CS61A Notes 02 – Procedure As Data [Solutions v1.0]

What in the World is `lambda`?

QUESTIONS: What do the following evaluate to?

`(lambda (x) (* x 2))`
```
#[closure arglist=(x) e16fd0]
```

`((lambda (y) (* (+ y 2) 8)) 10)`
```
96
```

`((lambda (b) (* 10 ((lambda (c) (* c b)) b))) ((lambda (e) (+ e 5)) 5))`
```
1000
```

`((lambda (a) (a 3)) (lambda (z) (* z z)))`
```
9
```

```
((lambda (n) (+ n 10))
 ((lambda (m) (m ((lambda(p) (* p 5)) 7))) (lambda (q) (+ q q))))
80
```

Procedures As Arguments

QUESTIONS

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

2. **What about his new best friend?**
   `((lambda(y z) (z y)) * (lambda(a) (a 3 5)))`
   ```
   15
   ```

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

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

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)
            (cond ((empty? sent) #f)
                  ((pred? (first sent)) (first sent))
                  (else (first-satisfies pred? (bf sent)))))
   ```

Procedures As Return Values

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?
As we discussed in lecture, (< 6) evaluates to #t, not a procedure, since keep requires a procedure, it fails miserably.

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.
The insight is that (lessthan 6) must return a procedure. In fact, it must return a procedure that checks if a given number is less than 6. So...

```
(define (lessthan n)
        (lambda(x) (< x n)))
```

c. Now, how would we go about making this legal?
`(keep (< 6) '(4 5 6 7 8))`
The tricky thing here is that (< 6) must also return a procedure as we did up there. That requires us to redefine what '<' is, since '<' the primitive procedure obviously doesn't return a procedure.

```
(define (< n)
      (lambda(x) (> n x)))
```

Note also that we can't use '<' in the body as a primitive!

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`.**

This is pretty hard. Consider writing the normal numeric exponents:

```
(define (expt base power)
         (if (= power 0)
             1
             (* base (expt base (- power 1)))))
```

So, if power is 0, we have what's called the "identity" – 1. That is, raising something to the power of 0 returns the identity. Similarly, raising a function to the power of 0 should return the "identity function", which is a function that doesn't do anything to the argument. It makes sense – applying a function zero times is like not applying it at all.

In the recursive case, we'll want to apply the function power-1 times first (through recursion), and then apply it one more time.

```
(define (f-expt func power)
         (if (= power 0)
             (lambda(x) x)
             (lambda(x) (func ((f-expt func (- power 1)) x)))))
```