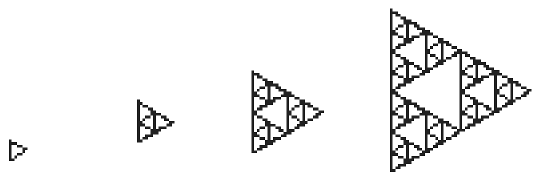
Lecture 3. Recursion

3.1 "Thinking Recursively"

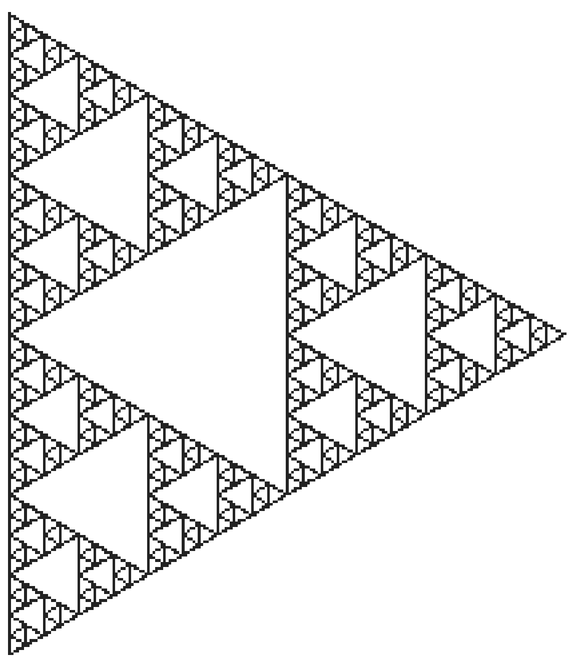
Reading: *Programming in MacScheme*, Chapters 4 and 5.

3.2 The Sierpinski Triangle

```
(define (nested-triangle size level)
      (cond ((= level 0) 0)
            (else (repeat 3
                          (nested-triangle (/ size 2) (- level 1))
                          (fd size)
                          (rt 120)))))
```



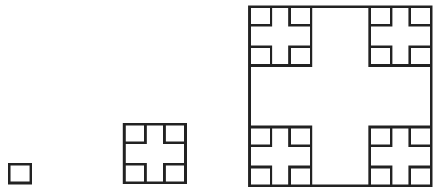
```
(nested-triangle 5 1)
(nested-triangle 10 2)
(nested-triangle 20 3)
(nested-triangle 40 4)
```



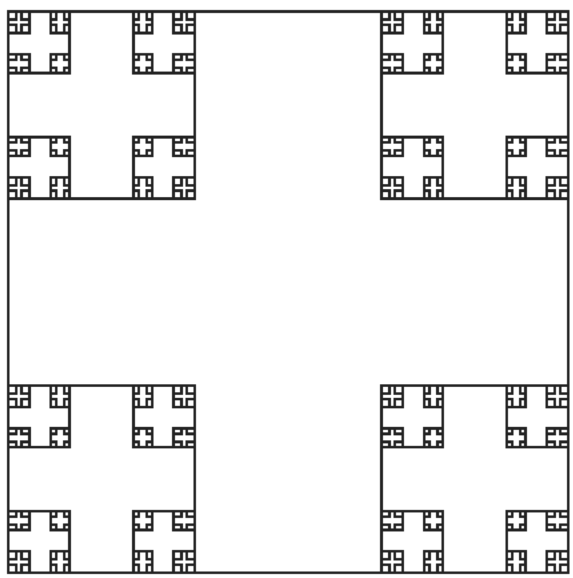
```
(nested-triangle 160 6)
```

3.3 A "Sierpinski Square"?

```
(define (nested-square size level)
  (cond ((= level 0) 0)
        (else (repeat 4
                      (nested-square (/ size 3) (- level 1))
                      (fd size)
                      (rt 90)))))
```



