CS61A Week 2

Scheme Basics, Order of Evaluation, Recursion [a]

(v1.0)

[a] based on Chung Wu's notes

Some fun with Scheme

QUESTIONS: What do the following evaluate to? (based on CS 3 Spring 2001 Midterm)

1.
```
(equal? '1 (se (1)))
```

2.
```
(if (= 1 '1) + -)
```

3.
```
(if < 3 4)
```

4.
```
(* (+ 2 (/ 7 1)
     (*)))
```

5.
```
(define mantra '(cal is great))

  (and (if #f #f #t)
       'a
       '(b)
       (not (= 1 2))
       mantra)
```

6.
```
(define (weird a b c) '(/ 100 a))

  (weird 0 1 2)
```

7.
```
(cond (and (< 2 1) #t 'boo)
       (else 'awesome))
```

Applicative vs. Normal Order

Applicative order is where Scheme first evaluates completely all the procedures and arguments before executing the function call. Normal order, on the other hand, is to expand out complex expressions involving defined procedures into one involving only primitive operators and self-evaluating values, and then perform the evaluation. In other words, we defer evaluation of the arguments; we substitute the unevaluated expression (as opposed to evaluated expression for applicative order) into the body of a procedure.

Let's define a few functions to play with:

```
(define (double x) (+ x x))

(define (square y) (* y y))

(define (f z) (+ (square (double z)) 1))
```

QUESTIONS:

1. Evaluate `(f (+ 2 1))` in both applicative and normal order. Will you get the same result? Why or why not? Which order is more efficient?

   Applicative order:

   Normal order:

2. Is applicative order always more efficient than normal order? If so, why? If not, can you give a counter example?

recursion (n.) see recursion

A recursive procedure is one that calls itself in its body. The classic example is factorial:

```
(define (fact n)
    (if (= n 0)
        1
        (* n (fact (- n 1)))))
```

Note that, in order to calculate the factorial of n, we tried to first calculate the factorial of `(- n 1)`. This is because of the observation that 5! = 5 * 4!, and that, if we know what 4! is, we could find out what 5! is. If this makes you a bit suspicious, it's okay - it takes a little getting used to. The trick is to use the procedure as if it is already correctly defined. This will become easier with practice. The mantra to repeat to yourself is:

TRUST THE RECURSION!

QUESTIONS:

1. Write a procedure `(expt base power)` which implements the exponentiation function. For example `(expt 3 2)` returns 9.

2. I have a bag of n oranges and m apples. If I eat them one at a time, how many ways can I eat all oranges and apples?

3. Define a procedure `subsent` that takes in a sentence and an integer i. It returns a sentence with ith item in the original sentence to the last item (The first element being the 0th item). For example, `(subsent '(6 4 2 7 5 8))` returns `(7 5 8)`.

4. Define a procedure `sum-of-sents` that takes in two sentences of numbers and returns a sentence with the sum of respective elements from both sentences. Note that the sentences *do not* have to be the same size! For example, `(sum-of-sents '(1 2 3) '(4 5 6))` will return `(5 7 9)`, whereas `(sum-of-sents '(1 2 3 4 5) '(6 7))` will return `(7 9 3 4 5)`.

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 [list of 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?

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

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

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

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