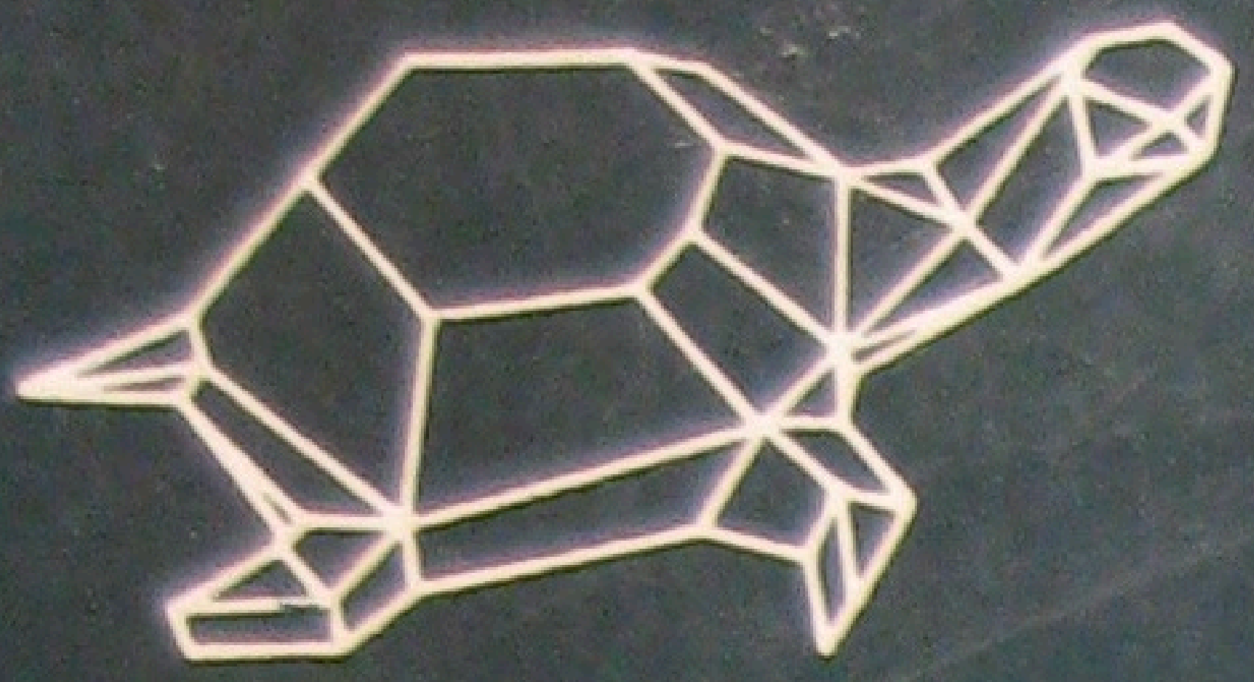
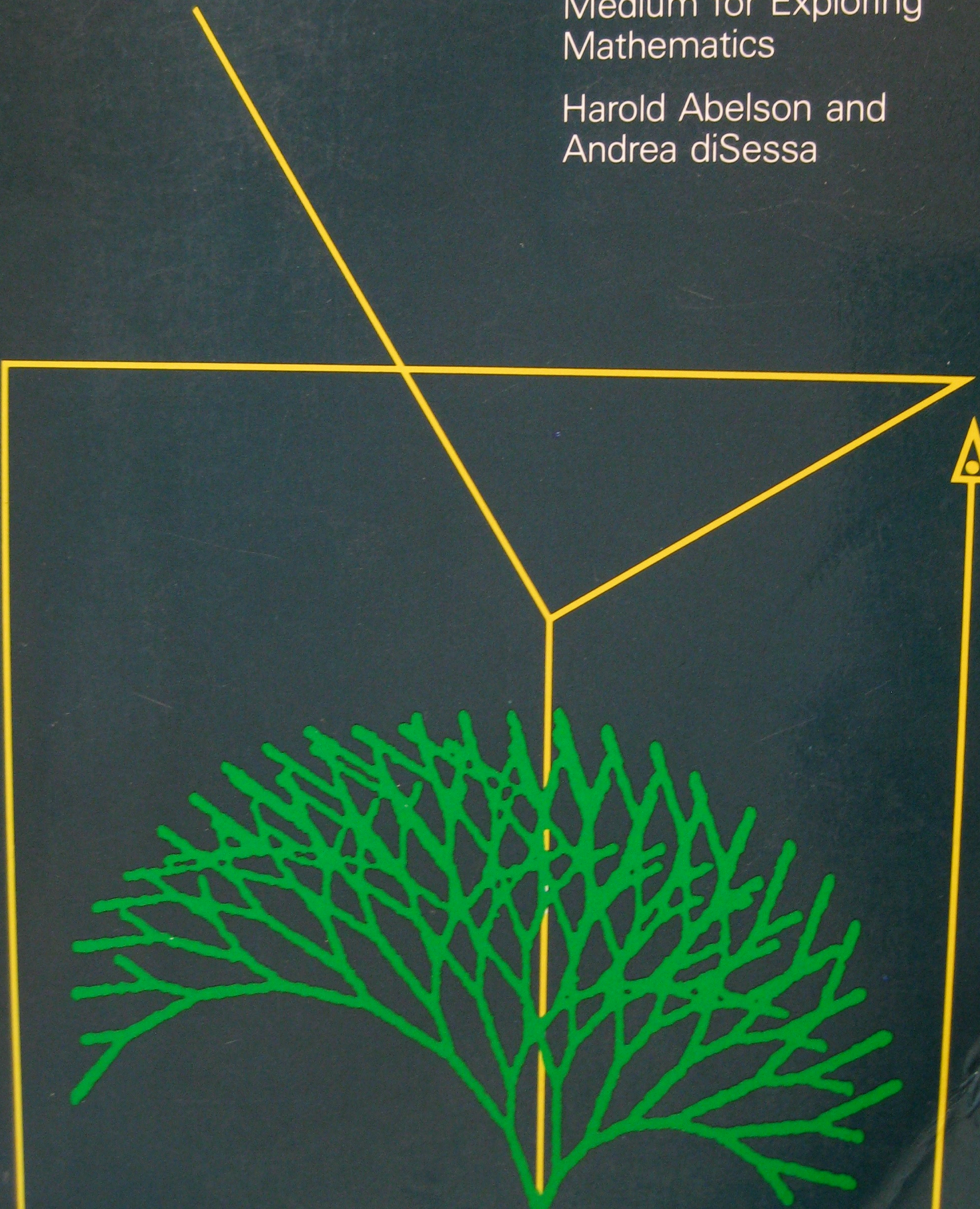


Turtle Geometry

The Computer as a Medium for Exploring Mathematics

Harold Abelson and Andrea diSessa



This is a book about exploring mathematics, and the most important thing about exploring mathematics is for you to do it rather than just passively read what we've written. Many of the sections (in particular chapter 2, the latter part of chapter 3, and chapters 6 through 9) contain extended descriptions of computer projects for you to implement and investigate. Additional open-ended projects, indicated with a [P], are listed with the exercises at the end of each section. In the text and exercises we have taken pains to show both the breadth and the depth of possible work with turtle geometry. What is most encouraging to us, however, is that we are certain that we have only scratched the surface. So do not hesitate to depart from the problems and projects we've included in order to follow your own ideas.

The computer used to undertake these projects must be capable of producing drawings in response to "turtle graphics" commands. Fortunately, most computer graphics systems can be readily adapted to execute turtle commands. A detailed explanation of how to do this is given in chapter 3, and appendix B includes a typical implementation of turtle commands in BASIC. In addition, there are (as of January 1981) a few commercially available computer systems that have turtle graphics built in. The most widespread of these is the Pascal system for the Apple II. (See appendix B for others.)

In writing a book about computer graphics projects we had to select a notation in which to describe the algorithms. Using a standard programing language such as BASIC would have forced us to specify numerous implementation-dependent "computerish" details, which would have obscured the simplicity of the programs. Our response was to choose a notation in which turtle algorithms can be expressed simply, and yet which is close enough to real programing languages so that you should have little troubling translating our programs into the language of your choice. In fact, our notation is quite similar to the programing language Pascal (if one ignores details about declaring variables and data types), and even closer to Logo, a language developed at MIT especially for this kind of educational use. Appendix A provides details on our "computer language" notation, and appendix B illustrates sample turtle programs translated into standard programing languages.

Besides computer projects, we've included numerous exercises to test understanding, suggest inquiry, or just pique the imagination. These range in scope from simple problems to extended research topics. The more difficult ones are indicated with a [D]. There are also a few problems marked [DD]. If you tackle one of them, be prepared for a real challenge. We've provided two separate answer sections, one of hints and one of more complete answers; exercises are marked with [H], [A], or [HA], depending on whether we've furnished a hint, an answer, or both. If you get stuck on an [HA] exercise, try working with the hint before looking up the answer.

You should not feel constrained to read this book straight through. Indeed, we hope even casual readers will sample various sections, digging into what they find most interesting. The closest things to prerequisites for later chapters are as follows: The first few sections of chapter 1 introduce the basics of turtles and turtle geometry. Chapter 3 contains the most essential technical material, vectors and coordinate methods, which we use freely in explanations and programs in later chapters. Chapter 5 broaches the new area of nonflat geometries, which will occupy the rest of the book, but even here only the most general notions are necessary to proceed. By and large we have placed the most important material toward the beginnings of chapters and sections.

The tradition of calling our display creatures "turtles" started with Grey Walter, a neurophysiologist who experimented in Britain during the early 1960s with tiny robot creatures he called "tortoises." These inspired the first turtles designed at MIT—computer-controlled robots that moved around on the floor in response to the commands `FORWARD` and `RIGHT`. Work in the present mathematical and computer-graphics context followed directly and inherited the turtle terminology.

1

Introduction to Turtle Geometry

We start with the simplest vocabulary of images, with "left" and "right" and "one, two, three," and before we know how it happened the words and numbers have conspired to make a match with nature: we catch in them the pattern of mind and matter as one.

Jacob Bronowski, *The Reach of Imagination*

This chapter is an introduction on three levels. First, we introduce you to a new kind of geometry called turtle geometry. The most important thing to remember about turtle geometry is that it is a mathematics designed for exploration, not just for presenting theorems and proofs. When we do state and prove theorems, we are trying to help you to generate new ideas and to think about and understand the phenomena you discover.

The technical language of this geometry is our second priority. This may look as if we're describing a computer language, but our real aim is to establish a notation for the range of complicated things a turtle can do in terms of the simplest things it knows. If you wish to actually program a computer-controlled turtle using one of the standard programing languages, you will need to know more details than are presented here; see appendixes A and B.

Finally, this chapter will introduce some of the important themes to be elaborated in later chapters. These themes permeate not only geometry but all of mathematics, and we aim to give you rich and varied experiences with them.

1.1 Turtle Graphics

Imagine that you have control of a little creature called a turtle that exists in a mathematical plane or, better yet, on a computer display screen. The turtle can respond to a few simple *commands*: `FORWARD` moves the turtle, in the direction it is facing, some number of units. `RIGHT` rotates it in place, clockwise, some number of degrees. `BACK` and `LEFT` cause the opposite movements. The number that goes with a command to specify how much to move is called the command's *input*.