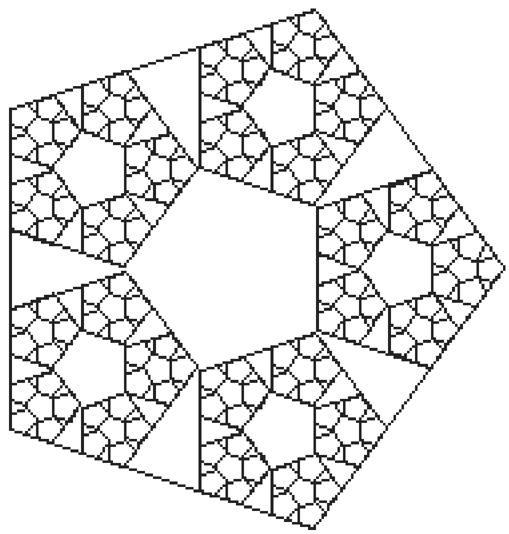
```
(nested-square 5 1)
(nested-square 15 2)
(nested-square 45 3)
```



```
(nested-square 140 5)
```

3.3 A General Nested-Polygon Procedure

```
(define (nested-poly nsides ratio size level)
  (cond ((= level 0) 0)
        (else (repeat nsides
                      (nested-poly nsides ratio
                                   (/ size ratio) (- level 1))
                      (fd size)
                      (rt (/ 360 nsides))))))
```

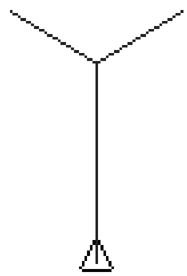


(nested-poly 5 2.65 80 4)

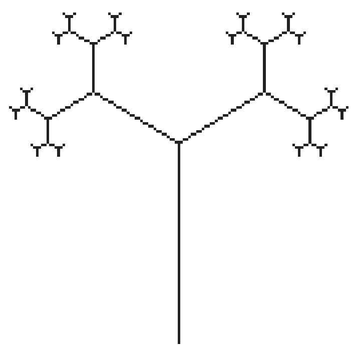
3.4 Making Tree-Like Shapes

```
(define (branch length level)
  (cond ((= level 0) 0)
        (else (fd length)
              (lt 60)
              (branch (/ length 2) (- level 1))
              (rt 120)
              (branch (/ length 2) (- level 1))
              (lt 60)
              (bk length))))
```

(branch 50 1)

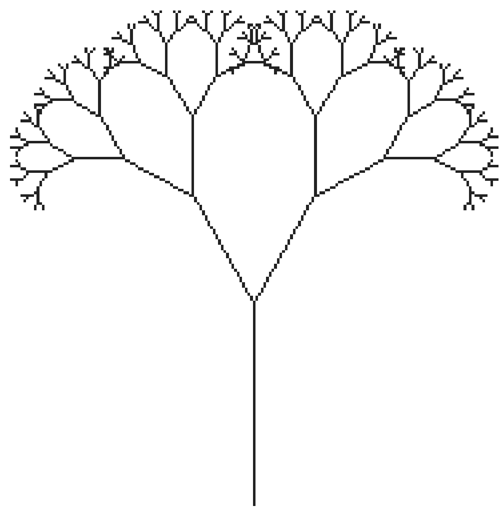


(branch 50 2)



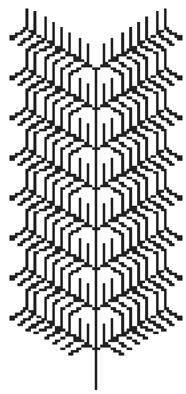
(branch 50 6)

```
(define (branch length angle ratio level)
  (cond ((= level 0) 0)
        (else
          (fd length)
          (lt angle)
          (branch (/ length ratio) angle ratio (- level 1))
          (rt (* 2 angle))
          (branch (/ length ratio) angle ratio (- level 1))
          (lt angle)
          (bk length))))
```



```
(branch 50 30 1.6 8)
```

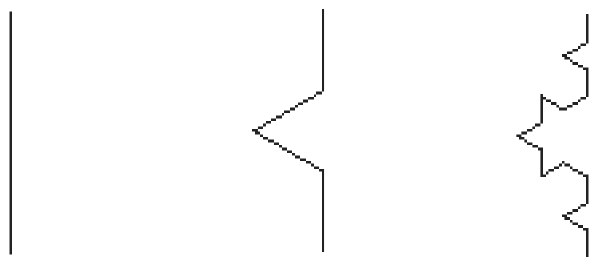
```
(define (feather size level)
  (cond ((= level 0) 0)
        (else (repeat 8 (fd (/ size 8))
                        (lt 60)
                        (feather (/ size 4) (- level 1))
                        (rt 120)
                        (feather (/ size 4) (- level 1))
                        (lt 60))
              (bk size))))
```



