CSE 341 — Scheme Discussion Questions

1. Suppose we evaluate the following Scheme expressions:

   (define x '(snail clam))
   (define y '(octopus squid scallop))

   Draw box-and-arrow diagrams of the result of evaluating the following expressions. In particular, note carefully what parts of the list are created fresh, and which are shared with the variables x and y.

   (a) (define a (cons 'geoduck x))
   (b) (define b (cons y y))
   (c) (define c (append x y))
   (d) (define d (cdr y))

2. Given the variables defined in Question 1, what is the result of evaluating the following expressions?

   (a) (eq? (car a) 'geoduck)
   (b) (eq? d '(squid scallop))
   (c) (equal? d '(squid scallop))
   (d) (eq? d (cdr y))

3. Continuing with the same variables used in Question 1, show how the box-and-arrow diagram changes after evaluating each of the following expressions.

   (set-car! x 'tuna)
   (set-car! y 'dolphin)
   (set-cdr! (cdr y) ())

4. Suppose we define some functions and variables:

   (define (test1 n)
     (set-cdr! n ())
     (display n))

   (define (test2 n)
     (set! n ())
     (display n))

   (define x '(snail clam))
   (define y '(octopus squid scallop))

   What gets printed when we evaluate

   (test1 x)
   (test2 y)

   What are the values of x and y afterward?

5. What is the result of evaluating the following Scheme expressions?
(a) \( \text{(let ((x (+ 2 4))}
   (y 100))
   (+ x y)) \)

(b) \( \text{(let ((x 100)}
   (y 5))
   (let ((x 1))
     (+ x y))) \)

6. Define a function `mylength` to find the length of a list.

7. Define a recursive function `add1` that takes a list of numbers, and returns a new list of numbers, each being 1 plus the original. For example, \( \text{(add1 ' (10 20 30))} \) should evaluate to \( (11 21 31) \).

8. Define a non-recursive version of `add1` that uses `map` and `lambda`.

9. Using `map` and `lambda`, define a function `averages` that accepts two lists of numbers, and returns a list of the average of each pair. For example:
   \( (\text{averages ' (1 2 3) ' (11 12 13)}) \) => \( (6 7 8) \)

10. Aloysius Q. Hacker, 341 student, is puzzled by the following code.

    \( (\text{define incr ()}) \)
    \( (\text{define get ()}) \)

    \( (\text{let ((n 0)}) \)
        \( (\text{set! incr (lambda (i) (set! n (+ n i)))}) \)
        \( (\text{set! get (lambda () n)))} \)

    Aloysius is unsure how the functions `incr` and `get` can possibly work... if `n` is in a stack frame, why do `incr` and `get` still work correctly even though we're done with evaluating the `let`?

    Is there something special about `let` at the top level? So Aloysius tries an experiment:

    \( (\text{define newincr ()}) \)
    \( (\text{define newget ()}) \)

    \( (\text{define (test k)}) \)
    \( (\text{let ((n 0)}) \)
        \( (\text{set! newincr (lambda (i) (set! n (+ n i k)))}) \)
        \( (\text{set! newget (lambda () n)))} \)

    Then he evaluates `(test 100)`. This time the `let` is embedded in a function, and so (Aloysius reasons) certainly its stack frame will go away when `test` returns.

    What is the result when Aloysius evaluates each of the following expressions in turn?

    \( (\text{newget}) \)
    \( (\text{newincr 10}) \)
    \( (\text{newget}) \)

    Explain (or at least make some reasonable hypotheses).

11. Define a tail-recursive version of “map” for 1-argument functions. (Avoid side effects if possible, but use them if necessary.)