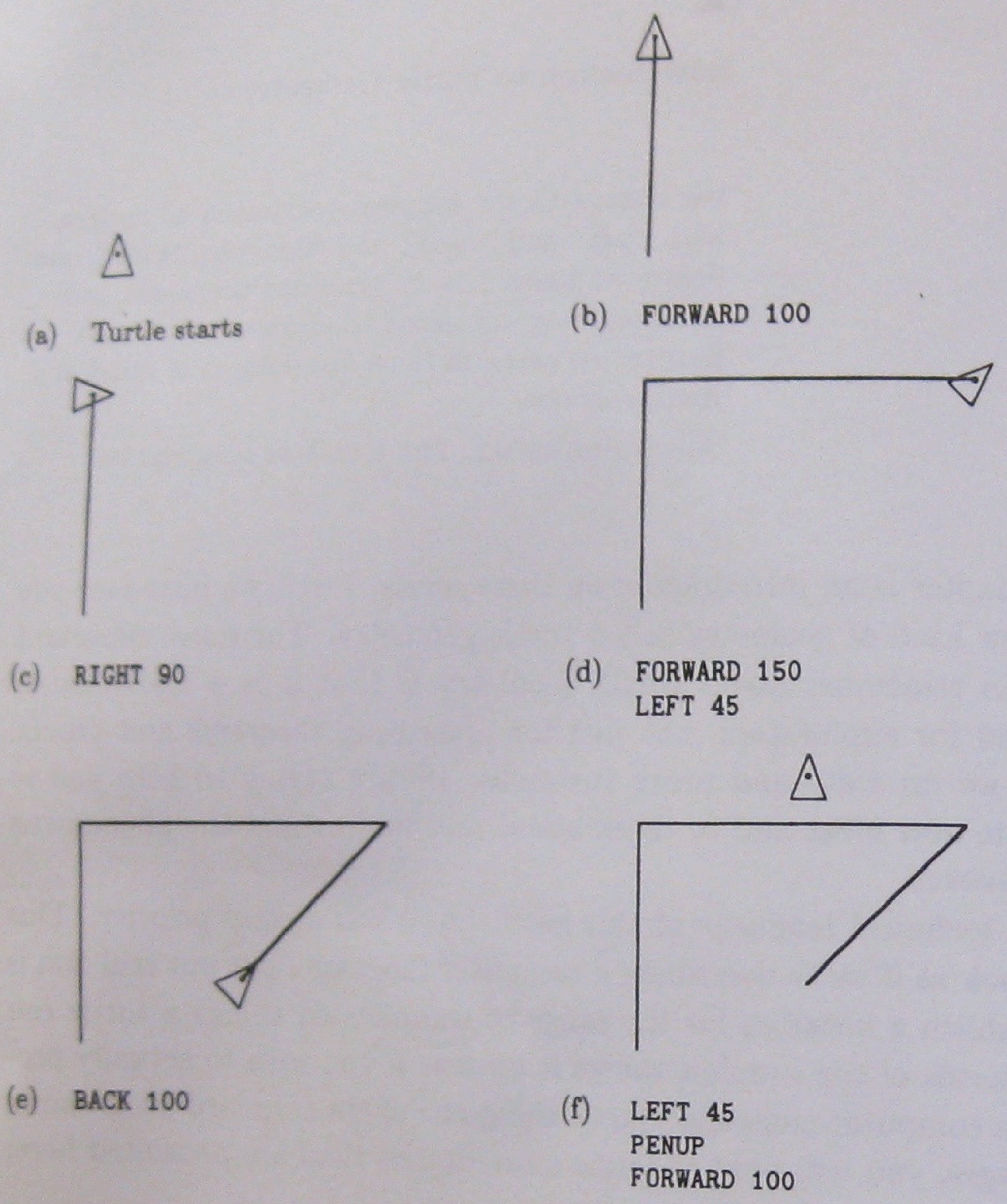


Figure 1.1
A sequence of turtle commands.

In describing the effects of these operations, we say that FORWARD and BACK change the turtle's *position* (the point on the plane where the turtle is located); RIGHT and LEFT change the turtle's *heading* (the direction in which the turtle is facing).

The turtle can leave a trace of the places it has been: The position-changing commands can cause lines to appear on the screen. This is controlled by the commands PENUP and PENDOWN. When the pen is down, the turtle draws lines. Figure 1.1 illustrates how you can draw on the display screen by steering the turtle with FORWARD, BACK, RIGHT, and LEFT.

1.1.1 Procedures

Turtle geometry would be rather dull if it did not allow us to teach the turtle new commands. But luckily all we have to do to teach the turtle a new trick is to give it a list of commands it already knows. For example, here's how to draw a square with sides 100 units long:

```
TO SQUARE
   FORWARD 100
   RIGHT 90
   FORWARD 100
   RIGHT 90
   FORWARD 100
   RIGHT 90
   FORWARD 100
```

This is an example of a *procedure*. (Such definitions are also commonly referred to as programs or functions.) The first line of the procedure (the *title line*) specifies the procedure's name. We've chosen to name this procedure SQUARE, but we could have named it anything at all. The rest of the procedure (the *body*) specifies a list of instructions the turtle is to carry out in response to the SQUARE command.

There are a few useful tricks for writing procedures. One of them is called *iteration*, meaning repetition—doing something over and over. Here's a more concise way of telling the turtle to draw a square, using iteration:

```
TO SQUARE
   REPEAT 4
      FORWARD 100
      RIGHT 90
```

This procedure will repeat the indented commands FORWARD 100 and RIGHT 90 four times.

Another trick is to create a SQUARE procedure that takes an input for the size of the square. To do this, specify a name for the input in the title line of the procedure, and use the name in the procedure body:

```
TO SQUARE SIZE
   REPEAT 4
      FORWARD SIZE
      RIGHT 90
```

Now, when you use the command, you must specify the value to be used for the input, so you say SQUARE 100, just like FORWARD 100.

The chunk FORWARD SIZE, RIGHT 90 might be useful in other contexts, which is a good reason to make it a procedure in its own right:

```
TO SQUAREPIECE SIZE
   FORWARD SIZE
   RIGHT 90
```

Now we can rewrite SQUARE using SQUAREPIECE as

```
TO SQUARE SIZE
   REPEAT 4
      SQUAREPIECE SIZE
```

Notice that the input to SQUARE, also called SIZE, is passed in turn as an input to SQUAREPIECE. SQUAREPIECE can be used as a *subprocedure* in other places as well—for example, in drawing a rectangle:

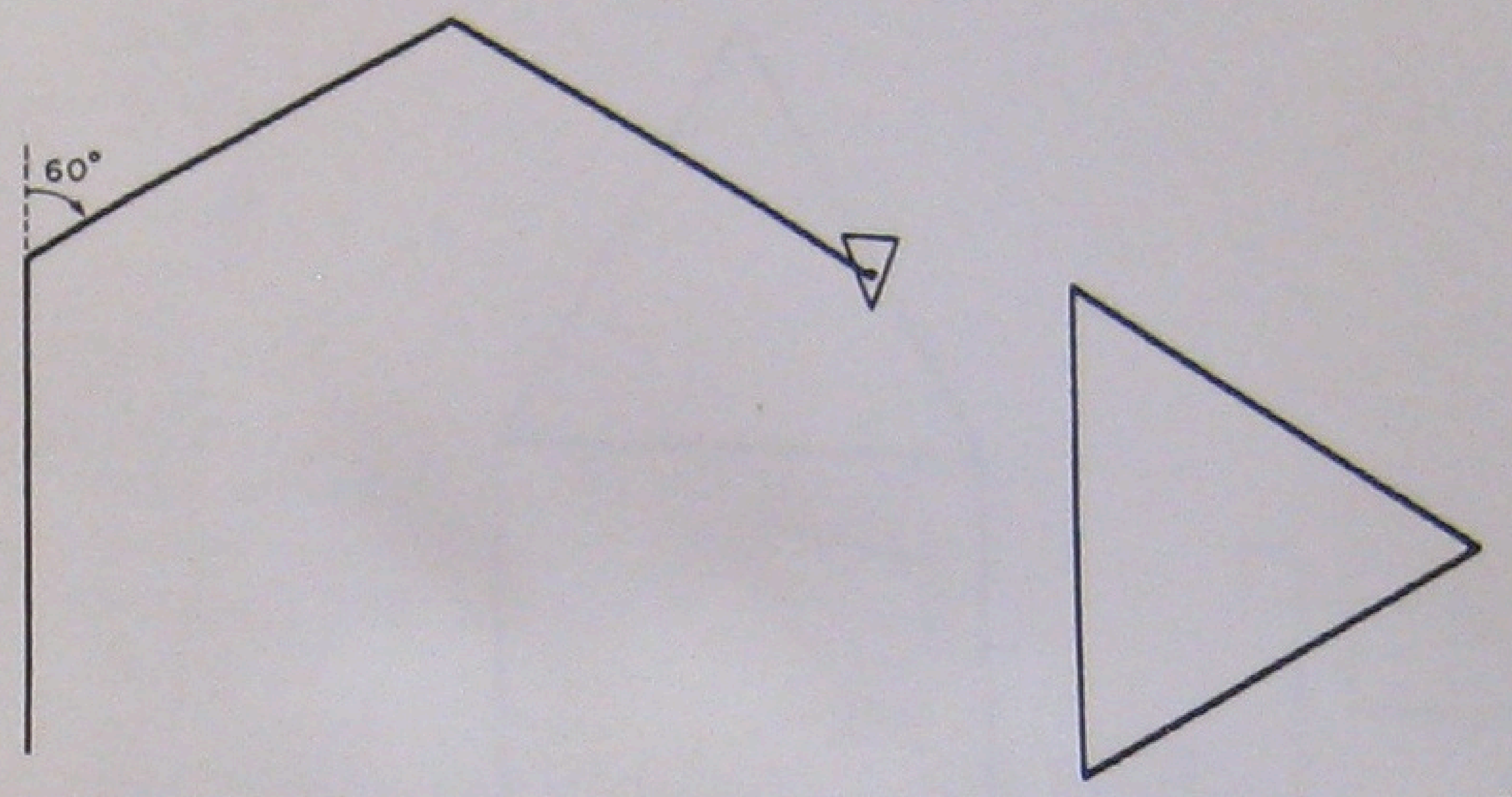
```
TO RECTANGLE SIDE1 SIDE2
   REPEAT 2
      SQUAREPIECE SIDE1
      SQUAREPIECE SIDE2
```

To use the RECTANGLE procedure you must specify its two inputs, for example, RECTANGLE 100 50.

When programs become more complex this kind of input notation can be a bit hard to read, especially when there are procedures such as RECTANGLE that take more than one input. Sometimes it helps to use parentheses and commas to separate inputs to procedures. For example, the RECTANGLE procedure can be written as

```
TO RECTANGLE (SIDE1, SIDE2)
   REPEAT 2
      SQUAREPIECE (SIDE1)
      SQUAREPIECE (SIDE2)
```

If you like, you can regard this notation as a computer language that has been designed to make it easy to interact with turtles. Appendix A gives some of the details of this language. It should not be difficult to rewrite these procedures in any language that has access to the basic turtle commands FORWARD, BACK, RIGHT, LEFT, PENUP, and PENDOWN.



```
TO TRY.ANGLE
   REPEAT 3
      FORWARD 100
      RIGHT 60
```

```
TO TRIANGLE
   REPEAT 3
      FORWARD 100
      RIGHT 120
```

Figure 1.2
Attempt to draw a triangle.

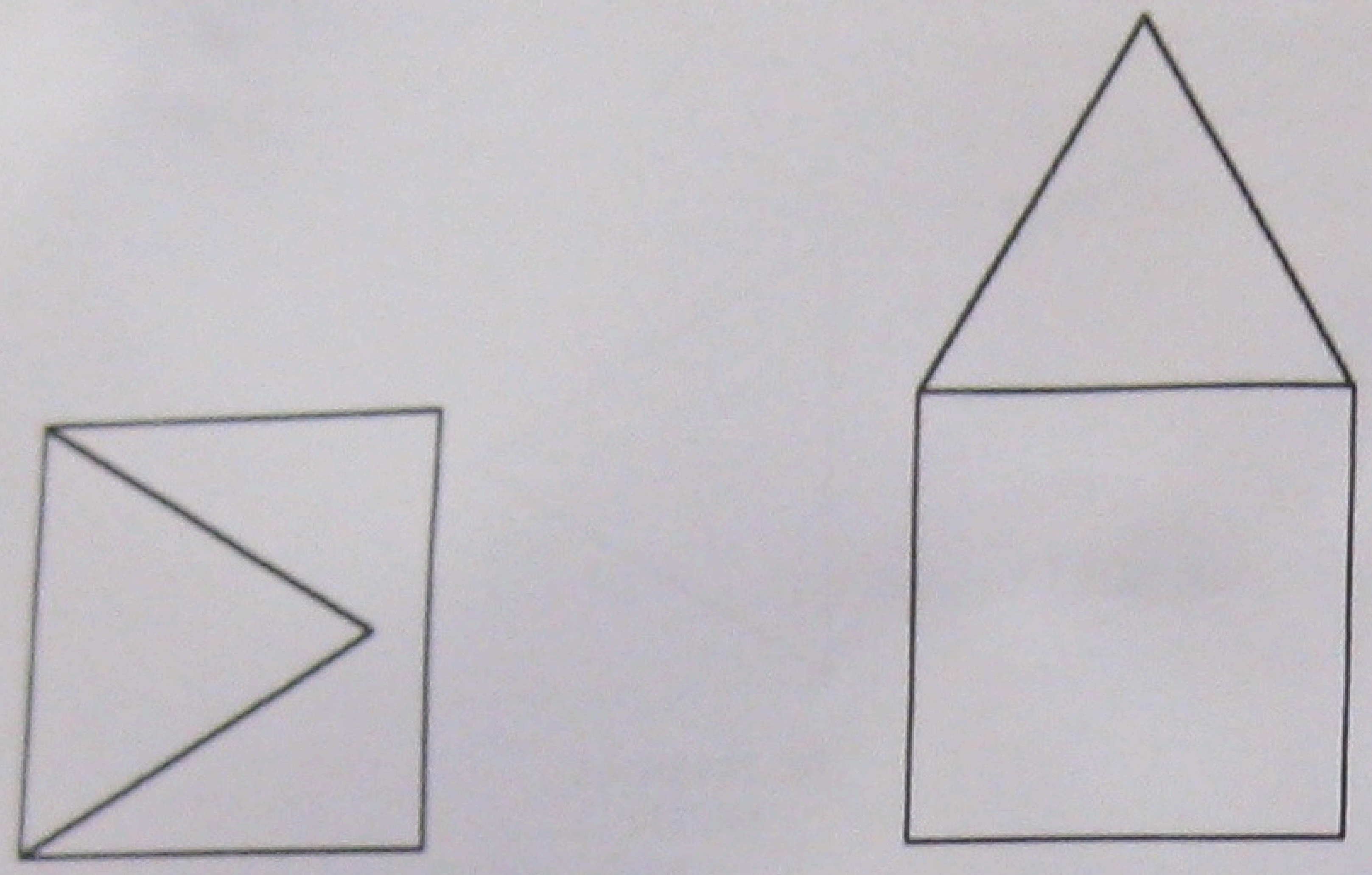
Appendix B gives some tips on how to implement these commands in some of the more common computer languages, and includes sample translations of turtle procedures.

