

Leone Learning Systems, Inc.
Wonder. Create. Grow.

Leone Learning Systems, Inc. Phone 847 951 0127
237 Custer Ave Fax 847 733 8812
Evanston, IL 60202
Email tj@leonelearningsystems.com

Drawing with Recursion

TJ Leone
April 2005

⋮

Introduction

In the unit on *Combining Functions*, you learned that you can invoke a procedure you wrote from inside another procedure. In the example below, the procedure `triangle.inches` invokes the procedure `feet.to.inches`:

```
to feet.to.inches :feet
  op product 12 :feet
end

to triangle.inches :side
  fd feet.to.inches :side
  rt 120
  fd feet.to.inches :side
  rt 120
  fd feet.to.inches :side
  rt 120
end
```

One of the things that make Logo so powerful is that Logo procedures can also invoke themselves. Here's an example (if you try it, you'll need to use the Halt button to stop it):

```
to circle
  fd 1
  rt 1
  circle
end
```

This technique of writing a procedure that invokes itself is called *recursion*. Recursion can be used to draw simple figures like circles or polygons. It can also be used to draw complex shapes called *fractals*. Fractals can be mathematical structures like the Sierpinski triangle, Koch snowflake, Peano curve, Mandelbrot set and Lorenz attractor. Recursive drawings can also describe many real-world objects that do not have simple geometric shapes, such as clouds, mountains, turbulence, and coastlines.

In this unit, we will start with simple designs to explore the basics of recursion, and then move on to some fractals.

⋮

Circle

Let's start by taking a closer look at the circle procedure described in the introduction. With MSWLogo open, click on the Edall button and type this procedure into the editor window:

```
to circle
  fd 1
  rt 1
  circle
end
```

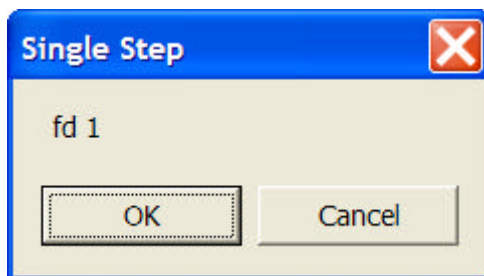
Now select Save and Exit from the File menu to save the procedure.

Next, press the Trace button to change its label to UnTrace and press the Step button to change its label to UnStep.

Type the following into the Input Box:

```
pd circle
```

When you press the Execute button (or hit the Enter key on your keyboard), you should see the Single Step window:



Each step of your procedure will appear in the Single Step window. Your procedure waits until you press OK, executes the step, and then shows you the next step. Click OK a number of times. What happens when you invoke `circle` from within the `circle` procedure?

You will also notice a trace of your procedure's execution in the Recall List Box as you step through your procedure:

```
pd circle
  ( circle )
  ( circle )
  ( circle )
```

⋮

Circle (continued)

The trace in the Recall List Box shows invocations of procedures that you have written. Every time a procedure is invoked by another procedure, the name of the procedure gets indented. Another example might help clarify. Here's a trace of the `triangle.inches` procedure in the introduction:

```
cs pd triangle.inches 6
( triangle.inches 6 )
  ( feet.to.inches 6 )
    feet.to.inches outputs 72
      ( feet.to.inches 6 )
        feet.to.inches outputs 72
          ( feet.to.inches 6 )
            feet.to.inches outputs 72
          triangle.inches stops
```

The first line is the line that was typed into the Input Box. This line always gets copied into the Recall List Box, whether you're tracing procedures or not.

When a procedure is invoked, the trace shows the name of the procedure followed by the inputs to the procedure. The procedure name together with the procedure inputs are enclosed in parentheses. When a procedure is invoked from the Input Box, the trace line is flushed all the way to the left.

Notice that `feet.to.inches` is slightly indented under `triangle.inches`. This signifies that `feet.to.inches` is invoked by `triangle.inches`. If a procedure takes inputs, the trace shows the input that is handed to that procedure. For example, since `triangle.inches` was executed with an input of 6, that 6 was passed to `feet.to.inches`, so we see `(feet.to.inches 6)` when `feet.to.inches` is invoked.

