COMP 210, FALL 2000
Lecture 22: Generative Recursion–-QuickSort & Sierpinski

Reminders:
1. Homework is due **Friday** in class. (*mea culpa, mea culpa*)

Review

1. We introduced **lambda** as a second mechanism for defining functions in Scheme. (It is, in fact, the older mechanism and dates back before **define**.) We worked a couple of examples and talked about using **lambda** as an alternative to a local–-when the function is called from one place, **lambda** is often a cleaner solution.

Generative Recusion (See Section 25 of the text for more detailed treatment)
In the last homework, you built a pair of sorting programs–-one based on the idea of an insertion sort, and one based on the idea of a merge sort. Let's look at a third way of sorting numbers–-an algorithm called QuickSort.

Simple idea, simple algorithm
- ⇨ Pick a representative element of the list to be sorted and call it the pivot
- ⇨ Divide the remainder of the list into two lists, one containing elements smaller than the pivot and one containing elements larger than the pivot.
- ⇨ Sort those smaller lists (using QuickSort, unless they are trivial lists)
- ⇨ Create a sorted version of the original list by combining the sorted list of smaller elements, the pivot element, and the sorted list of larger elements.

Work a couple of examples
(list 11 8 14 7)
(list 1 5 3 6)

How would we develop the program qsort? Contract, purpose, header, & template …

```
;; qsort:  list-of-number -> list-of-number
;; Purpose: sort the list of numbers into ascending order
(define (qsort  alon)
   (cond
     [(empty? alon)  …]
     [(cons?    alon)
          … (first alon)  … (qsort  … (rest alon)) …]))
```

We know that this is filled in with **empty** by reading the contract–-qsort returns a list-of numbers

Can we fill in the rest from the English description?

```
;; qsort:  list-of-number -> list-of-number
;; Purpose: sort the list of numbers into ascending order
(define (qsort  alon)
   (cond
     [(empty? alon)  empty]
     [(cons?    alon)
        (local [(define pivot (first alon))]
         … (first alon)  … (qsort  … (rest alon)) …
         ) ] ))
```

The template is not doing what we need. We don't need to run qsort on the rest of the code. Instead, we need to run it on the list of numbers smaller than the pivot and on the list of numbers larger than the pivot. This is not what the template (and the methodology to date) derives.

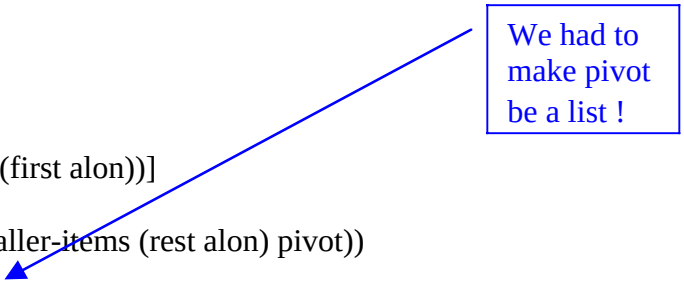
We really want something similar to

```
;; qsort:  list-of-number -> list-of-number
;; Purpose: sort the list of numbers into ascending order
(define (qsort  alon)
   (cond
     [(empty? alon)  empty]
     [(cons?    alon)
        (local [(define pivot (first alon))]
                … (qsort (smaller-items (rest alon) pivot))
                … (qsort (larger-items (rest alon) pivot)) …
         ) ] ))
```

(Assuming the existence of smaller-items and larger-items)

Finally, we can fill it in with

```
;; qsort:  list-of-number -> list-of-number
;; Purpose: sort the list of numbers into ascending order
(define (qsort  alon)
   (cond
     [(empty? alon)  empty]
     [(cons?    alon)
        (local [(define pivot (first alon))]
                (append
                    (qsort (smaller-items (rest alon) pivot))
                    (list pivot)
                    (qsort (larger-items (rest alon) pivot)) )
         ) ] ))
```



```
(define (smaller-items alon threshold)
   (filter (lambda (n) (<  n threshold))  alon))

(define (larger-items alon threshold)
   (filter (lambda (n) (> n threshold))  alon))
```