1.1.2 Drawing with the Turtle

Let's draw a figure that doesn't use 90° angles—an equilateral triangle. Since the triangle has 60° angles, a natural first guess at a triangle procedure is

```
TO TRY.ANGLE SIZE
   REPEAT 3
      FORWARD SIZE
      RIGHT 60
```

But TRY.ANGLE doesn't work, as shown in figure 1.2. In fact, running this "triangle" procedure draws half of a regular hexagon. The bug in the procedure is that, whereas we normally measure geometric figures by their interior angles, turtle turning corresponds to the exterior angle at the vertex. So if we want to draw a triangle we should have the turtle turn 120°. You might practice "playing turtle" on a few geometric figures until it becomes natural for you to think of measuring a vertex by how much the turtle must turn in drawing the vertex, rather than by



```
TO HOUSE SIDE
   SQUARE SIDE
   TRIANGLE SIDE
```

```
TO HOUSE SIDE
   SQUARE SIDE
   FORWARD SIDE
   RIGHT 30
   TRIANGLE SIDE
```

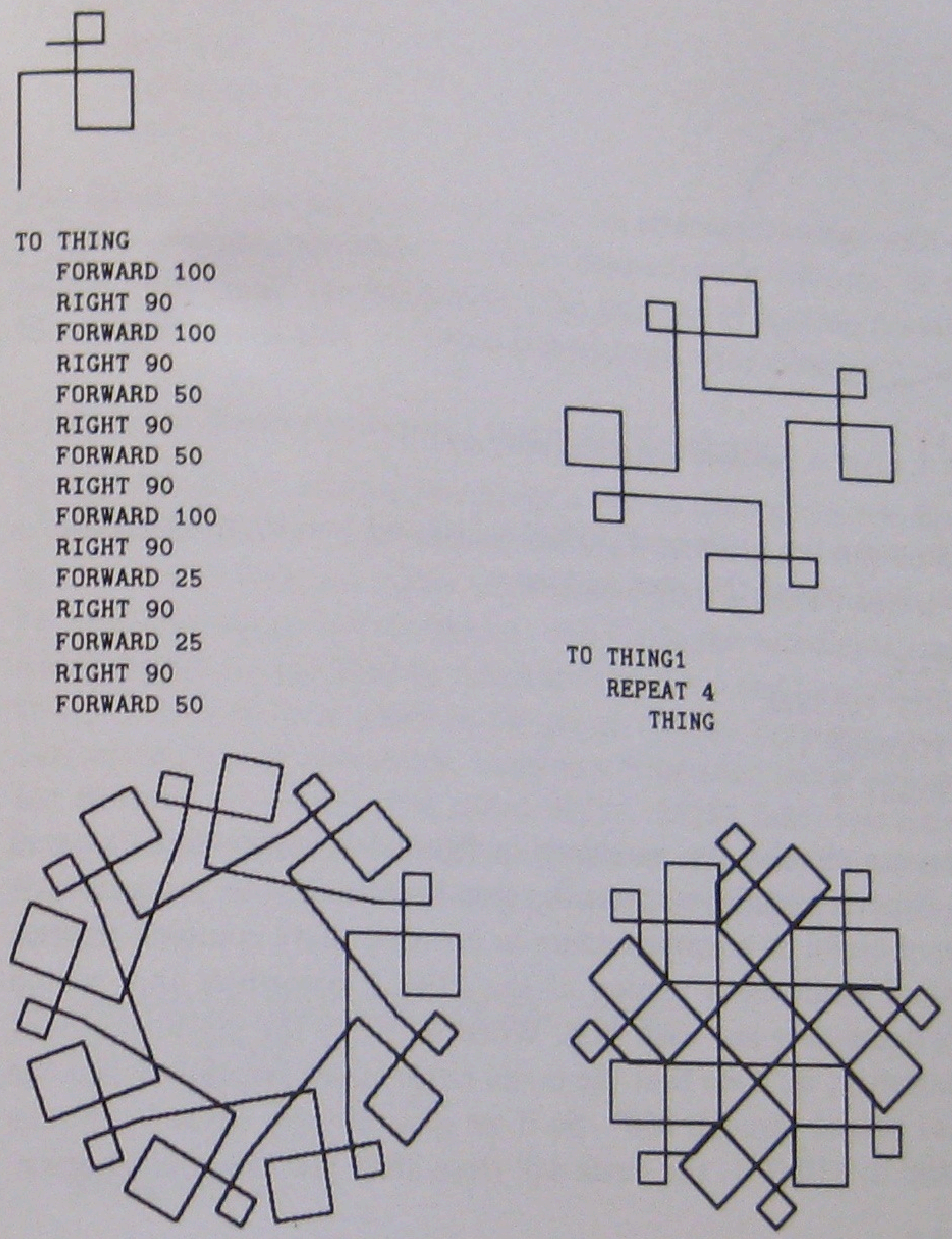
Figure 1.3
(a) Initial attempt to draw a house fails. (b) Interface steps are needed.

the usual interior angle. Turtle angle has many advantages over interior angle, as you will see.

Now that we have a triangle and a square, we can use them as building blocks in more complex drawings—a house, for example. But figure 1.3 shows that simply running SQUARE followed by TRIANGLE doesn't quite work. The reason is that after SQUARE, the turtle is at neither the correct position nor the correct heading to begin drawing the roof. To fix this bug, we must add steps to the procedure that will move and rotate the turtle before the TRIANGLE procedure is run. In terms of designing programs to draw things, these extra steps serve as an interface between the part of the program that draws the walls of the house (the SQUARE procedure) and the part that draws the roof (the TRIANGLE procedure). In general, thinking of procedures as a number of main steps separated by interfaces is a useful strategy for planning complex drawings.

Using procedures and subprocedures is also a good way to create abstract designs. Figure 1.4 shows how to create elaborate patterns by rotating a simple "doodle."

After all these straight line drawings, it is natural to ask whether the turtle can also draw curves—circles, for example. One easy way to do



```
TO THING
   FORWARD 100
   RIGHT 90
   FORWARD 100
   RIGHT 90
   FORWARD 50
   RIGHT 90
   FORWARD 50
   RIGHT 90
   FORWARD 100
   RIGHT 90
   FORWARD 25
   RIGHT 90
   FORWARD 25
   RIGHT 90
   FORWARD 50
```

```
TO THING1
   REPEAT 4
      THING
```

```
TO THING2
   REPEAT FOREVER
      THING
      RIGHT 10
      FORWARD 50
```

```
TO THING3
   REPEAT FOREVER
      THING
      LEFT 45
      FORWARD 100
```

Figure 1.4
Designs made by rotating a simple doodle.

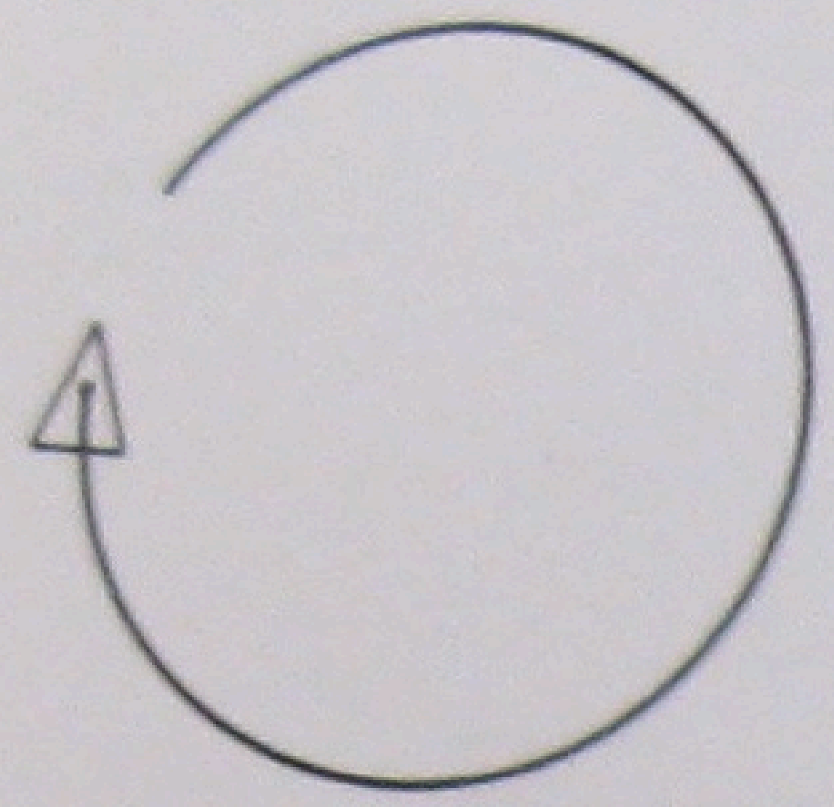


Figure 1.5
FORWARD 1, RIGHT 1, repeated draws a circular arc.

this is to make the turtle go FORWARD a little bit and then turn RIGHT a little bit, and repeat this over and over:

```
TO CIRCLE
   REPEAT FOREVER
      FORWARD 1
      RIGHT 1
```

This draws a circular arc, as shown in figure 1.5. Since this program goes on "forever" (until you press the stop button on your computer), it is not very useful as a subprocedure in creating more complex figures. More useful would be a version of the CIRCLE procedure that would draw the figure once and then stop. When we study the mathematics of turtle geometry, we'll see that the turtle circle closes precisely when the turtle has turned through 360°. So if we generate the circle in chunks of FORWARD 1, RIGHT 1, the circle will close after precisely 360 chunks:

```
TO CIRCLE
   REPEAT 360
      FORWARD 1
      RIGHT 1
```

If we repeat the basic chunk fewer than 360 times, we get circular arcs. For instance, 180 repetitions give a semicircle, and 60 repetitions give a 60° arc. The following procedures draw left and right arcs of DEG degrees on a circle of size R:

```
TO ARCR R DEG
   REPEAT DEG
      FORWARD R
      RIGHT 1
```

```
TO ARCL R DEG
   REPEAT DEG
      FORWARD R
      LEFT 1
```

(See figure 1.6 and exercise 3 for more on making drawings with arcs.)

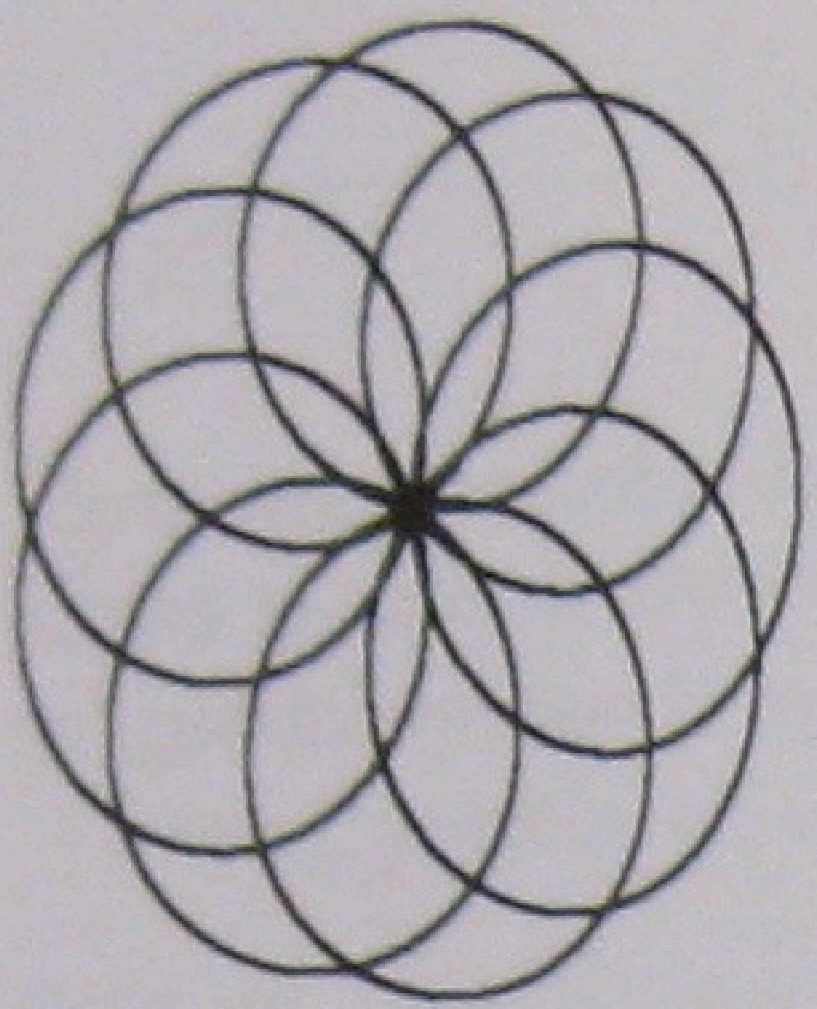
The circle program above actually draws regular 360gons, of course, rather than "real" circles, but for the purpose of making drawings on the display screen this difference is irrelevant. (See exercises 1 and 2.)

