Lecture 2. Turtle Graphics Programming

2.1 Scheme Programming: Where We Left Off Last Time

In the previous lecture, we just began to work with the Scheme interpreter and to get a sense for the rules that it uses to evaluate expressions. Just to recap: in dealing with the interpreter we type in expressions after the prompt; then type ENTER (thereby telling the interpreter to read and evaluate the just-completed expression); and the interpreter prints out the result of evaluating the expression, assuming there is no error. If the expression is (for some reason) not a valid Scheme expression, the interpreter prints out an error message and places us in the interactive debugger—which, for the moment, we will refrain from investigating.

2.1.1 The Scheme Interpreter and Its Rules

We can think of the interpreter as an abstract machine that takes expressions as input, and evaluates those expressions to return data objects of some sort. Our experience with the interpreter is thus far limited, but we have seen some examples in which the interpreter evaluates expressions to return numbers. Over the course of the next several weeks, part of our agenda will be to develop an increasingly finer-grained picture of what the interpreter does—more specifically, what rules it uses to evaluate expressions. As it happens, these rules are remarkably simple—in total, they can fit on a page or two of description—but they are also tremendously expressive. A major reason for this quality of expressiveness is something that we have already glimpsed in the case of numeric and REPEAT expressions: namely, that the same rule-set that evaluates simple expressions can be used to evaluate progressively more complex expressions built from the simpler ones.

2.1.2 Evaluating Numbers; Evaluating Calls to Primitive Procedures

One rule that we have already encountered:

Numbers evaluate to themselves.

Examples:
```
>>> 7
7

>>> 44.5
44.5
```

We have also seen examples of expressions in which we are using *primitive arithmetic procedures*:

```
>>> (+ 9 5)
14

>>> (- 16 7.8)
8.2
```

The rule for evaluating expressions of this type can be stated as follows:

To evaluate a primitive procedure call, first evaluate each of the argument expressions; the results of evaluating these expressions will be values that can be passed as arguments to the primitive procedure. The interpreter applies the primitive procedure "directly" and returns the result.

What is meant by saying that the primitive procedure is applied "directly"? The idea is that the interpreter has certain procedures (such as addition) built into it: we treat these primitive procedures as "black boxes" that the interpreter can use to perform tasks such as arithmetic. Later in this lecture, we will acquire methods for building our own procedures (using the interpreter's primitive procedures as building blocks).

Just to step through a particular example: consider the expression

```
>>> (+ 9 5)
```

The interpreter first evaluates the two argument expressions, 9 and 5. These are numeric expressions which (according to our very first rule) simply evaluate to the numbers 9 and 5. The interpreter then uses the addition procedure with these numbers as arguments, and returns the number 14 as the overall result.

2.1.3 Nested Expressions

The primitive procedure evaluation rule can be used without alteration to evaluate nested expressions like the following:

```
>>> (+ 3 (* 6 8))
51

>>> (* 4.2 (+ 21 2))
96.6

>>> (+ (* 2 3) (- 8 6))
8
```

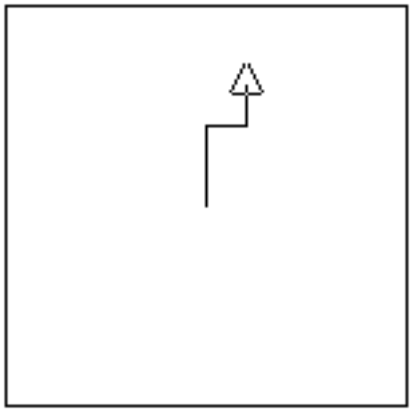
Again, all that we are doing here is evaluating the argument expressions (which may themselves involve calls to primitive procedures); the resulting number values are then used as arguments to the overall arithmetic procedure. As an example, consider the first expression:

```
>>> (+ 3 (* 6 8))
```

Here, the first argument expression is a number, which evaluates to 3; the second argument expression is a call to a primitive procedure, which (using the primitive procedure rule) evaluates to 48. These values are then used as the arguments to the addition procedure, and the interpreter finally returns the value 51 as the final result.

