This PDF includes a chapter from the following book:

Turtle Geometry

The Computer as a Medium for Exploring Mathematics

© 1986 MIT

License Terms:

Made available under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International Public License

https://creativecommons.org/licenses/by-nc-nd/4.0/

OA Funding Provided By:

The open access edition of this book was made possible by generous funding from Arcadia—a charitable fund of Lisbet Rausing and Peter Baldwin.

The title-level DOI for this work is:

doi:10.7551/mitpress/6933.001.0001

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

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*.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

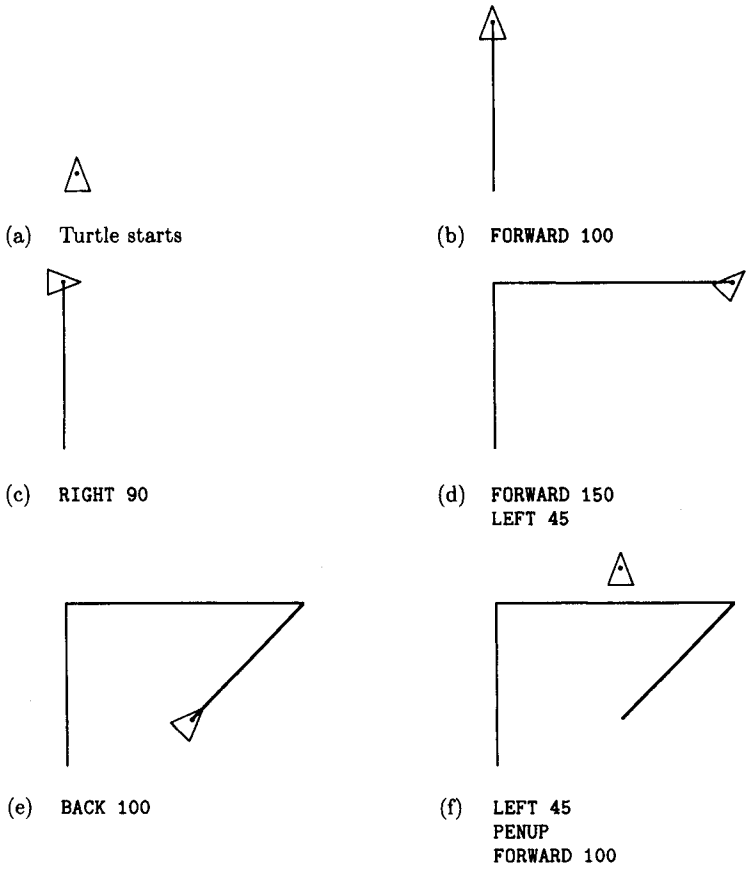


Figure 1.1
A sequence of turtle commands.

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.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

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
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

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
```

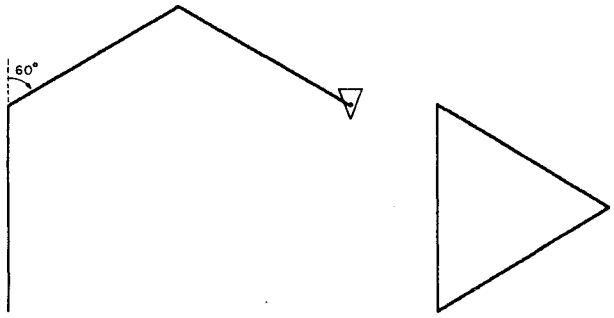
To use the RECTANGLE procedure you must specify its two inputs, for example, RECTANGLE 100 50.

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.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022




```
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.

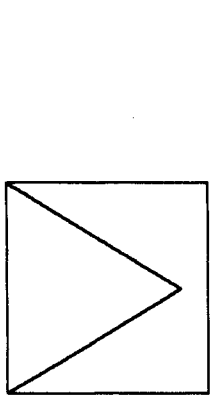
1.1.2 Drawing with the Turtle

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

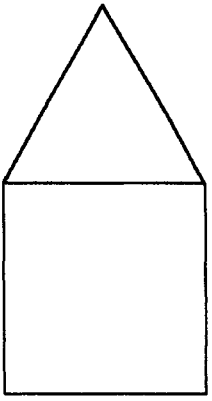
```
TO TRY.ANGLE SIZE
   REPEAT 3
      FORWARD SIZE
      RIGHT 60
```

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

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022



```
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.

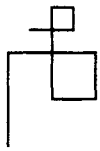
the usual interior angle. Turtle angle has many advantages over interior angle, as you will see.

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.

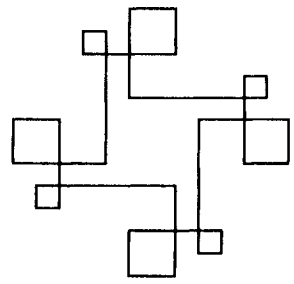
Using procedures and subprocedures is also a good way to create abstract designs. Figure 1.4 shows how to create elaborate patterns by rotating a simple "doodle."

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

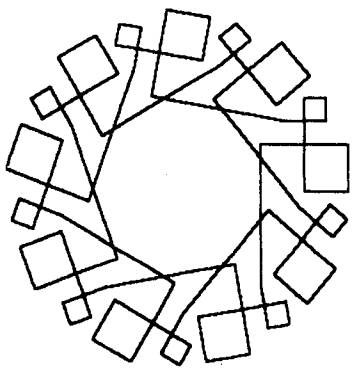
Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022



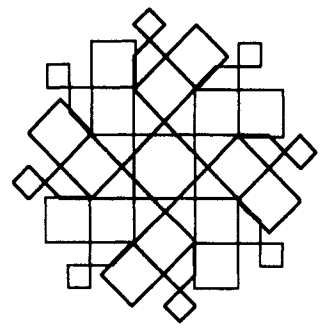
```
TO THING
   FORWARD 100
   RIGHT 90
   FORWARD 100
   RIGHT 90
   FORWARD 50
   RIGHT 90
   FORWARD 50
   RIGHT 90
   FORWARD 100
   RIGHT 90
   FORWARD 25
   RIGHT 90
   FORWARD 25
   RIGHT 90
   FORWARD 50
```



```
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.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

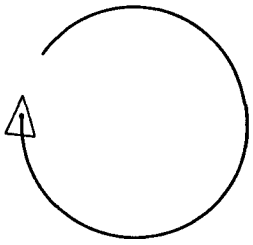


Figure 1.5
FORWARD 1, RIGHT 1, repeated draws a circular arc.

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
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

```
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.)

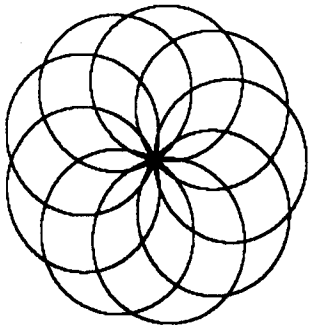
1.1.3 Turtle Geometry versus Coordinate Geometry

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

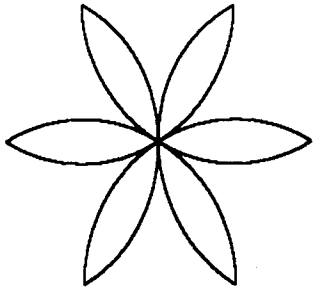
```
TO CARTESIAN.RECTANGLE (WIDTH, HEIGHT)
   SETXY (WIDTH, 0)
   SETXY (WIDTH, HEIGHT)
   SETXY (0, HEIGHT)
   SETXY (0, 0)
```

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

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

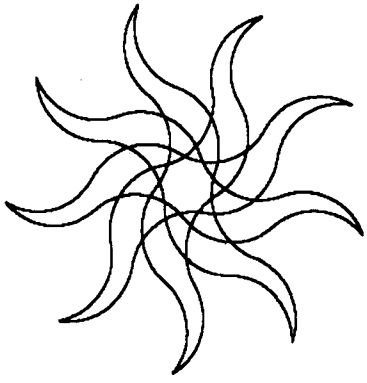


```
TO CIRCLES
   REPEAT 9
      ARCR 1 360
      RIGHT 40
```



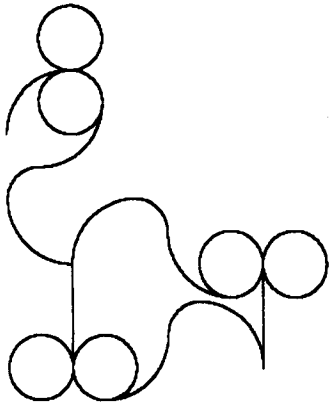
```
TO PETAL SIZE
   ARCR SIZE 60
   RIGHT 120
   ARCR SIZE 60
   RIGHT 120

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



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

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



MONSTER

Figure 1.6
Some shapes that can be made using arcs.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

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

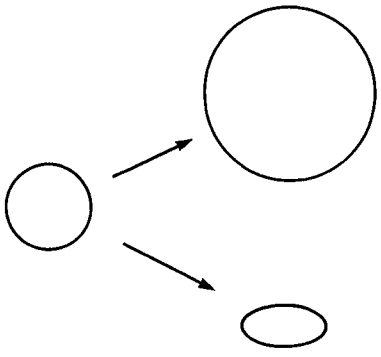
One major difference between turtle geometry and coordinate geometry rests on the notion of the *intrinsic* properties of geometric figures. An intrinsic property is one which depends only on the figure in question, not on the figure's relation to a frame of reference. The fact that a rectangle has four 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

