EECS 345: Programming Language Concepts

Programming Exercise 2

Due Friday, February 20

For questions 1-8, write Scheme definitions for the following functions using tail recursion and continuation passing style (CPS). The continuation argument should be the last argument. For example, if you were asked to write factorial, the "normal" recursive function is:

```
(define factorial
  (lambda (n)
    (if (zero? n)
        1
        (* n (factorial (- n 1))))))
```

and so your answer should be

```
(define factorial-cps
  (lambda (n return)
    (if (zero? n)
        (return 1)
        (factorial-cps (- n 1) (lambda (v) (return (* n v)))))))
```

You do not have to convert simple scheme built-in non-recursive functions like `null?`, `eq?`, `list?`, `number?`, `car`, `cons`, `cdr`, `list` to CPS, but all other helper functions you create should be in CPS.

Try to write these using no helper functions other than the other functions in this homework.

1. `multiplyby` takes a number and a list of numbers and returns a list that is the input list with each element multiplied by the input number.
   You can write this function without any helper functions.

   ```
   > (multiplyby 5 '(1 2 3 4 10 11))
   (5 10 15 20 50 55)
   ```

2. `crossmultiply` takes two lists of numbers, each list represents a vector. Returns the *outer product* of the two vectors. The outer product is a matrix (a list of lists) and each list is the result of multiplying the second list by the corresponding value of the first list.

   ```
   > (crossmultiply '(1 2 3) '(3 0 2))
   ((3 0 2) (6 0 4) (9 0 6))
   > (crossmultiply '(8 -1 4 3) '(3 1))
   ((24 8) (-3 -1) (12 4) (9 3))
   ```

3. `maxnumber` takes a list of numbers that contains at least one number and returns the largest number in the list.
   You can write this function without any helper functions.

   ```
   > (maxnumber '(3 1 5 2 7 5 3 8 1 2))
   8
   ```

4. `partialsums*` takes a list that may contain sublists. The output should have the same list structure as the input, but the car of each list (and each sublist) should be the sum of all numbers in that list.

There should be no other values in the list.
You can write this function without any helper functions.

```
> (partialsums* '(1 a 2 b))
(3)
> (partialsums* '((1) (2) (3)))
(6 (1) (2) (3))
> (partialsums* '((1 2) (10 20) (100 200)))
(333 (3) (30) (300))
> (partialsums* '(1 a (10 20 x y) c 2 (x y z) (a b 1 (c 200 d))))
(234 (30) (0) (201 (200)))
```

5. `trimatoms` takes a list, possibly containing sublists, and a list of atoms. It returns the list of atoms with the first *k* atoms of the list removed where *k* is the number of non-null atoms in the first list.
You can write this function without any helper functions.

```
> (trimatoms '(a b c) '(1 2 3 4))
(4)
> (trimatoms '(((a)) (b ((c)))) '(1 2 3 4 5))
(4 5)
> (trimatoms '(((a)) () () (b ((c)))) '(1 2 3 4 5))
(4 5)
```

6. `exchange` takes a list (possibly containing sublists) and a list of atoms. You may assume that both lists contain the same number of non-null atoms. The output should be identical to the first list except that the atoms of the list should be the same as the atoms of the second list, in order.

```
> (exchange '(((a)) (b ((c)))) '(1 2 3))
(((1)) (2 ((3))))
> (exchange '((() a) b) c) '(1 2 3))
((() 1) 2) 3)
```

7. `removesubsequence*` takes a list of atoms and a general list. The first list is a *subsequence* of the second list. The method should return the second list with the first occurence of the subsequence removed. So, if the first list is `'(a b c)`, the first `a` if the second list is removed, the first `b` that appears after the removed `a` is removed, and the first `c` that appears after the removed `b` is removed - no matter how deep the atoms are nested.

```
> (removesubsequence* '(a b) '(w (x b) ((a) ((y z))) b) (lambda (v1 v2) v2))
(w (x b) (() ((y z))))
```

   As a hint, you need to keep track of two values between recursive calls. So use two values in the continuation function: `(removesubsequence* '(a b) '(w (x b) ((a) ((y z))) b) (lambda (v1 v2) v2))`
8. The function `split` takes a list and returns a list containing two sublists, the first with the elements at the odd indices and the second with the elements at the even indeces (assuming the first element is at index 1: Try writing this using only two cases: `(null? l)` and `else`. You should do something similar to the `removesubsequence*` where you keep track of two values and use a different continuation function.

```
> (split '())
(()())
> (split '(1))
((1)())
> (split '(1 2 3 4 5))
((1 3 5) (2 4))
```

9. **Write the following function without external helper functions or additional parameters.** You

may use `letrec` to create an internal helper function that uses continuation passing style.
The function `suffix` takes an atom and a list and returns a list containing all elements that occur *after* the last occurrence of the atom.

```
(suffix 'x '(a b c))  ==> (a b c)
(suffix 'x '(a b x c d x e f)) ==> (e f)
```

10. **Write a second version of suffix that uses `call/cc` instead of the "normal" continuation passing style.** You may not use external helper functions or additional parameters, but you may use `letrec`.

```
(suffix2 'x '(a b c))  ==> (a b c)
(suffix2 'x '(a b x c d x e f)) ==> (e f)
```