2.1.4 Beginning Turtle Graphics: Forward and Right

In the previous lecture, we also encountered the fundamental procedures used in SchemePaint's turtle-graphics programming: namely, `forward` and `right`. These procedures are used, respectively, to move the "turtle" (the little programmable cursor) forward in the direction that its "head" is pointing, and to change its direction.



```
(forward 20)
(right 90)
(forward 10)
(right -90)
(forward 10)
```

If you have played a bit with these two turtle-graphics procedures, you have probably already discovered that they can take any numeric values (including non-integer and negative values) as arguments. For instance, the expression

```
(right -90)
```

above effectively turns the turtle left 90 degrees.

Since turning 360 degrees is the same as turning in a complete circle, the expressions

```
(right 360)
```

and

```
(right 0)
```

have the same effect (namely, none!) on the turtle's direction. For that matter,

```
(right 720)
```

or

```
(right 1080)
```

likewise have no effect. And by the same token,

```
(right -90)
```

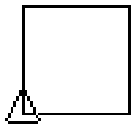
```
(right 270)
```

```
(right 630)
```

and so forth, all have the effect of turning the turtle left 90 degrees.

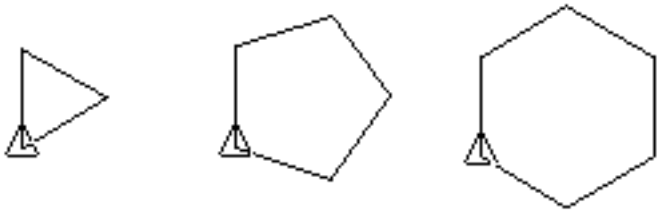
2.1.5 Repeating sequences of turtle steps

Besides `forward` and `right`, the previous lecture also introduced `repeat`, which is SchemePaint's technique for repeating sequences of turtle-graphics expressions. For instance, by repeating four forward-moves-and-right-face-turns, we effectively tell the turtle to draw a square:



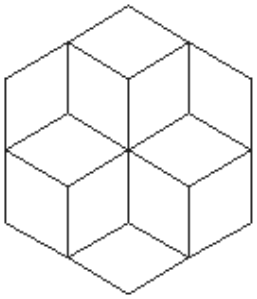
```
(repeat 4 (forward 25) (right 90))
```

In similar fashion, repeating simple forward moves (followed by turns of the appropriate angles) can produce any regular closed polygon:



```
(repeat 3 (forward 25) (right 120))

(repeat 5 (forward 25) (right 72))

(repeat 6 (forward 25) (right 60))
```

And we noted that—much like arithmetic expressions—`repeat` expressions may be nested within larger, more complex `repeat` expressions:



```
(repeat 6 (repeat 6 (forward 25) (right 60)) (right 60))
```

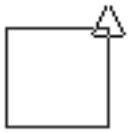
2.2 Expanding the Turtle-Graphics Vocabulary

As a first step beyond the topics of the previous lecture, we may as well add a bit to our vocabulary of turtle-graphics procedures. First, although `forward` and `right` are perfectly good procedure names, SchemePaint includes equivalent procedures `fd` and `rt` with (respectively) the same meanings. Thus, we could produce our earlier repeated-hexagon pattern by typing:

```
(repeat 6 (repeat 6 (fd 25) (rt 60)) (rt 60))
```

Because using `fd` and `rt` makes our expressions briefer to read and type, we tend to use these procedures instead of their more verbosely-named equivalents.

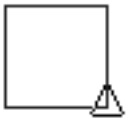
There are other procedures worth knowing about. Instead of moving the turtle forward, we can move it backward using the `backward` (or, equivalently, `bk`) procedure. Compare the following expression with the earlier square-generating expression:



```
(repeat 4 (bk 25) (rt 90))
```

We don't have to use `bk`—if we prefer, we could instead use negative arguments to the `fd` procedure—but sometimes using `bk` is a bit more expressive.

Similarly, instead of using a negative argument for the `rt` procedure, we can instead use the `left` (or equivalently, `lt`) procedure to turn the turtle to its left:



```
(repeat 4 (fd 25) (lt 90))
```

There are also turtle-graphics procedures that allow us to "pick up the turtle's pen" (and to "put it down"). The idea is that when the turtle moves, it will either draw a line (if the pen is down), or not draw anything (if the pen is up). The `turtle-penup` procedure (or, equivalently, `pu`) is a procedure of no arguments that, when called, tells the turtle to pick up its pen; subsequent turtle-moves will not draw lines (until the pen is put back down). Similarly, the `turtle-pendown` (or `pd`) procedure tells the turtle to put its pen down:

```
(repeat 4 (pd) (fd 10) (pu) (fd 10) (rt 90))
```