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022



```
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

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

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.

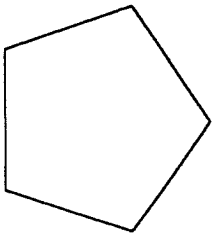
POLY is a generalization of some procedures we've already seen. Setting the angle inputs equal to 90, 120, and 60, we get, respectively, squares, equilateral triangles, and regular hexagons. Setting the angle input equal to 1 gives a circle. Spend some time exploring POLY, examining how the figures vary 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),

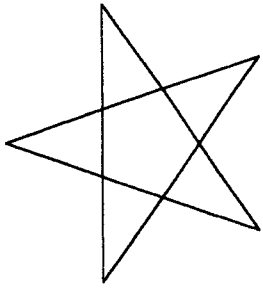
```
TO NEWPOLY SIDE ANGLE
   REPEAT FOREVER
      FORWARD SIDE
      RIGHT ANGLE
      FORWARD SIDE
      RIGHT (2 * ANGLE)
```

(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.

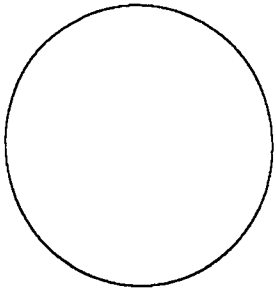
Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022



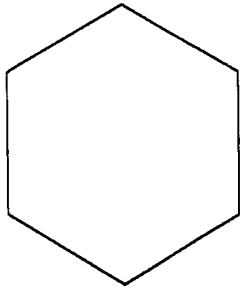
ANGLE = 72



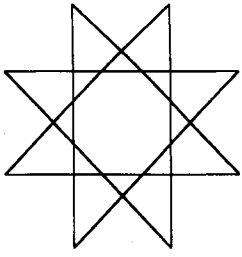
ANGLE = 144



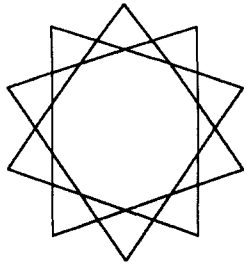
ANGLE = 1



ANGLE = 60



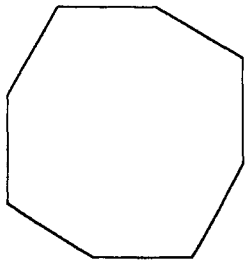
ANGLE = 135



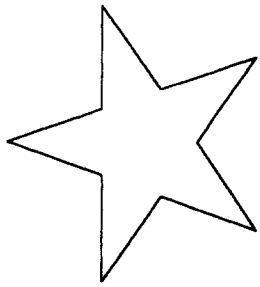
ANGLE = 108

Figure 1.8
Shapes drawn by POLY.

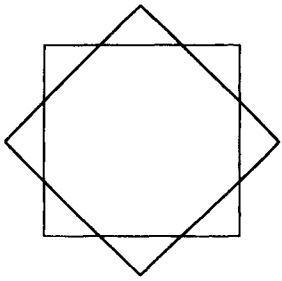
Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022



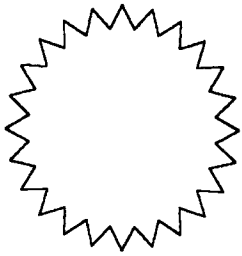
ANGLE = 30



ANGLE = 144



ANGLE = 45



ANGLE = 125

Figure 1.9
Shapes drawn by NEWPOLY.

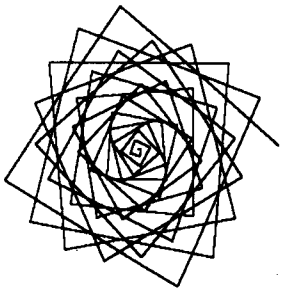
Recursion

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

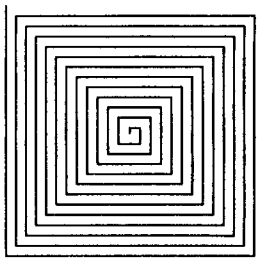
```
TO POLY SIDE ANGLE
   FORWARD SIDE
   RIGHT ANGLE
   POLY SIDE ANGLE
```

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

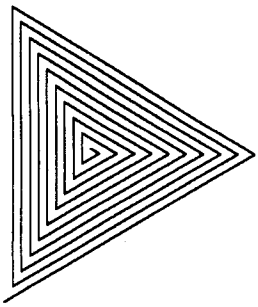
Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022



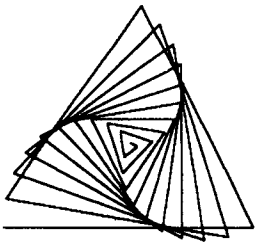
ANGLE = 95



ANGLE = 90



ANGLE = 120



ANGLE = 117

Figure 1.10
Shapes drawn by POLYSPI.

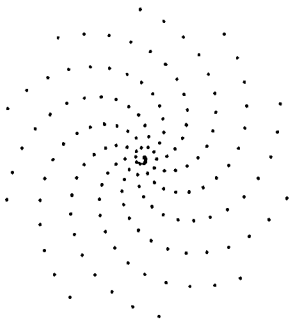
One advantage of this slightly different way of representing POLY is that it suggests some further modifications to the basic program. For instance, when it comes time to do POLY again, call it with different inputs:

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

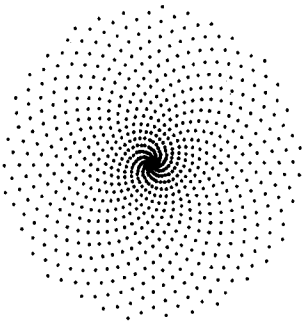
Figure 1.10 shows some sample POLYSPI figures. Look carefully at how the program generates these figures: Each time the turtle goes FORWARD it goes one unit farther than the previous time.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

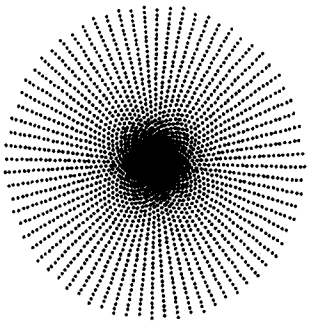
$10 \times$ blowup, 1 arm right



natural scale, 10 arms left



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



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

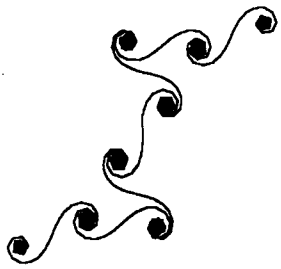
Figure 1.11
The vertices of a POLYSPI.

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

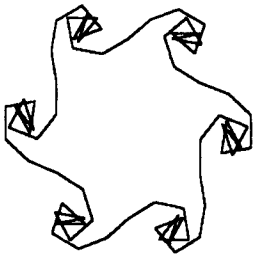
```
TO POLYSPI (SIDE, ANGLE, INC)
   FORWARD SIDE
   RIGHT ANGLE
   POLYSPI (SIDE + INC, ANGLE, INC)
```

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).

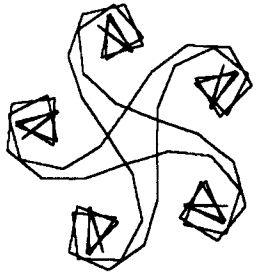
Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022



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.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

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 $1 \div$ 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
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

In particular, how does the symmetry of DOUBLEPOLY relate to that of POLY with the same ANGLE input? [HA]

9. [D] Which encloses the larger area—POLY(5,5) or POLY(6,6)? [HA]

10. [P] Find inputs to INSPI that give a nonclosed figure. Can you give a convincing argument that the figure is really nonclosed rather than, say, a closed figure too big to fit on the display screen? [A]

11. [P] If the display system you are using allows "wraparound," you can get some interesting effects by trying POLYs with very large sides. Explore these figures. [H]

12. There are three kinds of "interchanges" we can perform on turtle programs: interchanging RIGHT and LEFT, interchanging FORWARD and BACK, and (for programs that terminate) reversing the sequence of instructions. Describe in geometric terms the effect of each of these operations, both by itself and in combination with the others. Start with the class of programs that close (return the turtle to its initial position and heading). [HA]

13. [P] The pattern made by the vertices of POLYSPI can be an interesting object of study. The dots seem to group into various "arms," either straight or curving left or right. To draw these patterns, you can use the procedures

```
TO SPIDOT ANGLE
   SUBSPIDOT 0 ANGLE

TO SUBSPIDOT SIDE ANGLE
   FORWARD SIDE
   DOT
   RIGHT ANGLE
   SUBSPIDOT (SIDE + 1, ANGLE)

TO DOT
   PENDOWN
   FORWARD 1
   BACK 1
   PENUP
```

For example, predict what you will see between SPIDOT 90, which has four arms, and SPIDOT 120, which has three. Can you explain the sequence of figures you actually do see? Figure 1.11 shows how the figure

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

drawn by the same SPIDOT program seems to have different numbers of spiral arms when viewed at different scales of magnification, which can be accomplished by changing the increment to SIDE in SUBSPIDOT. Study this phenomenon. [H]

14. [P] Suppose we have a function called RANDOM that outputs a random digit (0 through 9). Play around with the procedure

