CSC172 LAB 16

THE “L” WORD

1 Introduction

The labs in CSC172 will follow a pair programming paradigm. Every student is encouraged (but not strictly required) to have a lab partner. Labs will typically have an even number of components. The two partners in a pair programming environment take turns at the keyboard. This paradigm facilitates code improvement through collaborative efforts, and exercises the programmers cognitive ability to understand and discuss concepts fundamental to computer programming. The use of pair programming is optional in CSC172. It is not a requirement. You can learn more about the pair programming paradigm, its history, methods, practical benefits, philosophical underpinnings, and scientific validation at http://en.wikipedia.org/wiki/Pair_programming .

Every student must hand in their own work, but every student must list the name of their lab partner if any on all labs.

This lab has six parts. You and your partner(s) should switch off typing each part, as explained by your lab TA. As one person types the lab, the other should be watching over the code and offering suggestions. Each part should be in addition to the previous parts, so do not erase any previous work when you switch.

The textbook should present examples of the code necessary to complete this lab. However, collaboration is allowed. You and your lab partner may discuss the lab with other pairs in the lab. It is acceptable to write code on the white board for the benefit of other lab pairs, but you are not allowed to electronically copy and/or transfer files between groups.

2 Lambda in Scheme

1. As everyone familiar with popular culture knows the “L” word is “lambda”. It's the Greek letter: “λ”. In mathematical logic and computer science, **lambda calculus**, also **λ-calculus**, is a formal system designed to investigate function definition, function application and recursion. It was introduced by Alonzo Church and Stephen Cole Kleene in the 1930s as part of an investigation into the foundations of mathematics, but has emerged as a useful tool in the investigation of problems in computability or recursion theory, and forms the basis of a paradigm of computer programming called functional programming[1]. Let's start this lab with the following observations. Type in the following and examine the results (be sure to use the **Pretty Big** language):

1 Henk Barendregt, "The Impact of the Lambda Calculus in Logic and Computer Science." *The Bulletin of Symbolic Logic*, Volume 3, Number 2, June 1997.

```
(define x 3)
(define y 5)
(display x) (newline)
(display y) (newline)
(+ x y)

(define foo (list + x y))
(define bar (list + 'x 'y))

(display foo) (newline)
(display bar) (newline)

(eval foo)
(eval bar)

(set! x 123)
(set! y 456)

(eval foo)
(eval bar)
```

The point of all this is to illustrate how lists can be both “data” and “programs”. The use of the single quote escaped the evaluation. In the “foo” list, the variables were dereferenced and the values were baked into the definition. In the “bar” list, the single quote escaped evaluation so that the symbols were baked into the list. The concept that programs can be data and data can be programs is what this lab is all about.

2. Consider three functions. One is our old friend the triangle sum. The second is the sum of squares. The third is the sum of cubes.

```
(define (sum-ints a b)
  (if (> a b)
    0
    (+ a (sum-ints (+ a 1) b))))

(sum-ints 1 3)
```

```
(sum-ints 1 10)

(define (square x) (* x x))
(define (cube x) (* x x x))

(define (sum-sqrs a b)
  (if (> a b)
    0
    (+ (square a) (sum-sqrs (+ a 1) b))))

(sum-sqrs 1 5)

(define (sum-cubes a b)
  (if (> a b)
    0
    (+ (cube a) (sum-cubes (+ a 1) b))))

(sum-cubes 1 5)
```

3. What you notice, of course, is the similarity in the structure between the three functions. Wouldn't it be nice if we could abstract the summation machinery away from the function? Recall that in mathematical "sigma notation" we have different terms for the "series generation" and the "function" $\sum_{x=a}^{b} f(x)$. In such an expression the $\sum_{x=a}^{b}$ represents the "series generation" and the $f(x)$ represents the "function".

   We can do this in scheme. All we need to do is make the function a parameter. Consider the following, noting the structural similarity to the three functions above:

```
(define (summer term a b)
  (if (> a b)
    0
    (+ (term a) (summer term (+ a 1) b))))
```

   Now we can re-define two of the three functions using our generic summation machine.

```
(define (sum-cubes2 a b) (summer cube a b))
(define (sum-sqrs2 a b) (summer square a b))

(sum-sqrs2 1 5)
(sum-cubes2 1 5)
```

If we wanted to define a trivial identity function to use as the term, we can re-define the simple triangle sum.

```
(define (id x) x) ; identity is a reserved function
(define (tri-sum2 a b) (summer id a b))
(tri-sum2 1 5)
```

4. This is pretty neat, because we gain the power of abstraction by being able to pass in the function we want. However, having to predefine the function is a bit of a cumbersome step. It would be more flexible if we could define an "inline" function, on the fly without having to name it. This is what the "lambda" form is all about. The lambda is used to create procedures. The syntax is : (lambda (<*formal-parameters*>) <*body*>).