Also notice that a trace shows when a procedure stops. If the procedure is an operation like `feet.to.inches`, we can tell that it stops when we see a line like `feet.to.inches outputs 72`. If the procedure is a command, we can tell that it stops when we see a line like `triangle.inches stops`.

In the trace of `circle`, the only procedure invoked is `circle`. The line `(circle)` keeps getting indented because each `circle` is invoked by the one above it. Why don't we see "circle stops"?

To let the procedure continue without stepping, click the Cancel button. What does the turtle do on the main screen? What happens with the trace in the Recall List Box? When you're ready to move on, press the Halt button and read further.

⋮

The Stop Rule

As you must have noticed in the last section, the `circle` procedure does not stop unless you press the Halt button or quit out of MSWLogo. This is considered rude behavior for procedures. So how do we get recursive procedures like `circle` to stop by themselves? We add a *stop rule* to keep the procedure from invoking itself when we want it to stop.

Here's a procedure called `circle.stop`, that has a stop rule:

```
to circle.stop
  fd 1
  rt 1
  if heading = 0 [stop]
  circle.stop
end
```

The `stop` command tells Logo to end the procedure. In `circle.stop`, the `stop` command is executed if the predicate `heading = 0` is true. Will this allow us to draw a proper circle and then stop? Yes and no.

Suppose the turtle has a heading of 0 when we execute `circle.stop`. The first thing `circle.stop` tells Logo to do is to move the turtle `fd 1`, then turn it `rt 1`. When this happens, the turtle's heading changes from 0 to 1. Next, `circle.stop` tells Logo to check the heading of the turtle. If the heading is 0, the procedure should end. At this point, the heading of the turtle is 1, so we go to the next command, which is `circle.stop`. This starts us all over again.

Each time `circle.stop` is called, the heading increases by 1 until it reaches 359. When the turtle's heading is 359, Logo moves it `fd 1` according to the commands listed in `circle.stop`. But now, when Logo turns the turtle `rt 1`, the heading becomes 0. If this sounds weird to you, check it out for yourself. Execute these commands:

```
seth 359
show heading
rt 1
show heading
```

With the heading equal to zero, the predicate `heading = 0` is true, and `stop` is executed, ending the `circle.stop` procedure.

We have just shown that the stop rule for `circle.stop` will work if the turtle starts out with a heading of 0. However, there are many cases in which this particular stop rule doesn't work correctly. We will discuss these exceptions later and develop a better stop rule in the Solved Problems section.

⋮

Recursive Poly

Since recursive procedures are just procedures that invoke themselves, we can also have recursive procedures with inputs. For example, here's a recursive procedure for drawing polygons:

```
to poly :size :angle
  fd :size
  rt :angle
  if heading = 0 [stop]
  poly :size :angle
end
```

This procedure has the same weak stop rule as the circle procedure above, so it requires that your turtle start with a heading of 0. You'll have a chance to build a better stop rule for this procedure later.

Try running a trace on the `poly` procedure. Click on the Trace button if necessary to change its label to UnTrace. Then execute a `poly` command. I used this:

```
cs pd poly 100 90
```

When I executed my command I saw the following in my Recall List Box:

```
cs pd poly 100 90
( poly 100 90 )
  ( poly 100 90 )
    ( poly 100 90 )
      ( poly 100 90 )
        poly stops
      poly stops
    poly stops
  poly stops
poly stops
```

Remember our earlier trace of `triangle.inches`? In that trace, the lines showing the invocation of `feet.to.inches` was indented to show that `feet.to.inches` was invoked by `triangle.inches`.

But `poly` actually invokes itself. The first time we see `(poly 100 90)`, it is flushed all the way to the left. This appears when `poly` is invoked from the Input Box.

The next time we see `(poly 100 90)`, it is indented. This part of the trace appears when a new `poly` is called from the first `poly`.

⋮

Recursive Poly (continued)

Every time one `poly` invokes another, the trace adds a new line of `(poly 100 90)` that is indented farther to the right, and Logo moves the turtle forward 100, turns it right 90, and checks the turtle's heading to decide what to do next. The invocation of `poly` with the trace of `(poly 100 90)` that is farthest to the right is called the innermost invocation of `poly`.

