the entire program, as you need a lot of information to understand how this page relates to the whole. The classic book on programming in the large is [Bro 75].

Programming in the small

Small programs, of the kind discussed in this book, challenge our technical know-how and inventiveness. *Algorithmic issues* dominate the programmer's thinking: Among several algorithms that all solve the same problem, which is the most efficient under the given circumstances? How much time and space does it take? What data structures do we use? In contrast to large programs, small programs are usually *deep*, consisting of short, compact code many of whose statements are executed very often. Understanding a small program may also be difficult, at least initially, since the chain of thought is often subtle. Once you understand it thoroughly, you can reproduce it at any time with much less effort than was first required. Mastery of interesting small programs is the

best way to get started in computer science. We encourage the reader to work out all the details of the examples we present.

This book is concerned only with programming in the small. This decision determines our choice of topics to be presented, our style of presentation, and the notation we use to express programs, explanations, and proofs, and heavily influences our comments on techniques of programming. Our style of presentation appeals to the reader's intuition more than to formal rigor. We aim at highlighting the key idea of any argument that we make rather than belaboring the details. We take the liberty of using a free notation that suits the purpose of any specific argument we wish to make, trusting that the reader understands our small programs so well that he can translate them into the programming language of his choice. In a nut shell, we emphasize substance over form.

The purpose of Part II is to help engender a fluency in using different notations. We provide yet other examples of unconventional notations that match the nature of the problem they are intended to describe, and we show how to translate them into Pascal-like programs. Since much of the difference between programming languages is merely syntactic, we include two chapters that cover the basics of syntax and syntax analysis. These topics are important in their own right; we present them early in the hope that they will help the student see through differences of notation that are merely "syntactic sugar".

Documentation versus literature: is it meant to be read?

It is instructive to distinguish two types of written materials, and two corresponding types of writing tasks: documents and literature. *Documents* are constrained by requirements of many kinds, are read when a specific need arises (rarely for pleasure), and their quality is judged by criteria such as formality, conformity to a standard, completeness, accuracy, and consistency. *Literature* is a form of art free from conventions, read for education or entertainment, and its quality is judged by aesthetic criteria much harder to enumerate than the ones above. The touchstone is the question: Is it meant to be read? If the answer is "only if necessary", then it's a document, not literature.

As the name implies, the documentation of large programs is a typical document-writing chore. Much has been written in software engineering about documentation, a topic whose importance grows with the size and complexity of the system to be documented. We hold that small programs are not documented, they are explained. As such, they are literature, or ought to be. The idea of programs as literature is widely held (see, e.g. [Knu 84]). The key idea is that an algorithm or program is part of the text and melts into the text in the same way as a paragraph, a formula, or a picture does. There are also formal notations and systems designed to support a style of programming that integrates text and code to form a package that is both readable for humans and executable by machines [Knu 83].

Whatever notation is used for literate programming, it has to describe all phases of a program's evolution, from idea to specification to algorithm to program. Details of a good program cannot be understood, or at least not appreciated, without an awareness of the grand design that guided the programmer. Whereas details are usually well expressed in some formal notation, grand designs are not. For this reason we renounce formality and attempt to convey ideas in whatever notation suits our purpose of insightful explanation. Let us illustrate this philosophy with some examples.

A snowflake

Fractal pictures are intuitively characterized by the requirement that any part of the picture, of any size, when sufficiently magnified, looks like the whole picture. Two pieces of information are required to define a specific fractal:

1. A picture primitive that serves as a building-block: Many copies of this primitive, scaled to many different sizes, are composed to generate the picture.
2. A recursive rule that defines the relative position of the primitives of different size.

A picture primitive is surely best defined by a drawing, and the manner of composing primitives in space again calls for a pictorial representation, perhaps augmented by a verbal explanation. In this style we define the fractal 'Snowflake' by the following *production* rule, which we read as follows: A line segment, as shown on the left-hand side, must be replaced by a polyline, a chain of four shorter segments, as shown at the right-hand side (Exhibit 4.1). We start with an initial configuration (the zero-generation) consisting of a single segment (Exhibit 4.2). If we apply the production rule just once to every segment of the current generation, we obtain successively a first, second, and third generation, as shown in Exhibit 4.3. Further generations quickly exhaust the resolution of a graphics screen or the printed page, so we stop drawing them. The curve obtained as the limit when this process is continued indefinitely is a *fractal*. Although we cannot draw it exactly, one can study it as a mathematical object and prove theorems about it.

