LAMBDA, HIGHER ORDER FUNCTIONS 3

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

June 23, 2010

1 Review from Last Lecture

1.1 CS Illustrated

Functions as Arguments: http://csillustrated.berkeley.edu/PDFs/functions-as-data.pdf.

1.2 Procedures We Learned

```
sentence ;Joins together multiple words in a single sentence
se ; same as sentence
empty? ; tests if a sentence or a word is empty
```

1.3 Review of Recursion

If we examine both of the procedures from yesterday, we see that they both kind of have that same mapping pattern. You take the first element, you do something to it, and then you combine it with the recursive call on the sentence except for the first element. This pattern is very common. In fact, you'll give it a name in the homework. Unfortunately, not everything follows this pattern. Let's look at the code for Pascal's Triangle.

```
(define (pascal row col)
  (cond ((= col 0) 1)
        ((= col row) 1)
        (else (+ (pascal (- row 1) (- col 1))
                 (pascal (- row 1) col) ))))
```

Note that both row and column numbers start at zero.

1.4 Another Higher-Order-Function Example: Keep

```
;;;;; cs60a/lectures/1.3/keep.scm
(define (evens nums)
  (cond ((empty? nums) '())
        ((= (remainder (first nums) 2) 0)
```

```
         (se (first nums) (evens (bf nums))) )
        (else (evens (bf nums))) ))
(define (ewords sent)
  (cond ((empty? sent) '())
        ((member? 'e (first sent))
         (se (first sent) (ewords (bf sent))) )
        (else (ewords (bf sent))) ))
(define (pronouns sent)
  (cond ((empty? sent) '())
        ((member? (first sent)
           '(I me you he she it him her we us they them))
         (se (first sent) (pronouns (bf sent))) )
        (else (pronouns (bf sent))) ))
```

We can again see there is a generalization here:

```
(define (keep pred sent)
  (cond((empty? sent) '())
       ((pred (first sent))
        (se (first sent)(keep pred (bf sent))))
      (else (keep pred (bf sent)))))
```

2 Anonymous Functions

2.1 Motivating Factors

Let's get back to the generalized sum function we coded yesterday. Just as a reminder, it computes $\sum_{i=a}^{b} f(i)$.

```
(define (sum-of func a b) ;Sums of func from a to b
   (if (> a b)
       0
       (+ (func a) (sum-of (+ a 1) bb)) ))
```

So now, say we want to do the sum of sines. We can use it as follows:

```
> (define (sinsq num) (* (sin x) (sin x)))
> (sum sinsq 5 8)
```

But this seems kind of kludge. If this is the only place we are using the `sinsq` function, it seems like it's kind of useless to have to make something if we're just going to throw it away after one use. Thus, we turn to *anonymous functions*. Note that this is kind of confusing if you haven't seen this point before. This is because when we first learned about functions, we always think about them as very tied together with the idea of names, right? A function like $f(x) = x^3$ is *named* f. But, we can think of the same function x^3 and think of it the same way, just without a name. Note, I'm not talking about the *value* of x^3, which is a number. I'm still talking about a function, just a nameless one.

2.2 Lambda

2.2.1 The Syntax

```
(lambda (args) body) ; lambda expression returns a procedure
((lambda (a b) (+ a b)) 3 4) ; Returns 7
```

The blue above is evaluated into the function itself, whereas the red are the arguments.

2.2.2 Explanation

This is something that lets us create functions. Remember, this is actually the first thing we've seen that actually created a function. When we did define, it turns out that the real definition of `(define (square num) (* num num))` actually is `(define square (lambda (num) (* num num))`. Thus, the two notations are actually the same. It's always `(define variable value)`.
Remember what I was talking about when I spoke about the difference between the expression and the value? A lambda expression generates a procedure as a value. Thus, the difference between the *expressions* from the *values* is significant.
Let's look at the picture again, with understanding of the last section of the drawing.
http://csillustrated.berkeley.edu/PDFs/functions-as-data.pdf

2.3 First Class Data Type

Just terminology. We won't ask you to list the 4 things, but the idea here is to think that you can do to first-class data what you can do to any other piece of data. Typically, numbers are first class in every language, and other things may or may not be. A few things must be true.

