COMP 210, Spring 2001
Lecture 19: Introduction to General Recursion

Reminders:
1. Exam next Wednesday evening at 7 P.M.

Reading (for exam):
1. Local definitions and local scope: Intermezzo 3 (Section 18)
2. Functional abstraction: Sections 19-22
3. Lambda: Intermezzo 4 (Section 24)
4. Accumulators: Sections 31-32

Reading (new material)
1. Generative (general) recursion: Sections 25-28, 30
2. Computational complexity: Intermezzo 5 (Section 29)

Review

- ⇨ We introduced the notion of accumlators to simplify the solution of some problems and to provide a more efficient approach to solving others.

Introduction to General Recusion (Reference: Section 25, HTDP)

In the last homework, you built a pair of sorting programs–-one based on the idea of successively inserting numbers in an already sorted list and one based on the idea of merging already sorted lists. 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
- ⇨ Partition the remainder of the list into two lists, one containing elements ≤ the pivot and one containing elements ≥than the pivot.
- ⇨ Sort those smaller lists (using Quicksort, unless they are trivial lists)
- ⇨ Create a sorted version of the original list by concatenating the sorted list of smaller elements and the sorted list of larger elements.

Note: the Quicksort algorithm presented here is an adaptation of a famous algorithm for sorting arrays (vectors in Scheme terminology) to the problem of sorting lists. Quicksort is the premier algorithm for sorting arrays. On lists, it is not necessarily better than mergesort, but it works surprisingly well.

Note; the version of Quicksort presented in the book is flawed; it fails on lists with equal 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 num) -> (list-of num)
;;   sorts the list of numbers alon into ascending
(define (qsort alon)
  (cond [(empty? alon)  …]
        [else
         … (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 num) -> (list-of num)
;; Purpose: sort the list of numbers into ascending order
(define (qsort alon)
  (cond [(empty? alon) empty]
        [else
         (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 **alon**. Instead, we need to run it on the list of numbers ≤ **pivot** and on the list of numbers > **pivot**. This decomposition of the problem is not what the template provides.

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]
    [else
     (local
         [(define pivot (first alon))]
            (append (qsort (smaller-items alon pivot)))
                    (qsort (larger-items alon pivot))))]))
```

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

Finally, we define the helper functions

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

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

Is this program correct? What happens if we try the computation

```
  (qsort (list 1))
= (append (qsort (smaller-items (list 1) 1) (larger-items (list 1) 1)))
```

```
= (append (qsort (list 1)) …)
```

What is wrong? The code does not ensure that the generated subproblems are smaller than the original problem. One solution to this problem is to partition the list into three pieces: the pivot element, the elements in **`(rest alon)`** that are ≤ pivot, and the elements in **`(rest alon)`** that are > **`pivot`**. The piece consisting of the pivot element obviously does not need to be sorted. The code for this version is:

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

Another solution is to force the typical (non-base) case to have at least two elements and force each partition to contain at least one element. In the array version of Quicksort, there is a very slick way to achieve this result but it does not directly adapt to the list version. The following code is similar in spirit to the array version. (See the Cormen, Leiserson, Rivest book on algorithms for the array version.

```
; A (Pair-of T) is a structure
;   (make-Pair l r) where l and r are T's.
(define-struct Pair (left right))

; qsort: (list-of num) -> (list-of num)
; Purpose: sort the list of numbers into ascending order
(define (qsort alon)
  (cond
    [(empty? alon) empty]
    [(empty? (rest alon)) alon]
    [else
      (local [(define pair (partition (first alon) (rest alon)))]
        (append (qsort (Pair-left pair))
                (qsort (Pair-right pair))))]))