Exhibit 4.1: Production for replacing a straight-line segment by a polyline

Exhibit 4.2: The simplest initial configuration

Exhibit 4.3: The first three generations

The production rule drawn above is the essence of this fractal, and of the sequence of pictures that lead up to it. The initial configuration, on the other hand, is quite arbitrary: If we had started with a regular hexagon, rather than a single line segment, the pictures obtained would really have lived up to their name, snowflake. Any other initial configuration still generates curves with the unmistakable pattern of snowflakes, as the reader is encouraged to verify.

After having familiarized ourselves with the objects described, let us turn our attention to the method of description and raise three questions about the formality and executability of such notations.

1. Is our notation sufficiently formal to serve as a program for a computer to draw the family of generations of snowflakes? Certainly not, as we stated certain rules in colloquial language and left others completely unsaid, implying them only by sample drawings. As an example of the latter, consider the question: If a

segment is to be replaced by a "plain with a mountain in the center", on which side of the segment should the peak point? The drawings above suggest that all peaks stick out on the same side of the curve, the outside.

2. Could our method of description be extended and formalized to serve as a programming language for fractals? Of course. As an example, the production shown in Exhibit 4.4 specifies the side on which the peak is to point. Every segment now has a + side and a – side. The production above states that the new peak is to grow over the + side of the original segment and specifies the + sides and – sides of each of the four new segments. For every other aspect that our description may have left unspecified, such as placement on the screen, some notation could readily be designed to specify every detail with complete rigor. In “Syntax” and “Syntax analysis” we introduce some of the basic techniques for designing and using formal notations.

Exhibit 4.4: Refining the description to specify a "left-right" orientation.

3. Should we formalize this method of description and turn it into a machine-executable notation? It depends on the purpose for which we plan to use it. Often in this book we present just one or a few examples that share a common design. Our goal is for the reader to understand these few examples, not to practice the design of artificial programming languages. To avoid being sidetracked by a pedantic insistence on rigorous notation, with its inevitable overhead of introducing formalisms needed to define all details, we prefer to stop when we have given enough information for an attentive reader to grasp the main idea of each example.

Hilbert's space-filling curve

Space-filling curves have been an object of mathematical curiosity since the nineteenth century, as they can be used to prove that the cardinality of an interval, considered as a set of points, equals the cardinality of a square (or any other finite two-dimensional region). The term *space-filling* describes the surprising fact that such a curve visits every point within a square. In mathematics, space-filling curves are constructed as the limit to which an infinite sequence of curves C_i converges. On a discretized plane, such as a raster-scanned screen, no limiting process is needed, and typically one of the first dozen curves in the sequence already paints every pixel, so the term *space-filling* is quickly seen to be appropriate.

Let us illustrate this phenomenon using Hilbert's space-filling curve (David Hilbert, 1862–1943), whose first six approximations are shown in Exhibit 4.5. As the pictures suggest, Hilbert curves are best-described recursively, but the composition rule is more complicated than the one for snowflakes. We propose the two productions shown in Exhibit 4.6 to capture the essence of Hilbert (and similar) curves. This pictorial program requires explanation, but we hope the reader who has once understood it will find this notation useful for inventing fractals of her own. As always, a production is read: "To obtain an instance of the left-handside, get instances of all the things listed on the right-handside", or equivalently, "to do the task specified by the left-hand side, do all the tasks listed on the right-hand side".