It may be worth calling attention to the fact that `pd` and `pu` are procedures that happen to take no arguments; thus, in the expression above, when we wish to call, say, the `pd` procedure, we call it just the same way as any other procedure would be called, but without any argument expressions:

```
(pd)
```

There really isn't anything strange about this: some procedures (such as `fd` and `rt`) take single arguments; some (such as `+` or `*`) take two arguments[1];

---

[1]In point of fact, these particular procedures can take variable numbers of arguments; but for the moment, we will think of them as two-argument procedures.

while others, such as pd and pu, take zero arguments. In every case, the manner in which the Scheme interpreter evaluates procedure calls is perfectly consistent: evaluate all argument expressions (if any) and use the results as the arguments (if any are needed) for the given procedure.

2.2.1 A Summary of Turtle-Graphics Procedures Thus Far

FORWARD procedure of one (numeric) argument
FD
Moves the turtle forward by a given amount:

```
(fd 20)
```

BACKWARD procedure of one (numeric) argument
BK
Moves the turtle backward by a given amount:

```
(bk 20)
```

RIGHT procedure of one (numeric) argument
RT
Turns the turtle to its right by a given number of degrees:

```
(rt 45)
```

LEFT procedure of one (numeric) argument
LT
Turns the turtle left by a given number of degrees:

```
(lt 90)
```

TURTLE-PENUP procedure of no arguments
PU
Tells the turtle to stop drawing lines when it moves:

```
(pu)
```

TURTLE-PENDOWN procedure of no arguments
PD
Tells the turtle to draw lines when it moves:

```
(pd)
```

REPEAT special form (not a procedure)
Tells the turtle to repeat a sequence of expressions a given number of times:

```
(repeat 8  (fd 25) (rt 45))
```

2.2.2 Turtle Expressions Also Return Values

There is one other point that should perhaps be cleared up before we go any further. You may have noticed that in the discussion of arithmetic expressions, we placed great emphasis on the fact that the Scheme interpreter evaluates expressions and returns values as the result of the interpretation process: for instance, evaluating the expression

```
(+ 1 2)
```

returns the value 3.

On the other hand, what about turtle expressions? When we evaluate the expression

```
(fd 20)
```

we certainly see the turtle move on the screen; but how does this relate to the notion of a "returned value"?

In point of fact, the Scheme interpreter *does* return a value for turtle-graphics expressions, just as it does for arithmetic expressions. You might have noticed, for instance, that when you evaluate the expression `(fd 20)` at the Scheme interpreter, you actually see the following in the transcript window:

```
>>> (fd 20)
20
```

In this case, the Scheme interpreter has returned 20 as the value of the call to the `fd` procedure.

So the Scheme interpreter does in fact return values for calls to `fd` (and it returns values for `rt` expressions, `repeat` expressions, and so forth). But, having said that, we also note that we almost never care about these returned values. That is, we evaluate calls to `fd` in order to make the turtle move, and not to use the returned value itself.

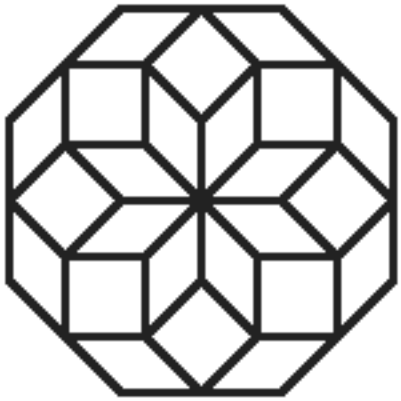
The issue that we are raising here is often referred as the question of whether expressions are evaluated "for value" or "for effect". In the case of our sample arithmetic expressions, these are evaluated "for value"; while turtle-graphics expressions are evaluated "for effect" (i.e., we generally do not intend to use the expression's returned value for any particular purpose). We will return to this topic later on.

