CS61A Notes 02 – Procedure As Data (v1.0)

What in the World is `lambda`?

No, here, `lambda` is not a chemistry term, nor does it refer to a brilliant computer game that spawned an overrated mod. `lambda`, in Scheme, means "a procedure". The syntax:

```
(lambda ([formal parameters])
      [body of the procedure] )
```

So let's use a rather silly example:

```
(lambda (x)
      (* x x))
```

Read: a procedure that takes in a single argument, and returns a value that is that argument multiplied by itself. When you type this into Scheme, the expression will evaluate to something like:

```
#[closure arglist=(x) 14da38]
```

This is simply how Scheme prints out a "procedure". Since `lambda` just gives us a procedure, we can use it the way we always use procedures – as the first item of a function call. So let's try squaring `3`:

```
( (lambda (x) (* x x)) 6)
```

Read that carefully and understand what we just did. Our statement is a function call, whose function is "a procedure that takes in a single argument, and returns a value that is that argument multiplied by itself", and whose argument is the number `6`. Compare with `(+ 2 3)`; in that case, Scheme evaluates the first element – some procedure – and passes in the arguments `2` and `3`. In the same way, Scheme evaluates the first element of our statement – some procedure – and passes in the argument `6`.

A lot of you probably recognized the above `lambda` statement as our beloved "`square`". Of course, it's rather annoying to have to type that `lambda` thing every time we want to square something. So we can bound it to a name of our desire. Oh, let's just be spontaneous, and use the name "`square`":

```
(define square
      (lambda (x) (* x x)))
```

Note the similarities to `(define pi 3.1415926)`! For `pi`, we told Scheme to "bind to the symbol '`pi`' the value of the expression `3.1415926` (which is self-evaluating)". For `square`, we told Scheme to "bind to the symbol '`square`' the value of the `lambda` expression (which happens to be a procedure)". The two cases are not so different after all. If you type `square` into the Scheme prompt, it would print out something like this again:

```
#[closure arglist=(x) 14da38]
```

As we said before, this is how Scheme prints out a procedure. Note that this is not an error! It's simply what the symbol "`square`" evaluates to – a procedure. It's as natural as typing in "`pi`" and getting `3.1415926` back (if you've already defined `pi`, of course).

It's still a little annoying to type that `lambda` though, so we sugar-coat it a bit with some "syntactic sugar" to make it easier to read:

```
(define (square x) (* x x))
```

Ah, now that's what we're used to. The two ways of defining `square` are exactly the same to Scheme; the latter is just easier on the eyes, and it's probably the one you're more familiar with. Why bother with "`lambda`" at all, then? That is a question we will eventually answer, with vengeance.

QUESTIONS: What do the following evaluate to?

```
(lambda (x) (* x 2))
```

```
((lambda (y) (* (+ y 2) 8)) 10)
```

```
((lambda (b) (* 10 ((lambda (c) (* c b)) b))) ((lambda (e) (+ e 5)) 5))
```

```
((lambda (a) (a 3)) (lambda (z) (* z z)))
```

```
((lambda (n) (+ n 10))
 ((lambda (m) (m ((lambda(p) (* p 5)) 7))) (lambda (q) (+ q q))))
```

First-Class Procedures!

In programming languages, something is *first-class* if it doesn't chew with its mouth open. Also, as SICP points out, it can be bound to variables, passed as arguments to procedures, returned as results of procedures and used in data structures. Obviously we've seen that numbers are first-class – we've been passing them in and out of procedures since the beginning of time (which was about last week), and have been putting them in data structures – sentences – since the dinosaurs went extinct (also last week).

What is surprising – and exceedingly powerful – about Scheme, is that **procedures are also awarded first-class status**. That is, procedures are treated just like any other values! This is one of the more exotic features of LISP (compared to, say C or Java, where passing functions and methods around as arguments require much more work). And as you'll see, it's also one of the coolest.