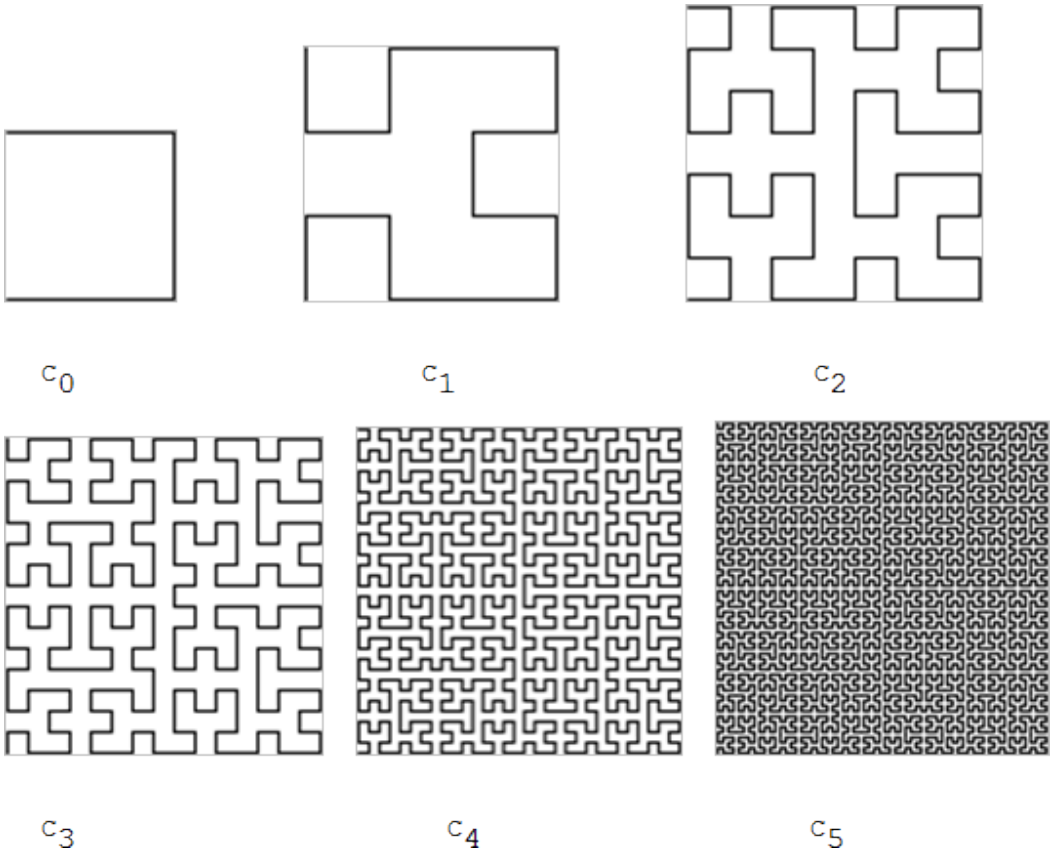

Exhibit 4.5: Six generations of the family of Hilbert curves

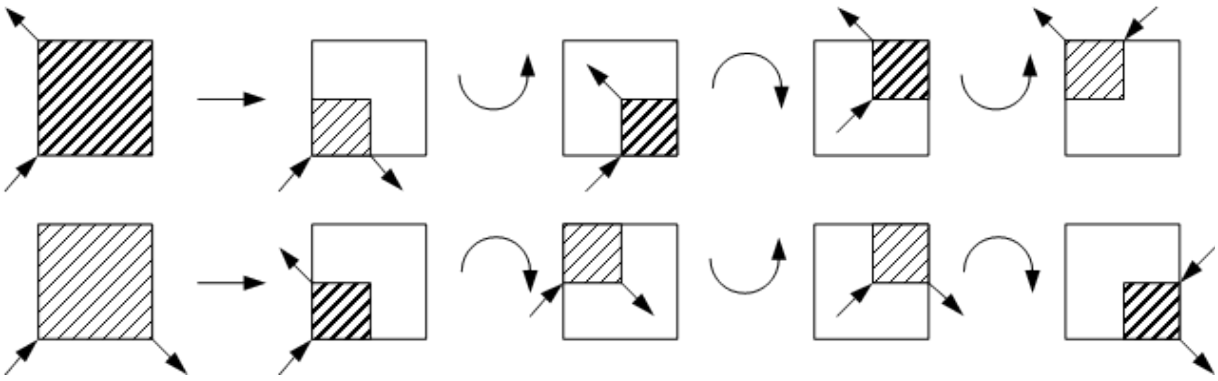

Exhibit 4.6: Productions for painting a square in terms of its quadrants

The left-hand side of the first production stands for the task: paint a square of given size, assuming that you enter at the lower left corner facing in the direction indicated by the arrow and must leave in the upper left corner, again facing in the direction indicated by that arrow. We assume turtle graphics primitives, where the state of the brush is given by a position and a direction. The hatching indicates the area to be painted. It lies to the right of the line that connects entry and exit corners, which we read as "paint with your right hand", and the hatching is in thick strokes. The left-hand side of the second production is similar: Paint a square "with your left hand" (hatching is in thin strokes), entering and exiting as indicated by the arrows.