```
TO RANDPOLY SIDE ANGLE
   REPEAT FOREVER
      IF RANDOM = 0 THEN PENDOWN
                ELSE PENUP
      FORWARD SIDE
      RIGHT ANGLE
```

Use this program as the basis for some psychology experiments. For instance, what is the average number of sides that must be drawn before people can recognize which POLY it is?

15. [D] Find some local and intrinsic way to describe an ellipse. Write a program that makes the turtle draw ellipses, where the inputs specify the size and eccentricity of the ellipse. [A]

1.2 POLYs and Other Closed Paths

This section develops some general theorems about turtle programs by studying one of the simplest of them, POLY—which, when it closes, exhibits clearly some properties shared by all closed paths, no matter how complicated. Even when POLY doesn't close, it can serve as a model that clarifies symmetry and other important properties of a very general class of programs. Careful and patient study of such a simple program will be richly rewarded.

1.2.1 The Closed-Path Theorem and the Simple-Closed-Path Theorem

You have probably already noticed that POLY with an angle input of $360/n$ draws a regular n-sided polygon. But it is not always true that (number of sides)$\times$(angle) $= 360$. If you try running POLY with an angle of 144 you will see that it draws a five-pointed star, and $5 \times 144 = 720$, not 360. Noticing that 720 is exactly twice 360 might lead us to guess the following formula:

(number of sides) $\times$ (angle) $= 360 \times$ (an integer).

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

It's not hard to see why this formula is true. The number of sides times the angle is precisely the *total turning* done by the turtle in walking once around the figure—the net change in heading. If the path is to close legitimately, and not just cross itself, then the turtle must end its trip with the same heading it started out with. Thus, the total turning must be some multiple of 360°.

Total turning is the central concept here. It certainly need not be restricted to POLY. One can imagine any turtle program keeping a running count of its turning, adding in RIGHTs and subtracting LEFTs. Because only RIGHTs and LEFTs change heading, this total turning is always exactly the total change in heading. In particular, if the path is a *closed path* (one which restores the turtle's initial position and heading), we can be confident that the net turning (= change of heading) is a multiple of 360°. This gives us our first turtle-geometric theorem:

Closed-Path Theorem The total turning along any closed path is an integer multiple of 360°.

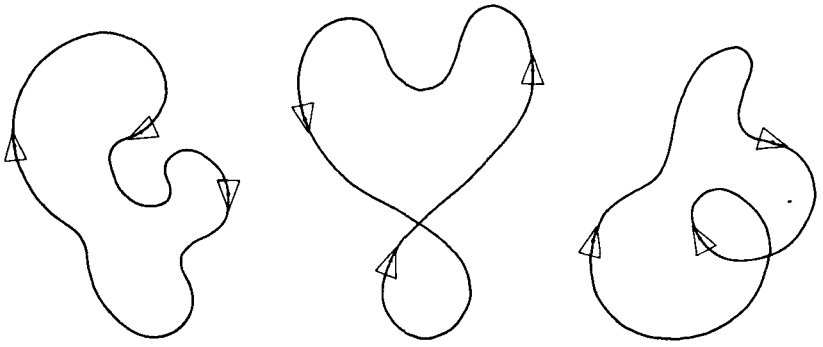
Total turning is an intrinsic property of a path. It does not depend on where the path starts, or how it is oriented with respect to "vertical." The total turning of a closed path is frequently summarized simply by the particular integer that multiplies 360. That integer is called the *rotation number* of the path. As an exercise, follow the turtle around the sample paths in figure 1.13 and compute the rotation numbers.

Does your experience with POLY suggest an improvement to the closed-path theorem? A little experimentation should convince you that there are two essentially different classes of POLY paths: simple polygons (such as squares, triangles, and hexagons); and star polygons (such as five-pointed stars), which are characterized by the fact that the paths cross themselves. The simple polygons always appear to have total turning equal to $+360°$ or $-360°$, depending upon the direction in which the turtle traverses the path. The star polygons, however, always have total turning different from $\pm 360°$.

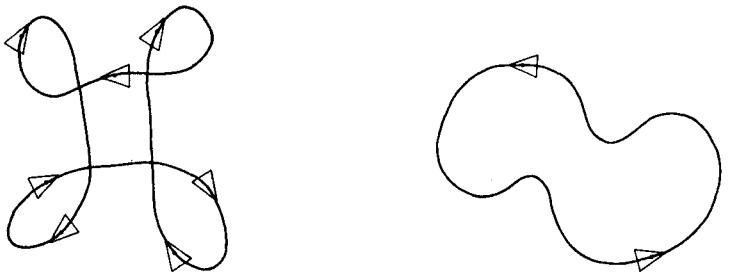
One wonders if this experimental correlation has general significance. It is not hard to prove its validity for POLYs (see exercise 11 below). But the more important conjecture involves generalizing from POLYs to any *simple closed path* (a closed path that does not cross itself):

Simple-Closed-Path Theorem The total turning in a simple closed path is 360° (to the right or to the left). That is to say, the rotation number of any simple closed path is ± 1.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022



rotation number 1 rotation number 0 rotation number 2



rotation number 3 rotation number —1

Figure 1.13
Rotation numbers of closed paths.

Take a look at some examples of simple closed paths to convince yourself of the plausibility of this theorem, which is difficult to prove rigorously. We will return to it later, in chapter 4. For now you should note that this theorem says that there is a relation between two very different aspects of a closed path—the turning and the crossing points. That makes it considerably less obvious than the closed-path theorem, but also much more powerful. We give one example of the power here and several more in the exercises.

The simple-closed-path theorem says that the sum of the exterior angles of any simple polygon is 360°. For triangles, we can rewrite this in terms of the three interior angles A, B, and C to get

$$(180 - A) + (180 - B) + (180 - C) = 360,$$

and thus

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

$(A + B + C)$ = the sum of interior angles $= 3 \times 180 - 360 = 180$.

So, as a corollary of the simple-closed-path theorem, we have derived the familiar result that the interior angles of a triangle must sum to 180°. (Exercises 6–9 detail some other applications of the simple-closed-path theorem.)

1.2.2 The POLY Closing Theorem

The POLY procedures we've written so far, iterative and recursive, have one fault: They never stop. That makes it generally impossible to use them as subprocedures in more complicated programs. Moreover, the "inefficiency" of a drawing program that doesn't know when it is done may simply offend one's sensibilities. The problem of making a POLY program that stops is a mathematical one with two fundamentally different approaches.

The *global approach* is as follows: Sit back and look ahead. Given an ANGLE input, compute how many times the turtle must run the basic POLY step, FORWARD SIDE, RIGHT ANGLE, before the path closes and starts again. Then you need only repeat the POLY step that many times. The *local approach* revolves around questions like the following: How can the turtle know, as it is walking along, when it is done? What clue can the turtle be watching for? We will take the second approach here, as it turns out to be simpler. The first approach, however, is mathematically rich and is pursued in section 1.4.

Consider: How could a turtle, while walking along drawing a POLY, know when the figure has been completed? (A computer turtle cannot see the lines it is drawing.) Thinking locally, the turtle knows only two things, position and heading. Neither of these is truly local, for to measure them usually involves a coordinate system. But the one locally computable quantity we know about—total turning—can do the trick. The closed-path theorem says that if the path closes, then total turning must be a multiple of 360°. How about the converse: If the total turning reaches a multiple of 360°, will the path be closed? This is not true for turtle paths in general, but it is true for POLY:

POLY Closing Theorem A path drawn by the POLY procedure will close precisely when the total turning reaches a multiple of 360°.

There is one bug in this theorem, one exceptional case: If the angle of the POLY is equal to 0 then the turtle just walks off along a straight line. The path never closes, even though at every point the total

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

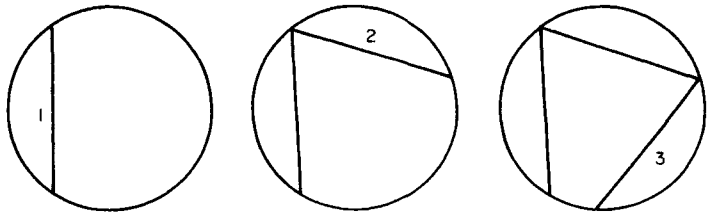


Figure 1.14
POLY lays down a sequence of equal chords.

turning is 0, a perfectly good multiple of 360. But this exceptional case, FORWARD SIDE, RIGHT 0, is transparent enough so that we can just leave it out of consideration in most instances. Any multiple of 360° will, of course, have the same effect as a turn of 0.

We'll outline two different proofs of the POLY closing theorem.

Sketch of Proof 1 Have you noticed the important fact that the vertices of POLY lie on a circle? (Everything about POLY seems to be circular!) We leave the proof of this geometric fact to you in exercise 2. Using this fact, one can redescribe POLY as the sequential laying down of fixed-length chords on a fixed circle as shown in figure 1.14. The point is that there is only one chord of the required length that can be produced by the turtle starting at any given heading. (Actually there are two, but one of them has the wrong sense—the turtle would turn off the circle after traversing the chord.) Thus, whenever the turtle returns to its initial heading (total turning = any multiple of 360°) it will be about to retrace the first chord and so should stop. Notice how this proof breaks down for the exceptional case FORWARD SIDE, RIGHT 0. The turtle must do some turning or else the vertices will lie on a straight line rather than on a circle.

An alternative proof is inspired less by geometry and more by ideas from the theory of computation. It proceeds as follows.

Sketch of Proof 2 Assume that we have a turtle following a POLY procedure, and that at some time the turtle returns to its initial heading (heading change = a multiple of 360°) but not to its initial position. We will show that this assumption leads to a contradiction. (The trick of the proof is to show that the turtle must walk off to infinity in some

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

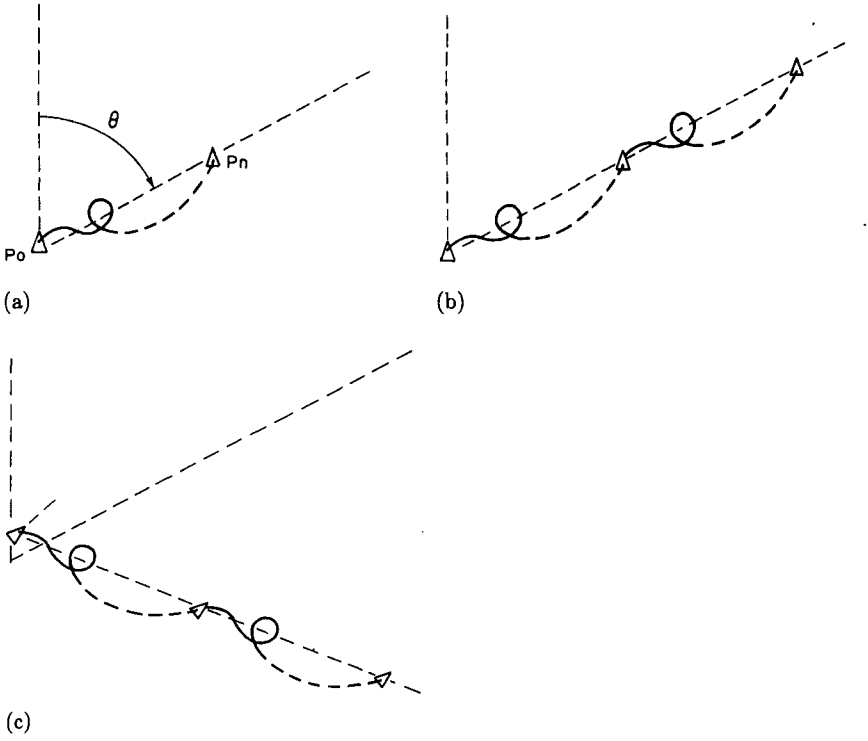


Figure 1.15
The POLY closing theorem. (a) Suppose the turtle returns to his initial heading, but not his initial position. (b) n more steps must do the same thing again. (c) From a new heading (after one POLY step), chunks of n steps carry the turtle away on a new line.

direction. Then, by regrouping the sequence of commands, we'll show that the turtle runs off to infinity in a different direction.)

By assumption, the turtle returns to its initial heading after some number (say, n) repetitions of the POLY step. (Notice that n cannot be 1 if we neglect the exceptional case ANGLE = 0.) Draw a dotted line connecting the turtle's initial position p_0 to its position p_n after n repetitions. This line makes some angle θ with the turtle's initial heading (figure 1.15a).

Now let the turtle continue for n more repetitions of the POLY step. Since the turtle starts out from p_n with the same heading it had when it started at p_0, the effect of n more POLY steps will be to do the same

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

thing again, moving the turtle farther out along the same line, and again bring it back to the initial heading (figure 1.15b). Continuing with n more repetitions, and n more, and so on, we see that the turtle must run off infinitely far in the direction of the dotted line. Moreover, at no point can the turtle's path stray very far from the line, since the turtle must get back to it at the end of every n POLY steps.

Now let's return the turtle to the initial state and run the POLY step for one iteration. We will now see the turtle at a new position p_1 with a different heading. If we continue with n repetitions from here, the turtle will end up on a new dotted line that lies at angle θ to this new heading. (figure 1.15c).

But the problem is obvious now. Running another sequence of n, then another and another, forces the turtle off infinitely far along this new line. But the turtle cannot remain close to both dotted lines as it marches off to infinity. This contradiction means that our assumption that the turtle does not come back to the initial position must have been wrong. This completes the proof.

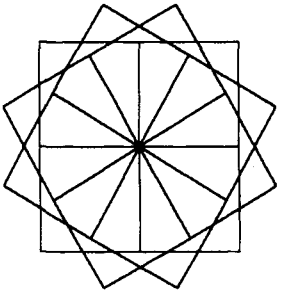
This second proof demonstrates an important computational strategy: Divide a process into meaningful chunks (for example, the parts of the POLY between equal headings), then pay close attention to the net action of the chunks. Structuring a complex program as a group of subprocedures illustrates the same strategy.

Here finally is our POLY with stop rule:

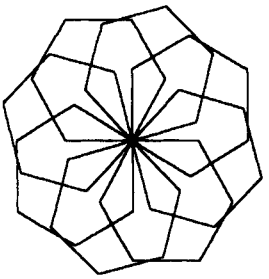
```
TO POLYSTOP SIDE ANGLE
   TURN ← 0
   REPEAT
      FORWARD SIDE
      RIGHT ANGLE
      TURN ← (TURN + ANGLE)
   UNTIL REMAINDER (TURN, 360) = 0
```

Note the use of the new symbol ←, which means "assign to the variable on the left the value given on the right." The procedure REMAINDER is a function that computes the value of its first input modulo its second input. The program also makes use of the iteration construct "REPEAT ... UNTIL (some condition)", which keeps repeating the indented portion until the condition is true (and always does the indented part at least once).

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022



POLYROLL 100 90 30



POLYROLL 100 60 45

Figure 1.16
Examples of POLYROLL.

This program allows us to use POLYs as building blocks in more complex figures; for example (see figure 1.16),

