Introduction to algorithms

Article *in* Resonance · September 1996

DOI: 10.1007/BF02837777

1 author:

R. K. Shyamasundar
Indian Institute of Technology Bombay
291 PUBLICATIONS **1,602** CITATIONS

Introduction to Algorithms

4. Turtle Graphics

R K Shyamasundar

R K Shyamasundar is Professor of Computer Science at TIFR, Mumbai and has done extensive research in various foundation areas of computer science.

The primary purpose of a programming language is to assist the programmer in the practice of her art. Each language is either designed for a class of problems or supports a different style of programming. In other words, a programming language turns the computer into a 'virtual machine' whose features and capabilities are unlimited. In this article, we illustrate these aspects through a language similar to *logo*. Programs are developed to draw geometric pictures using this language.

Programming Languages

A programming language is more than a vehicle for instructing existing computers. It primarily assists the programmer in the most difficult aspects of her art, namely program design, documentation and debugging. In other words, a good programming language should help express how the program is run, and what it intends to accomplish. It should achieve this at various levels, from the overall strategy to the details of coding and data representation. It should help establish and enforce programming disciplines that ensure harmonious cooperation of the parts of a large program developed separately and finally assembled together. It must assist in developing and displaying a pleasant writing style. In a broader sense, a programming language transforms the underlying machine into a '*virtual machine*' at a different level. The features and the capabilities of the virtual machine are limited only by the imagination of the language designer; of course, the virtual machine should in fact be efficiently 'implementable'. Ease of reading of programs is much more important than ease of writing.

A programming language primarily assists the programmer in program design, documentation and debugging.

In the previous articles of this series, we have looked at the basic control structures of languages. In this article, we shall take a look at an integration of such structures with simple graphic commands and illustrate how such an integration aids the programmer to solve problems in *turtle-geometry*. Finally, we provide a glimpse of a few of the programming languages that support different styles of programming.

A good programming language should help express how the program should run, and what it intends to accomplish.

A Simple Programming Language

First let us try to write algorithms to draw graphical (geometric) pictures using the following Turtle-like (Logo-like) commands. The Turtle-commands are used with the basic control structures such as sequencing, test and iteration described in earlier parts of the series.

Capabilities of Turtle

Let us first enumerate the basic capabilities of our Turtle:

- The Turtle moves on a plane.
- It starts at a coordinate point and has a direction.
- It can move from one point to another with or without drawing a line as it moves.
- It can rotate clockwise or anti-clockwise while remaining at the same point.

With such primitive capabilities, we can build the basic commands as shown in *Table 1*.

Table 1. Basic Commands for the Turtle

line n	Move n units and draw a straight line in the direction of the Turtle
skip n	Move n units in the direction of the Turtle without drawing a line
Anti-clock n	Rotate the Turtle by n degrees in the anti-clockwise direction
Clockwise n	Rotate the Turtle by n degrees in the clockwise direction

Programming Notation

As elaborated in the earlier articles, algorithms must be written in an unambiguous formal way. Algorithms intended for automatic execution by computers are called *programs* and the formal notations used to write *programs* are called *programming languages*. The concept of a programming language has been around since the mid-fifties. In 1945, the German mathematician Konrad Zuse invented a notation called Plankalkül. Statements in the language had a two-dimensional format: variables and their subscripts were aligned vertically and operations on them were laid out along the horizontal axis. Zuse wrote Plankalkül programs on paper including one that made simple chess moves. Even though the language was not implemented, many of the ideas developed by Zuse have been introduced in subsequent programming languages.

Drawing a Square

An algorithm to draw a square using the turtle-commands (essentially the way we draw a square) is shown in *Table 2*. Note that from the initial value of *i* and the termination condition of the loop (i.e., *i* > 0 is false), we can conclude that four lines have been drawn. From the sequencing of the commands for rotation, we can deduce that the figure should be a square.

To draw a square of length 10 units one writes the command:

call square (10)

The above command draws the square shown in *Figure 1(a)*.