1.1.3 Turtle Geometry versus Coordinate Geometry

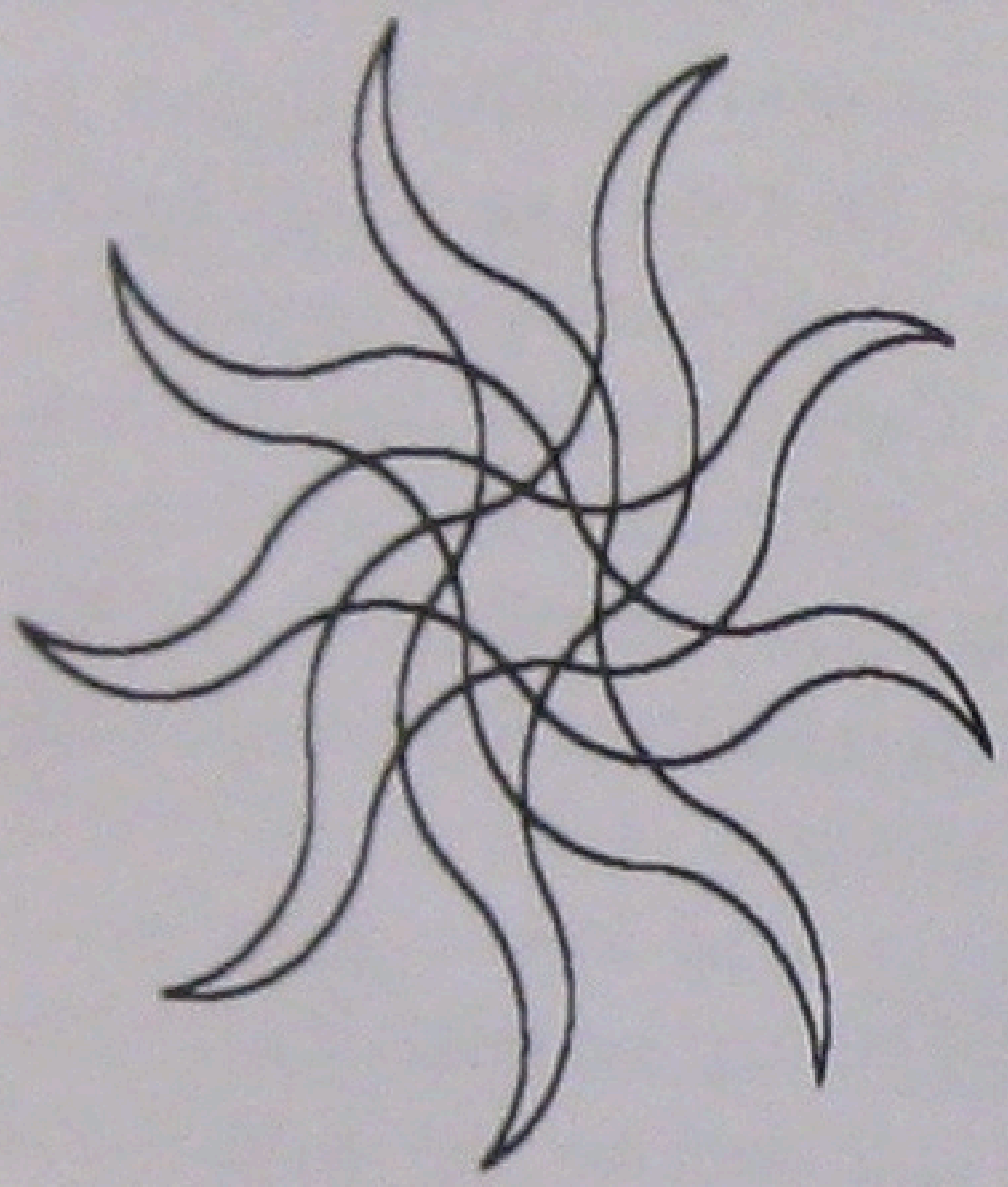
We can think of turtle commands as a way to draw geometric figures on a computer display. But we can also regard them as a way to describe figures. Let's compare turtle descriptions with a more familiar system for representing geometric figures—the Cartesian coordinate system, in which points are specified by two numbers, the x and y coordinates relative to a pair of axes drawn in the plane. To put Cartesian coordinates into our computer framework, imagine a "Cartesian turtle" whose moves are directed by a command called SETXY. SETXY takes two numbers as inputs. These numbers are interpreted as x and y coordinates, and the turtle moves to the corresponding point. We could draw a rectangle with SETXY using

```
TO CARTESIAN.RECTANGLE (WIDTH, HEIGHT)
   SETXY (WIDTH, 0)
   SETXY (WIDTH, HEIGHT)
   SETXY (0, HEIGHT)
   SETXY (0, 0)
```

You are probably familiar with the uses of coordinates in geometry: studying geometric figures via equations, plotting graphs of numerical relationships, and so on. Indeed, Descartes' marriage of algebra and geometry is one of the fundamental insights in the development of mathematics. Nevertheless, these kinds of coordinate systems—Cartesian, polar, or what have you—are not the only ways to relate numbers to geometry. The turtle FORWARD and RIGHT commands give an alternative way of measuring figures in the plane, a way that complements the coordinate viewpoint. The geometry of coordinates is called *coordinate geometry*; we shall refer to the geometry of FORWARD and RIGHT as *turtle geometry*. And even though we will be making use of coordinates later

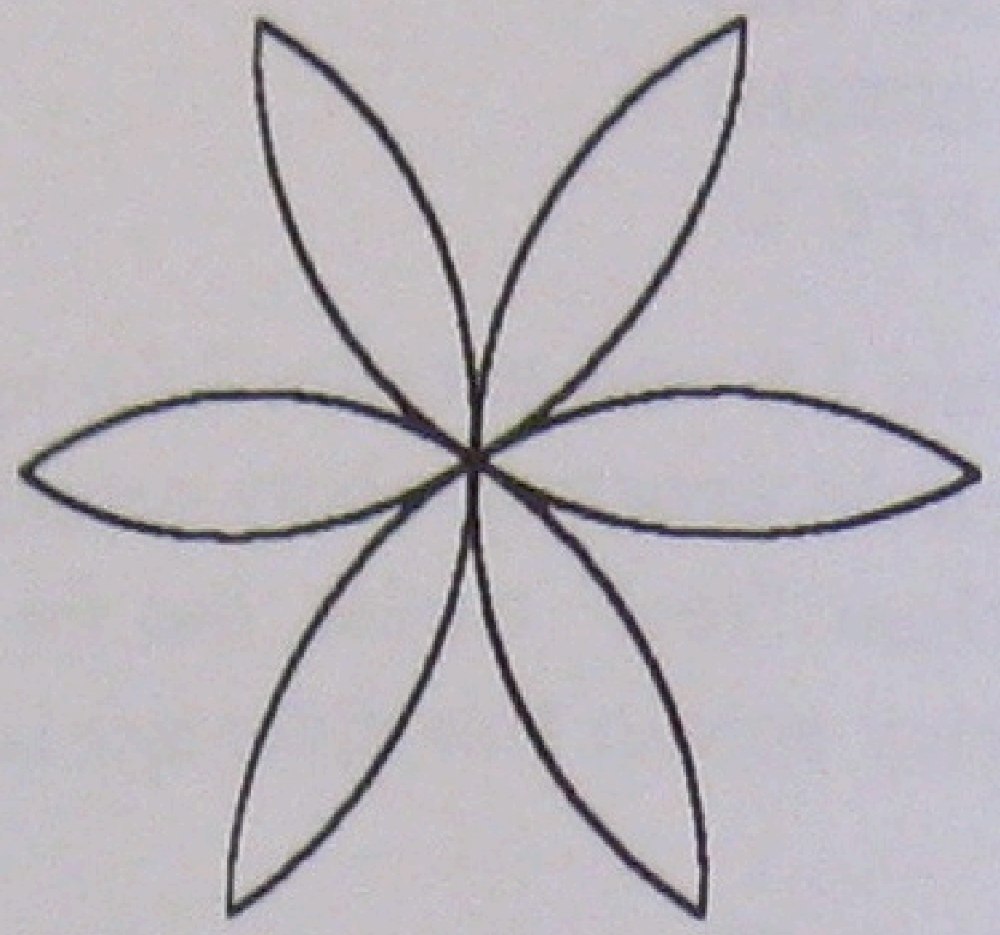


```
TO CIRCLES
   REPEAT 9
      ARCR 1 360
      RIGHT 40
```



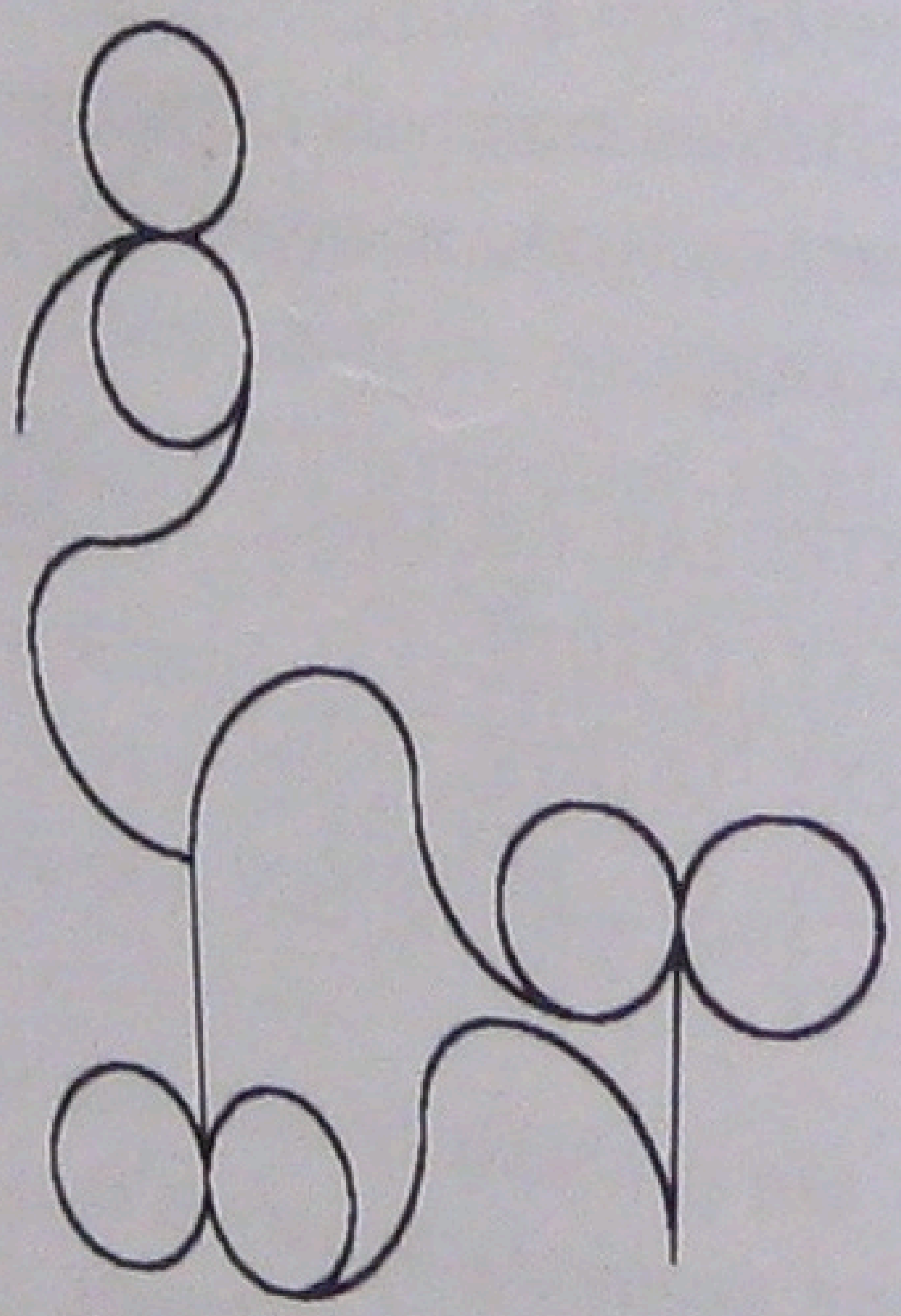
```
TO RAY R
   REPEAT 2
      ARCL R 90
      ARCR R 90

TO SUN SIZE
   REPEAT 9
      RAY SIZE
      RIGHT 160
```



```
TO PETAL SIZE
   ARCR SIZE 60
   RIGHT 120
   ARCR SIZE 60
   RIGHT 120

TO FLOWER SIZE
   REPEAT 6
      PETAL SIZE
      RIGHT 60
```



MONSTER

Figure 1.6
Some shapes that can be made using arcs.

on, let us begin by studying turtle geometry as a system in its own right. Whereas studying coordinate geometry leads to graphs and algebraic equations, turtle geometry will introduce some less familiar, but no less important, mathematical ideas.