2.3 Continuing Experiments with REPEAT; Nested REPEATS

Now let's get back to experimenting with the turtle. Here's a recipe for a regular octagon:

```
(repeat 8 (fd 20) (rt 45))
```

And, as suggested by the earlier examples, we can generate a rotated sequence of octagons:

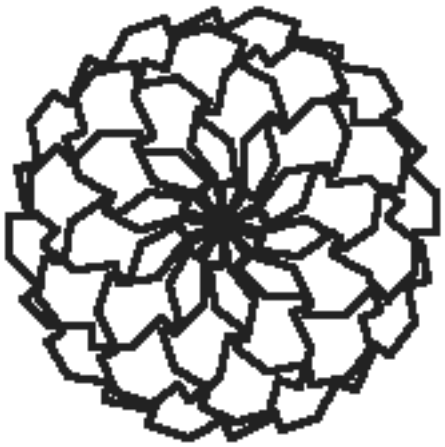


```
(repeat 8 (repeat 8 (fd 20) (rt 45))
          (rt 45))
```

There are plenty of possibilities for experimentation; for instance, rather than repeat simple forward-right sequences, we can repeat slightly more complicated patterns. Here, we've repeated a "jagged" forward move to make a kind of "buzz-saw-like octagon":

```
(repeat 8 (fd 10) (rt 90) (fd 5) (lt 90) (fd 10) (rt 45))
```

We can rotate these "jagged octagons" to make a flower-like pattern:



```
(repeat 10
        (repeat 8 (fd 10) (rt 90) (fd 5) (lt 90) (fd 10) (rt 45))
        (rt 36))
```

And we can incorporate pu and pd expressions to pick up the turtle's pen and have it make a series of "flower-like" objects. Try following through the sequence of turtle commands to see how this pattern was made:

```
(repeat 3
        (fd 30)
        (repeat 10
                (repeat 8 (fd 4) (rt 90) (fd 2) (lt 90) (fd 4) (rt 45))
                (rt 36))
        (bk 30)
        (rt 90) (pu) (fd 40) (lt 90) (pd))
```

This is a recipe for a five-pointed star. Again, try working through the turtle moves to see how this shape was generated. You might also experiment with similar repeat patterns to generate (e.g.) six- or seven-pointed stars.

```
(repeat 5 (fd 30) (rt 144) (fd 30) (lt 72))
```

```
(repeat 6 (repeat 5 (fd 10) (rt 144) (fd 10) (lt 72))
          (pu) (fd 40) (pd) (rt 60))
```

```
(repeat 4
        (repeat 6 (repeat 5 (fd 10) (rt 144) (fd 10) (lt 72))
                  (pu) (fd 40) (pd) (rt 60))
        (pu) (fd 20) (pd) (rt 90))
```

2.4 The SchemePaint Interface

This is a good time to pause a bit on the "theoretical" side of Scheme programming, and to spend more time on the use of the MacScheme interface (and the additions to it that have been incorporated in our SchemePaint interface).

2.4.1 The General MacScheme Interface

If you have been playing with the SchemePaint system, you have probably by now become familiar with several features of the MacScheme interface. Many of these are described at length in the MacScheme manual and in the "Programming in MacScheme" text; here, we will only mention a few salient points.

There are four menus at the top of the screen that are supplied by the basic MacScheme system: the **File**, **Edit**, **Command**, and **Window** menus. Over time, the selections available on these various menus will become more familiar; one that you might make use of now is the Reset choice on the **Command** menu. This choice will reset the Scheme system, and is handy if you wish to exit from the interactive debugger (using this choice is an alternative to typing "q" at the debugger prompt, as mentioned in the previous lecture).

If you are familiar with standard commercial Macintosh programs, you will probably find no difficulty in using most of the **Edit** menu choices. For instance, you can copy expressions within the transcript window by using the Cut and Paste (or Copy and Paste) choices. This is especially helpful if you have typed in a long expression (like one of those nested `repeat` expressions shown earlier), and you wish to reuse this expression as part of a larger one: in this event, you can select the expression-to-copy with the mouse; select Copy from the **Edit** menu; move to the point in the transcript window where you want to place the expression-to-copy; and finally, select Paste from the **Edit** menu.

The **Window** menu is used to select between multiple windows on the MacScheme screen. We haven't needed this menu just yet, and won't until we discuss the topic of opening up new windows in MacScheme for the creation and editing of files—a topic we will discuss in the next lecture. Several of the **File** menu choices will be introduced in that same discussion.

2.4.2 Using SchemePaint's Paint Menu

In addition to the MacScheme menus, there are two additional menus provided by the SchemePaint system. The Paint menu includes a number of handy features for working with SchemePaint—though several of them will still seem opaque for the present. The three that are most useful for our purposes are the Clear Windows choice (which clears out the SchemePaint window); the Toggle Grid Mode choice (which determines whether the next clear-windows operation will draw a gray grid on the SchemePaint window or leave it blank); and the Toggle Palette Size choice, which allows you to move back and forth between smaller and larger sets of colors.