Drawing Regular Polygons

In the procedure for drawing a square, we have used "length" to be the *formal parameter*. Now, if we allow the number of sides to be another parameter then we can get polygons easily. In other

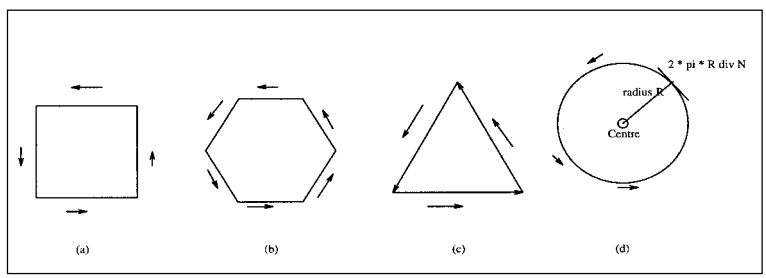

Figure 1 Polygons Drawn

Table 2 Drawing a square

```
procedure square (length:integer);
        i: = 4;
        while i > 0 do
                line length;          (* Draw a line *)
                anti-clock 90;        (* Rotate by 90 in the anti-clockwise direction *)
                i: = i-1
        end;                          (* 4 lines have been drawn *)
endprocedure
```

words, we have refined further the procedural abstraction of the *square*; that is, square becomes a particular polygon; but not the other way. The procedure shown in *Table 3* draws a polygon of *N* sides of length *L*. As our turtle draws in "integer" steps, we should use such *N*s that divide 360 exactly so that the polygon is a *closed* (beginning and end-points coincide) figure.

Thus the commands

call polygon (6, 13) and} call polygon (3,10)

draw a hexagon with sides of length 13 units and an equilateral triangle with sides of length 10 units respectively.

Drawing a Circle

We can draw a circle by noting the fact that a circle can be approximated as a regular polygon with a large number of sides. For example,

call polygon (60, 8)

will draw a polygon of 60 sides with each of length 8 units, and will look like a circle on most computer screens! The usual way of drawing a circle is to draw it with a given radius, say R. To draw in such a way, one could draw a polygon of radius R around a centre point. A way to draw a circle of radius, R, around a centre, C, would be to draw N sides, each time starting from the centre, jumping to the edge, drawing a side and then jumping back to the centre again.

Note

- The assumption that the Turtle is oriented is important. Depending on its orientation, the line is drawn.
- The reader should understand the use of *parameters* discussed inthe previous article of this series.

Table 3 Procedure for Drawing a Polygon

```
procedure polygon (N:integer, L:integer);
        i: = N;
        while i > 0 do
                line L;
                anti-clock (360 ¸ N);
                i: i = i – 1
        endwhile
endprocedure
```

To be able to draw the circle using the method outlined, we need to use a command described earlier (but not used so far): *skip from one point to another point without drawing a line*. To recollect, the command and its effect is given in *Table 4.*

Let us see how we can devise an algorithm for drawing a circle using such an approach. Now, we know that circumference of a circle of radius R, is equal to $2\pi R$. As our turtle moves in integer units, we shall approximate π by 3. Thus, the length of the side becomes $(2 * 3 * \text{radius}) \div$ sides. A procedure implementing these ideas is described in *Table 5*.

Exercise: Are there any drawbacks of the approximation used?

The start and end-points may not coincide! However, the centre of the circle will be where it was intended to be. It may also be observed that the *skip*-command is somewhat independent of other commands. One of the interesting questions often raised is the independence (or orthogonality) of basic commands and constructs from various considerations. Some of these aspects will be featured in future articles.

Table 4
SKIP Command

skip n – The turtle moves n units from the current position in the direction pointed to by the turtle, without drawing a line.

skip –n – The turtle moves n units from the current position in the direction opposite to that pointed by the turtle, without drawing a line.

Turing Award

The Turing Award — highest award in Computer Science — is presented annually "to an individual selected for contributions of a technical nature to the computing community that are judged to be of lasting and major importance to the field of computing science." The award commemorates Alan M Turing, an English mathematician whose work "captured the imagination and mobilized the thoughts of a generation of scientists" to quote Alan J Perlis, the first recipient of the award.

Programming is fundamental to the computing field. Programs play a dual role both as the notations that influence thought and also the directions for an abstract computing machine. The influence can be seen by the fact that in the first 20 years half of the Turing awards have recognized contributions to programming languages, programming methodology, and programming systems. The award lectures by Alan J Perlis, Edsgar W Dijkstra, Donald E Knuth, Dana Scott, Michael Rabin, John Backus, Robert Floyd, C A R Hoare, Ken Thompson, Dennis Ritchie and Niklaus Wirth provide insights into the various computational aspects as seen by the pioneers.

Complex Patterns

Let us consider the problem of drawing a simple flower which consists of a number of petals. For simplicity, let us assume that the petal has a simple shape as shown in *Figure 2(a)*.