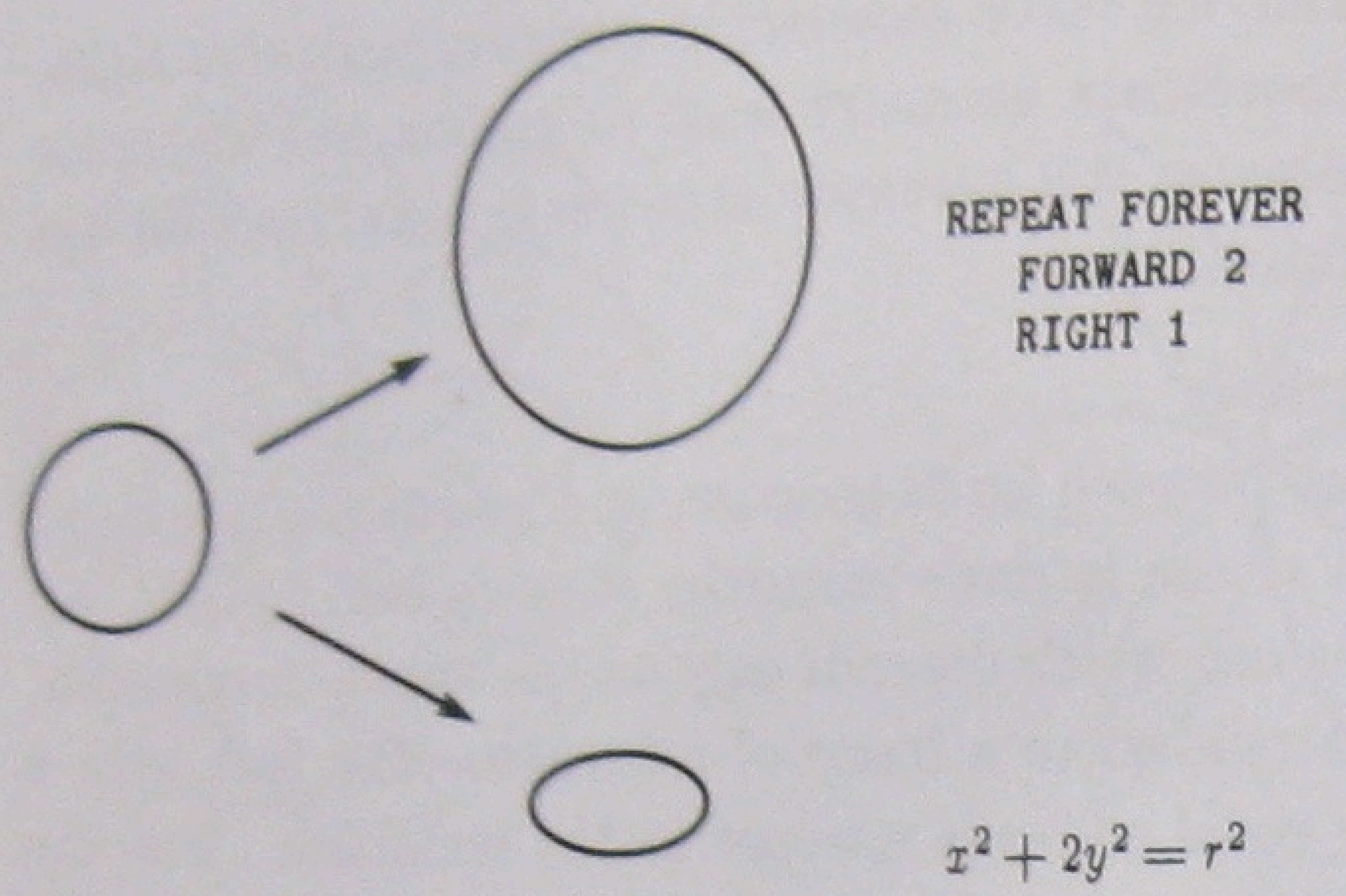
Intrinsic versus Extrinsic

One major difference between turtle geometry and coordinate geometry rests on the notion of the *intrinsic* properties of geometric figures. An intrinsic property is one which depends only on the figure in question, not on the figure's relation to a frame of reference. The fact that a rectangle has four equal angles is intrinsic to the rectangle. But the fact that a particular rectangle has two vertical sides is *extrinsic*, for an external reference frame is required to determine which direction is "vertical." Turtles prefer intrinsic descriptions of figures. For example, the turtle program to draw a rectangle can draw the rectangle in any orientation (depending on the turtle's initial heading), but the program `CARTESIAN.RECTANGLE` shown above would have to be modified if we did not want the sides of the rectangle drawn parallel to the coordinate axes, or one vertex at $(0, 0)$.

Another intrinsic property is illustrated by the turtle program for drawing a circle: Go `FORWARD` a little bit, turn `RIGHT` a little bit, and repeat this over and over. Contrast this with the Cartesian coordinate representation for a circle, $x^2 + y^2 = r^2$. The turtle representation makes it evident that the curve is everywhere the same, since the process that draws it does the same thing over and over. This property of the circle, however, is not at all evident from the Cartesian representation. Compare the modified program

```
TO CIRCLE
   REPEAT FOREVER
      FORWARD 2
      RIGHT 1
```

with the modified equation $x^2 + 2y^2 = r^2$. (See figure 1.7.) The drawing produced by the modified program is still everywhere the same, that is, a circle. In fact, it doesn't matter what inputs we use to `FORWARD` or `RIGHT` (as long as they are small). We still get a circle. The modified equation, however, no longer describes a circle, but rather an ellipse whose sides look different from its top and bottom. A turtle drawing an ellipse would have to turn more per distance traveled to get around its "pointy" sides



```
REPEAT FOREVER
   FORWARD 2
   RIGHT 1
```

$x^2 + 2y^2 = r^2$

Figure 1.7
Modifying the turtle program still produces a circle. Modifying the equation gives an ellipse.

than to get around its flatter top and bottom. This notion of "how pointy something is," expressed as the ratio of angle turned to distance traveled, is the intrinsic quantity that mathematicians call *curvature.* (See exercises 2 and 4.)

Local versus Global
The turtle representation of a circle is not only more intrinsic than the Cartesian coordinate description. It is also more *local;* that is, it deals with geometry a little piece at a time. The turtle can forget about the rest of the plane when drawing a circle and deal only with the small part of the plane that surrounds its current position. By contrast, $x^2 + y^2 = r^2$ relies on a large-scale, *global* coordinate system to define its properties. And defining a circle to be the set of points equidistant from some fixed point is just as global as using $x^2 + y^2 = r^2$. The turtle representation does not need to make reference to that "faraway" special point, the center. In later chapters we will see how the fact that the turtle does its geometry by feeling a little locality of the world at a time allows turtle geometry to extend easily out of the plane to curved surfaces.

Procedures versus Equations
A final important difference between turtle geometry and coordinate geometry is that turtle geometry characteristically describes geometric objects in terms of procedures rather than in terms of equations. In formulating turtle-geometric descriptions we have access to an entire range of procedural mechanisms (such as iteration) that are hard to capture in

the traditional algebraic formalism. Moreover, the procedural descriptions used in turtle geometry are readily modified in many ways. This makes turtle geometry a fruitful arena for mathematical exploration. Let's enter that arena now.

1.1.4 Some Simple Turtle Programs

If we were setting out to explore coordinate geometry we might begin by examining the graphs of some simple algebraic equations. Our investigation of turtle geometry begins instead by examining the geometric figures associated with simple procedures. Here's one of the simplest: Go FORWARD some fixed amount, turn RIGHT some fixed amount, and repeat this sequence over and over. This procedure is called POLY.

```
TO POLY SIDE ANGLE
   REPEAT FOREVER
      FORWARD SIDE
      RIGHT ANGLE
```

It draws shapes like those in figure 1.8.

POLY is a generalization of some procedures we've already seen. Setting the angle inputs equal to 90, 120, and 60, we get, respectively, squares, equilateral triangles, and regular hexagons. Setting the angle input equal to 1 gives a circle. Spend some time exploring POLY, examining how the figures vary as you change the inputs. Observe that rather than drawing each figure only once, POLY makes the turtle retrace the same path over and over. (Later on we'll worry about how to make a version of POLY that draws a figure once and then stops.)

Another way to explore with POLY is to modify not only the inputs, but also the program; for example (see figure 1.9),

```
TO NEWPOLY SIDE ANGLE
   REPEAT FOREVER
      FORWARD SIDE
      RIGHT ANGLE
      FORWARD SIDE
      RIGHT (2 * ANGLE)
```

(The symbol "*" denotes multiplication.) You should have no difficulty inventing many variations along these lines, particularly if you use such procedures as SQUARE and TRIANGLE as subprocedures to replace or supplement FORWARD and RIGHT.

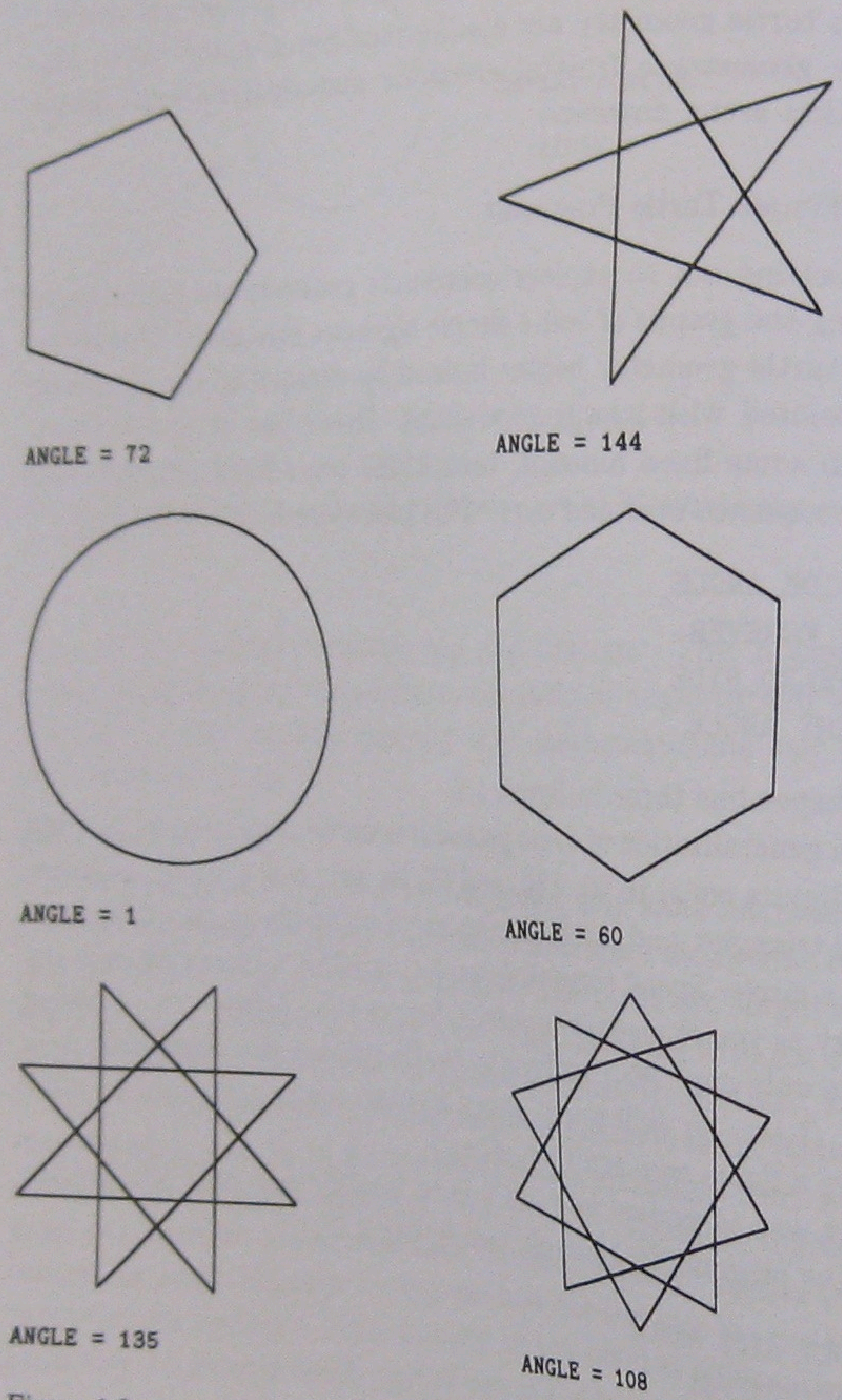


Figure 1.8
Shapes drawn by POLY.