```
TO POLYROLL SIDE ANGLE1 ANGLE2
   REPEAT FOREVER
      POLYSTOP SIDE ANGLE1
      RIGHT ANGLE2
```

Exercises for Section 1.2

1. The simple-closed-path theorem has a serious bug as it stands. It purports to give the precise multiple of 360 that describes the total turning for a set of paths. Unfortunately, one can insert a step, RIGHT 360, that does not change the path at all, yet changes the multiple of 360 given by total turning. These gratuitous 360s must be pruned from the program before the theorem can hold. However, the pruning can be somewhat complicated if the gratuitous 360s are hidden—as, for example, LEFT 160 followed by RIGHT 360 being written as RIGHT 200. Give general rules for pruning. (Think of writing a procedure that takes the text of a turtle procedure as input and returns the pruned version.) Try your method on the following program:

```
TO PRUNE.ME
   FORWARD 5
   RIGHT 360
   FORWARD 5
   LEFT 240
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

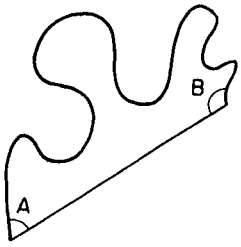


Figure 1.17
Relate the angles A and B to the total turning over the arc.

```
FORWARD 10
LEFT 120
FORWARD 0
LEFT 120
FORWARD 10
RIGHT 120
```

Can you give some motivation for pruning other than that it makes the simple-closed-path theorem true? [A]

2. Fill in the details in the first proof of the POLY closing theorem (see "sketch of proof 1"), including a proof that the vertices of POLY lie on a circle. [H]

3. Prove that if the angle input to POLY is an irrational number, the turtle never returns to its initial position, and yet always remains within a finite distance from it. [A]

4. [P] Invent some variations on the POLYROLL program, perhaps modeled after POLYSPI and INSPI.

5. Rewrite the POLYSTOP program recursively, so that it doesn't use the REPEAT command. [A]

6. What is the sum of the interior angles of an n-gon? What is the interior angle of a regular n-gon? Show how these formulas can be easily derived by using the simple-closed-path theorem. [HA]

7. Suppose we have a simple arc (an arc that does not cross itself) and that we join the endpoints of the arc by a straight line. Suppose further that the line and the arc do not intersect except at the endpoints (figure 1.17). Use the simple-closed-path theorem to give a formula relating the

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

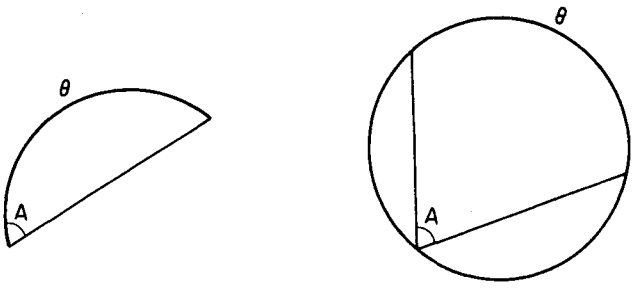


Figure 1.18
Solve for A in terms of θ.

total turning over the arc to the (interior) angles that the arc makes with the line. [HA]

8. Apply the result of the previous exercise to find the angle between a chord of a circle and the arc that it subtends (figure 1.18a). [A]

9. Use the previous exercise to compute the arc of a circle subtended by an inscribed angle (figure 1.18b). [HA]

10. Proof 1 of the POLY closing theorem was based on the fact that the vertices of a POLY all lie on a circle. Use the simple-closed-path theorem to show that the amount of arc on the circumscribed POLY circle from one vertex to the next is just the angle input to the POLY procedure. [H]

11. We said that we would delay giving a proof of the simple-closed-path theorem until chapter 4. Give a proof of the theorem in the special case where the simple closed path is a POLY figure. [H]

12. If you take a bicycle and lock the front wheel at angle θ from straight ahead (where θ is rather small), the bicycle will turn in a circle. What is the radius of the circle, given that the length between wheel centers of the bicycle is D? [HA]

1.3 Looping Programs

We said that the turtle approach allows us to take concepts that are useful in thinking about computation and apply them to the study of

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

geometry. One such concept is that of *state*. Of course the idea of state is not unique to computer science. It is important in physics, chemistry, and any other field involving configurations that are subject to change. But we do not generally look upon geometry in this way; geometric figures are usually regarded as static objects. Turtle geometry provides a more dynamic perspective—the geometry is tied to movements.

The state of the turtle is given by specifying its position and its heading. From the state point of view, the basic turtle commands—FORWARD, BACK, LEFT, and RIGHT—are *state-change operators:* They cause the turtle to change state. In this section we will look at a sequence of turtle commands purely in terms of its net effect in changing the initial state to the final state, ignoring what comes between. Thus, a sequence of turtle commands can be summarized as a single state-change operator. At this level of abstraction all programs that generate a closed path are the same—they are all *state-change-equivalent.* They correspond to the simplest of all state-change operators, the one that does nothing and leaves the initial state *invariant.*

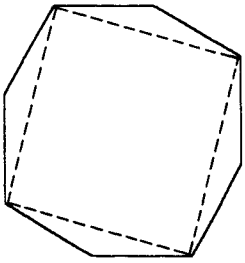
1.3.1 The Looping Lemma

There is something striking about the paths drawn by the modifications to POLY discussed at the end of section 1.1. It is as if their descent from POLY cannot be suppressed! You should have noticed the same phenomenon in many of your own programs. Figure 1.19 shows a POLY skeleton in dotted lines underlying the elaborate surface structures of NEWPOLY and INSPI. Can we understand this phenomenon?