When the heading of the turtle becomes 0, the stop rule for the innermost `poly` kicks in, and that `poly` stops and the trace prints `poly stops` in the Recall List Box. The `poly` that invoked the innermost `poly` invokes `poly` in its last line and then ends. Since the innermost `poly` is now finished, the `poly` that invoked it can stop.

The stops keep coming a couple more times until the outermost `poly` has stopped. Notice that the indentations of `poly stops` follow the indentations of the invocations of the corresponding `(poly 100 90)`.

⋮

Polyspi

In the last section, the `poly` procedure was invoked from the Input Box with the inputs 100 and 90. Whenever a recursive call was made to invoke `poly` once more, `poly` was invoked again with the same inputs, 100 and 90.

Now let's look at a new procedure called `polyspi`:

```
to polyspi :side :angle :increment
  fd :side
  lt :angle
  if :side > 200 [stop]
  polyspi :side + :increment :angle :increment
end
```

Try it out with some different inputs and see what it does. Some to try include:

```
polyspi 4 90 4
polyspi 1 95 1
polyspi 0 95 10
polyspi 7 117 11
polyspi 100 120 20
```

Here's a trace I ran on `polyspi`:

```
cs pd polyspi 100 120 20
( polyspi 100 120 20 )
  ( polyspi 120 120 20 )
    ( polyspi 140 120 20 )
      ( polyspi 160 120 20 )
        ( polyspi 180 120 20 )
          ( polyspi 200 120 20 )
            ( polyspi 220 120 20 )
              polyspi stops
            polyspi stops
          polyspi stops
        polyspi stops
      polyspi stops
    polyspi stops
  polyspi stops
polyspi stops
```

Notice how the first input to `polyspi` changes with each invocation. Why did this happen? The `polyspi` procedure was first invoked with the inputs 100 for `:side`, 120 for `:angle`, and 20 for `:increment` from the Input Box.

⋮

Polyspi (continued)

The `polyspi` procedure move forward 100 and turns left 20 because its first input was 100 and its second input was 20. Since 100 is not greater than 200, the `polyspi` procedure continues to the next line, which tells it to invoke another `polyspi` with first input $100 + 20 = 120$, second input 120, and third input 20.

This inner `polyspi` now has inputs 120 for `:side`, 120 for `:angle`, and 20 for `:increment`. It moves the turtle forward 120, turns it right 120, and checks to see if 120 is greater than 200. Since it isn't, this `polyspi` invokes another `polyspi` with first input $120 + 20 = 140$, second input 120, and third input 20.

Each `polyspi` calls a new one until one of them gets an input of `:side` that's greater than 200. When that happens, we've called the innermost `polyspi`. It is stopped by the stop rule `if :side > 200 [stop]`, which allows its calling `polyspi` to get to its end, and so on until all the `polyspi` procedures have stopped.

You can create lots of recursive procedures that make changes in this way. Here are a couple of other examples from *Turtle Geometry* by Harold Abelson and Andrea diSessa (Abelson & diSessa, 1986):

```
to inspi :side :angle :increment
  fd :side
  rt :angle
  if :angle > 3600 [stop]
  inspi :side :angle + :increment :increment
end
```

```
to eqspi :side :angle :scale
  fd :side
  lt :angle
  if :angle > 3600 [stop]
  eqspi :side * :scale :angle :scale
end
```

Again, we have some stop rules that aren't that great. You'll get to improve them later on.

⋮

Triangle.inches revisited

Let's go back again to the `triangle.inches` example. Here are the procedures:

```
to feet.to.inches :feet
  op product 12 :feet
end

to triangle.inches :side
  fd feet.to.inches :side
  rt 120
  fd feet.to.inches :side
  rt 120
  fd feet.to.inches :side
  rt 120
end
```

Notice that `feet.to.inches` is invoked three different times following each `fd` command in `triangle.inches`.

Here's the trace of `triangle.inches` that was shown earlier:

```
cs pd triangle.inches 6
( triangle.inches 6 )
( feet.to.inches 6 )
feet.to.inches outputs 72
( feet.to.inches 6 )
feet.to.inches outputs 72
( feet.to.inches 6 )
feet.to.inches outputs 72
triangle.inches stops
```

Each time that `feet.to.inches` is invoked, trace writes a line `(feet.to.inches 6)`. Every time the `feet.to.inches` procedure ends, we see `feet.to.inches outputs 72`. Remember, we see the “outputs” line when an operation ends and the “stops” line when a command ends.

