EECS 345: Programming Language Concepts

Programming Exercise 1

Due Saturday, January 31

The purpose of this programming exercise is to learn the basic functional programming paradigm and become comfortable using recursion. In this homework, you are to create a number of Scheme functions. You are to follow a strict function programming style. That means you need to follow the style we used in class and use only functions, parameters, and recursion. You may write helper functions if needed, and you may use functions created for one problem to solve another. Please do not use built in Scheme functions except the ones we used in class: `car`, `cdr` (and all their variants like `cadr` and `cdar`), `cons`, `append`, `null?`, `pair?`, `list?`, `number?`, `=`, `eq?` `zero?`, `if`, `cond`, and all the standard arithmetic and logic functions.

Please include a comment at the top of the file giving your name, and please include a comment at the top of each function briefly explaining the function. Scheme comments start with a semicolon.

Do not nest `cond` statements. Nor have more than two `if` statements nested inside each other. Instead, rearrange your logic so that you can write your function with a single `cond` of multiple cases.

You can assume all input is in the proper format.

Write the following functions:

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. `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
   ```

3. `removelast` takes a list of atoms containing at least one atom and returns the same list minus the last atom of the list
   You can write this function without any helper functions.

   ```
   > (removelast '(a b c d e))
   (a b c d)
   ```

4. `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))
```

5. `interleave3` takes three lists of atoms and returns a list where the elements are interleaved in order in the pattern: *(list1, list2, list3, list1, list2, list3, ...)*. The lists do not have to be the same length.

```
> (interleave3 '(a b c d) '(1 2) '(P Q R S T U))
(a 1 P b 2 Q c R d S T U)
```

6. `reverse*` takes a list that may contain sublists and reverses the contents of the list. The contents of each sublist is also reversed.
   You can write this function without any helper functions.

```
> (reverse* '(a b (c d e) ((f g) h)))
((h (g f)) (e d c) b a)
```

7. `reverselists` takes a list, possibly containing sublists. The outerlist is not reversed, but each sublist is reversed. Any sublists inside those lists should stay in the original order, but sublists of the sublists should be reversed. This pattern should repeat for as many layers of sublists are in the list.

```
> (reverselists '(a b c d))
(a b c d)
> (reverselists '(a b (c d e f) g h (i j k l)))
(a b (f e d c) g h (l k j i))
> (reverselists '(a b (c (d e) f) g h ((i (j k)) l)))
(a b (f (d e) c) g h (l (i (k j))))
```

8. `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)
```

9. `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)))
```

10. `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)
```