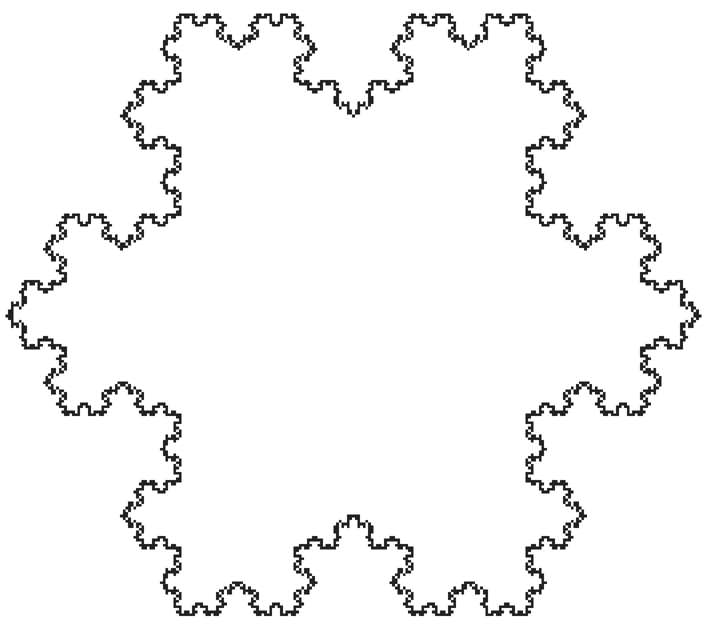
```
(feather 80 3)
```

3.5 The Koch Snowflake

```
(define (snowflake size level)
  (cond ((= level 0) (fd size))
        (else (snowflake (/ size 3) (- level 1))
              (lt 60)
              (snowflake (/ size 3) (- level 1))
              (rt 120)
              (snowflake (/ size 3) (- level 1))
              (lt 60)
              (snowflake (/ size 3) (- level 1)))))
```



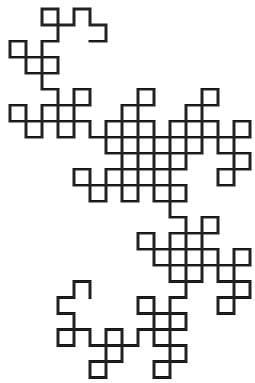
```
(snowflake 60 0)
(snowflake 60 1)
(snowflake 60 2)
```



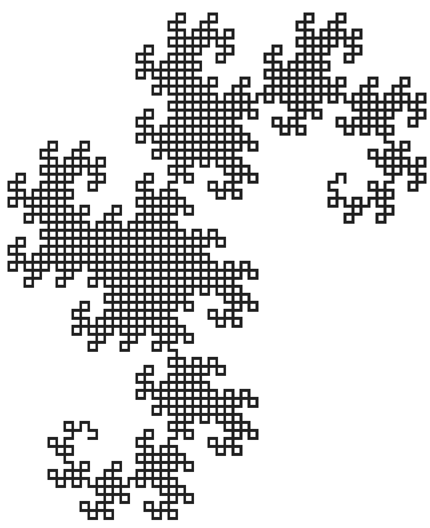
```
(repeat 3 (snowflake 150 5) (rt 120))
```

3.6 The Dragon Curve

```
(define (left-dragon size level)
  (cond ((= level 0) (fd size))
        (else (left-dragon size (- level 1))
              (lt 90)
              (right-dragon size (- level 1)))))

(define (right-dragon size level)
  (cond ((= level 0) (fd size))
        (else (left-dragon size (- level 1))
              (rt 90)
              (right-dragon size (- level 1)))))
```