The right-hand sides of the productions are now easily explained. They say that in order to paint a square you must paint each of its quadrants, in the order indicated. They give explicit instructions on where to enter and exit,

what direction to face, and whether you are painting with your right or left hand. The last detail is to make sure that when the brush exits from one quadrant it gets into the correct state for entering the next. This requires the brush to turn by 90°, either left or right, as the curved arrows in the pictures indicate. In the continuous plane we imagine the brush to "turn on its heels", whereas on a discrete grid it also moves to the first grid point of the adjacent quadrant.

These productions omit any rule for termination, thus simulating the limiting process of true space-filling curves. To draw anything on the screen we need to add some termination rules that specify two things: (1) when to invoke the termination rule (e.g. at some fixed depth of recursion), and (2) how to paint the square that invokes the termination rule (e.g. paint it all black). As was the case with snowflakes and with all fractals, the primitive pictures are much less important than the composition rule, so we omit it.

The following program implements a specific version of the two pictorial productions shown above. The procedure 'Walk' implements the curved arrows in the productions: the brush turns by 'halfTurn', takes a step of length s, and turns again by 'halfTurn'. The parameter 'halfTurn' is introduced to show the effect of cumulative small errors in recursive procedures. 'halfTurn = 45' causes the brush to make right-angle turns and yields Hilbert curves. The reader is encouraged to experiment with 'halfTurn = 43, 44, 46, 47', and other values.

```
program PaintAndWalk;
const  pi = 3.14159;  s = 3;  { step size of walk }
var    turtleHeading: real;  { counterclockwise, radians }
       halfTurn, depth: integer;  { recursive depth of painting }

procedure TurtleTurn(angle: real);
{ turn the turtle angle degrees counterclockwise }
begin  { angle is converted to radian before adding }
  turtleHeading := turtleHeading + angle · pi / 180.0
end;  { TurtleTurn }

procedure TurtleLine(dist: real);
{ draws a straight line, dist units long }
begin
  Line(round(dist · cos(turtleHeading)), round(-dist·sin(turtle
Heading)))

end;  { TurtleLine }

procedure Walk (halfTurn: integer);
begin  TurtleTurn(halfTurn);  TurtleLine(s);  TurtleTurn(halfTurn)
end;

procedure Qpaint (level: integer; halfTurn: integer);
begin
  if  level = 0  then
    TurtleTurn(2 · halfTurn)
  else  begin
    Qpaint(level - 1, -halfTurn);
    Walk(halfTurn);
    Qpaint(level - 1, halfTurn);
    Walk(-halfTurn);
    Qpaint(level - 1, halfTurn);
    Walk(halfTurn);
    Qpaint(level - 1, -halfTurn)
  end
end;  { Qpaint }

begin  { PaintAndWalk }
```

```
      ShowText;  ShowDrawing;
      MoveTo(100, 100);  turtleHeading := 0;  { initialize turtle
state }
      WriteLn('Enter halfTurn 0 .. 359  (45 for Hilbert curves): ');
      ReadLn(halfTurn);
      TurtleTurn(-halfTurn);  { init turtle turning angle }
      Write('Enter depth 1 .. 6: ');  ReadLn(depth);
      Qpaint(depth, halfTurn)
   end.  { PaintAndWalk }
```

As a summary of this discourse on notation, we point to the fact that an executable program necessarily has to specify many details that are irrelevant from the point of view of human understanding. This book assumes that the reader has learned the basic steps of programming, of thinking up such details, and being able to express them formally in a programming language. Compare the verbosity of the one-page program above with the clarity and conciseness of the two pictorial productions above. The latter state the essentials of the recursive construction, and no more, in a manner that a human can understand "at a glance". We aim our notation to appeal to a human mind, not necessarily to a computer, and choose our notation accordingly.

Pascal and its dialects: lingua franca of computer science

Lingua franca (1619):

1. A common language that consists of Italian mixed with French, Spanish, Greek and Arabic and is spoken in Mediterranean ports
2. Any of various languages used as common or commercial tongues among peoples of diverse speech
3. Something resembling a common language

(From *Webster's Collegiate Dictionary)*

Pascal as representative of today's programming languages

The definition above fits Pascal well: In the mainstream of the development of programming languages for a couple of decades, Pascal embodies, in a simple design, some of the most important language features that became commonly accepted in the 1970s. This simplicity, combined with Pascal's preference for language features that are now well understood, makes Pascal a widely understood programming notation. A few highlights in the development of programming languages may explain how Pascal got to be a lingua franca of computer science.