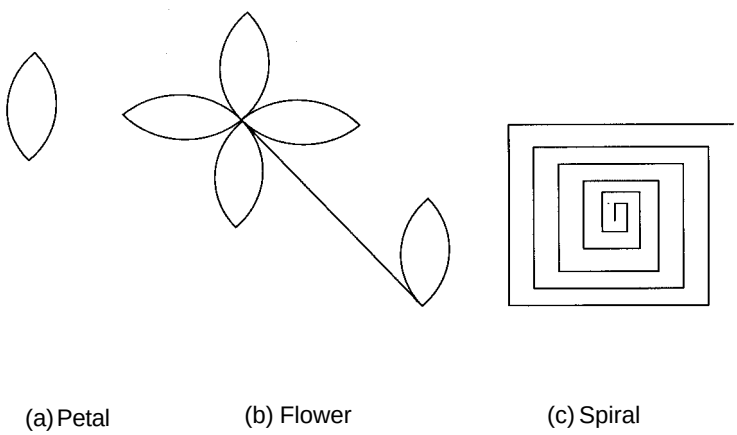

Figure 2 Patterns

Table 5 Procedure for Drawing a Circle

```
procedure circle (sides:integer, radius:integer);
        i := 1;
        while i ≤ sides do
                        skip radius;
                        clockwise 90;
                        skip ( – (3 * radius ÷ sides);
                        line (2 * (3 * radius )) ÷ sides);
                        skip (– (3 * radius)) ÷ sides);
                        anti-clock 90;
                        skip (– radius);
                        clockwise (360 ÷ sides)
        endwhile
endprocedure
```

Drawing a Petal

We have shown above how we can draw circles and arcs. Using the same technique, it should be easy to arrive at a procedure to draw a petal. For simplicity, the reader can think of a petal as made up of two quadrants of a circle with some radius say size. Since the orientation of the turtle is important, let us enforce the condition that the turtle returns to its original position and orientation after the petal is drawn. Let us denote the procedure by

procedure PETAL (SIZE: integer)

The code for this procedure is left as an exercise to the reader. Now, a command of PETAL (5) will draw a petal similar to that shown in *Figure 2(a)* with 5 units as the radius. Now, we can program a simple flower using PETAL as a procedure. Let us assume that a flower has a head, with several petals in a circle, and a stem with a leaf, which looks just like a petal. A program to draw a flower with *N* petals of size *S* is shown in *Table 6*.

One can draw spirals of various shapes and sizes

Table 6 Drawing a Flower

```
procedure FLOWER (N: integer; S: integer)
        i:= 0;
        while i < N do
                call PETAL (S);          (* draw a petal of size S *)
                                          (* as per assumption the turtle returns to its initial
                                             position on completion *)
                anti-clock ((360 ÷N));  (* Rotate the turtle for the next petal *)
                i:=i+1;
        end;
        anti-clock 180;                  (* turn the direction of the turtle *)
        line (4 * S);                    (* draws a stem of length equal to 4 times the size
                                             of the petal *)
        anti-clock 90;                   (* position the turtle for the leaf *)
call PETAL (S)
endprocedure
```

Using the above procedures, the user can create further patterns such as a bunch of flowers etc.

Drawing a Rectangular Spiral Pattern

One can draw spirals of various shapes and sizes. Consider the shape shown in *Figure 2(c)*. The growing spiral could be considered as made up of lines and rotations on the turtle position which is captured in the program shown in *Table 7*. Now a program to draw a spiral is shown in *Table 8* which uses the procedure STEP given in *Table 7*. Size 1 and size 2 indicate the lengths of the least and the maximum sides in the spiral respectively.

```
procedure step (size:integer, angle: integer);
        line size;
        clockwise angle;
endprocedure
```

Table 7 Basic Step in a Spiral

Table 8 Program for Drawing a Spiral

```
procedure SPIRAL (size1, size2: integer, angle:integer);
        i:=1;                (* used for the offset from side to side *)
                              (* assumed to be 1 here *)
        if size 1 < size 2 then
                call step (size1, angle);
                size 1:= size 1+i;  (* increment the side of the spiral *)
                call SPIRAL (size 1, size 2, angle) (* a recursive call *)
end
endprocedure
```

Exercise

1. Try and trace the above procedure with some values for the parameters; this will give you a good understanding of the recursive program structure and the correspondence between formal and actual parameters.
2. The interested reader may write a program to generate logarithmic spirals.

It may be noted that the program for drawing a spiral is recursive.