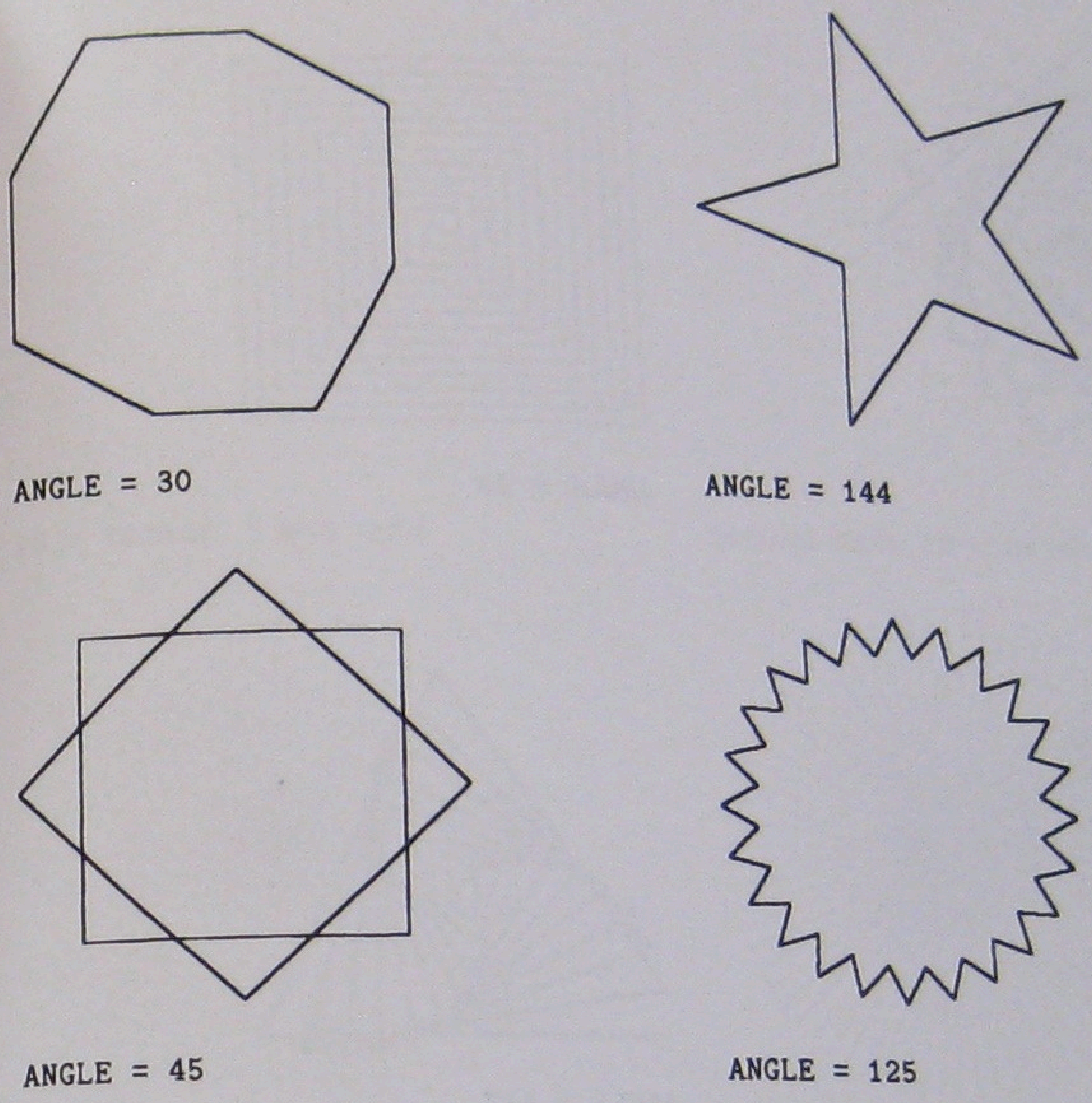


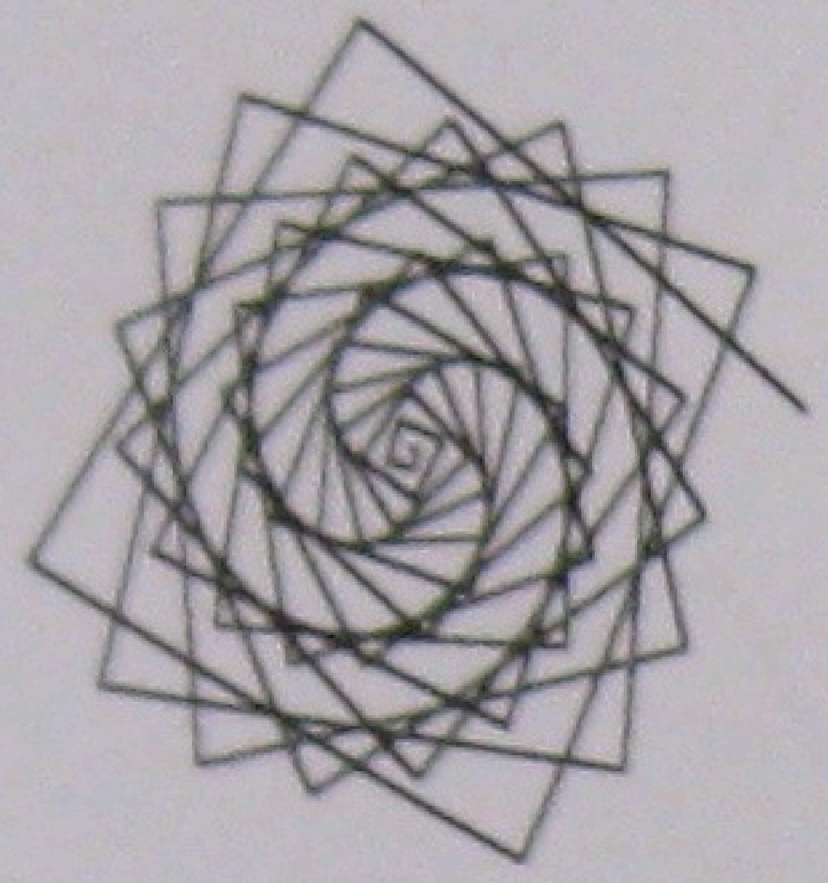
Figure 1.9
Shapes drawn by NEWPOLY.

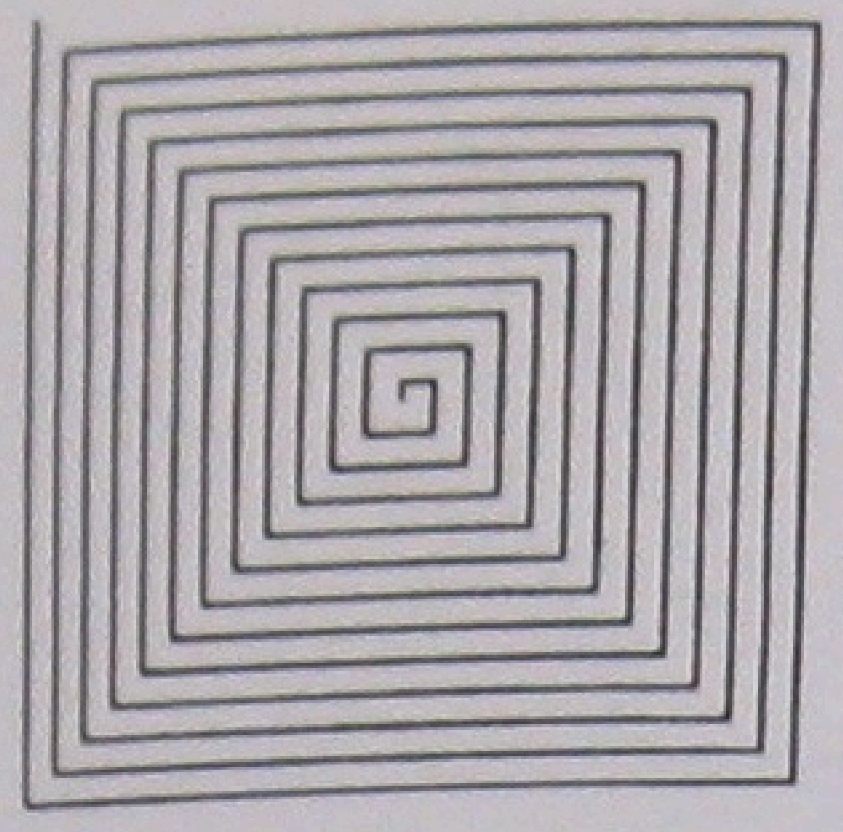
Recursion

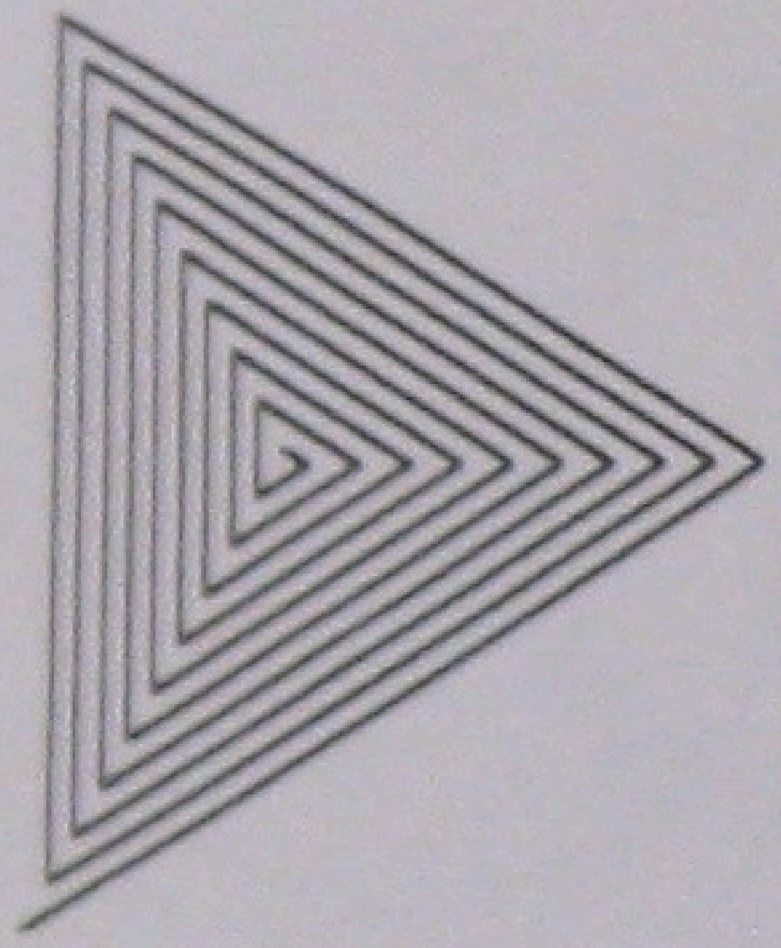
One particularly important way to make new procedures and vary old ones is to employ a program control structure called *recursion;* that is, to have a procedure use itself as a subprocedure, as in

```
TO POLY SIDE ANGLE
   FORWARD SIDE
   RIGHT ANGLE
   POLY SIDE ANGLE
```

The final line keeps the process going over and over by including "do POLY again" as part of the definition of POLY.