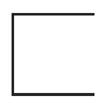
```
(left-dragon 4 8)
```



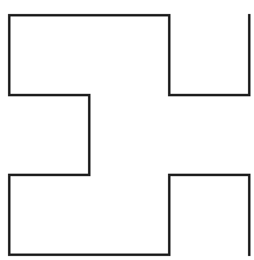
```
(left-dragon 2 11)
```

3.7 The Hilbert Curve

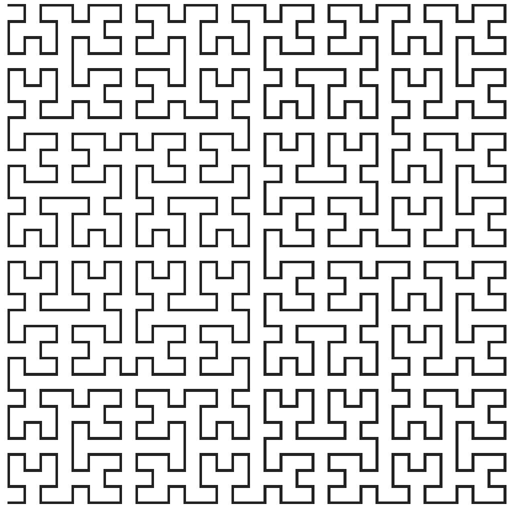
```
(define (left-hilbert side level)
  (cond ((= level 0) 0)
        (else
          (left 90)
          (right-hilbert side (- level 1))
          (fd side)
          (right 90)
          (left-hilbert side (- level 1))
          (fd side)
          (left-hilbert side (- level 1))
          (right 90)
          (fd side)
          (right-hilbert side (- level 1))
          (left 90))))

(define (right-hilbert side level)
  (cond ((= level 0) 0)
        (else
          (right 90)
          (left-hilbert side (- level 1))
          (fd side)
          (left 90)
          (right-hilbert side (- level 1))
          (fd side)
          (right-hilbert side (- level 1))
          (left 90)
          (fd side)
          (left-hilbert side (- level 1))
          (right 90))))
```



```
(left-hilbert 20 1)
```



```
(left-hilbert 20 2)
```



```
(right-hilbert 4 5)
```

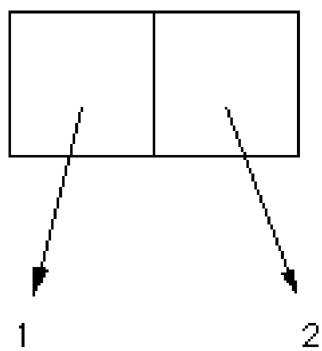
3.8 Pairs: Gluing Data Objects Together

The reading for most of the material in the remainder of this lecture can be found in Chapter 8 of *Programming in MacScheme*.

The `cons` procedure creates new pairs:

```
>>> (cons 1 2)
(1 . 2)
```

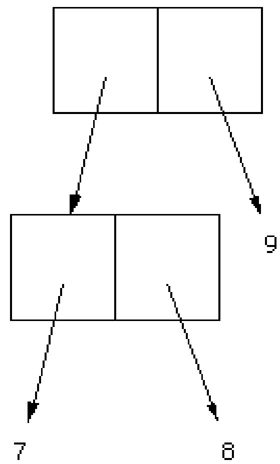
The result of evaluating this call to `cons` is a new pair object which, in box-and-pointer notation, would be drawn as follows:




```
>>> (define new-pair (cons 5 6))
new-pair
>>> new-pair
(5 . 6)
>>> (car new-pair)
5
>>> (cdr new-pair)
6
```

3.9 Pairs can be combined into larger structures

The box-and-pointer notation for `three-numbers-glued` would look like this:



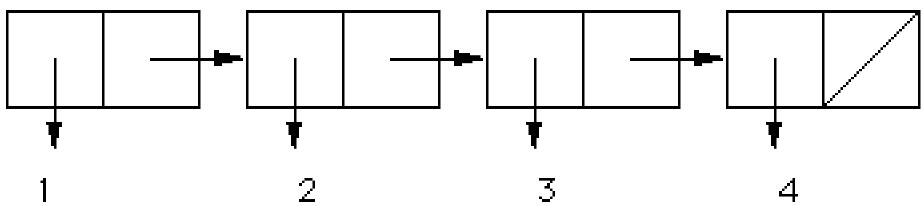

```
>>> (define three-numbers-glued
      (cons (cons 7 8) 9))
three-numbers-glued
>>> three-numbers-glued
((7 . 8) . 9)
>>> (car three-numbers-glued)
(7 . 8)
>>> (cdr three-numbers-glued)
9
>>> (car (car three-numbers-glued))
7
```