- Can be value of variable
- Can be argument to procedure
- Can be return value from procedure
- Can be member of aggregate data type

2.4 Let

2.4.1 Syntax

```
;let with 1 argument
(let ((var1 val1))
     body)
;same as:
((lambda (var1) body) val1)
;let with multiple arguments
(let ((var1 val1)
      (var2 val2)
      (var3 val3))
     body)
;same as:
((lambda (var1 var2 var3) body) val1 val2 val3)
```

2.4.2 Motivation and Explanation

Let's consider the following equation: $ax^2 + bx + c = 0$. You guys should recognize this as a quadratic polynomial, which you may remember has a solution using the quadratic equation: $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. Thus, there are 2 roots.
Let's program that.

```
(define (roots a b c)
  (se (/ (+ (- b) (sqrt (- (* b b) (* 4 a c)))) (* 2 a))
      (/ (- (- b) (sqrt (- (* b b) (* 4 a c)))) (* 2 a))
)
```

Note how it computes that entire ugly thing twice. And let's assume that the square root is kind of hard to compute. On modern machines, this isn't really the case, but assume, for the sake of argument, that this square root is kind of hard to compute. Well, what we want to really do is compute it once, write it down, and then use it twice. Well, we know how to give something a name by using it as an argument to a function:

```
(define (roots a b c)
  (define (roots-helper d)
    (se (/ (+ (- b) d) (* 2 a))
        (/ (- (- b) d) (* 2 a)) ))
  (roots-helper (sqrt (- (* b b) (* 4 a c)))) )
```

We can see that `d` is going to get bound to that argument which is that ugly square root expression. The problem is we never really need this `roots-helper` procedure. So, let's replace it with a `lambda`.

```
(define (roots a b c)
  ((lambda (d)
    (se (/ (+ (- b) d) (* 2 a))
        (/ (- (- b) d) (* 2 a)) )) ;closes the lambda
   (roots-helper (sqrt (- (* b b) (* 4 a c))) ))
```

However, this is really really awkward, isn't it? We say we have a variable named `d` at the top, but we don't say what the value of that variable is until the very end. That's so messy! Fortunately, this is something that's so common that there's a solution, for a special piece of syntax called `let`.

```
(define (roots a b c)
  (let ((d (sqrt (- (* b b) (* 4 a c))) ))
    (se (/ (+ (- b) d) (* 2 a))
        (/ (- (- b) d) (* 2 a)) ) ) ) ;closes the lambda
```

3 Efficiency and Asymptotic Analysis

3.1 Motivation

Okay! We're moving from talking about functions to kind of mentally taking a few steps back and talking about how to talk about functions. In particular, although most of this course we're focused on writing programs that work, we'll be talking about being able to judge the efficiency of a program today and tomorrow.

First, let's think about this problem. The obvious way to test the quality of a program is to just run the darn thing, right? Just break out a stopwatch, and set the timer, and see how long it takes to run. But the problem with this is to think of all the constraints of reality we've talked about. What if you have other programs running on your computer in the background? What if your CPU kind of sucks, and it's really old? Well, what we want to do, is divorce the hardware that you're running these programs on with the abstract software programs that you are writing.

3.2 An Example[2]

Suppose you're writing a program to manage the inventory for a retail store. Suppose that it takes **10,000ms** to load the initial inventory, and **10ms** to process each transaction.

Well, if we're talking about a small number of transactions, say < 1000, then it's pretty straightforward to see that the initial load time is what takes a long time. However, note that we're only talking about *20 seconds* total here. We don't really care about the differences here when it's not such a big deal.

[2]This section adapted from Jonathan Shewchuk's lecture notes.

However, if we're talking about a large number of transactions, say $> 1,000,000$, then the initial load time becomes massively outweighed by the time per transaction. For comparison, with 1 million transactions, the time to process the transactions is about 3 hours, whereas the 10 second load time really doesn't matter much.
Thus, whatever we use to think about runtimes of programs must be *more focused on large problems*, and less focused on smaller problems.