Notice that the trace indents each line of `(feet.to.inches 6)` by the same amount. This is because `feet.to.inches` is invoked each time by the same `triangle.inches` procedure, i.e., the `triangle.inches` procedure that was invoked through the Input Box with `cs pd triangle.inches 6`.

⋮

Branching out

Now consider the following procedure from Brian Harvey's book, *Symbolic Computing* (Harvey, 1997):

```
to tree :size
  if :size < 4 [stop]
  fd :size
  lt 20
  tree :size / 2
  rt 40
  tree :size / 2
  lt 20
  bk :size
end
```

Try this out with some different values and see what it does. Notice that this procedure invokes itself twice in the two lines with the commands `tree :size / 2`. Here's what a trace of the procedure looks like:

```
cs pd tree 6
( tree 6 )
  ( tree 3 )
    tree stops
  ( tree 3 )
    tree stops
tree stops
```

Notice that the two lines `(tree 3)` are indented by the same amount. The two invocations of `(tree 3)` were made by the `tree` procedure that was invoked from the Input Box, i.e., the one that had the input 6.



Solved Problems

1. *Problem:* Write a new version of `circle.stop` with a new stop rule so that it will stop after drawing a complete circle, no matter what the heading of the turtle is when you execute the procedure.

Solution:

We start by taking another look at `circle.stop`:

```
to circle.stop
  fd 1
  rt 1
  if heading = 0 [stop]
  circle.stop
end
```

The stop rule for this procedure is

```
if heading = 0 [stop]
```

There are two problems with this stop rule. First of all, if the turtle starts out with a heading that is some counting number 1, 2, 3... up to 359, the heading of the turtle will change back to 0 in less than 360 steps, so a complete circle will not be drawn.

Second, if the turtle starts out with a heading that contains a fractional part, for example 0.5, 127.127, or 300.9126, the heading will never reach 0 because we only change the heading by adding 1. For example, if the turtle starts with a heading of 0.5, when the turtle goes forward 1 and right 1 the heading changes to 1.5. The next time the turtle goes forward 1 and right 1 the heading changes to 2.5, then 3.5, and so on, until the heading changes to 359.5. After that, the heading changes to 0.5 and the turtle goes around again. Since the heading is never exactly 0, the turtle will never stop. When a procedure keeps going and going without stopping, we say that the procedure is in an *infinite loop*.

We could handle the problem of the infinite loop by changing the stop rule to this:

```
if heading < 1 [stop]
```

Even if the turtle started with a heading that had a fractional part, the turtle would eventually reach a heading zero point something and stop. In general, rules with `<` or `>` are better stop rules than rules with `=`.

⋮

Solved Problems (continued)

But fixing the problem of the infinite loop does not fix the problem of incomplete circles drawn when the turtle starts with a heading that isn't zero.

We need a way to make sure we stop after the circle is completed, and not sooner. However, we can't tell whether the circle is completed just by looking at the turtle's heading or position. We need to pass on information about the turtle's starting state (heading or position) or keep track of the total amount of turning as we go.

Here's one way to solve the problem:

```
to circle.h
  do.circle.h heading
end

to do.circle.h :start.heading
  fd 1
  rt 1
  if heading = :start.heading [stop]
  do.circle.h :start.heading
end
```

The `circle.h` procedure invokes `do.circle.h` with one input—the heading of the turtle. The `do.circle.h` stores this input in the variable `:start.heading`. Since nothing is done to change the value of `:start.heading`, it always contains the heading that the turtle had before `do.circle.h` was invoked.

Each time `do.circle.h` is invoked, the turtle moves forward 1 and right 1. Then the value output by the operation `heading` is compared to the value stored in `:start.heading`. The value reported by `heading` changes every time the turtle turns, but the value stored in `:start.heading` is always the same. When the value reported by `heading` matches the value stored in `:start.heading`, the turtle's heading is back to where it started and we stop.