2.4.3 Using SchemePaint's Turtle Menu

The choices on the **Turtle** menu should, in most cases, be clear given the introduction to turtle-graphics programming thus far. The Pen Up and Pen Down choices permit you to specify via the menu whether the turtle will draw lines or not (as we have seen, there are SchemePaint procedures that perform the same functions). The Home and Center choices are similar but not identical: both, when selected, will move the turtle back to the point (0, 0) at the center of the **SchemePaint** window and will reset the turtle's heading to due north, but the Home command causes the turtle to draw a line (when the pen is down) while the Center command will simply center the turtle without drawing lines regardless of the turtle's pen-state. (Most people regard the Center command as more useful than the Home command—its effect is typically the one desired.) Finally, the Show or Hide Turtle command will show the turtle (if it is not visible) and hide it (if it is). Note that when the turtle moves beyond the portion of the plane visible in the **SchemePaint** window (for example, if we first move the turtle forward 200 steps from the center position), then the turtle will not show up on the screen even if it is nominally "visible": we can only see a visible turtle when it is actually within the visible range of the **SchemePaint** window.

2.4.4 Using the Paint Window

The **Paint** window in SchemePaint allows you (among other features) to choose particular pen colors and pen-widths. For the time being, these are probably the only features of the **Paint** window that you should expect to use (in one or two cases, unsuccessful experimentation can prove frustrating).

By selecting one of the sixteen color-boxes toward the left of the **Paint** window (for now ignore the two boxes with circles in the bottom row), you can choose a color in which the pen (and turtle) will draw lines; similarly, the three dots of different sizes at the top of the fourth column specify three different pen widths you can use. When you have selected a color and pen-width, you can use the mouse to draw lines by hand on the **SchemePaint** window; these can be combined freely with patterns drawn by the turtle.

If you wish to fill a given region in a particular color, you can use the `fill` box at the top right of the **Paint** window. Here's how it operates: you first select the color in which you want to fill; then select the `fill` box; then click the mouse twice anywhere in the region that you want to fill within the **SchemePaint** window (sometimes this takes rather delicate hand-eye coordination, for tiny areas!). The program will, after a moment's wait, fill the selected region with the given color. When you wish to go back to drawing, select one of the pen-width boxes in the fourth column.

You should actively *avoid* using the `click` or (especially) the `proc` choices in the **Paint** window for the present. These will be explained more fully in a later lecture.

2.5 Defining New Names

Back to theory.

Thus far, we have written only a few types of expressions for the Scheme interpreter to evaluate. It is now time to expand our repertoire.

Scheme allows us to *bind* names to objects by using the define special form, as shown in the following expressions:

```
>>> (define a 3)
a
>>> (define b 40)
b
```

The way to think about these expressions is that we are creating new names, `a` and `b`, for the numbers 3 and 40, respectively. That is, we now think of `a` as a name for the number 3, and `b` as a name for the number 40. If we now tell the Scheme interpreter to evaluate these names, the interpreter returns the objects to which the names refer (or, to put it a bit more formally, the interpreter returns the values to which the names have been bound):

```
>>> a
3
>>> b
40
```

In these two expressions, we have first asked the Scheme interpreter to evaluate the name `a` (the interpreter returns the value 3) and then the name `b` (the interpreter returns 40).

Let us formalize these observations in a new rule:

Names evaluate to the objects to which they have been bound.

Note that, by using this rule, we can now predict what will happen when names are used as parts of larger expressions:

```
>>> (+ a (* b 2))
83
```

2.5.1 Define as a Special Form

We have already noted that not all compound Scheme expressions are, strictly speaking, procedure calls: in particular, we mentioned that `repeat` is a special form (rather than a Scheme procedure). Similarly, `define` is not a Scheme procedure, but is instead a special form. [See *Programming in MacScheme*, pp. 24-27.]

The upshot of this discussion is that we have a new rule for Scheme programming. This one is sort of a "rule collection"—it tells us that there are certain Scheme expressions that don't obey the standard procedure-evaluation rules:

To evaluate a compound expression beginning with a special form, you cannot use the primitive-procedure evaluation rule. Each special form has its own unique rule for evaluation. That's why they're called special forms.