3.10 Lists

Generally, Scheme and Lisp programmers use a kind of "standard" arrangement for pairs in which a number of pairs are linked together so that the final CDR arrow points to the special object "nil":

```
>>> (define one-thru-four
      (cons 1 (cons 2 (cons 3 (cons 4 '())))))
one-thru-four
```

This is now the list object to which the name `one-thru-four` is bound (or, a bit more accurately, you should think of the name as being bound to the leftmost pair object):




```
>>> one-thru-four
(1 2 3 4)
>>> (car one-thru-four)
1
>>> (car (cdr one-thru-four))
2
>>> (car (cdr (cdr one-thru-four)))
3
```

We could have defined this list equivalently via the list primitive:

```
>>> (define one-thru-four (list 1 2 3 4))
one-thru-four
```

3.11 A review of list-related primitives

CONS
Purpose: to create new pairs
```
>>> (cons 0 one-thru-four)
(0 1 2 3 4)

>>> (define new-list (cons 4 (cdr one-thru-four)))
new-list

>>> new-list
(4 2 3 4)
```

CAR
Purpose: returns the object pointed to by the first arrow in a pair
```
>>> (car one-thru-four)
1

>>> (car (cdr new-list))
2
```

CDR
Purpose: returns the object pointed to by the second arrow in a pair
```
>>> (cdr one-thru-four)
(2 3 4)

>>> (cdr (cdr (cdr one-thru-four)))
(4)

>>> (cdr (cdr (cdr (cdr one-thru-four))))
()
```

LIST
Purpose: to create new lists (shorthand for nested CONS calls)
```
>>> (list 1 2 3 4)
(1 2 3 4)

>>> (list 2 (list 3 4))
(2 (3 4))
```

APPEND
Purpose: to create new lists by concatenating the elements of two lists together.
```
>>> (append new-list (list 5 6))
(4 2 3 4 5 6)
```

NULL?
Purpose: returns #T if its argument is the empty (null) list, or #F otherwise.
```
>>> (null? (cdr new-list))
#F

>>> (null? (cdr (list 2)))
#T
```

3.12 Some standard recursive list procedures

```
(define (list-length lis)
  (cond ((null? lis) 0)
        (else (+ 1 (list-length (cdr lis))))))


(define (sum-of-elts lis)
  (cond ((null? lis) 0)
        (else (+ (car lis)
                 (sum-of-elts (cdr lis))))))

>>> (list-length one-thru-four)
4
>>> (sum-of-elts one-thru-four)
10

(define (list-average lis)
  (/ (sum-of-elts lis) (list-length lis)))

>>> (list-average one-thru-four)
2.5
```

As an exercise, you should definitely work through the problem of defining a `list-reverse` procedure:

```
>>> (list-reverse one-thru-four)
(4 3 2 1)
```

3.13 Making "Turtle-Point" Objects

First, we make a "turtle-point constructor":

```
(define (make-turtle-point x y)
  (list x y))
```

Now, two "turtle-point selectors":

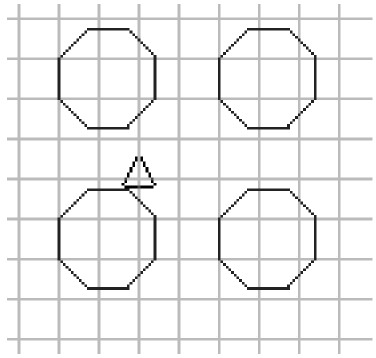
```
(define (turtle-point-x tpt)
  (car tpt))

(define (turtle-point-y tpt)
  (cadr tpt))
```

A procedure to move the turtle to a given turtle-point (this uses the built-in `setpos` primitive):

```
(define (move-turtle-to tpt)
  (setpos (turtle-point-x tpt)
          (turtle-point-y tpt)))
```

Now, we could begin to build a library of procedures that could be used with turtle points. For instance, here are some procedures that reflect turtle points vertically, horizontally, and through the origin:

```
(define (reflect-x tpt)
  (make-turtle-point (* -1 (turtle-point-x tpt))
                     (turtle-point-y tpt)))

(define (reflect-y tpt)
  (make-turtle-point (turtle-point-x tpt)
                     (* -1 (turtle-point-y tpt))))

(define (reflect-in-origin tpt)
  (reflect-x (reflect-y tpt)))

(define (poly-from-startpt nsides sidelength startpt)
  (pu)
  (move-turtle-to startpt)
  (pd)
  (polygon nsides sidelength))


(define (make-four-polys nsides sidelength startpt)
  (poly-from-startpt nsides sidelength startpt)
  (poly-from-startpt nsides sidelength (reflect-x startpt))
  (poly-from-startpt nsides sidelength (reflect-y startpt))
  (poly-from-startpt nsides sidelength (reflect-in-origin startpt)))
```



