COMP 210, FALL 2000
Lecture 21: From Lambda to QuickSort

Reminders:

1. Exam handed out today, due Wednesday at 5pm in my office, DH 2065. If you missed class, copies of the exam are available outside my office. Exam covers through *Lecture 18 (on the web site)*,the text, through Intermezzo 3 (on local), and the *lab lectures up to lab 5* (on the web site).
2. Homework is due **Friday** in class. (*mea culpa, mea culpa*)
3. Review session this afternoon ***DH 1064 or DH 1070***

Review

1. We derived the Scheme function **filter** and derived an appropriate contract for it. The contract was new (to our experience) because it contained a variable **alpha** rather than a concrete **type**.

   ```
   filter :  (alpha -> boolean)  list-of-alpha  -> list-of-alpha
   ```

   Since I admonished all of you to go to lab lecture, you should now be familiar with **map**, **filter**, **foldl**, and **foldr**.

2. You may not use these abstract functions on the test. (Reminder on the cover)

3. We talked about the fact that programs are values, so we can use them anywhere that a value is legal. We also learned that some parts of Scheme are not values (we call them keywords or commands) and cannot be used as values. Someone, after class, pointed out that we can redefine the built-in functions such as >, <, and +. Yes, we can do that. No, it isn't a particularly good idea.

Introduction

Today's lecture will introduce one new piece of Scheme syntax (lambda) and, time permitting, move on to a major new idea that dominates the final section of the course – *generative recursion*.

Consider the Scheme program **double-all**

```
;; double-all:  list-of-number -> list-of-number
;; Purpose:  double all of the numbers in the input list
(define (double-all alon)
   (cond
         [(empty? alon) empty]
         [(cons?   alon)
           (cons  (* 2 (first alon)) (double-all (rest alon)))]
   ))
```

After lab, you should recognize that this function can be written more simply using **map**.

```
;; double:  number -> number
;; Purpose: consume n, produce 2n
(define (double num)
  (*  2 n))    ;; compiler person would write it as (+ n n)

;; double-all:  list-of-number -> list-of-number
;; Purpose:  double all of the numbers in the input list
 (define (double-all alon)
   (map  double alon))
```

If we are going to make use of these abstract functions, we will quickly get tired of making up names for all the little helper functions that we need. We could, of course, encapsulate them inside a local

```
;; double-all:  list-of-number -> list-of-number
;; Purpose:  double all of the numbers in the input list
(define (double-all alon)
   (local [(define (double n) (* 2 n))]
          (map  double alon) ))
```

This hides **double** from the world outside **double-all** and avoids the potential for a name conflict. However, there are two problems with writing **double-all** this way.

1. It forces you to invent a name for double. (minor hassle)
2. It violates the whole philosophical purpose of using local. The real justifications for using a local are:
   - ⇨ To avoid computing some complicated value more than once.
   - ⇨ To make complicated expressions more readable by introducing helper functions that break the expression up into more tractable parts.

   (Notice that avoiding the use of invariant parameters might fall under either case!)

This example fits neither criterion. The expression is not complicated; in fact, it is about as simple as a Scheme expression can get. The expression is not used in many places; it is used exactly once. The only reason for introducing double is because we need a function (number->number) that we can pass to **map** – this lets us avoid writing a lot of code by using the abstract function.

To handle this situation, Scheme includes a construct called λ. Unfortunately, DrScheme operates under the limited typographic conventions of computer keyboards, so we end up writing it out as **lambda**. Lambda lets us create unnamed programs –- it is a second way to write out a program (without using **define**).

```
(define (double n)
  (* 2 n))
```

```
(lambda (n)
  (* 2 n))
```

These are equivalent, in the sense that they both create programs that "do" the same thing. They differ, in the sense that you can use **double** anywhere that its name can be seen, while the **lambda** expression occurs somewhere in the code, is created, is evaluated, and cannot be used elsewhere *because it has no name*.

Using **lambda**, we could rewrite **double-all** as

```
(define (double-all alon)
   (map (lambda (n) (* 2 n))  alon) )
```

Formally, lambda is written

```
(lambda
  (arg1  arg2 … argn)
  body
)
```

where arg1, arg2, …, argn and body are arbitrary Scheme expressions.

To evaluate a lambda expression, DrScheme rewrites it as

```
(local  [(define (a-unique-new-name  arg1  arg2 … argn)
                 body)]
        a-unique-new-name)
```

The body-expression cannot refer to *a-unique-new-name* because the programmer does not know how to write it. The unique name is introduced by the rewriting process, not by the programmer, so the programmer cannot write a lambda expression that *directly* calls itself. To slightly simplify the explanation, assume the list being sorted contains no duplicates.

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)) …
          ) ] ))
```

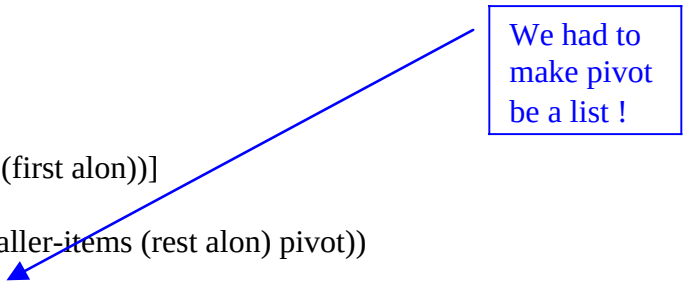
(Assuming the existence of smaller-items and larger-items)

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.

Next class, we'll look at another example of generative recursion and then figure out how to derive a template for generative recursion.