ANGLE = 95


ANGLE = 90


ANGLE = 120

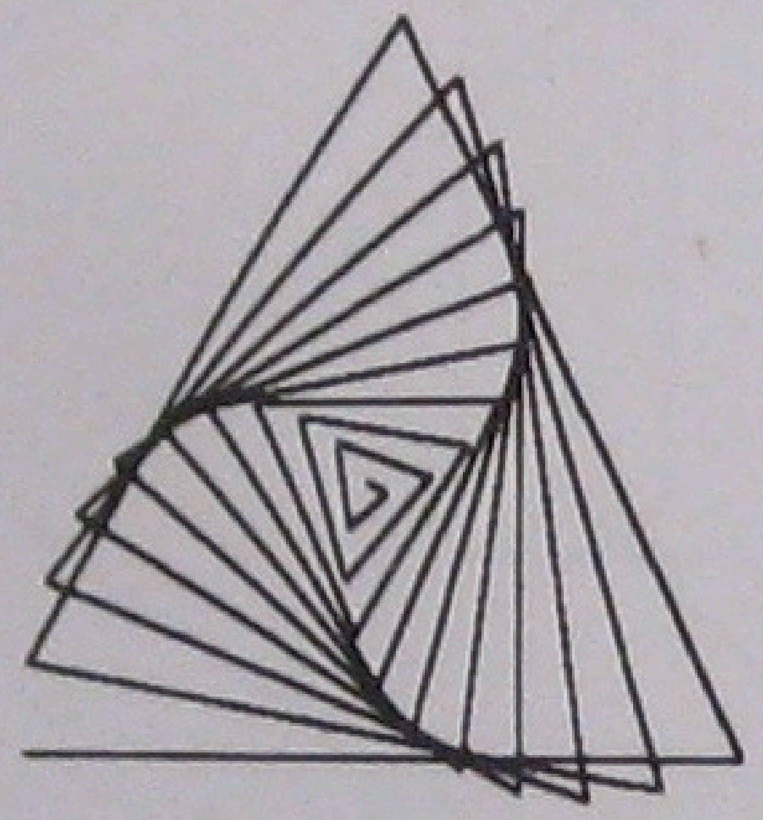
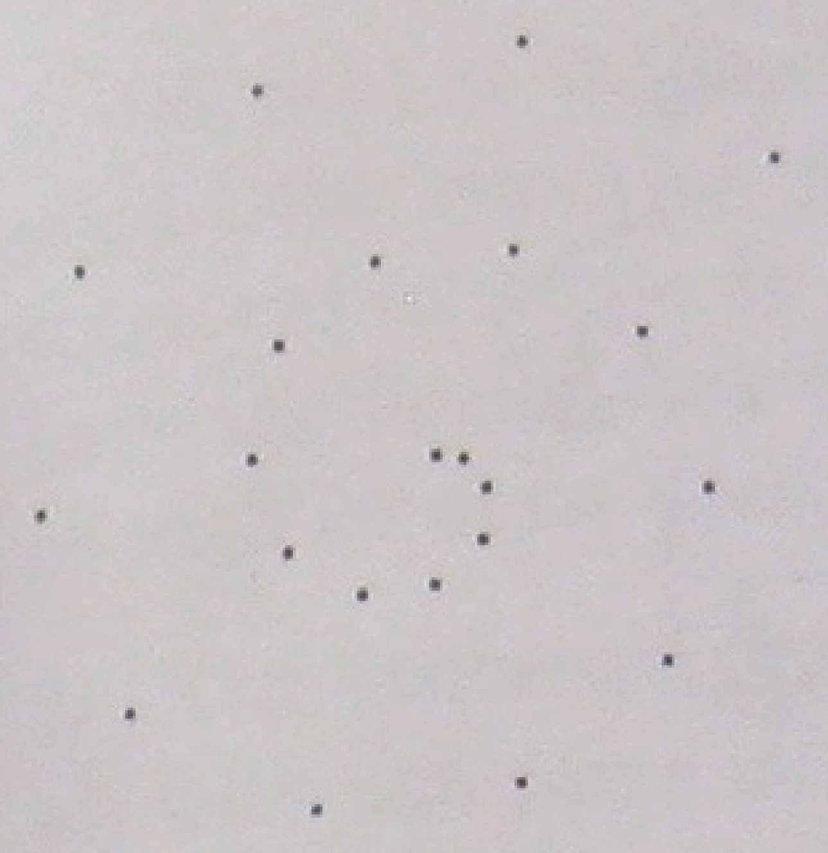

ANGLE = 117

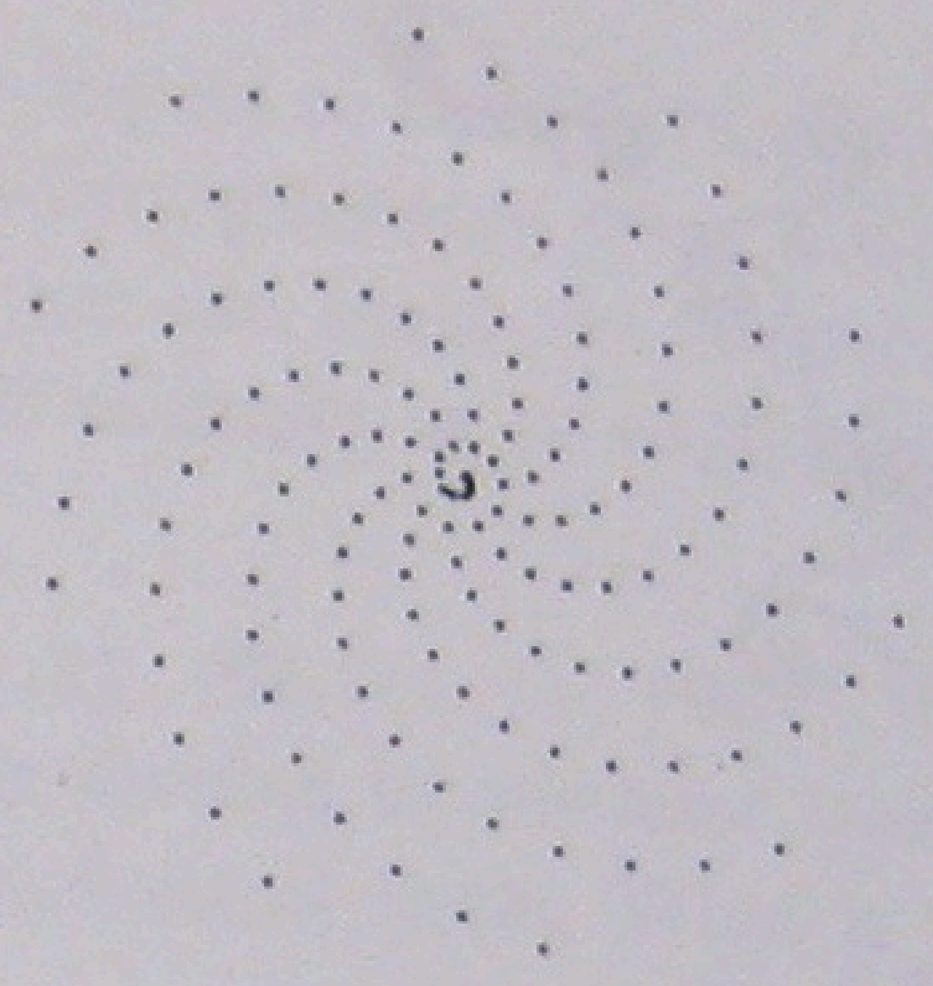
Figure 1.10
Shapes drawn by POLYSPI.

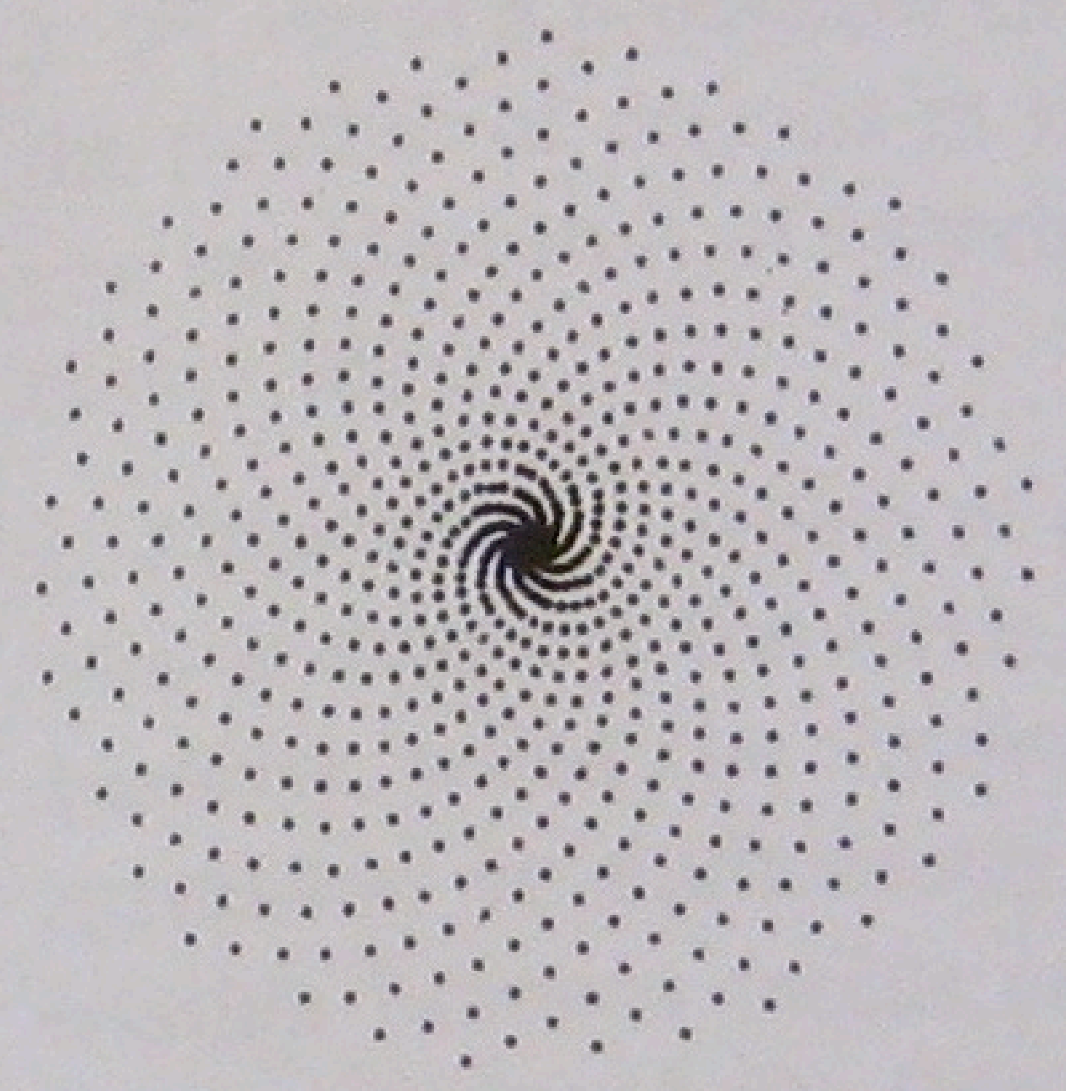
One advantage of this slightly different way of representing POLY is that it suggests some further modifications to the basic program. For instance, when it comes time to do POLY again, call it with different inputs:

```
TO POLYSPI SIDE ANGLE
   FORWARD SIDE
   RIGHT ANGLE
   POLYSPI (SIDE + 1, ANGLE)
```

Figure 1.10 shows some sample POLYSPI figures. Look carefully at how the program generates these figures: Each time the turtle goes FORWARD it goes one unit farther than the previous time.