Something similar could be done by saving the original position of the turtle with `pos` (or `xcor` and `ycor`) and then comparing the stored value of `pos` (or `xcor` and `ycor`) with the turtle's actual `pos` (or `xcor` and `ycor`) each time the turtle moves forward 1 and turns right 1.

⋮

Solved Problems (continued)

With my version of MSWLogo, the `do.circle.h` stop rule still doesn't work sometimes, for example, when the turtle starts with a heading of 14.7. I'm not exactly sure why. I suppose it has something to do with the way that decimal values are stored or computed for headings. There's a small discrepancy of $1.06581410364015e-14$ degrees when the turtle has completed a circle, so the stop rule fails and the turtle doesn't stop.

Here's a solution that seems to always work:

```
to circle.t
  do.circle.t 1
end

to do.circle.t :total.turn
  fd 1
  rt 1
  if :total.turn = 360 [stop]
  do.circle.t :total.turn + 1
end
```

In this procedure, we invoke `do.circle.t` with an input of 1. The value of 1 is stored in `:total.turn`, which keeps track of the total amount of turning that the turtle has done so far. Why is the input 1? When we start the procedure, the turtle hasn't turned at all, so the `:total.turn` should be 0, shouldn't it?

The reason I used an input of 1 is that, by the time we get to the stop rule for the first time, the turtle has already moved forward 1 and turned right 1. So I count the total turning from 1 instead of 0. How could I change the stop rule if `do.circle` was invoked from `circle.t` with an input of 0?

When `do.circle.t` invokes itself, it gives `:total.turn` so far plus 1 as the input. That way, the value of `:total.turn` always matches the total number of `rt 1` commands executed so far. Once the turtle has turned right 360 times, the stop rule fires and we're done. We say that a rule *fires* in an if statement when the predicate outputs `true` and the instruction list which is the second input to the if procedure is run. In the procedure above, the rule `if :total.turn = 360 [stop]` fires when the `:total.turn` is 360.

⋮

Solved Problems (continued)

2. *Problem:* Write a new version of `poly` with a new stop rule so that it will stop after drawing a complete figure, no matter what the heading of the turtle is when you execute the procedure.

Solution:

We start by taking another look at `poly`:

```
to poly :size :angle
  fd :size
  rt :angle
  if heading = 0 [stop]
  poly :size :angle
end
```

One important difference between the `poly` procedure and the `circle` procedure is that each time `circle` is invoked, the turtle turns right 1 degree, but each time `poly` is invoked, the turtle turns right `:angle` degrees where `:angle` is whatever number was given as the second input to `poly`.

This means that if we want to keep track of total turning, we need to add the `:angle` to the `:total.turn` each time, instead of just 1:

```
to poly.stop :size :angle
  do.poly.stop :size :angle :angle
end

to do.poly.stop :size :angle :total.turn
  fd :size
  rt :angle
  if :total.turn = 360 [stop]
  do.poly.stop :size :angle :total.turn + :angle
end
```

This looks like a nice solution. It works for inputs like 120, 90, 72, and 60. But it doesn't work for 144, or 111. Why not? Try executing `poly.stop` with inputs of 100 for the `:size` and 144 for the `:angle`. What happens? We know that the `:total.turn` is getting bigger and bigger, so let's change the stop rule to make the procedure stop when `:total.turn` is greater than 360.

⋮

Solved Problems (continued)

Here's the revised `do.poly.stop` procedure, with the `=` operation changed to a `>` operation:

```
to do.poly.stop :size :angle :total.turn
  fd :size
  rt :angle
  if :total.turn > 360 [stop]
  do.poly.stop :size :angle :total.turn + :angle
end
```

Try running this version by executing `poly.stop` with an input of 144. Now the stop rule will fire, but it fires too soon. What's going on? Let's try a trace:

```
cs poly.stop 100 144
( poly.stop 100 144 )
( do.poly.stop 100 144 144 )
( do.poly.stop 100 144 288 )
( do.poly.stop 100 144 432 )
do.poly.stop stops
do.poly.stop stops
do.poly.stop stops
poly.stop stops
```

The first time `do.poly.stop` is invoked, it is invoked by `poly.stop` with a `:size` of 100, an `:angle` of 144, and a `:total.turn` of 144. The next time `do.poly.stop` is invoked, an `:angle` of 144 has been added to `:total.turn`, so `:total.turn` becomes 288. The time after that, `do.poly.stop` gets inputs of 100 for `:size`, 144 for `:angle`, and $288+144=432$ for `:total.turn`.

