RECURSION, EVALUATION, FUNCTIONS 2

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

June 22, 2010

1 Functions from Last Lecture

1.1 CS Illustrated

Words and sentences: http://csillustrated.berkeley.edu/PDFs/word-sentence-selectors.pdf

1.2 Procedures We Learned

```
first ; Returns the first element of a sentence or a word
butfirst ;Returns the rest of the sentence or a word
bf ;Abbreviation for butfirst
last ;returns the last element of a sentence or a word
butlast ; returns all but the last element
bl ; abbreviation for butlast
word ;takes 2 things and combines them into a single word
' ; Quotes a word, so prevents evaluation
equal? ; tests if two things are equal
if ;Syntax: (if (PREDICATE?) TRUE-EXPRESSION FALSE-EXPRESSION)
cond ;Syntax: (cond ((PRED-1?) (DO-1)) ((PRED-2?) (DO-2)) (else (DO-3)))
member? ; (member? 'a 'hello)⇒#f    (member? 'a 'halo) ⇒ #t
```

2 Recursion:

At the end of lecture, we talked about this:

```
;;;;; In file cs61a/lectures/1.1/pigl.scm
(define (pigl wd)
  (if (pl-done? wd)
      (word wd 'ay)
```

```
      (pigl (word (bf wd) (first wd)))))
(define (pl-done? wd)
  (vowel? (first wd)))
(define (vowel? letter)
  (member? letter '(a e i o u)))
```

When we trace it, we get some interesting things. Let's look at why by first explaining a bit about how computers work. I'm gong to step a bit on the toes of CS61C, but here we go. There are a lot of really little people in your machine. Each of these little people, are unfortunately not very bright. They really only know how to do a *tiny* thing.
Let's look at another example, called sent-sum. This example sums together all the numbers in a sentence:

```
(define (sent-sum sent)
   (if (empty? sent)
       0
       (+ (first sent) (sent-sum (bf sent)))))
```

Now let's look at the recursion monkeys! http://scratch.mit.edu/projects/ColleenBerkeley/213678
Looking back, hopefully this answers why there's so many different traces. Each pair of lines that are indented identically are actually a separate monkey. Therefore, the monkeys must go back one at a time. That is why you have 4 monkeys all returning the same thing.
As a last example, let's look at a procedure argue. This procedure returns the opposite of whatever you put in.

```
;;;;; In file cs61a/lectures/1.1/argue.scm
> (argue '(i like spinach))
(i hate spinach)
> (argue '(broccoli is awful))
(broccoli is great)
(define (argue s)
  (if (empty? s)
      '()
      (se (opposite (first s))
          (argue (bf s)))))
(define (opposite w)
  (cond ((equal? w 'like) 'hate)
        ((equal? w 'hate) 'like)
        ((equal? w 'wonderful) 'terrible)
        ((equal? w 'terrible) 'wonderful)
        ((equal? w 'great) 'awful)
        ((equal? w 'awful) 'great)
        ((equal? w 'terrific) 'yucky)
        ((equal? w 'yucky) 'terrific)
        (else w) ))
```

This computes a function (the opposite function) of each word in a sentence. It works by dividing the problem for the whole sentence into two subproblems: an easy subproblem for the first word of the sentence, and another subproblem for the rest of the sentence. This second subproblem is just like the original problem, but for a smaller sentence.

3 Order of Evaluation

Scheme is something called APPLICATIVE ORDER. Basically, when you call a procedure, it will evaluate all the arguments first, and then use the values of those arguments in the body. For example, if we ask Scheme to evaluate `(+ (* 4 5) (- 6 2))`, then this is translated into `(+ 20 4)`, which is then evaluated to yield 24. In other words, if we say: `(define (f x y) (+ x y))`, and then call `(f (* 4 5) (- 6 2))`.

4 Functions as Arguments

4.1 Overview

There is essentially a brick wall in your mind between things and actions. As an example, in language, these are separated into nouns and verbs. This is pretty much true in just about every language I know about. Incidentally, this is why calculus is so difficult. When you take the derivative of a function, what you get is not a number, but a function. Thus, the derivative and the integral are what we'll call 'higher order functions'. In other words, they are a function that take in another function.
Let's look at a cute picture: http://csillustrated.berkeley.edu/PDFs/functions-as-data.pdf.

4.2 Introduction

Let's try to breach this topic by generalizing a pattern. Below is the code for defining the area of four shapes.

```
(define pi 3.141592654)
(define (square-area r) (* r r))
(define (circle-area r) (* pi r r))
(define (sphere-area r) (* 4 pi r r))
(define (hexagon-area r) (* (sqrt 3) 1.5 r r))
```

The pattern is for each of these, area $= \_ * r^2$. Instead of defining each of these separately, we can start by generalizing the pattern as follows:

```
(define (area shape r) (* shape r r))
(define square 1)
(define circle pi)
(define sphere (* 4 pi))
(define hexagon (* (sqrt 3) 1.5))
```

Note that the thing we've generalized here is that instead of taking in a fixed number, we are kind of using a variable as a number to do something.

4.3 Functions as an Argument

We have not yet begun to use a function as a part in another function. For that, let's take a look at this:

```
(define (square x) (* x x))
(define (cube x) (* x x x))
(define (sumsquare a b) ;Sums of squares from a to b
   (if (> a b)
       0
       (+ (square a) (sumsquare (+ a 1) bb)) ))
(define (sumcube a b)
```

```
    (if (> a b)
        0
        (+ (cube a) (sumcube (+ a 1) bb)) ))
```

If we look at what we've done here, we can see a pattern in what we've done. Let's look at it:

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

If this is the pattern, then we should instead just take in the function instead:

```
(define (sum-of func a b) ;Sums of func from a to b
    (if (> a b)
        0
        (+ (func a) (sum-of (+ a 1) bb)) ))
(sum-of square 1 5)
```

WHOA. We just did something here. The `square` in this call to `sum-of` is a very different thing to do than what you are used to doing with square. The call in here is passing the procedure as a piece of data. That's something very powerful, as we'll see, and very confusing for many students.