Thus far, `repeat` and `define` are the only special forms that we have encountered. A few more will be introduced very shortly.

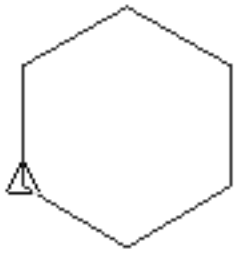
2.6 Defining New Procedures

See Programming in MacScheme, Chapter 3 for a description of how the `define` special form may also be used to define new procedures, and how user-defined procedure calls (so-called "compound procedure calls") are evaluated by the Scheme interpreter.

Here are two initial examples of turtle-graphics procedures created using define. Note that these two first examples are procedures that take no arguments:

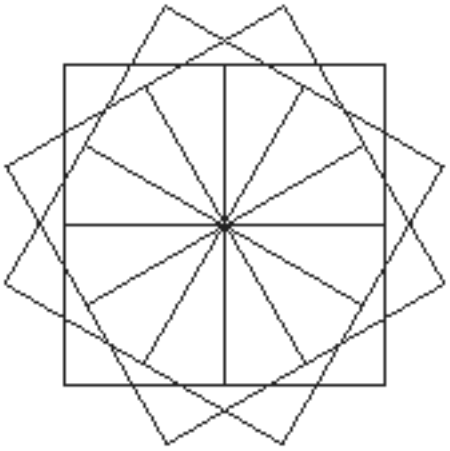
```
(define (square)
  (repeat 4 (fd 40) (rt 90)))

(define (hexagon)
   (repeat 6 (fd 30) (rt 60)))
```

We can call our new procedures on no arguments:



```
(hexagon)
```

And of course we can use these procedure calls within repeat expressions:

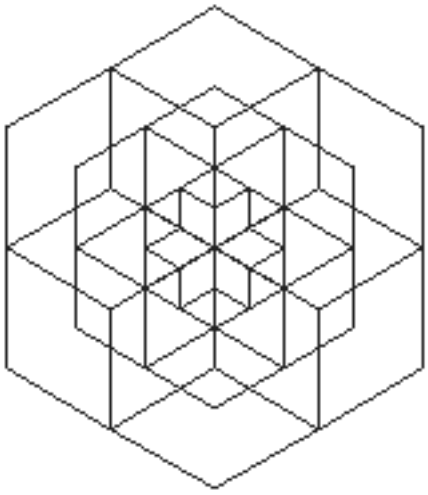


```
(repeat 12 (square) (rt 30))
```

We can pursue this procedure-writing technique to create procedures that take various numbers of arguments. For instance, let us rewrite our square and hexagon procedures so that they take a single argument corresponding to the side-length of the given shape:

```
(define (square side)
  (repeat 4 (fd side) (rt 90)))

(define (hexagon side)
  (repeat 6 (fd side) (rt 60)))
```

Now we have a method for making squares (or hexagons) of any desired size:



```
(repeat 6 (hexagon 10) (hexagon 20) (hexagon 30) (rt 60))
```

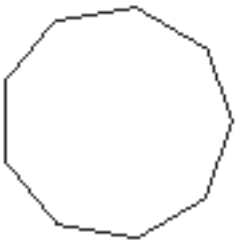
You may have noticed that there is a general pattern for producing regular polygons of n sides: the turtle moves forward by some amount and then turns right 360/n degrees, repeating this movement n times. Consider, for instance, our earlier recipes for triangles, pentagons, and octagons

```
(repeat 3 (fd 40) (rt 120))
(repeat 5 (fd 40) (rt 72))
(repeat 8 (fd 40) (rt 45))
```

We can capture this idea by writing a general polygon procedure that takes two arguments: the number of sides of the desired polygon, and the side-length (or, if you like, the "size") of the polygon:

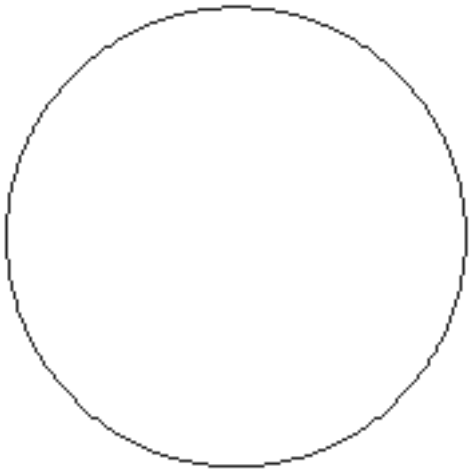
```
(define (polygon n-sides side-length)
  (repeat n-sides (fd side-length) (rt (/ 360 n-sides))))
```