Fortran emerged in 1954 as the first high-level programming language to gain acceptance and became *the* programming language of the 1950s and early 1960s. Its appearance generated great activity in language design, and suddenly, around 1960, dozens of programming languages emerged. Three among these, Algol 60, COBOL, and Lisp, became milestones in the development of programming languages, each in its own way. Whereas COBOL became the most widely used language of the 1960s and 1970s, and Lisp perhaps the most innovative, Algol 60 became the most influential in several respects: it set new standards of rigor for the definition and description of a language, it pioneered hierarchical block structure as the major technique for organizing large programs, and through these major technical contributions became the first of a family of mainstream programming languages that includes PL/1, Algol 68, Pascal, Modula-2, and Ada.

The decade of the 1960s remained one of great ferment and productivity in the field of programming languages. PL/1 and Algol 68, two ambitious projects that attempted to integrate many recent advances in programming language technology and theory, captured the lion's share of attention for several years. Pascal, a much smaller

project and language designed by Niklaus Wirth during the 1960s, ended up eclipsing both of these major efforts. Pascal took the best of Algol 60, in streamlined form, and added just one major extension, the then novel type definitions [Hoa 72]. This lightweight edifice made it possible to implement efficient Pascal compilers on the microcomputers that mushroomed during the mid 1970s (e.g. UCSD Pascal), which opened the doors to universities and high schools. Thus Pascal became the programming language most widely used in introductory computer science education, and every computer science student must be fluent in it.

Because Pascal is so widely understood, we base our programming notation on it but do not adhere to it slavishly. Pascal is more than 20 years old, and many of its key ideas are 30 years old. With today's insights into programming languages, many details would probably be chosen differently. Indeed, there are many "dialects" of Pascal, which typically extend the standard defined in 1969 [Wir 71] in different directions. One extension relevant for a publication language is that with today's hardware that supports large character sets and many different fonts and styles, a greater variety of symbols can be used to make the source more readable. The following examples introduce some of the conventions that we use often.

"Syntactic sugar": the look of programming notations

Pascal statements lack an explicit terminator. This makes the frequent use of begin-end brackets necessary, as in the following program fragment, which implements the insertion sort algorithm (see chapter 17 and the section "Simple sorting algorithms that work in time"); –∞ denotes a constant ≤ any key value:

```
A[0] := -∞;
for i := 2 to n do  begin
   j := i;
   while  A[j] < A[j - 1]  do
      begin  t := A[j];  A[j] := A[j - 1];  A[j - 1] := t;  j := j - 1  end;
end;
```

We aim at brevity and readability but wish to retain the flavor of Pascal to the extent that any new notation we introduce can be translated routinely into standard Pascal. Thus we write the statements above as follows:

```
A[0] := -∞;
for i := 2 to n do  begin
   j := i;  { comments appear in italics }
   while  A[j] < A[j - 1]  do  { A[j] :=: A[j - 1];  j := j - 1 }
   { braces serve as general-purpose brackets, including begin-end }
   { :=: denotes the exchange operator }
end;
```

Borrowing heavily from standard mathematical notation, we use conventional mathematical signs to denote operators whose Pascal designation was constrained by the small character sets typical of the early days, such as:

```
  ≠ ≤ ≥ ≠ ¬ ∧ ∨ ∈ ∉ ∩ ∪ \ |x|  instead of
<> <= >= <> not and or in not in · + - abs(x) respectively

We also use signs that may have no direct counterpart in Pascal, such
as:
```

⊃ ⊇ ⊄ ⊂ ⊆	Set-theoretic relations
∞	Infinity, often used for a "sentinel" (i.e. a number larger than all numbers to be processed in a given

	application)
±	Plus-or-minus, used to define an interval [of uncertainty]
ΣΠ	Sum and product
⌈x⌉	Ceiling of a real number x (i.e. the smallest integer ≥ x)
⌊x⌋	Floor of a real number x (i.e. the largest integer ≤ x)
√	Square root
log	Logarithm to the base 2
ln	Natural logarithm, to the base e
iff	If and only if

Although we may take a cavalier attitude toward notational differences, and readily use concise notations such as ∧ ∨ for the more verbose 'and', 'or', we will try to remind readers explicitly about our assumptions when there is a question about semantics. As an example, we assume that the boolean operators ∧ and ∨ are *conditional*, also called 'cand' and 'cor': An expression containing these operators is evaluated from left to right, and the evaluation stops as soon as the result is known. In the expression x ∧ y, for example, x is evaluated first. If x evaluates to 'false', the entire expression is 'false' without y ever being evaluated. This convention makes it possible to leave y undefined when x is 'false'. Only if x evaluates to 'true' do we proceed to evaluate y. An analogous convention applies to x ∨ y.