Why didn't the template work? How did we think of this unusual approach? Our two earlier sorting methods came from the data. Insertion sort sticks one number into a sorted list of numbers, and follows the template. The form of the program follows the data. Merge sort takes two sorted lists and combines them. It then takes the insightful step of recognizing that the trivial list–-the one element list, is sorted and that we can break any list down to a hierarchy of lists ending with trivial lists. The form of the program follows the data.

Quicksort is different. It takes a list and sorts it.

Not all computer science can be generated by templates derived from the data. Sometimes, it takes a novel thought, an original insight, a clever trick. Quicksort is one of those cases. In Quicksort, we needed insights about the nature of the data and the nature of the problem we were trying to solve.

The kind of recursive programming that we've done *until today* is called structural recursion. Structural recursion arises naturally from the structure of the information. In writing structural recursion, the key is to get the data definitions right. Remember how we felt our way around with family trees and with directories. We learned that it sometimes takes a process of development and refinement to get the data definitions right.

Quicksort is an example of another fundamental form of recursion that we will call generative recursion. In generative recursion, we generate new instances of a problem based on some insight about the nature of the problem and (perhaps) the values of the data. We solve those new problems by recurring on our process.

Sierpinski Triangles

Fractals are mathematical abstractions that have garnered a lot of interest in the last fifteen years. They have been used in computer graphics, in modeling, and numerous other fields. One critical property of a fractal is that it has a similar structure when looked at on several different scales.

One simple fractal is called the Sierpinski triangle (see page 367 in the text).

… Draw several iterations of the Sierpinski triangle …

We want to write a program that consumes three vertices of the original (equilateral) triangle and draws the Sierpinski triangle. It should return **true**. Of course, we need to stop the recursion somewhere, so we will provide a threshold to serve as a lower bound on the lengths of a triangle's sides. Assume that we have **draw-triangle** which takes three points and a function **too-small?** which takes three points and returns a boolean.

Finally, we need a representation of a point. The book calls these **posn**s, for "position".

```
;; a posn is a
;; (make-posn x y) where x and y are numbers
(define-struct posn ( x y ))
```

Given all this mechanism, we still don't have any insight into how to build the program **sierpinski : posn posn posn -> boolean**. Clearly, there will be a case for the triangle that is so small as to be unworthy of further exploration. We will use the helper function **too-small?** to determine when a triangle meets that criterion. The other case, when the triangle is not **too-small?**, requires that we carry out the fundamental steps for constructing the next generation of Sierpinski triangles–-finding the midpoints of all three sides, drawing the triangle defined by those sides, and then recurring to subdivide the outer three triangles formed by this subdivision.

This suggests developing the program along these lines:

```
;; sierpinski : posn posn posn -> boolean
;; Purpose: draw Sierpinski's triangle to a resolution defined by the
;;             function too-small?
(define (sierpinski p1  p2 p3)
   (cond
      [(too-small? p1 p2 p3)    true]   ;; must return a boolean; value forced by and
      [else
         ( local [(define  p1-p2  (midpoint p1 p2))
                  (define  p1-p3  (midpoint p1 p3))
                  (define  p2-p3  (midpoint p2-p3))]

                  (and  (draw-triangle p1-p2 p2-p3  p1-p3)
                        (sierpinski  p1  p1-p2  p1-p3)
                        (sierpinski  p1-p2, p2, p2-p3)
                        (sierpinski  p2-p3 p1-p3  p3))
          )]))
```

where **midpoint** is a helper function that maps two posns into a third that is midway between the two arguments.

```
;; midway:  posn  posn -> posn
(define (midway  p1 p2)
  (make-posn (/ (+ (posn-x p1) (posn-x p2)) 2)
                  (/(+ (posn-y p1) (posn-y  p2)) 2)))
```

There is one final complication with this version of the program. It correctly captures the pattern of recursion needed to generate the various generations of triangles. However, it never draws the lines for the outermost triangle.