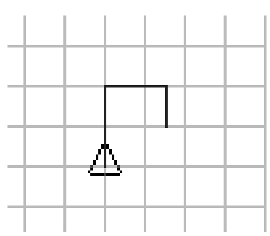
```
(make-four-polys 8 10 (make-turtle-point 20 20))
```

Here, we make a procedure that takes as argument a list of turtle-points and progressively moves the turtle to each of the points in turn:

```
(define (connect-turtle-dots list-of-dots)
  (cond ((null? list-of-dots) '())
        (else (move-turtle-to (car list-of-dots))
              (connect-turtle-dots (cdr list-of-dots)))))
```

It's probably handy to move the turtle to the first point in the list without drawing a line:

```
(define (move-and-connect list-of-dots)
  (pu)
  (move-turtle-to (car list-of-dots))
  (pd)
  (connect-turtle-dots (cdr list-of-dots)))


(define list-of-turtle-pts
  (list (make-turtle-point 5 0)
        (make-turtle-point 5 10)
        (make-turtle-point -10 10)
        (make-turtle-point -10 -10)))
```



```
(move-and-connect list-of-turtle-pts)
```

The material covered in this portion of the lecture can be found in Chapter 8 of *Programming in MacScheme*.

3.14 Symbols: a New Kind of Object

Names can be bound to symbols (just as they can be bound to numbers, pairs, booleans, and procedures):

```
>>> (define test-symbol 'hi)
test-symbol
>>> test-symbol
hi
```

Likewise, we can group symbols into lists (just as we can with numbers and other types of objects):

```
>>> (define list-of-symbols
      (list 'apple 'banana 'orange))
list-of-symbols
>>> list-of-symbols
(apple banana orange)
```

Symbols are compared for equality with the `eq?` predicate procedure:

```
>>> (eq? 'a 'b)
#f
>>> (eq? 'a 'a)
#t
```

3.15 The Quote Special Form

Let's look back at that very first expression again:

```
>>> (define test-symbol 'hi)
test-symbol
```

Why do we need that quote mark just before the "hi" in this expression? First, imagine what this would mean if the quote mark weren't there:

```
>>> (define test-symbol hi)
```

This is a perfectly good Scheme expression *assuming* that the name `hi` is bound to some object. The way the interpreter would treat this expression is by first evaluating the name `hi` and then binding the name `test-symbol` to whatever object the name `hi` was bound to. In other words, if we were to evaluate the following two expression in sequence

```
>>> (define hi 4)
hi
>>> (define test-symbol hi)
test-symbol
```

then the Scheme interpreter would happily evaluate the second expression, binding the name `test-symbol` to the number object 4.

But of course this isn't what we want—instead, we want the name `test-symbol` to be bound to the symbol `hi`, and not the result of evaluating that symbol. So we need some way of telling the Scheme interpreter to treat a particular symbolic expression not as a name to be evaluated but simply as "what it looks like"—namely, a symbol. This is the purpose of the quote mark; it's rather like a "don't-evaluate" marker for the interpreter:

```
>>> (define test-symbol 'hi)
test-symbol
```

The quote mark is actually shorthand for a special form named `QUOTE`, which could be used as follows:

```
>>> (define test-symbol (quote hi))
test-symbol
```

This is entirely equivalent to the previous expression. In both cases we are essentially telling the interpreter to treat the name `hi` as an un-evaluated symbol.