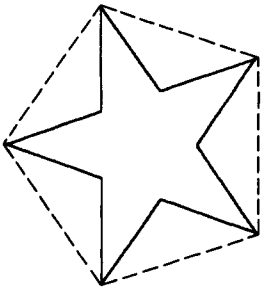
The key observation is this: Between successive vertices of the POLY skeleton, the program does the same thing. We might say that the program is just a decorated version of the underlying POLY. That the paths of NEWPOLY and INSPI consist of a collection of identical pieces is evident from the pictures they draw. In the case of NEWPOLY the repetitive or looping behavior is clear in the program structure. INSPI's program structure will require a second look. For now we proceed on the basis of the visual evidence of the paths and consider the class of programs that do the same thing over and over, regardless of the complexity of the *basic loop* (the thing that is repeated).

We can peel away the decoration by focusing on the net result of the basic loop. What is the difference between the initial state and the final state of the turtle? By the nature of the turtle only two things can

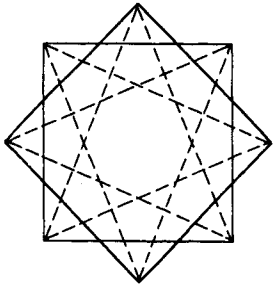
Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022



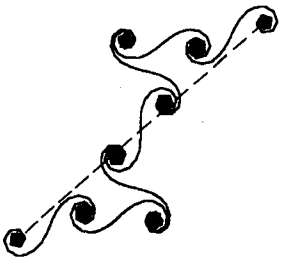
90



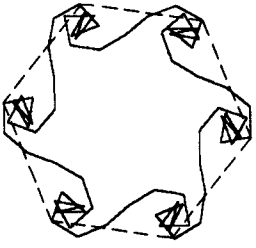
72



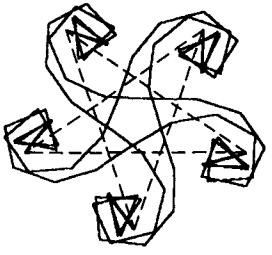
135



180



60



144

Figure 1.19
NEWPOLYs and INSPIs with "POLY skeletons" indicated by dashed lines. (Numbers are skeleton angles.)

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

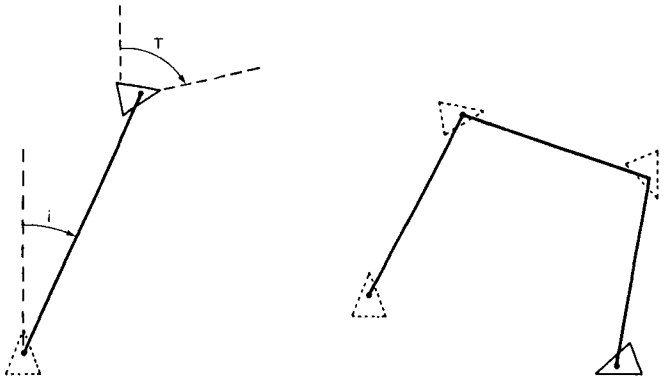


Figure 1.20
(a) The net result (state change) of some basic loop. (b) Repeating the net result lays down the POLY skeleton.

happen: a net change of position and a net change of heading. Figure 1.20a shows the general case. Notice that we cannot assume that the change of position is in the same direction as the turtle's original heading; hence, we must include an angle i between the initial heading and the change-of-position line.

When the basic loop is repeated, the same change of state must occur, but now relative to the new heading and hence rotated by the heading change which the turtle underwent in the first loop. We will call this change of heading T (for total turn). The next loop must follow the same pattern, and so on. Figure 1.20b shows the repeated process laying down the skeleton POLY. The angle i is *i*rrelevant to the intrinsic properties of the underlying POLY—it just determines the relative orientation of the POLY with respect to the initial heading of the turtle. The important quantity is the heading change from beginning to end of the basic loop, the total turning T in the basic loop. This is the angle of turning from one segment of the POLY to the next, and it determines the figure's properties. We can state this result more formally:

Looping Lemma Any program that is just a repetition of some basic loop of turtle instructions has precisely the structure of POLY with an angle input equal to T, the total turning in the loop.

You should be able to say what "has the structure of POLY" means in detail. It includes such things as repeatedly touching base with a circle

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

if total turning is not equal to a multiple of 360°, and touching base with a line if it is. It also includes the fact that the symmetry type of the figure is the same as that of the underlying POLY. For instance, if the total turning of the basic loop is 90°, the repeated loop will have the fourfold symmetry of a square, necessarily closing in four iterations of the loop.

1.3.2 Examples of Looping Programs

Let's analyze some simple looping programs. In NEWPOLY the total turning is $3 \times$ ANGLE. If ANGLE is 144, then $T = 3 \times 144 = 432$, which is equivalent to $72 = 360/5$. Hence, the five-pointed star NEWPOLY actually has the structure of a pentagon (not a POLY with ANGLE 144)—something that might not have been apparent from just looking at the path. (The fact that the path is simple is a clue. Also, observe that the program visits the vertices of the underlying pentagon in sequence, as does POLY with an ANGLE of 72, rather than skipping between vertices, as does POLY with ANGLE 144.)

Let's take a look at INSPI—in particular INSPI with ANGLE equal to 2 and INCREMENT equal to 20, which draws the decorated five-pointed star shown in figure 1.12. The program simply alternates FORWARD SIDE with turning RIGHT an ever-increasing angle which is tabulated in the following:

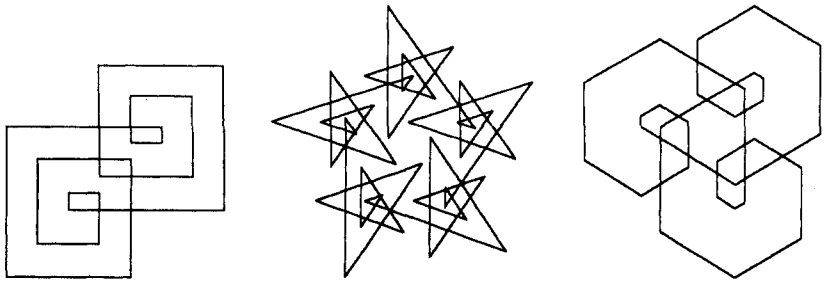
```
RIGHT 2
RIGHT 2 + 20
RIGHT 2 + 2*20
RIGHT 2 + 3*20
.
.
RIGHT 2 + 17*20
RIGHT 2 + 18*20 = 2 + 360
```

The last command has the same effect as the first, and the one to follow, RIGHT 2 + 19 × 20 = RIGHT 2 + 20 + 360, is the same as the second. The program is clearly staging a repeat performance of the first 18 steps. Computing the total turning, we find

$$2 + (2 + 20) + (2 + 2 \times 20) + \cdots + (2 + 17 \times 20)$$
$$= 18 \times 2 + (1 + 2 + \cdots + 17) \times 20 = 3,096,$$

which is equal to 216, or −144, modulo 360. (When computing the

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022



ANGLE = 90
MAX = 10

ANGLE = 144
MAX = 8

ANGLE = 60
MAX = 10

Figure 1.21
Spirolaterals.

heading change, we need only consider the turning modulo 360°.) We now see the origin of the five-pointed star POLY skeleton. INSPI is typical of the way in which, because of the modulo-360 effect, many seemingly ever-changing programs actually form repeating loops.

Another interesting repeating-loop program draws the family of figures called spirolaterals shown in figure 1.21:

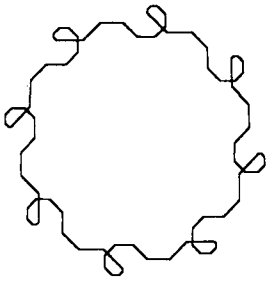
```
TO SPIRO (SIDE, ANGLE, MAX)
   REPEAT FOREVER
      SUBSPIRO (SIDE, ANGLE, MAX)

TO SUBSPIRO (SIDE, ANGLE, MAX)
   COUNT ← 1
   REPEAT
      FORWARD (SIDE * COUNT)
      RIGHT ANGLE
      COUNT ← (COUNT + 1)
   UNTIL COUNT > MAX
```

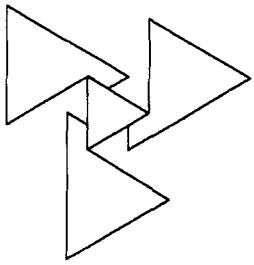
If you study this procedure you will see that it amounts to having the turtle draw an initial chunk of a POLYSPI (in fact, the first MAX lines of a POLYSPI) and repeat this over and over. It is easy to see that the basic loop in SPIRO has total turning ANGLE $\times$ MAX.

We will refer to the figures drawn by SPIRO as *simple spirolaterals*. A more general kind of spirolateral (shown in figure 1.22) maintains a basic loop in which each vertex has the same amount of turning, but allows the turtle to turn left rather than right at some of the vertices. To specify

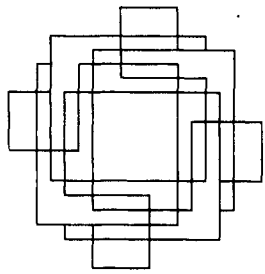
Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022



ANGLE = 45
MAX = 11
LIST = [8 9]



ANGLE = 120
MAX = 6
LIST = [1 3]



ANGLE = 90
MAX = 11
LIST = [3 4 5]

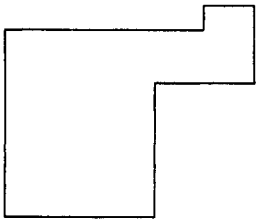
Figure 1.22
Generalized spirolaterals.

these figures we need to indicate the direction of the turtle's turning at each vertex. The corresponding GSPIRO procedure takes four inputs: a side length (the length of the shortest side in the figure), an angle through which the turtle turns left or right at each vertex, a number MAX telling how many steps are in the basic loop, and a list of numbers specifying the vertices at which the turtle should turn left. If the vertex number is a member of the list, then the turtle turns left at the vertex; otherwise the turtle turns right. (The MEMBER command is used to tell whether or not something is a member of a list.) Thus, the GSPIRO procedure is

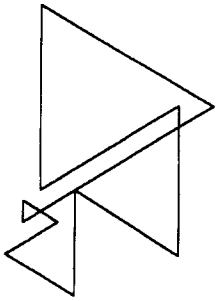
```
TO GSPIRO (SIDE, ANGLE, MAX, LIST)
   REPEAT FOREVER
      SUBGSPIRO (SIDE, ANGLE, MAX, LIST)

TO SUBGSPIRO (SIDE, ANGLE, MAX, LIST)
   COUNT ← 1
   REPEAT
      FORWARD SIDE * COUNT
      IF MEMBER (COUNT, LIST)
         THEN LEFT ANGLE
         ELSE RIGHT ANGLE
      COUNT ← COUNT + 1
   UNTIL COUNT > MAX
```

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022



ANGLE = 90
MAX = 8
LIST = [4 8]



ANGLE = 120
MAX = 11
LIST = [3 4 6 7]

Figure 1.23
Unexpectedly closed spirolaterals.