   Consider:

```
(define (plus-three x) (+ x 3)) ; standard
     (plus-three 9)

(define plus-four (lambda (x) (+ x 4))) ; same using lambda
(plus-four 13)

(lambda (x) (+ x 4)) ; a stand alone method

((lambda (x) (+ x 5)) 21) ; take some time to digest this
; we are building a nameless function on the fly
; and then using it at the head of a list
; and passing it "21" as a parameter
; similarly
((lambda (x y) (* x y)) 7 13)
; the function only exists for the moment

; now, we can use lambda to build in-line functions
(define (tri-sum3 a b) (summer (lambda (x) x) a b))
```

```
(define (sum-sqrs3 a b) (summer (lambda (x) (* x x)) a b))
(define (sum-cubes3 a b) (summer (lambda (x) (* x x x)) a b))

(tri-sum3 1 5)
(sum-sqrs3 1 5)
(sum-cubes3 1 5)
```

The point being, we didn't need to have to predefine the identity, square, or cube functions.

So you should now be able to use this lambda form to define `geo-sum:` $\sum_{x=a}^{b} \frac{1}{x}$ by using the summer function and a lambda expression. Do so to complete this section of the lab.

5. Another use of lambda is the creation of local variables. We are used to defining local variables to help us in programming languages. If you were asked to write a method to compute:

$$f(x,y)=x(1+xy)^2+y(1-y)+(1+xy)(1-y)$$

You would likely first compute local variables: $a=1+xy$ and $b=1-y$,

and then compute: $f(x,y)=xa^2+yb+ab$.

Since doing so would save you a multiply, an addition, and a subtraction operation.

So, consider using lambda to define the method and then pass the local variables in as parameters. (Take some time to digest how this is working)

```
(define (f x y)
  ((lambda (a b)
    (+ (* x (* a a))
       (* y b)
       (* a b))
  ) ; end of "the function"
  (+ 1 (* x y)) ; local variable “a”
  (- 1 y) ; local variable “b”
))
(f 1 1)
(f 8 7)
```

Powerful? Yes. Elegant? Yes! Slightly abstruse syntactically? Perhaps.

6. Local variables are used so often that a special, (perhaps slightly less abstruse) syntax has been

developed which more naturally expresses the traditional idea of a local variable. This is the let expression.

The general form of the let expression is

```
(let ((<var1> <expression1>)
      (<var2> <expression3>)
      .....
      (<varn> <expressionn>))
     <body>)
```

As you might expect, in the <body> , var_1 gets the value of $expression_1$ var_2 gets the value of $expression_2$, etc.

```
(define (f2 x y)
  (let ((a (+ 1 (* x y )))
        (b (- 1 y)))
       (+ (* x (* a a))
          (* y b)
          (* a b )
       ) ; end of "+"
  ) ; end of "let"
)  ; end of "define"

(f2 1 1)
(f2 8 7)
; note, we use the above syntax to define “local” variables
(display a) ; try this, a is not defined
```

3 Hand In

Hand in the source code from this lab at the appropriate location on the blackboard system at my.rochester.edu. You should hand in a single compressed/archived (i.e. “zipped” file that contains the following.)

1. A plain text file named README that includes your contact information, your partner's name, a brief explanation of the lab (A one paragraph synopsis. Include information identifying what class and lab number your files represent.). And one sentence explaining the contents of all the other files you hand in.
2. The Racket file containing the definitions you wrote in the lab as well as those copied from the lab prompt with author information commented out at the top.
3. A plain text file named OUTPUT that includes your interactions from DrRacket.

4 Grading

90% Functionality

- 60% Demonstration of given code for parts 1-6
- 30% Implementation of geo-sum from part 4

10% README and OUTPUT files

5 References

Many of the examples in this lab exercise have been adapted from the “Wizard” book. (Hal Abelson's, Jerry Sussman's and Julie Sussman's *Structure and Interpretation of Computer Programs* (MIT Press, 1984; ISBN 0-262-01077-1) .