COMP 210, FALL 2000
Lecture 21: Finishing up Lambda & Review for Second Exam

Reminders:

1. Exam handed out today, due Friday at 5pm in my office, DH 2065. If you missed class, copies of the exam are available outside my office. Exam covers through today's lecture, all of the lab lectures (through this week 's lab lecture that covered abstract functions – lab 8 on the web site), and the book material through section 24.
2. Homework will appear on the web site today or this weekend, due Monday 10/30.
3. Review session this afternoon ***DH 1064 or DH 1070***

Review

- We finished up functional abstraction. We introduced lambda.

Introduction

Today's lecture will introduce one new piece of Scheme syntax (lambda) and include some review material for the test.

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.

- It forces you to invent a name for double. (minor hassle)
- 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)                    (lambda (n)
  (* 2 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.

Notice that lambda is the original way that Scheme defined functions. The expressions

```
(define f                              (define (f  x)
    (lambda(x) (* x x)))                   (* x x))
```

are entirely equivalent. DrScheme introduced the latter form because it is somehow seen as friendlier than the former. However, the two are entirely equivalent.

Exam Review:

Exam covers lectures 11 through 21 (today's lecture)

Exam covers all the lab lectures, through lecture 8

Exam covers book through Intermezzo 4 (just before the start of Section V, on Generative Recursion). I will not ask any questions from the section on mathematical examples, but those of you considering a career in science or engineering should look at that material before the semester is over. You won't see similar material again until you take CAAM 353.

The Exam will be a two-hour, closed-notes, closed-book take-home exam. It is conducted under the Rice honor code. If this is your first take home exam, recall that you should set aside enough time to do the exam in one sitting (2 hours, fifteen minutes). Get all the supplies you need, including paper. Record your name, starting time, stopping time on the front of the test. Write the pledge, ***correctly***, and sign it. Staple your exam pages together.

Be sure to write your name legibly on the front of the test. Every test we get one or two where we cannot read the writing!

The Big Picture

Four major subjects:

- ***Family trees, directory structures***

  Key ideas:

  - Template for each data definition,
  - Draw the arrows to get the mutual recursions correct
  - Keep in mind that, sometimes, you need to summarize a list at the level above the list–-this happened in the siblings problem in the homework
  - Similarly, when you need to look down inside some other data definition, that's a good place to call a helper function

- ***Functions that take two complex arguments***

  Key ideas:

  - Build a table of the possible cases or conditions that your program might encounter
  - Write down the predicates for those cases and use them to build your template
  - Use a similar table to write down the actions that should happen in each case
  - Test examples should include the trivial case–- like (merge empty empty)

- ***Local***

  Key ideas:

  - Local creates a box where we can introduce new definitions and use them
  - When DrScheme evaluates **local**, it rewrites all the newly defined names with unique names –- for any occurrence inside the local, including locals nested inside the current local. This has the effect of creating a new "scope," in that rewriting can (1) hide externally visible names by changing them to refer to the local copy and (2) prevent external (outside the box or outside the local) expressions from referencing names defined inside the local (inside the box).
  - You should use local:

1. To avoid computing some complicated value more than once
2. To make some complicated expression more readable by introducing local helper functions to break the expression into tractable parts (& clean up the mess). Local lets you hide these names.

➢ Local is the only way we have given you to preserve a value in the middle of a computation! It is quite limited, but it is there.

- ***Abstract functions***

  ➢ When you see common patterns in the code, as we saw with the whole series of repetitive examples (keep-*xx*-*yy*), you should extract the essence and create an abstract version of the code.

  ➢ Make the differences be arguments to the abstract version—that is, parameterize the code by the differences.

  ➢ DrScheme provides you with a number of built-in functions, including **filter, map, foldr,** and **foldl**. You saw them in lab.

  ➢ Programs are just values in Scheme. Using lambda to define a function makes that absolutely clear. You can make a program be an argument to a function. A program can return a function. The built-in abstract functions take function arguments.

  ➢ Lambda allows you to define an anonymous program, a program with no name. This is useful when the program exists solely to be passed to another program (as an argument). It gets you out of needing to think up a name for the lambda program. It tells DrScheme, in no uncertain terms, that you are not going to use the program again.

  (Remember, we mentioned that DrScheme runs around behind your back, finding objects that can never be referenced and reclaiming them. A lambda expression, used as an argument to **filter**, can never be referenced after filter returns.)