The basic loop SUBGSPIRO makes MAX $-$ L right turns and L left turns where L is the number of elements in LIST, making a total turning of

$$(\texttt{MAX} - L) \times \texttt{ANGLE} - L \times \texttt{ANGLE} = (\texttt{MAX} - 2L) \times \texttt{ANGLE}.$$

One intriguing property of spirolaterals is that they may be closed even when the heading change is a multiple of 360°. Total turning a multiple of 360° would lead you to expect a POLY substrate that would march off on a straight line to infinity. But, by a remarkable coincidence, the sidelength of the underlying POLY (which we did not bother to compute for any of these other programs) might turn out to be 0 as well! This corresponds to having a looping program in which the basic loop closes all by itself. This phenomenon deserves a name: *unexpectedly closed*. Figure 1.23 gives some examples of unexpectedly closed spirolaterals.

1.3.3 More on the Looping Lemma

We end the body of this section with two remarks about the looping lemma—one about its implications beyond predicting the symmetry of looping programs, the other about increasing the strength of the lemma.

First, the looping lemma constrains the behavior of any looping program. Under many circumstances, it may be possible to exclude simple looping as a way of generating a class of paths. For example, any infinite spiral neither touches base on a fixed circle nor marches off to infinity around a line as POLY does, so it cannot be drawn by any looping program.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

Second, it can be very valuable to know how to identify which programs are looping programs without a detailed look at each particular case. We can give a purely program-structural criterion that serves to identify such programs: A program must loop (or terminate) if it consists of any combination of

- fixed and finite sequences of turtle commands FORWARD, BACK, LEFT, and RIGHT with specified numeric inputs (these are called *fixed instruction sequences*),
- repeats, and
- calls to programs that satisfy these properties.

Notice that this criterion is recursive in form.

1.3.4 Technical Summary

The following is a technical recap of results stated or implied in this section. The detailed proofs of these facts are left as exercises.

The Canonical Form of a Turtle State-Change Operator
Any fixed instruction sequence of turtle commands is state-change-equivalent to a POLY step sandwiched between a RIGHT I and LEFT I for some angle I:

```
RIGHT I
FORWARD D
RIGHT T
LEFT I
```

The angle T is precisely the total turning of the fixed instruction sequence.

Looping Lemma and Classification of Looping Programs
Any program that repeats a fixed instruction sequence (or the equivalent, as in INSPI) has the behavior of a POLY with angle T and side D in the following senses (the angle T may be simply determined as the total turning in the basic loop):

Boundedness If $T \neq 0$ then the figure drawn by the program will lie within a fixed distance from some circle, and hence will be bounded. In the exceptional case, $T = 0$, the figure will lie within a fixed distance from some line (figure 1.24).

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

$T \neq 0$ $T = 0$

Figure 1.24
Any looping program is confined to a region (a) near a circle ($T \neq 0$) or (b) in the exceptional case ($T = 0$) near a line.

Closing If T is a rational multiple of 360°, the program will always draw a closed path, with the usual exception, $T = 0$, which causes the program to walk off to infinity. The exception to the exception is when D is also zero, the equivalent of `POLY 0 0`, in which case the program is "unexpectedly" closed. If T is irrational the program will never close.

Symmetry The program must have the same rotational symmetry as `POLY D T`. In particular, if $T = 360s/r$ where s/r is a fraction in lowest terms, then the program will have r-fold symmetry. (We have not yet discussed symmetry in detail. This topic will form the basis of section 1.4.)

1.3.5 Nontechnical Summary

Doing the same thing over and over is either circular, straight-linish, or very dull.

Exercises for Section 1.3

1. [P] Draw at least three distinct (ignoring size) `INSPI` figures with sixfold symmetry.

2. Give a proof of the looping lemma and the classification of looping programs given in the technical summary. This may be done using the form of a general state-change operator (given above) or by modifying

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

Proof 2 of the POLY closing theorem of subsection 1.2.2. Give bounds for the "fixed distances" specified in the boundedness part in terms of the instructions in the basic loop.

3. [P] Figure 1.23 shows some unexpectedly closed spirolaterals. Find some more.

4. What is the heading change for INSPI with an ANGLE of A and an INCREMENT of 10? [A]

5. [D] What is the heading change for INSPI with an ANGLE of A and an INCREMENT of $360/n$, where n is an integer? [HA]

6. [DP] Compute the total turning, T, of the basic loop in the following looping program in terms of the angle inputs BOTTOM and TOP. Use your formula to draw at least three different (ignoring size) figures with threefold symmetry; fourfold, fivefold. [HA]

```
TO POLYARC (SIDE, BOTTOM, TOP)
   REPEAT FOREVER
      INSPI.STOP (SIDE, BOTTOM, TOP, 1)
      INSPI.STOP (SIDE, TOP, BOTTOM, -1)

TO INSPI.STOP (SIDE, START.ANG, END.ANG, INC)
   REPEAT
      FORWARD SIDE
      LEFT START.ANG
      START.ANG ← START.ANG + INC
   UNTIL START.ANG = END.ANG
```

7. [P] Make "even more general" spirolaterals by allowing the turtle to move BACK at certain of the vertices. Analyze this program. Find some unexpectedly closed figures.

8. [DD] Show that a simple spirolateral can never be unexpectedly closed. [H]

9. [DD] Can INSPI produce unexpectedly closed figures?

10. [P] Invent some disguised looping programs like INSPI. Give a formula for the total turning of the basic loop in terms of the inputs to the procedure. Find inputs that draw figures with simple symmetry. Find inputs that draw unbounded figures. Determine whether any of these figures are unexpectedly closed.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

11. When we summarize turtle paths as state-change operators, the closed paths are precisely those operators that leave the turtle's state unchanged. This suggests the generalization of studying operators that, when run twice (or three times, and so on), leave the state unchanged. Describe the paths corresponding to these operators. [A]

12. [P] Can you characterize "looping programs" in terms of the commands used in writing them? In particular, consider the following program structures, where X and Y are variables and n is some fixed number:

- do loops
- goto statements
- assignment statements of the form `X ← n`
- assignments of the form `X ← Y + n`
- conditional statements of the form `IF X = n ...`

For each kind of structure, say whether a program that repeats a block of instructions consisting of basic turtle commands together with that particular structure must necessarily be equivalent to a program which repeats a fixed instruction sequence. Are there bad combinations, in the sense that two structures which separately lead to looping may not loop when combined in a single program?

1.4 Symmetry of Looping Programs

Section 1.3 showed how the symmetry of any looping program is determined by the symmetry of an underlying POLY skeleton. But what determines the symmetry of the POLY? To begin with, it is clear that the SIDE input in `POLY SIDE ANGLE` does not affect the shape of the figure at all but only determines the size. The real question is: How does the ANGLE input affect the symmetry of POLY? To be more precise, we can break this question into two questions:

For a given ANGLE input, how many vertices will the resulting POLY have?

Conversely, if we want to produce a POLY with a specified number of vertices, what number(s) can we use for the ANGLE input?

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

The purpose of this section is to answer these questions, and in doing so to provide a taste of the mathematics of number theory.

1.4.1 The Symmetry of POLY

We want to relate the number of vertices, n, to the input ANGLE, which we'll call A for short. The POLY closing theorem of 1.2.2 gives us a very good start. It says POLY is done when the turtle has turned a multiple of 360°, that is, when n turns of A each is some multiple of 360:

$$nA = 360R.$$

We've given the multiple the name R for a good reason: It is the rotation number of the figure, as defined in subsection 1.2.1.

But the above equation, which defines a common multiple of A and 360, doesn't tell the whole story. R and n aren't just any integers satisfying the equation; they are the smallest (positive) that do so, corresponding to the first time heading change reaches a multiple of 360. That is why the number $nA = 360R$ is called the *least common multiple* of A and 360, denoted LCM$(A, 360)$. The answer to our first question is:

For an ANGLE input of A, the number of vertices of the resulting POLY is $n = \text{LCM}(A, 360)/A$ and the rotation number is $R = \text{LCM}(A, 360)/360$.

What we've done so far is little more than giving the answer a name. How does one go about computing the least common multiple? One way, which assumes that A is an integer, is to express A and 360 as products of primes. Then each of the expressions nA and $360R$ will give a partial view of the factorization of LCM$(A, 360)$. For example, if $A = 144$ then we have $A = 2^4 \times 3^2$, $360 = 2^3 \times 3^2 \times 5$. Using this decomposition, we can deduce that

$$\begin{aligned}\text{LCM}(144, 360) &= n \times 2^4 \times 3^2 = R \times 2^3 \times 3^2 \times 5 \\ &= 2^4 \times 3^2 \times 5 = 720,\end{aligned}$$

for it is easy to see that the LCM must contain at least four factors of 2 (from A), one factor of 5 (from 360), and two factors of 3 (from either A or 360), and from this we derive that $n = 5$ and $R = 2$. So POLY 100 144 has fivefold symmetry (it consists of $n = 5$ identical pieces identically hooked together) and rotation number 2.