Now we can easily create any regular polygon just by using our new procedure. Here's a nonagon (nine-sided polygon) each of whose sides is of length 20:



```
(polygon 9 20)
```

Suppose we try calling the polygon with a really large number of sides: if we want the result to fit on the screen, we have to use a small side-length. For instance, let us create a 360-sided polygon (in which the turtle turns right by 1 degree after each forward move), with each of its 360 sides being of length 1:



```
(polygon 360 1)
```

The result, to the eye, looks very much like a circle. You can think of this experiment in a number of ways: one observation (which you may remember way back from high-school geometry) is that a circle may be thought of as a polygon with an infinite number of sides. Here, we have seen that polygons certainly get more "circle-like" as we increase the number of sides: we can imagine that if we were to try calling `polygon` with ever-larger number-of-sides arguments (and smaller side-length arguments) we would get shapes that would approximate circles ever more closely.

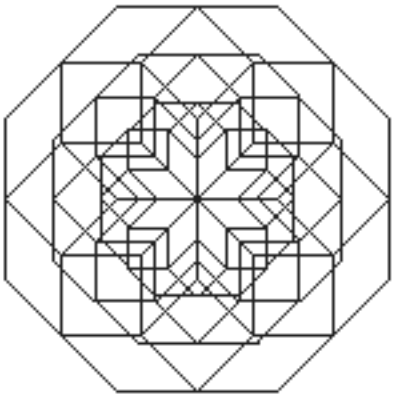
2.7 Using procedures as parts of more complex procedures

The same theme of "using simple building blocks to create more progressively more complex ideas" starts coming into its own once we create new procedures. For example, we could use our polygon procedure to write a new octagon procedure:

```
(define (octagon side)
  (polygon 8 side))
```

And now we can use the octagon procedure in creating still more complex patterns:

```
(define (octagon-pattern side)
  (repeat 8
          (octagon side)
          (octagon (* 1.5 side))
          (octagon (* 2 side))
          (rt 45)))
```



```
(octagon-pattern 10)
```

Here's another example: noting that our 360-sided polygon was a good approximation to a circle, let us create a procedure that makes only part of that circle. This procedure, named arcr, will take as its argument the number of degrees of the circle that we wish the turtle to draw (with 360 corresponding to the entire circle):

```
(define (arcr size)
  (repeat size (fd 1) (rt 1)))
```

Now we can adjoin two arcs of 60 degrees (one-sixth of the circle) to make something like a "petal":

```
(define (petal)
  (arcr 60) (rt 120) (arcr 60) (rt 120))
```

```
(petal)
```

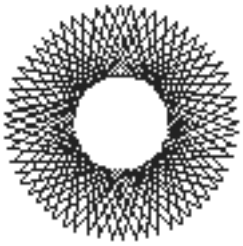
And now, repeating a bunch of petals makes a new flower-like pattern (this picture uses SchemePaint's fill feature to partially color the design):

```
(repeat 18 (petal) (rt 20))
```

Even with this still relatively recent exposure to programming, it is now feasible to explore a huge variety of projects. Just to take an example, suppose we want to see what would happen if we vary the polygon procedure a bit:

```
(define (newpolygon n-times side-length angle)
   (repeat n-times
          (fd side-length)
          (rt angle)
          (fd side-length)
          (rt (* 2 angle))))
```

In other words, the repeated sequence will not be a simple forward-right pattern, but rather a forward-right-forward-even-more-right pattern.



```
(newpolygon 100 20 142)
```

```
(newpolygon 100 20 144)
```

2.8 A summary of the Scheme interpreter's rules, thus far

Numbers evaluate to themselves.

To evaluate a primitive procedure call, first evaluate each of the argument expressions; the results of evaluating these expressions will be values that can be passed as arguments to the primitive procedure. The interpreter applies the primitive procedure "directly" and returns the result.

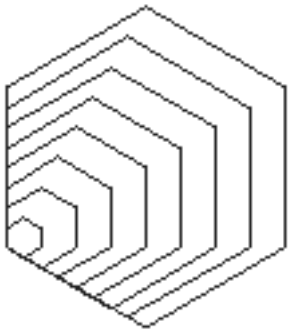
Names evaluate to the objects to which they have been bound.