Procedures As Arguments

A procedure which takes other procedures as arguments is called a **higher-order procedure**. You've already seen a few examples, in the lecture notes, so I won't point them out again; let's work on something else instead. Suppose we'd like to square or double every word in the sentence:

```
(define (square-every-word sent)
   (if (empty? sent)
       '()
       (se (* (first sent) (first sent)) (square-every-word (bf sent)))))

(define (double-every-word sent)
   (if (empty? sent)
       '()
       (se (* 2 (first sent)) (double-every-word (bf sent)))))
```

Note that the only thing different about `square-every-word` and `double-every-word` is just what we do to `(first sent)`! Wouldn't it be nice to generalize procedures of this form into something more convenient? When we pass in the sentence, couldn't we specify, also, what we want to do to each word of the sentence?

To do that, we can define a higher-order procedure called `every`. `every` takes in the procedure you want to apply to each element *as an argument*, and applies it indiscriminately. So to write `square-every-word`, you can simply do:

```
(define (square-every-word sent) (every (lambda(x) (* x x)) sent))
```

Now isn't that nice. Nicer still: the implementation of `every` is left as a homework exercise! You should be feeling all warm and fuzzy now.

The secret to working with procedures as values is to **keep the domain and range of procedures straight!** Keep careful track of what each procedure is supposed to take in and return, and you will be fine. Let's try a few:

QUESTIONS

1. **What does this guy evaluate to?**
   ```
   ((lambda(x) (x x)) (lambda(y) 4))
   ```

2. **What about his new best friend?**
   ```
   ((lambda(y z) (z y)) * (lambda(a) (a 3 5)))
   ```

3. **Write a procedure, `foo`, that, given the call below, will evaluate to `10`.**
   ```
   ((foo foo foo) foo 10)
   ```

4. **Write a procedure, `bar`, that, given the call below, will evaluate to `10` as well.**
   ```
   (bar (bar (bar 10 bar) bar) bar)
   ```

5. **Something easy: write `first-satisfies` that takes in a predicate procedure and a sentence, and returns the first element that satisfies the predicate test. Return `#f` if none satisfies the predicate test. For example, `(first-satisfies even? '(1 2 3 4 5))` returns 2, and `(first-satisfies number? '(a clockwork orange))` returns `#f`.**

```
(define (first-satisfies pred? sent)
```

Procedures As Return Values

The problem is often: write a procedure that, given _____, **return a procedure** that _____. The keywords – conveniently boldfaced – is that your procedure is supposed to return a procedure. This is often done through a call to lambda:

```
(define (my-wicked-procedure blah) (lambda(x y z) (...)))
```

Note that the above procedure, when called, will return a **procedure** that will take in three arguments. That is the common form for such problems (of course, never rely on these "common forms" as they'll just work against you on midterms!)

QUESTIONS

1. **In lecture, you were introduced to the procedure `keep`, which takes in a predicate procedure and a sentence, and throws away all words of the sentence that doesn't satisfy the predicate. The code for `keep` was:**

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

Recall that Brian said to `keep` numbers less than 6, this *wouldn't* work:
```
(keep (< 6) '(4 5 6 7 8))
```

a. Why doesn't the above work?

b. **Of course, this being Berkeley, and us being rebels, we're going to promptly prove the authority figure – the Professor himself – wrong. And just like some rebels, we'll do so by cheating. Let's do a simpler version; suppose we'd like this to do what we intended:**

```
(keep (lessthan 6) '(4 5 6 7 8))
```

Define procedure `lessthan` to make this legal.

c. **Now, how would we go about making this legal?**

```
(keep (< 6) '(4 5 6 7 8))
```

2. **Write a procedure exponents function, `(f-expt func power)` that returns a procedure which is equivalent to `func` applied `power` times. Assume `func` takes in only a single argument. For example, `((f-expt square 3) 2) ==> 256`, because `(square (square (square 2)))` is `256`.**