It may be noted that the program for drawing a spiral is recursive; in a sense, it is natural to treat it as a recursive program; by removing the part in the beginning or at the end, the pattern remains a spiral. In fact, the program reflects a growing spiral. That is, it starts from a basic step and adds to it in stages. It is also possible to write it in the reverse manner. An iterative program for the spiral is shown in *Table 9*. Iteration and recursion are two powerful techniques of programming; a comparative view would be presented in the forthcoming articles.

Fractals

A fractal is a geometric figure in which an identical motif repeats itself on an ever diminishing scale. Benoit B Mandelbrot, a French-American mathematician, is the godfather of fractals.

```
procedure SPIRAL (size 1, size 2: integer, angle:integer);
        i:=1;          (* used for the offset from side to side *)
                              (* assumed to be 1 here *)
while size 1 < size 2 do
        call step (size1, angle);
        size1:= size1+i; (* increment the side of the spiral *)
end
endprocedure
```

Table 9 An Iterative Program for Drawing a Spiral.

The extraordinary visual beauty of fractal images have made these endlessly repeating geometric figures widely familiar. They are more than just appealing visual patterns or simply part of pure mathematics and have proved to have a wide range of uses (widely used in Chaos theory) (*Resonance*, Vol 1, No 5). Once they were just mathematical curiosities; the advances in computer technology has made it possible to generate these patterns on even simple computers. In *Figure 3*, we illustrate a simple fractal (referred to as H-Fractal) which can be programmed in the language discussed above.

The H-fractal shown (arrows at the middle show one symmetry point)is called a dendrite after the Greek dendron, tree. The figure is singly-connected (with a single cut it becomes two figures not connected by any line). The structure is really like

The H-fractal is called a dendrite after the Greek dendron, tree.

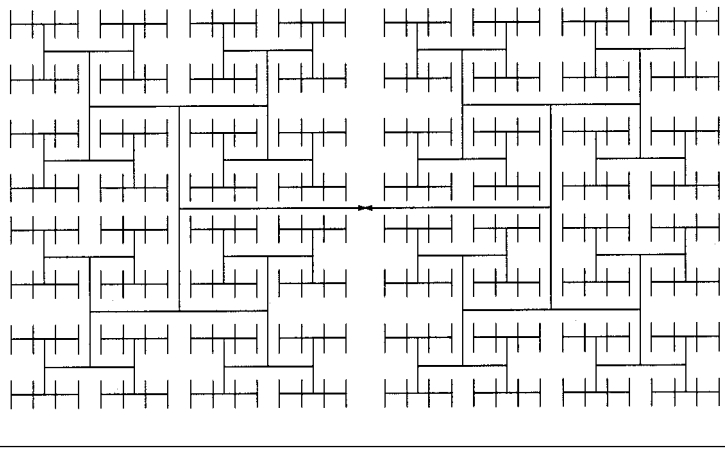

Figure 3 H – Fractal

that of a tree. A trunk separates into two side branches, each of which acts as a trunk for the following two smaller side branches, and so on. Writing a program to generate such a figure will show the power of recursion as well.

There has been a spectrum of programming languages catering to various styles and purposes.

Discussion

We have illustrated, using a turtle-like language, how one can arrive at a language for drawing geometric figures. The programmer visualizes a *virtual machine* of turtle-geometryusing a system with a turtle-language translator to regard the notations of programming as *languages* is a mixed blessing. On the one hand it is very helpful from the point of view that it provides a natural framework and terminologies such as *grammar, syntax* and *semantics* in understanding the notation clearly. On the other hand, it must be pointed out that the analogy with so called 'natural languages' is misleading since natural languages which are non-formalized derive both their weakness and strength from their vagueness and imprecision. There has been a spectrum of programming languages catering to various styles and purposes. In future articles, we will provide an overview of the different styles after discussing data-structures in the development of algorithms.

Suggested Reading

- ACM Turing award lectures: the first twenty years: 1966-1985, NewYork: ACM Press, Addison-Wesley, 1987. This book gives an insight into the views of the pioneers on computation and programming of computer science.
- S Papert. *Mindstorms: Children, Computers and Powerful Ideas*. Harvester Press, Breighton. 1980.
- H Abelson and A di Sessa. *Turtle Geometry: The computer as a medium for exploring mathematics.* M.I.T. Press, Cambridge, Mass. 1981.
- H Lauwerier. Fractals: Images of Chaos. Penguin Books. 1991. This book introduces fractals to a wide audience.

Address for Correspondence
R K Shyamasundar
Computer Science Group
Tata Institute of
Fundamental Research
Homi Bhabha Road
Mumbai 400 005, India.