CS 275
Exam I
March 10, 2020

My answers are in red:

1. Write procedure **(get n lat)** which returns the element of *lat* at index *n*. If *n* is greater than or equal to the length of *lat*, get should return the last element of lat. So (get 0 '(a b c d e)) returns a, (get 3 '(a b c d e)) returns d, and (get 100 '(a b c d e)) returns e.

```
(define get
    (lambda (n lat)
        (cond
            [(null? (cdr lat)) (car lat)]
            [(eq? 0 n) (car lat)]
            [else (get (- n 1) (cdr lat))])))
```

2. Write any way you wish procedure **(substitute* old new L)** which replaces each instance of *old* with *new* in the general list *L*. For example, (substitute 'a 'x '(a b a c)) returns '(x b x c)

Here's how The Little Schemer would do it:

```
(define substitute*
      (lambda (old new L)
            (cond
                  [(null? L) null]
                  [(atom? (car L)) (if (eq? old (car L))
                                       (cons new (substitute* old new (cdr L)))
                                       (cons (car L) (substitute* old new (cdr L))))]
                  [else (cons (substitute* old new (car L))
                              (substitute* old new (cdr L)))])))
```

Here's a shorter solution with map (you don't need apply here):

```
(define substitute*
      (lambda (old new L)
            (cond
                  [(null? L) null]
                  [(atom? L) (if (eq? old L) new L)]
                  [else (map (lambda (t) (substitute* old new t))L)])))
```

3. Use map and apply to write function **(nums L)**, which returns a flat list of all of the numbers in general list L. For example (nums '(a (2 bob) (3 (c (4 5 six)))) returns (2 3 4 5). Note that there is a primitive predicate (number? x) that returns #t if x is a number and #f if it isn't. You don't need to write (number? x).

```
(define nums
      (lambda (L)
            (cond
                  [(null? L) null]
                  [(atom? L) (if (number? L) (list L) null)]
                  [else (apply append (map nums L))])))
```

4. **What will this return?** Here (member? a L) is the usual predicate that returns #t if a is an element of list L.

(foldr (lambda (x y) (if (member? x y) y (cons x y))) null '(1 2 3 3 2 3 4))

This starts at the end of the list and works towards the front, appending on any element of the list that is not already in the answer. For this particular list it will return (1 2 3 4)

5. **What is the result** of the expression

```
(let ([f (lambda (x) (+ x 1))])
    (let ([f (lambda (y) (if (= y 0) 10 (* 2 (f 0))))])
        (f 3)))
```

The outer let makes a new environment, call it E1, that extends the top-level environment with a binding of f to the "add 1" function and evaluates the inner let within E1. The inner let evaluates
(lambda (y) (if (= y 0) 10 (* 2 (f 0))) within E1, producing a closure that I'll call C, makes a new environment E2 in which f is bound to C, and evaluates (f 3) in E2. Note that in C the parameter list is (y), the body is
(if (= y 0) 10 (* 2 (f 0))) and the closure environment is E1. When we call this with argument 3 we get (* 2 (f 0)) which is evaluated in E1. Within E1 (f 0) is 1, so altogether we get (* 2 1) or 2.

The result of the expression is thus 2.

6. Write **(Alternates lat)** which returns a pair of lists, the first with the even-indexed elements of *lat* and the second with the odd-indexed elements. The two returned lists should have elements in the same order as the original lat, For example, (Alternates '(a b c d e f g)) returns '((a c e g) (b d f)).

There are lots of convoluted ways to do this but there is also a straightforward way. Here is a function alt that takes every other element from a list: (alt '(a b c d e)) is (a c e):

```
(define alt
        (lambda (lat)
            (cond
                [(null? lat) null]
                [(null? (cdr lat)) lat]
                [else (cons (car lat) (alt (cddr lat)))])))
```

With this Alternates is easy:

```
(define Alternates
        (lambda (lat)
                (list (alt lat) (alt (cdr lat)))))
```

7. Let's say that a *count list* for a list L is a list of pairs, where the first element of each pair is one of the atoms of L and the second element of the pair is how often that atom occurs in L. So a count list for '(a b a c d d a) is ((a 3) (b 1) (c 1) (d 2)). Write function **(CountList lat)** that returns a count list for flat list *lat*. Your solution can list the elements of *lat* in any order you wish.

   If you recognize the building blocks this isn't too bad. At each step we want to take the next atom out of lat, count its occurrences within lat, and remove all of those occurrences from lat:

```
(define CountList
        (lambda (lat)
                (cond
                        [(null? lat) null]
                        [else (cons (list (car lat) (count (car lat) lat))
                                    (CountList (rember-all (car lat) lat)))])))
```

   The helper functions count and rember-all are *Little Schemer*-sorts of functions. There are many ways to write each of them; here they are with foldr:

```
(define count
        (lambda (a lat)
                (foldr (lambda (x y) (if (eq? x a) (+ y 1) y)) 0 lat)))

(define rember-all
        (lambda (a lat)
                (foldr (lambda (x y) (if (eq? x a) y (cons x y))) null lat)))
```