To evaluate a compound expression beginning with a special form, you cannot use the primitive-procedure evaluation rule. Each special form has its own unique rule for evaluation. That's why they're called special forms.

(So far, we have seen two special forms: `define` and `repeat`).

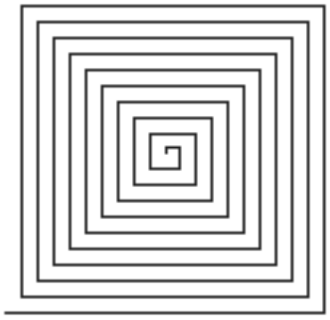
To evaluate a compound procedure call, first evaluate all the argument expressions; the results of evaluating these expressions will be values that can be passed as arguments to the compound procedure. Once you have these, evaluate the body of the procedure as though the procedure arguments were actually names for the just-evaluated argument values. The result of the procedure call is the value of the last expression in the body of the procedure.

```
(define (hexagon-countdown count)
  (cond ((= count 0) 0)
        (else (hexagon (* count 5))
              (hexagon-countdown (- count 1)))))
```



```
(hexagon-countdown 8)
```

```
(define (squiral count side-length)
  (cond ((= count 40) 0)
        (else (fd side-length)
              (rt 90)
              (squiral (+ 1 count) (+ 2 side-length)))))
```



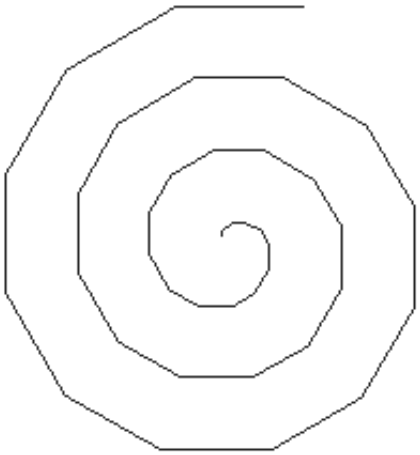
```
(squiral 0 1)
```

2.9.1 Non-turtle-graphics recursion

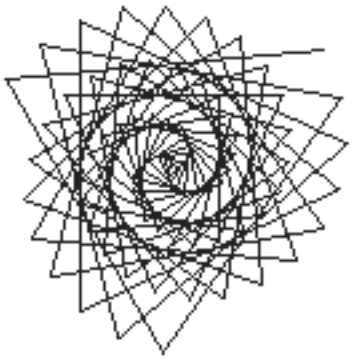
```
(define (factorial n)
  (cond ((= n 0) 1)
        (else (* n (factorial (- n 1))))))

>>> (factorial 20)
2432902008176640000

(define (fibonacci n)
   (cond ((= n 0) 0)
         ((= n 1) 1)
         (else (+ (fibonacci (- n 1)) (fibonacci (- n 2))))))

>>> (fibonacci 20)
6765
```

2.9.2 Variations on the spiral

```
(define (polyspi n-sides side angle)
  (cond ((= n-sides 0) 0)
        (else (fd side)
              (rt angle)
              (polyspi (- n-sides 1) (+ side 1) angle))))
```



(polyspi 40 1 30)



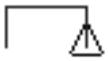
```
(polyspi 80 1 115)
```

2.9.2 The C-Curve

```
(define (c-curve size level)
  (cond ((= level 0) (fd size))
        (else (c-curve size (- level 1))
              (rt 90)
              (c-curve size (- level 1))
              (lt 90))))
```

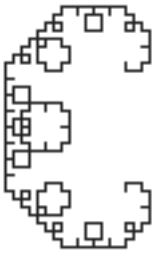
```
(c-curve 10 0)
```

```
(c-curve 10 1)
```

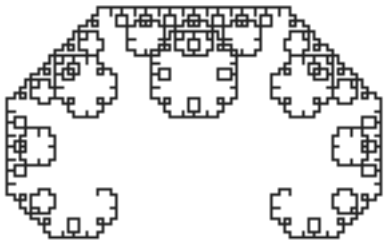


```
(c-curve 10 2)
```

```
(c-curve 10 3)
```



```
(c-curve 2 8)
```



```
(c-curve 1.5 10)
```