; partition: num (list-of num) -> (Pair-of (list-of num) (list-of num))
;   given non-empty orig-alon
;   splits (p) || orig-alon into (make-Pair l r) where
;   l || r = (p) || orig-alon and l,r are non-empty
(define (partition p orig-alon)
  (local
    [; alon is non-empty
     ; the alon || left || right = orig-alon
     ; left <= p < right
     (define (partition-help alon left right)
       (local [(define f (first alon))
               (define r (rest alon))
               (define to-left (<= f p))]
         (cond
           [(empty? r)
            ; place pivot on side that might otherwise be empty
            (cond [to-left (make-Pair (cons f left) (cons p right))]
                  [else (make-Pair (cons p left) (cons f right))])]
           [to-left (partition-help (rest alon) (cons f left) right)]
           [else (partition-help (rest alon) left (cons f right))])))]
      (partition-help orig-alon empty empty)))
```

Why doesn't the template work for Quicksort? It works for insertion sort. Insertion sort sticks the first number into a sorted list of the remaining numbers, and follows the template. The structure of the program follows the data. Quicksort is different. It takes a list and sorts it by cleverly decomposing the original problem into a simple combination (concatenation) of smaller subproblesm.

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* (with the exception of mergesort (which we will discuss in a moment) is called *structural* recursion. Structural recursion arises naturally from the structure of the data. 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 *general* recursion ("generative recursion" in the book). 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.

Mergesort revisited

Recall the mergesort program from Assignment ??. mergesort decomposes the sorting of a list into

- partitioning the list into two approximately equal sublists,
- sorting each piece, and
- merging the two sorted sublists.

The process "bottoms out" on a list of length 1 which is simply returned as the answer.

Like Quicksort, the mergesort algorithm does not fit a structural recursion template. What is the pattern in both of these cases? The orginal problem is decomposed into smaller problems of the same form as the original problem BUT these subproblems are not structural components of the original problem.

What About Our Methodology?
Our design methodology for structural recursion should be engraved in your brains by now. The steps are

- Data analysis and design, including examples of the data
- Contract, purpose, & header
- Construct test cases for the program
- Write the template
- Fill in the program's body
- Test the resulting program (against results of 3)

Do these same steps make sense for generative recursion? Most of them do. We still need to do data analysis and construct examples–-we cannot develop the program if we don't have the data definitions. Every program *needs* a contract, purpose, and header.

When we generate test cases, we need two kinds of test cases: special examples that test the limits or boundaries of the data and typical examples. [For example, what happens on QuickSort of an empty list and on a singleton list? The singleton list is the tricky case.] We also need examples that demonstrate how the program (or algorithm) operates. These should be similar to the worked out examples that we did on the board for Quicksort. These worked out examples help to solidify the operation of the algorithm–-the nuts and bolts of how it works.

We also need to use a template that is appropriate for generative recursion. The implementations of QuickSort and MergeSort have some common elements. Both use a **`cond`** that separates the trivial (or "base") case(s) from the recursive case. The recursive case decomposes the problem into smaller problems, solves them, and combines those solutions to form a solution for the original problem. The trivial case halts the recursion because the problem is small enough to solve directly. Picking out these cases requires problem specific knowledge.

```
(define (gen-recur-func problem-data)
   (cond
     [(trivial-to-solve? problem-data) (solve problem-data)]
     [else
      (combine-solutions
          … (gen-recur-func (generate-problem1 problem-data)) …
          …
          … (gen-recur-func (generate-problemk problem-data)) … )]))
```

This template doesn't give us as much specific guidance as the structural recursion template, but it does lay the groundwork for writing program that used generative recursion. In the structural recursion template, we just had to fill in the missing parts. In this template, you have to replace the various parts of the template with code that implements that part of the program. Thus, in QuickSort, the function ***trivial-to-solve?*** became the familiar test **empty?** and the solution for that case was to return **empty**.

In general, there are a series of questions that we need to ask about the problem before we can develop all the code. These questions will often lead to other, less formulaic, questions. Among the questions we should ask are:

- ⇨ What is a trivial instance of the problem?
- ⇨ What is the solution to a trivial instance?
- ⇨ How do we generate one or more smaller subproblems from the original problem?
- ⇨ How many subproblems should we generate?
- ⇨ Is the solution to the subproblem the solution to the original problem, or do we need to combine the solutions from several subproblems?
- ⇨ How do we combine the solutions from subproblems (if that is necessary)?

In the design of programs that use generative recursion, step 4 "write the template" is much more involved than it is in programs based on structural recursion. (The complex version of programs that work with multiple complex arguments began to have some analysis, but it was conceptually simpler than the process for generative recursion.)

In generating the subproblems, we must ensure that all of the subproblems are strictly smaller than the original problem. It is easy to make mistakes like we did in our first attempt at writing Quicksort.