CS61A Notes 02 – Procedures as Data

What in the world is `lambda`?

QUESTIONS: What do the following evaluate to? Describe the result: if it is self-evaluating, determine the value; if it is a procedure, describe the number of arguments it takes.

1. `(lambda (x) (* x 2))`

   This returns a procedure that takes in one argument.

2. `((lambda (y) (* (+ y 2) 8)) 10)`
   This creates a procedure with one argument and then calls this procedure on the value `10`. The final answer is `96`.
3. `(`**`(lambda (b) (* 10 ((lambda (c) (* c b)) b)))`**
    `((lambda (e) (+ e 5)) 5))`
   Notice that this statement is a call to the procedure defined in bold, whose argument is the result of calling *another* procedure on the value `5`. As a result, `b` will be given the value `10`. In the body of the bold procedure, another procedure is called with `b` as argument, and thus `c` will be given the value `10`. The statement thus evaluates to `1000`.
4. `(`**`(lambda (a) (a 3))`** `(lambda (z) (* z z)))`
   This is a call to the procedure defined in bold, which calls its argument on the value `3`. Since its argument is a squarer, this statement evaluates to `9`.
5. `(`**`(lambda (n) (+ n 10))`**
    `(`*`(lambda (m) (m ((lambda (p) (* p 5)) 7)))`* `(lambda (q) (+ q q))))`
   The substitution model helps here, as does giving each nameless procedure a name. Again, this is a call to the procedure defined in bold, with the rest of the statement as its only argument. The rest of the statement is itself a call to the procedure defined in italics, with `(lambda (q) (+ q q))` as its argument. The argument to the procedure defined in italics must be a procedure itself, because it will be called in the body. We encourage you to continue to trace this through; the answer is `80`, but more important is to understand how this answer came about.

Procedures As Arguments

Keep the domain and range of procedures straight!

QUESTIONS

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

   Notice that this is a call to the procedure returned by the first `lambda`. Notice also that whatever the argument `x` to the first lambda, it must be a one-argument procedure, since it will be called with one argument (which happens to be itself, though this is not necessarily the case).

In this case, this one-argument procedure is the procedure returned by the second `lambda`, which ignores its input and always returns `4`. Inside the body of the first `lambda`, then, the second lambda will be called with itself as an argument, and since the second `lambda` always ignores its input and returns `4`, the net result of the statement is `4`. The substitution model also tells us that the statement yields

```
((lambda (y) 4) (lambda (y) 4))
```

which returns `4`; the second `lambda` is never called, though it is evaluated.

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

   In this case, we are passing in the `*` primitive, which is also a procedure. This explains why, for example, we need the statement `(* 5 4)` to calculate the product of `5` and `4`: the `*` primitive is a procedure, and must be called. We know that to call a procedure, we must precede it with a left parenthesis and follow it with its arguments.

   The substitution model tells us that the above call yields

   ```
   ((lambda (a) (a 3 5) *)
   ```

   which further yields `(* 3 5)`, and thus 15.

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

   From the given call, we know that `foo` must take on two arguments (which, in this call, happen to be copies of the procedure itself), and it must return a procedure that takes in two arguments, since the result of the call to `foo` will again be called with arguments `foo` and `10`. A potential answer (out of many) is
   ```
   (define (foo x y) (lambda (a b) b))
   ```

   Another answer is `(define (foo x y) (lambda (a b) 10))`.

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

   From its example call, we notice at least that `bar` is a procedure that takes in two arguments. We can achieve the intended output by having `bar` return its first argument. A potential answer is then `(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. (A *predicate* is a special name for a procedure that only returns either `#t` or `#f`.) 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 don'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)))))
```

   To keep numbers less than 6, George and the TAs claim this wouldn't work:

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

   a) Why won't the above work?

   As we discussed in lecture, `(< 6)` evaluates to `#t`, not a procedure, and since `keep` requires a procedure as a second argument, this call fails miserably.

   b) Of course, this being Berkeley, and we being rebels, we're going to promptly prove the authoritative figures 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 in part (b). That requires us to redefine what < is, since < the primitive procedure obviously does not return a procedure.

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

Note also that we can't use < in the body as a primitive! (Why not?)