Program structure

Whereas the concise notations introduced above to denote operators can be translated almost one-to-one into a single line of standard Pascal, we also introduce a few extensions that may affect the program structure. In our view these changes make programs more elegant and easier to understand. Borrowing from many modern languages, we introduce a 'return()' statement to exit from procedures and functions and to return the value computed by a function.

Example

```
    function gcd(u, v: integer): integer;
    { computes the greatest common divisor (gcd) of u and v }
    begin  if  v = 0  then  return(u)  else  return(gcd(v, u mod v))
end;
```

In this example, 'return()' merely replaces the Pascal assignments 'gcd := u' and 'gcd := gcd(v, u mod v)'. The latter in particular illustrates how 'return()' avoids a notational blemish in Pascal: On the left of the second assignment, 'gcd' denotes a variable, on the right a function. 'Return()' also has the more drastic consequence that it causes control to exit from the surrounding procedure or function as soon as it is executed. Without entering into a controversy over the general advantages and disadvantages of this "flow of control" mechanism, let us present one example, typical of many search procedures, where 'return()' greatly simplifies coding. The point is that a search

routine terminates in one of (at least) two different ways: successfully, by having found the item in question, or unsuccessfully, because of a number of reasons (the item is not present, and some index is about to fall outside the range of a table; we cannot insert an item because the table is full, or we cannot pop a stack because it is empty, etc.). For the sake of efficiency as well as readability we prefer to exit from the routine as soon as a case has been identified and dealt with, as the following example from "Address computation:" illustrates:

```
function insert-into-hash-table(x: key): addr;
var  a: addr;
begin
   a := h(x);  { locate the home address of the item x to be
inserted }
   while  T[a] ≠ empty  do  begin
      { skipping over cells that are already occupied }
      if  T[a] = x  then  return(a);  { x is already present; return
its address }
      a := (a + 1) mod m    { keep searching at the next address }
   end;
   { we've found an empty cell; see if there is room for x to be
inserted }
   if  n < m - 1  then  {  n := n + 1;  T[a] := x }  else  err-
msg('table is full');
   return(a)  { return the address where x was inserted }
end;
```

This code can only be appreciated by comparing it with alternatives that avoid the use of 'return()'. We encourage readers to try their hands at this challenge. Notice the three different ways this procedure can terminate: (1) no need to insert x because x is already in the table, (2) impossible to insert x because the table is full, and (3) the normal case when x is inserted. Standard Pascal incorporates no facilities for "exception handling" (e.g. to cover the first two cases that should occur only rarely) and forces all three outcomes to exit the procedure at its textual end.

Let us just mention a few other liberties that we may take. Whereas Pascal limits results of functions to certain simple types, we will let them be of *any* type: in particular, structured types, such as records and arrays. Rather than nesting if-then-else statements in order to discriminate among more than two mutually exclusive cases, we use the "flat" and more legible control structure:

```
if B1 then S1 elsif B2 then S2 elsif … else Sn ;
```

Our sample programs do not return dynamically allocated storage explicitly. They rely on a memory management system that retrieves free storage through "garbage collection". Many implementations of Pascal avoid garbage collection and instead provide a procedure 'dispose(...)' for the programmer to explicitly return unneeded cells. If you work with such a version of Pascal and write list-processing programs that use significant amounts of memory, you must insert calls to 'dispose(...)' in appropriate places in your programs.

The list above is not intended to be exhaustive, and neither do we argue that the constructs we use are necessarily superior to others commonly available. Our reason for extending the notation of Pascal (or any other programming language we might have chosen as a starting point) is the following: in addressing human readers, we believe an open-ended, somewhat informal notation is preferable to the straightjacket of any one programming language. The latter becomes necessary if and when we execute a program, but during the incubation period when

our understanding slowly grows toward a firm grasp of an idea, supporting intuition is much more important than formality. Thus we describe data structures and algorithms with the help of figures, words, and programs as we see fit in any particular instance.

Programming project

1. Use your graphics frame program of “Graphics primitives and environments” to implement an editor for simple graphics productions such as those used to define snowflakes (e.g. 'any line segment gets replaced by a specified sequence of line segments'), and an interpreter that draws successive generations of the fractals defined by these productions.