Another way to compute the least common multiple is to solve the equation $nA = 360R$ exactly as the procedure POLY solves it: by running until it closes! Make a list of $n \times A$ for $n = 1, 2, 3, \ldots$ and see

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

when $n \times A$ is a multiple of 360. Applying this method to our example $A = 144$ gives

n	nA	multiple of 360?
1	144	no
2	288	no
3	432	no
4	576	no
5	720	yes

We can formulate this method as a procedure for computing the least common multiple:

```
TO LCM A B
   N ← 0
   REPEAT
      N ← N + 1
      MULTIPLE ← N * A
   UNTIL REMAINDER (MULTIPLE, B) = 0
   RETURN MULTIPLE
```

Such "brute force" methods of computation can be very useful. We will see in the next subsection the utility of a simple modification of this process, called Euclid's algorithm. The `LCM` procedure uses `RETURN`—a command we haven't seen before. Since the procedure is supposed to be computing some value, we need to have some method for "getting the value out of the procedure." This is what `RETURN` does. In practice the returned value will be used as an input to another operation, for example, `PRINT LCM(144,360)`.

Let's turn to our second question about the symmetry of `POLY` figures. Suppose we want to produce figures of a given symmetry; what `ANGLE` can we use? To answer this question, let's start again with the basic symmetry equation $nA = 360R$. Since we want to find values for A that will produce a given value for n, we should be able to use any A that satisfies $A = 360R/n$. The question is: What value(s) can we choose for R? For instance, we can take $R = 1, A = 360/n$, which always works—it makes a regular n-sided polygon. But $R = 2, A = 2 \times 360/n$ may not work. Here's an example: Suppose we want tenfold symmetry, $n = 10$. If we take $R = 2$, then A will be 72. This makes a pentagon, not a figure with tenfold symmetry.

We've been fooled. $A = 360R/n$ doesn't always give n-fold symmetry. Let's look more carefully at the above guess, $R = 2, n = 10$.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

Not only does the resulting pentagon not have tenfold symmetry, but it has rotation number 1, not 2. We clearly are not justified in naming our guesses 10 by n and 2 by R, so let's give them new names, n' and R'. How do these relate to the real n and R?

Let's compute the real n and R that correspond to $A = 360R'/n'$. We want the smallest positive integers n and R that satisfy $nA = 360R$. But this equation is satisfied by all pairs of integers R and n with the property that $R/n = R'/n'$. Since we want R and n to be the smallest pair with this property, we should take them to be the numerator and denominator of the fraction R'/n' reduced to lowest terms. In our example we had $R'/n' = 2/10 = 1/5$, so $n = 5$ and $R = 1$. To put this another way, $A = 360R'/n'$ will give n'-fold symmetry only when R'/n' cannot be reduced, that is to say, when R' and n' have no factors in common. Thus, the answer to the second question above is the following:

To generate a POLY with n-fold symmetry, take the ANGLE input to be $A = 360R/n$, where R is any positive integer that has no factors in common with n.

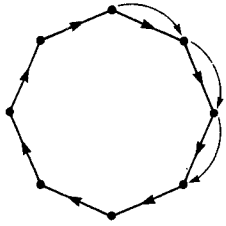
1.4.2 Common Divisors

We've answered our questions about the symmetry of POLY in terms of such concepts as common multiples and common factors. Let's take a detour from turtle geometry and turn this process around to see what knowing about POLY can tell us about these number-theoretic concepts.

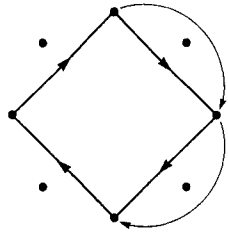
To begin with, the previous subsection led us to consider pairs of integers n and R that have no common factors. Such pairs are called *relatively prime*. We saw that $A = 360R/n$ draws an n-sided POLY precisely when R is relatively prime to n. We'll reinterpret this fact in terms of a new way of looking at the POLY process.

Think of the n vertices of POLY lying on a circle and numbered from 0 through $n - 1$. To construct the various n-sided POLYs we can connect the vertices in sequence using the following sorts of rules: (1) Connect each vertex to the very next one; (2) connect the vertices, skipping one in between; (3) connect the vertices, skipping 2 in between; and so on. Figure 1.25 illustrates the various patterns for $n = 8$. There are $n - 1$ possibilities, which correspond to $R = 1, 2, \ldots, n - 1$ in the formula $A = 360R/n$. For any choice of R, if we start at the vertex numbered 0, then that is connected to the vertex numbered R, which is connected to the vertex numbered $2R$, and so on. Since we're counting these ver-

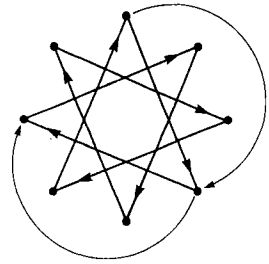
Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022



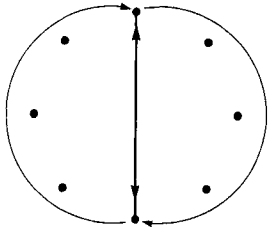
$0, 1, 2, 3, \ldots \pmod 8$



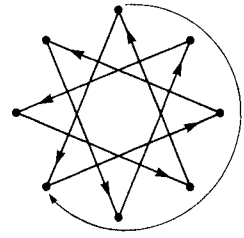
$0, 2, 4, 6, \ldots \pmod 8$



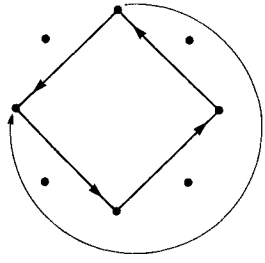
$0, 3, 6, 9, \ldots \pmod 8$



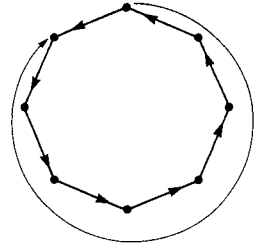
$0, 4, 8, 12, \ldots \pmod 8$



$0, 5, 10, 15, \ldots \pmod 8$



$0, 6, 12, 18, \ldots \pmod 8$



$0, 7, 14, 21, \ldots \pmod 8$

Figure 1.25
Patterns of connecting eight points, $n = 8$.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

tices modulo n, we see that the sequence of vertices hit are precisely the multiples $0, R, 2R, \ldots, (n-1)R$ taken modulo n.

Now we saw in the previous subsection that, if R and n are relatively prime, then the resulting POLY figure will have n vertices. In terms of the circle picture this means that all vertices on the circle $0, 1, 2, \ldots, n-1$ are reached. Consequently, if R and n are relatively prime, then the multiples of R taken modulo n must include all the numbers between 0 and $n-1$. In other words, if s is any integer between 0 and $n-1$, then there is some multiple of R, say pR, with $pR = s \pmod{n}$. Moreover, if R and n are not relatively prime, then at least one of the n vertices will not be touched by the POLY process, and so there will be some number s which is not equal to $pR \pmod{n}$ for any p. Restating the equality modulo n in terms of precise equality, we have the following:

Two integers R and n are relatively prime if and only if, for any integer s, we can find integers p and q such that $pR - qn = s$.

This condition can be written in an equivalent form (exercise 10) that reflects the fact that, if the vertex labeled 1 is hit, then all vertices must be:

Two integers R and n are relatively prime if and only if there exist integers p and q such that $pR + qs = 1$.

In summary: We have shown how to translate a condition about the relative primality of two integers into a rather different condition which has to do with representing integers as sums.

What exactly happens when R and n are not relatively prime? Figure 1.25 includes some examples: Taking $n = 8$ and $R = 2$ hits all the even vertices (the multiples of 2); $n = 8$ and $R = 4$ hits only 0 and 4 (multiples of 4); $n = 8$ and $R = 6$ hits all the multiples of 2. In general, the vertices which are hit—numbered $0, R, 2R, \ldots, (n-1)R$, taken modulo n—give precisely the multiples of some integer d. This integer d is called the *greatest common divisor* of n and R, and it can be defined by stating that $d = \mathrm{GCD}(n, R)$ is the largest integer that divides both n and R. The fact that the vertices which are hit are precisely the multiples of d follows from the fact that $\mathrm{GCD}(n, R)$ can alternatively be defined as the smallest positive integer that can be represented as $pR + qn$, where p and q also integers. We leave it to you (exercise 11) to verify these claims. You can see that the GCD of n and R is an

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

important quantity. When it is 1, n and R are relatively prime and the POLY has n-fold symmetry. When the GCD is not 1, it gives d, the integer whose multiples are the vertices actually hit.

Neither definition of GCD—as the largest common divisor, or as the smallest positive $pR + qn$—seems very helpful in actually computing the GCD of two given integers. One method for doing so is Euclid's algorithm. The idea of the algorithm is very simple: The common factors of n and R are the same as the common factors of $n - R$ and R. (Prove this.) And so the problem of finding the GCD of n and R can be reduced to finding the GCD of $n - R$ and R:

Euclid's Algorithm Start with two numbers. (1) If the two numbers are equal, then stop; the GCD is their common value. (2) Subtract the smaller number from the larger and throw away the larger. (3) Repeat the entire process using, as the two numbers, the smaller number and the difference computed in step 2.

The process produces ever smaller numbers and stops with two equal numbers. Take as an example 360 and 144. The sequence of pairs generated is

$$(360, 144) \rightarrow (216, 144) \rightarrow (72, 144) \rightarrow (72, 72) \rightarrow \text{done: GCD} = 72$$

Finding the GCD of 360 and 144 can be interpreted as follows: Any POLY with an integer angle will touch some subset of the vertices of a regular 360-gon. If the angle is 144, the vertices touched will be the multiples of 72.