We skipped over 360. That's why the stop rule

```
if :total.turn = 360 [stop]
```

never fired in the early version when the input angle was 144: the `:total.turn` went from 288 to 432 and kept on getting bigger, so it was never equal to 360. How do we fix this problem?

⋮

Solved Problems (continued)

You should have noticed by now that an angle input of 144 will draw a star if the `poly` procedure finishes properly. But when it's finished, the heading of the turtle should be the same as when it started. Why doesn't the total turning reach 360, like it does for triangles, or squares or pentagons?

Try this: Draw a five pointed star on a piece of paper. Use a small toy or cut out a small triangle to use for a turtle. Trace the star with the turtle, and see if you can tell how many times the turtle turns around before it completes tracing the path. How does the number of complete turns relate to the 144 degree turns needed to draw a star?

If you followed the path carefully, you noticed that the turtle makes two complete turns by the time it completely traces the star. One complete turn is 360 degrees, so two complete turns is 720 degrees. Let's try another version of

`do.poly.stop`:

```
to do.poly.stop :size :angle :total.turn
  fd :size
  rt :angle
  if :total.turn = 720 [stop]
  do.poly.stop :size :angle :total.turn + :angle
end
```

Now execute the command below.

```
cs poly.stop 100 144
```

We have a star! We can also still draw triangles and other regular polygons. The regular polygons still work because we draw them exactly twice. Your turtle probably moves too fast for you to see this, but you can slow the action down by clicking the Step button and trying the command below.

```
cs poly.stop 100 120
```

Are we done? What if the input angle is 111?

```
cs poly.stop 100 111
```

This one doesn't stop because the `:total.turn` goes from 111 to 222 to 333 to 444 to 555 to 666 to 777, skipping right past 720, and it keeps on growing, so it will never be equal to 720. We can try changing the `=` to `>` again, but, as you've probably guessed, we won't get a complete figure when the angle is 111.

⋮

Solved Problems (continued)

We need to think a little more carefully about how a turtle can return to its original heading. We saw that a turtle needs to turn around once, through 360 degrees, to get back to its original heading when it is drawing the path of a regular polygon. We also saw that it needs to turn around twice, through 720 degrees, to get back to its original heading when it is drawing the path of a five pointed star.

Can you imagine a figure for which the turtle would have to turn complete around three times? Or more?

The fact is, depending on the `:angle` input to `poly.stop`, the turtle might have to turn completely around any number of times. The key to our solution is the fact that any number of complete turns will be some multiple of 360. For example, if the turtle has to turn completely around 5 times, the `:total.turn` needed to complete the path will be $5 \times 360 = 1800$.

We could try adding lots of stop rules to `do.poly`, like this:

```
to do.poly.stop :size :angle :total.turn
  fd :size
  rt :angle
  if :total.turn = 360 [stop]
  if :total.turn = (2 * 360) [stop]
  if :total.turn = (3 * 360) [stop]
  if :total.turn = (4 * 360) [stop]
  if :total.turn = (5 * 360) [stop]
  if :total.turn = (6 * 360) [stop]
  if :total.turn = (7 * 360) [stop]
  do.poly.stop :size :angle :total.turn + :angle
end
```

But we couldn't list all possible cases. So let's look for something that all of these numbers have in common.

Well, if you divide a multiple of 360 by 360, you get a remainder of zero. Will that help us? Actually, it just so happens that there's a handy Logo primitive (built-in procedure) called `remainder`.

⋮

Solved Problems (continued)

Here are some examples that demonstrate the `remainder` primitive:

```
show remainder 10 4
2
show remainder 10 5
0
show remainder 720 360
0
show remainder 55 360
55
show remainder 420 360
60
show remainder 0 360
0
show remainder 415 360
55
```

Get the idea? The `remainder` operation outputs whatever's left over when you divide one number by another. So, when we want to see if one number is a multiple of another, we use the `remainder` operation and check to see if the remainder is 0.