10 × blowup, 1 arm right


natural scale, 10 arms left


$\frac{1}{4}$ reduction, 31 arms right, 41 left

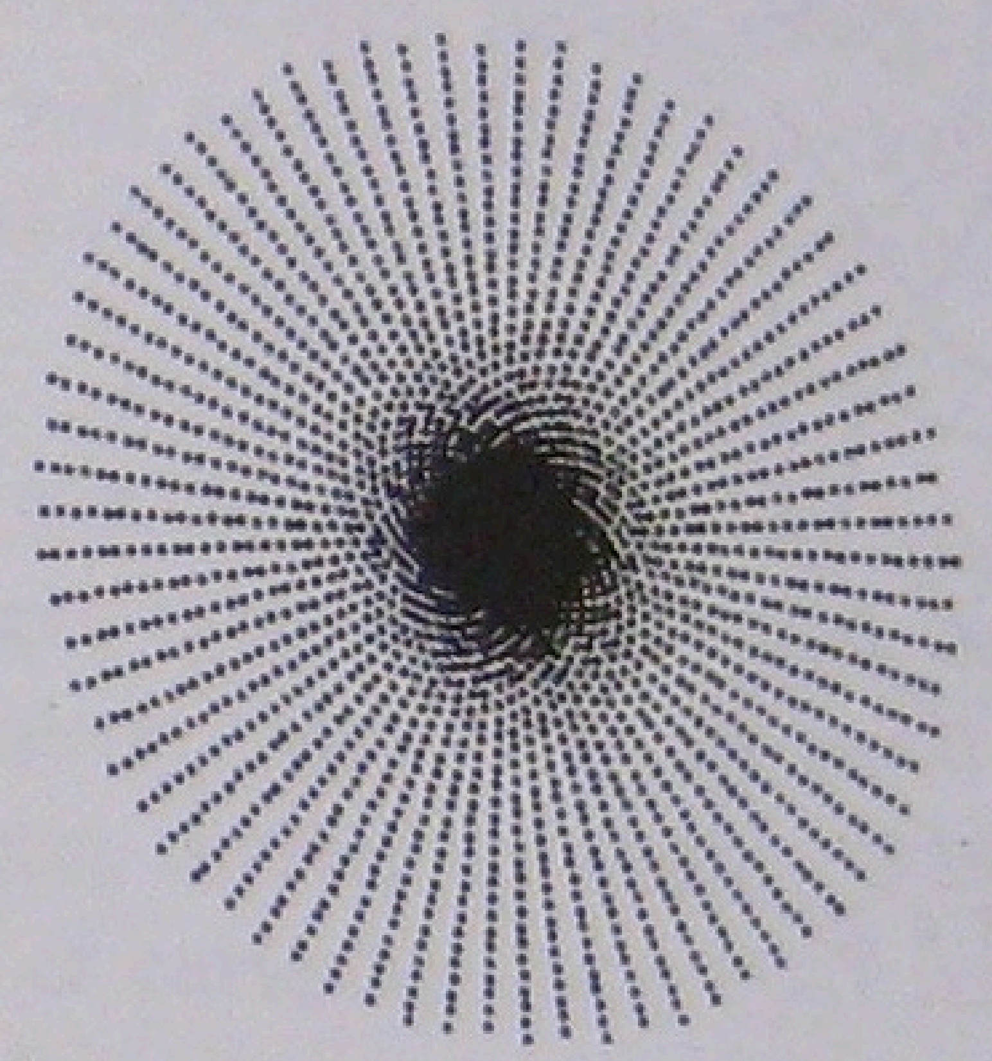
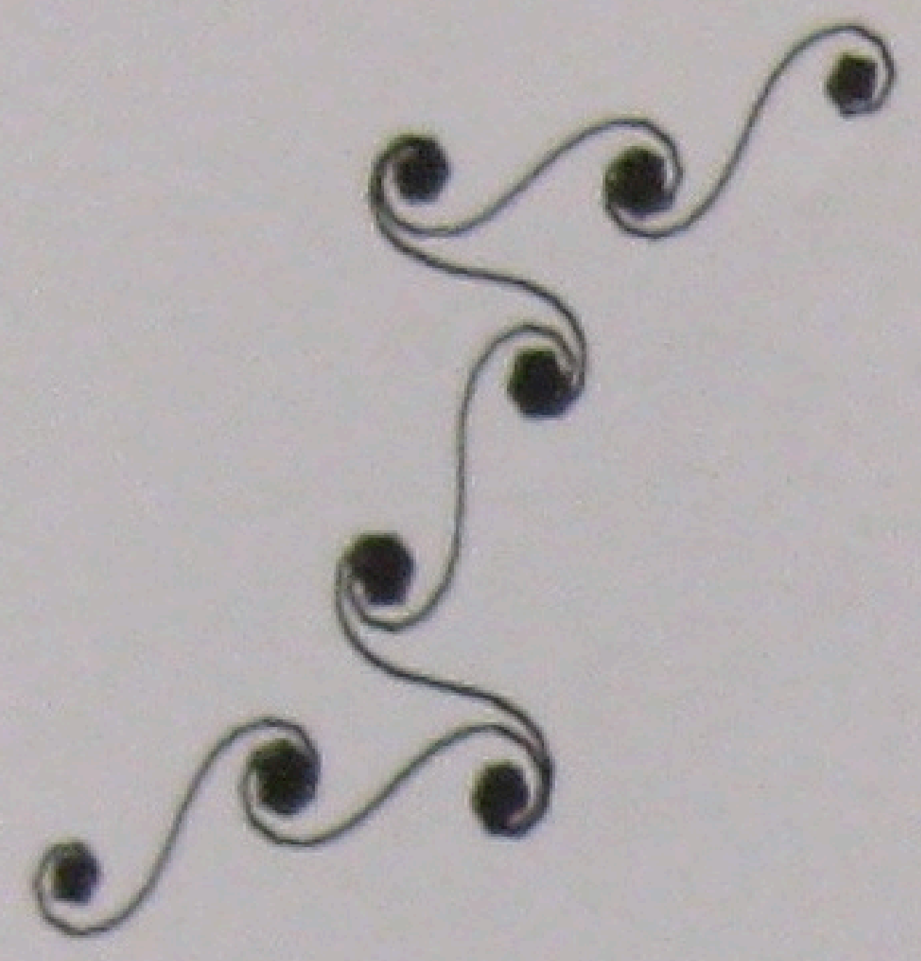

$\frac{1}{10}$ reduction, 72 arms straight

Figure 1.11
The vertices of a POLYSPI.

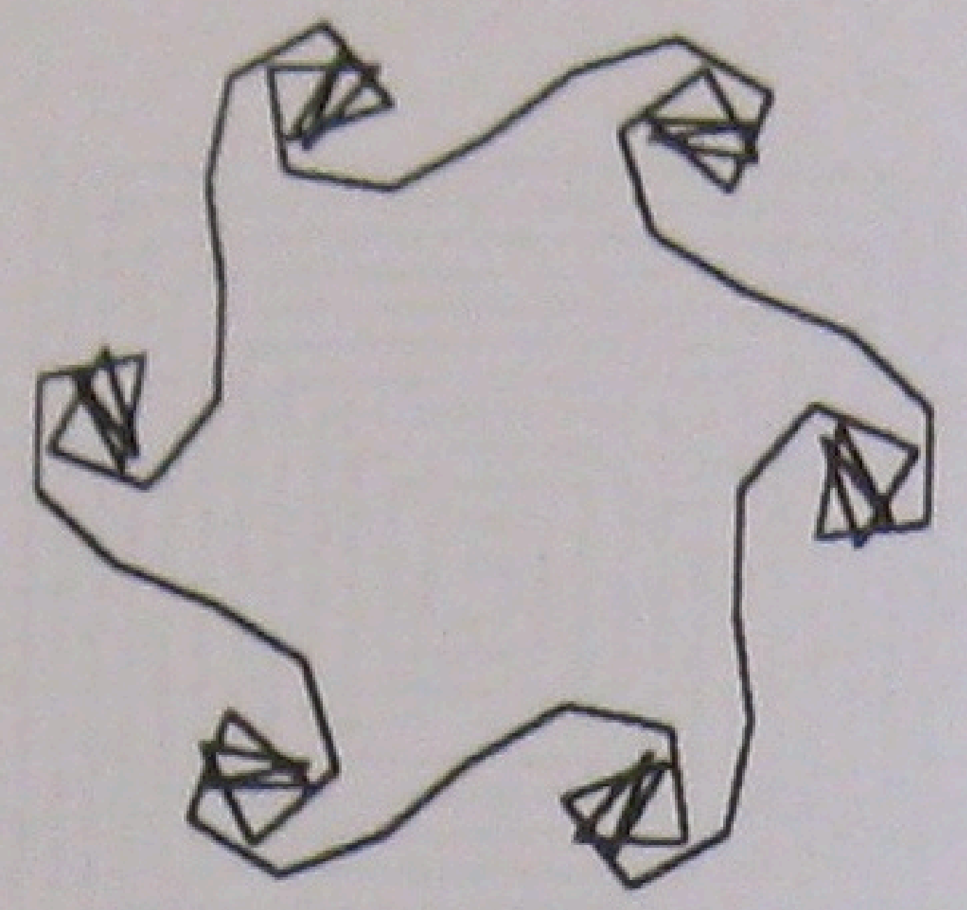
A more general form of POLYSPI uses a third input (INC, for increment) to allow us to vary how quickly the sides grow:

```
TO POLYSPI (SIDE, ANGLE, INC)
   FORWARD SIDE
   RIGHT ANGLE
   POLYSPI (SIDE + INC, ANGLE, INC)
```

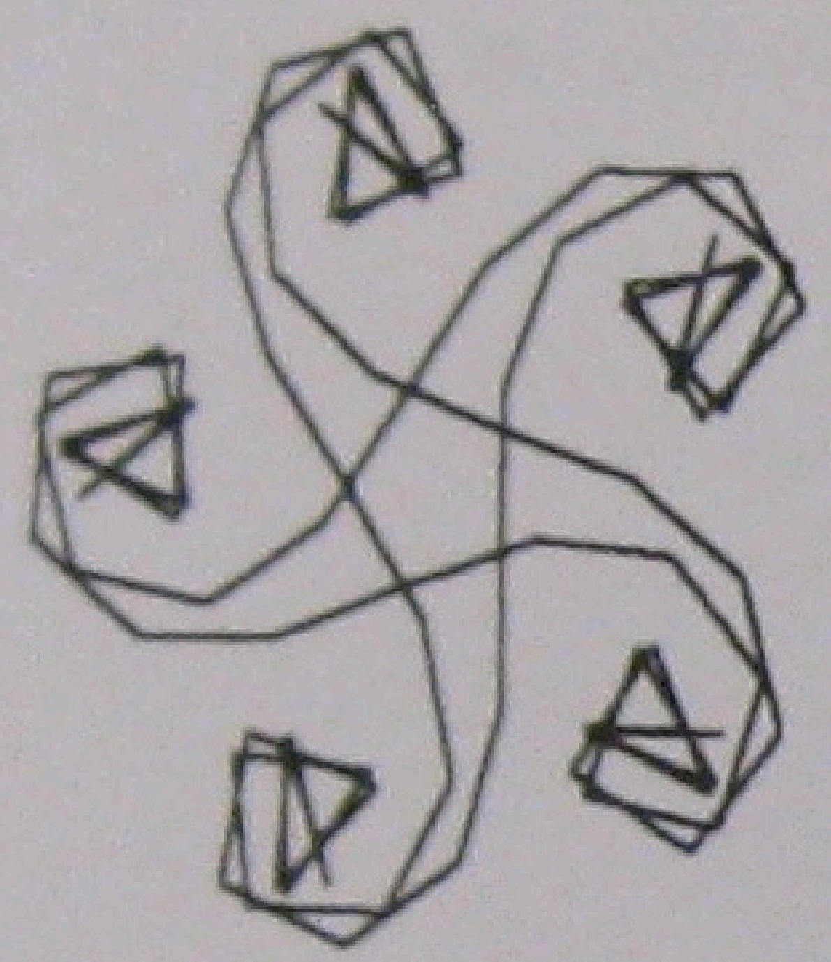
In addition to trying POLYSPI with various inputs, make up some of your own variations. For example, subtract a bit from the side each time, which will produce an inward spiral. Or double the side each time, or divide it by two. Figure 1.11 illustrates a pattern made drawing only the vertices of POLYSPI, shown at four scales of magnification (see exercise 13).



ANGLE = 0
INCREMENT = 7



ANGLE = 40
INCREMENT = 30



ANGLE = 2
INCREMENT = 20

Figure 1.12
Examples of INSPI.

Another way to produce an inward spiral (curve of increasing curvature) is to increment the angle each time:

```
TO INSPI (SIDE, ANGLE, INC)
   FORWARD SIDE
   RIGHT ANGLE
   INSPI (SIDE, ANGLE + INC, INC)
```

Run INSPI and watch how it works. The turtle begins spiraling inward as expected. But eventually the path begins to unwind as the angle is incremented past 180°. Letting INSPI continue, we find that it eventually produces a symmetrical closed figure which the turtle retraces over and over as shown in figure 1.12. You should find this surprising. Why should this succession of FORWARDs and RIGHTs bring the turtle back precisely to its starting point, so that it will then retrace its own path? We will see in the next section that this closing phenomenon reflects the elegant mathematics underlying turtle geometry.

Exercises for Section 1.1

1. We said in the text that when the inputs to the POLY procedure are small, the resulting figure will be indistinguishable from a circle. Do some experiments to see how large you can make the inputs and still have the figure look like a circle. For example, is an angle of 20° small enough to draw acceptable circles?

2. The sequence of figures POLY(2,2), POLY(1,1), POLY(.5,.5), ... all with the same curvature (turning divided by distance traveled), approaches "in the limit" a true mathematical circle. What is the radius of the circle? [HA]

3. [P] Write a procedure that draws circular arcs. Inputs should specify the number of degrees in the arc as well as the size of the circle. Can you use the result of exercise 2 so that the size input is the radius of the circle? [A]

4. Although the radius of a circle is not "locally observable" to a turtle who is drawing the circle, that length is intimately related to a local quantity called the "radius of curvature," defined to be equal to 1 ÷ curvature, or equivalently, to distance divided by angle. What is the relation between radius and radius of curvature for a POLY with small inputs as above? Do this when angle is measured in radians as well as in degrees. [A]

5. [P] Construct some drawings using squares, rectangles, triangles, circles, and circular arc programs.

6. [P] Invent your own variations on the model of POLYSPI and INSPI.

7. How many different 9-sided figures can POLY draw (not counting differences in size or orientation)? What angle inputs to POLY produce these figures? How about 10-sided figures? [A]

8. [PD] A rectangle is a square with two different side lengths. More generally, what happens to a POLY that uses two different side lengths as in the following program?

```
TO DOUBLEPOLY (SIDE1, SIDE2, ANGLE)
   REPEAT FOREVER
      POLYSTEP SIDE1 ANGLE
      POLYSTEP SIDE2 ANGLE
```