As we will see in subsequent examples, quote can also be used as a don't-evaluate marker before lists:

```
>>> (define test-list '(a b c))
test-list

>>> test-list
(a b c)
```

3.16 A Few Beginning Examples of Procedures Using Symbols

```
(define (filter-out-symbol sym lis)
  (cond ((null? lis) '())
        ((eq? (car lis) sym)
         (filter-out-symbol sym (cdr lis)))
        (else (cons (car lis)
                    (filter-out-symbol sym (cdr lis))))))

>>> (filter-out-symbol 'banana list-of-symbols)
(apple orange)


(define (replace-symbol symbol1 with-symbol2 lis)
  (cond ((null? lis) '())
        ((eq? symbol1 (car lis))
         (cons with-symbol2
               (replace-symbol symbol1 with-symbol2 (cdr lis))))
        (else (cons (car lis)
                    (replace-symbol symbol1 with-symbol2 (cdr lis))))))

>>> (replace-symbol 'rose 'horse '(a rose is a rose))
(a horse is a horse)
```

An idea which will come in handy for a sample program next week:

```
(define (choose-random-element lis)
  (list-ref lis (random (length lis))))
```

This procedure uses a few Scheme primitives:

`length` returns the length of a list (0 for the empty list); we wrote our own version of this in the previous lecture.

```
>>> (length '(a b c))
3
```

`random` takes a positive integer argument and return an integer between 0 and the integer - 1, chosen at random. (For instance, if we call `random` on the argument 7, we will get an integer from 0 to 6, chosen at random).

```
>>> (random 7)
2
```

`list-ref` takes a list and an integer *n* and returns the *n*th element of the list, starting the count from 0:

```
>>> (list-ref '(a b c) 0)
a
```

Putting these ideas together:

```
(define list-of-nouns
  '(cat dog horse birth death infinity))

>>> (choose-random-element list-of-nouns)
horse
```

3.17 Higher-Order Procedures: Nouning Verbs. (Or, "Data-ing Procedures.")

The material covered in this lecture is elaborated upon in Chapters 11 and 12 of *Programming in MacScheme.*

The rights of "first-class objects" in programming languages:

- This object can be the value of a name.
- This object can be passed as the argument to a procedure.
- This object can be returned as the result of a procedure call.
- This object can be grouped together into data structures (such as lists).

We're perfectly used to thinking of numbers as first-class objects:

Right 1: We can bind names to number objects.

```
(define pi 3.14159)
```

Rights 2 and 3: We can write procedures that take number objects as arguments, and return number objects as their results.

```
>>> (double  5)
10
```

Right 4: We can group number objects into lists.

```
(define list-of-numbers (list 1 2))
```

In Scheme (and, with minor modifications, in Lisp), procedures are first-class objects. Thus, we can (in accordance with right 2) create a procedure like the following:

Right 2: We can write procedures that take procedure objects as arguments.

```
(define (apply-to-3 proc)
  (proc 3))

>>> (apply-to-3 1+ )
4
```

Here, we have written a procedure `apply-to-3` that takes a single argument (which we intend to be a numeric procedure); when we call `apply-to-3` on a procedure argument, the result of the call to `apply-to-3` will be the result of applying the given procedure to the numeric argument 3.

3.18 A Classical Example of a Procedural Argument

```
(define (map-over-list proc lis)
  (cond ((null? lis) '())
        (else (cons (proc (car lis))
                    (map-over-list proc (cdr lis))))))

>>> (map-over-list 1+ '(2 3 4))
(3 4 5)
```

In point of fact, this procedure already exists as a Scheme primitive, with the name map:

```
>>> (map even? '(2 3 4))
(#t #f #t)
```

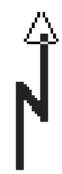
Another classical example:

```
(define (filter-by-predicate pred lis)
  (cond ((null? lis) '())
        ((pred (car lis))
         (cons (car lis) (filter-by-predicate pred (cdr lis))))
        (else (filter-by-predicate pred (cdr lis)))))

>>> (filter-by-predicate even? '(2 3 4 5))
(2 4)

>>> (map sqrt (filter-by-predicate positive? '(-5 6 7 -9)))
(2.449489742783178 2.6457513110645907)
```

3.19 Procedural Arguments in Turtle Graphics

First, let's make a simple turtle procedure. Here's a procedure that takes a single argument (corresponding to a "side-length") and makes a kind of zig-zag pattern scaled according to that argument:

```
(define (zig side)
  (fd side) (rt 144) (fd (/ side 2)) (lt 144) (fd side))
```



(zig 20)