The program draws a series of triangles with the broad face up and the point down. This reduces the amount of work required to go from the n^{th} sierpinski triangle to the $n+1^{st}$.

However, it means that the outer triangle–-the original triangle, is never drawn. We could reformulate our solution to draw the three outer triangles (with point facing up) rather than the one inner triangle. [For example, in going from the original triangle to the first divided triangle, we would draw a triangle of (p1, p1-p2 and p1-p3), another consisting of (p1-p2, p2, p1-p3) and a third of (p1-p2, p2-p3, p3). This involves three invocations of **draw-triangle** per generation of Sierpinski triangle rather than one, but that's just a constant factor of extra work –- 3x work per level.)

The alternative is to add an initial call to **draw-triangle** that handles the base case–-the first triangle. This would look like

```
;; sierpinski:  posn posn posn -> boolean
;; Purpose: elaborate & draw the Sierpinski triangle down to a size specified by
;;      the helper function too-small?
(define (sierpinski  p1 p2 p3)
   (local
     [  ;; sierp : posn posn posn -> boolean
        ;; Purpose: the workhorse of this program
       (define (sierp p1  p2 p3)
           (cond
               [(too-small? p1 p2 p3)    true]
               [else
                  ( local [(define  p1-p2  (midpoint p1 p2))
                           (define  p1-p3  (midpoint p1 p3))
                           (define  p2-p3  (midpoint p2-p3))]
                           (and  (draw-triangle p1-p2 p2-p3  p1-p3)
                                 (sierp  p1  p1-p2  p1-p3)
                                 (sierp  p1-p2 p2 p2-p3)
                                 (sierp  p2-p3 p1-p3  p3))
                    )]))
        ;; midway:  posn  posn -> posn
       (define (midway  p1 p2)
           (make-posn  (/ (+ (posn-x p1) (posn-x p2)) 2)
                       (/(+ (posn-y p1) (posn-y  p2)) 2)))
     ]
     (and
         (draw-triangle  p1  p2  p3)     ;; the missing outer lines
         (sierp   p1 p2 p3))))           ;; recursive routine for the inner triangles
```

Are there similarities between QuickSort and Sierpinski? Both programs seem to violate our template model. They contain "funny" recursion that does not arise from the structure of the information that they process. Instead, the recursion occurs as some innate part of the way that the problem was defined.

- In QuickSort, the algorithm operates by creating (at each step) two smaller lists that must be sorted and then merging them together with **append**. Here, the recursion comes from some insight into sorting.
- In Sierpinski, the algorithm derives the midpoints of the current triangle's sides. Connecting these midpoints creates four smaller triangles. The program draws the inner one and recurs on each of the outer ones.

Clearly, this is a new kind of recursion (at least, new in the context of COMP 210). We call this style of recursion "generative recursion," since the program proceeds by generating subproblems and solving them recursively.

In contrast, the style of recursion that we have used before today is called "structural recursion." With structural recursion, the recurrences arise directly from the structure of the data. With structural recursion, the data definitions become all important, because the recursion in the template and in the final program echoes the recursion in the templates.

With generative recursion, there need not be any recursion in the data definition. For example, Sierpinski operates on three arguments that are **posn**s. These have no innate recurrence. QuickSort operated on a list of numbers, but the algorithm did not follow the structure of the data [or else the structural recursion template would have worked].

Can we write a template that will help us with these generative recursion problems. Of course, this is COMP 210 –- Programming from templates. With these generative problems, the template will not provide as extensive help as it did with structural recursion. It will, however, get us started in the right direction.

Next class: templates for generative recursion.