Here's how we can use it in `do.poly.stop`:

```
to do.poly.stop :size :angle :total.turn
  fd :size
  rt :angle
  if (remainder :total.turn 360) = 0 [stop]
  do.poly.stop :size :angle :total.turn + :angle
end
```

Now `poly.stop` will work with any inputs.

⋮

Solved Problems (continued)

3. *Problem:* Write a recursive version of the `spin` program given by Brian Harvey on page 190 of *Symbolic Computing* (Harvey, 1997). What is the advantage of having a recursive version?

Solution:

Brian Harvey's version of `spin` looks like this:

```
to spin :turns :command
  repeat :turns [
    protect.heading :command right 360 / :turns
  ]
end

to protect.heading :squig
  local "oldheading
  make "oldheading heading
  run :squig
  setheading :oldheading
end
```

Before diving into the recursive version, let's think for a minute. What might be an advantage of doing a recursive version?

Our recursive version of `poly.stop` makes much more than regular polygons because we are not limited to total turns of 360 degrees. Now look at the `spin`. The total turn for `spin` will always be 360 degrees, because each turn is 360 divided by `:turns`. Let's write a recursive version of `spin` that allows total turns of 720, 1080, etc.

When we wrote `poly.stop`, our input was `:angle` instead of `:turn`. That allowed us to use any angle instead of just angles that divide into 360 some whole number of times. So our recursive version of `spin` should take `:angle` as input instead of `:turns`. It also has to keep track of the total amount the turtle has turned and check to see if this amount is a multiple of 360. Here's how it looks:

```
to spin.r :angle :command
  do.spin :angle :command :angle
end
```

⋮

Solved Problems (continued)

Here's `do.spin`:

```
to do.spin :angle :command :total.turn
  protect.heading :command rt :angle
  if equal? 0 remainder :total.turn 360 [stop]
  do.spin :angle :command sum :total.turn :angle
end
```

Now we can do spins like this:

```
spin.r 60 [squiggle]
spin.r 120 [squiggle]
spin.r 72 [squiggle]
spin.r 144 [squiggle]
spin.r 135 [squiggle]
```

The last two of these spins could not be drawn with Brian Harvey's version of `spin`.

⋮

Supplementary Problems

1. *Problem:* Write a version of `poly` that switches the amount of forward motion and the amount of turn each time it is recursively invoked.

Solution:

The trick is to switch the order of `:size` and `:angle` on the recursive call to `do.switchpoly`. Also, notice that `:size` gets added to `:total.turn` because `:size` is the value that gets passed on as the `:angle` when `do.switch.poly` is recursively invoked.

```
to switchpoly :size :angle
do.switchpoly :size :angle :angle
end

to do.switchpoly :size :angle :total.turn
fd :size
rt :angle
if equal? 0 remainder :total.turn 360 [stop]
do.switchpoly :angle :size :total.turn + :size
end
```

2. *Problem:* Modify the tree procedure to make the tree asymmetrical.

Solution:

```
to tree.a :size
if :size < 5 [fd :size back :size stop]
fd :size / 3
lt 30
tree.a :size * 2 / 3
rt 30
fd :size / 6
rt 25
tree.a :size / 2
lt 25
fd :size / 3
rt 25
tree.a :size / 2 lt 25
fd :size / 6
bk :size
end
```



References

Abelson, H., & diSessa, A. (1986). *Turtle Geometry: The Computer as a Medium for Exploring Mathematics* (First MIT Press paperback edition ed.). Cambridge, MA London, England: The MIT Press.

Harvey, B. (1997). *Symbolic Computing* (2nd ed. Vol. 1). Cambridge, MA: The MIT Press.



The Author

TJ Leone owns and operates Leone Learning Systems, Inc., a private corporation that offers tutoring and educational software. He has a BA in Math and an MS in Computer Science, both from the City College of New York. He spent two years in graduate studies in education and computer science at Northwestern University, and six years developing educational software there. He is a former Montessori teacher and currently teaches gifted children on a part time basis at the Center for Talent Development at Northwestern University in addition to his tutoring and software development work. His web site is <http://www.leonelearningsystems.com>.