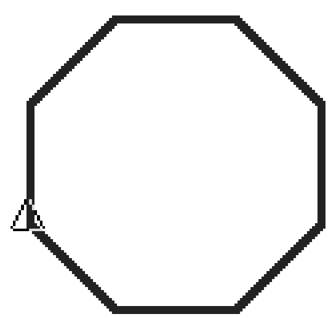
Now, instead of our usual octagon procedure:

```
(define (octagon side)
  (repeat 8 (fd side) (rt 45)))
```

let's make a version of octagon that takes two arguments: a "turtle-mover-procedure" argument and a side-length:

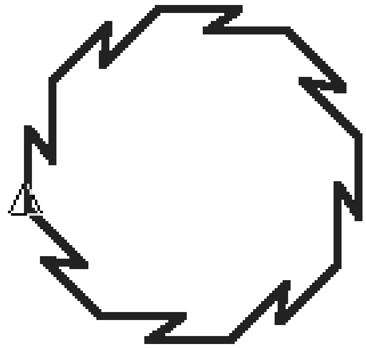
```
(define (octagon turtle-mover side)
  (repeat 8 (turtle-mover side) (rt 45)))
```

We can make a standard octagon using the procedure object to which fd is bound as our turtle-mover:

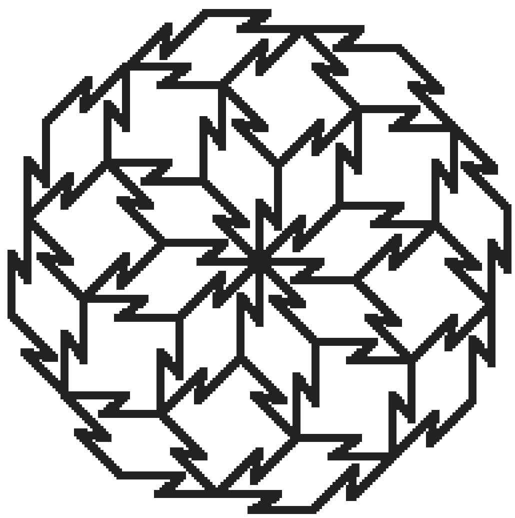


(octagon fd 30)

Or we could use the new zig procedure as our turtle-mover:



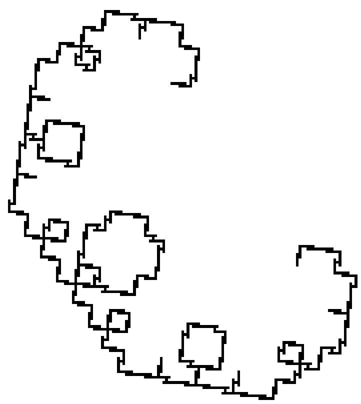
(octagon zig 20)



(repeat 8 (octagon zig 15) (rt 45))

We could try the same idea—using a "mover" argument—with some of our other turtle-graphics procedures.

```
(define (c-curve-with-mover mover side level)
  (cond ((= level 0) (mover side))
        (else (c-curve-with-mover mover side (- level 1))
              (rt 90)
              (c-curve-with-mover mover side (- level 1))
              (lt 90))))
```



```
(c-curve-with-mover zig 3 7)
```

3.20 Procedural Arguments: an Example from Math

We would like to express the idea of summing the value of a particular function at a series of values. In off-putting math notation, this idea is expressed using the character Σ:

$$\sum_{0}^{3} x^2 = 0 + 1 + 4 + 9 = 14$$

What does that "sum" character mean, anyway? We could think about this as something that requires three inputs—a function (like "x squared") a low integer value, and a high integer value—and returns a number:

```
(define (math-sum function lo hi)
  (sum (map function
            (make-list-from-lo-to-hi lo hi))))
```

This math-sum procedure takes three arguments: a function and two integers. It creates (using make-list-from-lo-to-hi) a list of the integers from the low value to the high value; here's an example of how this procedure works when called directly from the interpreter:

```
>>> (make-list-from-lo-to-hi 0 10)
(0 1 2 3 4 5 6 7 8 9 10)
```

The resulting list is then used as an argument to map, which maps the particular function over the entire list; and then we use that result as the argument to sum, a procedure which takes a list of numbers and sums them (we actually saw such a procedure way back in Lecture 4). Here's an example of our procedure in use:

```
>>> (math-sum double 0 5)
30
```