We can translate Euclid's algorithm into a recursive computer procedure:

```
TO EUCLID (N, R)
   IF N = R THEN RETURN N
   IF N > R THEN RETURN EUCLID (N - R, R)
   IF N < R THEN RETURN EUCLID (N, R - N)
```

There's an obvious way to speed up the algorithm: Subtract multiple copies of the smaller number from the larger in a single step. Even better, we can divide the smaller number into the larger, taking the smaller number and the remainder to start the next step. This has an additional advantage—we will know automatically that the remainder is smaller

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

than the original smaller number, so it will not be necessary to test to see which of the two inputs is smaller:

```
TO FAST.EUCLID (N, R)
   IF N = R THEN RETURN N
      ELSE RETURN FAST.EUCLID (R, REMAINDER (N, R))
```

Note that Euclid's algorithm is very nearly the reverse of the "brute force" method for finding the least common multiple given in subsection 1.4.1. In fact the algorithm gives us a new way of computing the LCM of two numbers because of the formula

$$\text{LCM}(p, q) \times \text{GCD}(p, q) = p \times q,$$

which is true for any integers p and q. (See exercise 16.)

As a final remark we point out that, whereas the REMAINDER function used in FAST.EUCLID is defined only for integers, the operations in the original EUCLID procedure make sense for any numbers. So we can define the GCD for any two numbers (not just integers) to be the value returned by the EUCLID procedure. (Exercises 15–17 invite you to investigate the properties of this generalized GCD.) We see here how the procedural formulation of a concept can suggest new insights and directions for exploring. Further explorations suggested by the EUCLID procedure are illustrated in exercises 18–25.

Exercises for Section 1.4

1. Determine the symmetry of POLY with $A = 350, 35, 37, 12\frac{1}{2}, 26\frac{2}{3}$. For each value of A, find p and q such that $A = 360p/q$ where p/q is a fraction reduced to lowest terms. [A]

2. Using only integer angles, what are all the possible values of n for which POLY can draw an n-sided figure? [HA]

3. [D] Show that the POLY symmetry determined by ANGLE $= 360 - A$ is the same as the symmetry of ANGLE $= A$. What about the symmetry of ANGLE $= 180 - A$? [HA]

4. Consider the process of finding the least common multiple of A and B. Show that the "brute force" method will find the same minimal number n such that $nA = RB$ as it will for $n(Ax) = R(Bx)$ where x is any number, and hence that $\text{LCM}(Ax, Bx) = x \times \text{LCM}(A, B)$. Use this idea to show how to compute least common multiples of (noninteger)

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

rational numbers. If $A = p/q$ and $B = r/s$, both fractions in lowest terms, show how to pick x so that one can use the prime factorization method to compute LCM(A, B). [A]

5. We saw that $A = 360R/n$ will produce an n-pointed figure for any integer R relatively prime to n. But how many of these Rs actually produce POLYs that look different? How many different POLYs are there with $n = 10, 36, 37$? How many in general? [A]

6. In solving the previous problem you may want to make use of the Euler ϕ function, which is defined for positive integers n to be the number of integers less than n and relatively prime to it. Euler gave a formula for $\phi(n)$, which works as follows: Suppose that $p_1, p_2, \ldots$ are the distinct prime factors of n, that is $n = p_1^a p_2^b \ldots$. (For example, $3960 = 2^3 \times 3^2 \times 5 \times 11$.) Then

$$\phi(n) = n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\cdots.$$

Use this formula to compute $\phi(1,000,000)$. [A]

7. [D] Use the fact that, for R relatively prime to n, the multiples of R include all the integers $1, 2, \ldots, n-1 \pmod{n}$ to prove Fermat's Little Theorem: If p is a prime and R is any positive integer less than p, then $R^{p-1} = 1 \pmod{p}$. [HA]

8. [P] Fermat's Little Theorem lies behind a recently discovered way to test by computer whether large numbers are prime. The idea is to start with a number p, pick a random number a less than p, and compute $a^{p-1} \pmod{p}$. If the answer is not 1, then p is not prime. Conversely, it is known that, in general, if p is not prime, then most of the numbers a less than p will not satisfy $a^{p-1} = 1 \pmod{p}$. So if we test, say, 10 different choices for a and they all satisfy Fermat's equation, then we can be virtually certain that p is prime. Implement this method in a computer program and use it to find, say, the ten largest primes less than one billion. (There are choices for p, called "Carmichael numbers," that will fool this test. They are nonprime, and yet satisfy the condition $a^{p-1} = 1 \pmod{p}$ for every a. But they are few and far between.)

9. [P] Write a procedure that, given a number n, returns a list of all the primes dividing n. Use this together with Euler's formula of exercise 6 to produce a procedure that calculates the Euler ϕ function.

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

10. Given integers R and n, show that there exist integers p and q such that $pR + qn = 1$ if and only if, for any integer s, there exist integers p_s and q_s such that $p_sR - q_sn = s$. [A]

11. [D] Prove that if we define the greatest common divisor $d = \mathrm{GCD}(n, R)$ as the largest integer that divides both n and R, then d is the smallest positive integer that can be expressed as $\langle\text{integer}\rangle R + \langle\text{integer}\rangle n$. Show also that the multiples of R taken modulo n are precisely the multiples of d. [H]

12. [D] Show that the EUCLID procedure works when its inputs are positive integers. That is, show that it will always terminate and that what it returns is in fact the GCD of its inputs.

13. [P] In subsection 1.2.2 we discussed the problem of writing a POLY procedure that draws the figure once and then stops. We implemented POLYSTOP using the "local" strategy of having the turtle count total turning. Write a version of POLYSTOP which uses the "global" strategy of taking the ANGLE input and computing in advance how many times to run the basic loop. Can you design the program so that it works not only for integer angles but for (noninteger) rational angles as well?

14. Show directly that if Euclid's algorithm returns d given R and n as inputs, then there exist integers p and q such that $pR + qn = d$. [H]

15. [D] Show that the EUCLID procedure terminates whenever its inputs are positive rational numbers, and thus allows us to extend the definition of greatest common divisor to all positive rational numbers. What is the "GCD" of $\frac{1}{2}$ and $\frac{2}{3}$? of a/b and c/d where a, b, c and d are integers? What can you say about the behavior of the algorithm when the inputs are not both rational? [A]

16. [D] Prove for integers p and q that $\mathrm{GCD}(p, q) \times \mathrm{LCM}(p, q) = p \times q$. Does this formula hold as well for rational numbers (with GCD defined as the result of the EUCLID procedure and LCM defined as indicated in exercise 4)?

17. [D] What can you say about the $\langle\text{integer}\rangle R + \langle\text{integer}\rangle n$ definition of GCD as it relates to the GCD for rational numbers (defined as the result of the EUCLID procedure)?

18. [D] Modify the EUCLID procedure to define a procedure DIO which not only computes $d = \mathrm{GCD}(n, R)$ but also returns integers p and q

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

such that $pR + qn = d$. The DIO procedure should take two inputs and return a list of three numbers p, q, d. [HA]

19. [D] Show how to speed up the DIO procedure by using the reduction method of the FAST.EUCLID procedure. Write the corresponding FAST.DIO program. [HA]

20. The name DIO comes from "Diophantine equations." These are equations which are to be solved for integer values of the unknowns. Use your DIO or FAST.DIO procedures to find integers x and y which satisfy the equation $17x + 117y = 1$; the equation $1234567x + 7654321y = 1$. [A]

21. Let T denote the transformation $(n, r) \rightarrow (r, n - r)$ which is used by the EUCLID program. Show that the transformation S defined by $(x, y) \rightarrow (x + y, x)$ is inverse to T in the sense that, for any pair (a, b), $T(S(a, b)) = S(T(a, b)) = (a, b)$. If we start with the pair $(1, 0)$ and repeatedly apply S we obtain

$$(1,0) \rightarrow (1,1) \rightarrow (2,1) \rightarrow (3,2) \rightarrow (5,3) \rightarrow (8,5) \rightarrow (13,8) \rightarrow \cdots.$$

The sequence of numbers formed by this operation is called the Fibonacci numbers; that is,

$$F(0) = 0, F(1) = 1, F(2) = 1, F(3) = 2, F(4) = 3, F(5) = 5, F(6) = 8,$$

and so on. Use the fact that S and T are inverse to show that for any integer n, $F(n)$ and $F(n - 1)$ are relatively prime. [A]

22. [P] Show that $F(n) = F(n - 1) + F(n - 2)$ and hence that the Fibonacci numbers can be generated by the procedure

```
TO FIB N
   IF N = 0 RETURN 1
   IF N = 1 RETURN 1
   RETURN FIB (N-1) + FIB (N-2)
```

Why does this procedure run so slowly? Can you find a faster method of computing the Fibonacci numbers? [A]

23. [DD] Expanding on the result of exercise 21, investigate the greatest common divisor of $F(n)$ and $F(n + k)$. State and prove a theorem about GCD$(F(a), F(b))$. [HA]

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022

24. [D] Use the `DIO` or `FAST.DIO` procedures (exercises 18,19) to solve Diophantine equations of the form $xF(n) + yF(n-1) = 1$ for integers x and y. What is the solution in general? [HA]

25. [P] For any pair of numbers p, q we can define a new sequence of numbers $F(p, q; n)$ by applying the transformation S of exercise 21 beginning with (p, q) rather than $(1, 0)$. Write a program to generate these sequences and investigate their properties. Can you express $F(p, q; n)$ in terms of the usual Fibonacci numbers? [A]

Downloaded from http://direct.mit.edu/books/book/chapter-pdf/1626040/9780262362740_caa.pdf by University of New Mexico user on 18 January 2022