Programming Languages
CSCI-GA.2110.001 Fall 2019

Scheme Assignment
Due Tuesday, October 22 at 11:55pm

Your assignment is to write a number of small Scheme functions. All code must be purely functional (no use of `set!`, `set-car!`, or `set-cdr!` allowed) and should concise and elegant.

Scheme programming will be on the mid-term exam on October 23, so it is essential that you finish this assignment on time. Please write your own code. You may discuss this assignment with the other students, and help each other, but if you don't write your own code then you will not do well on the midterm exam. You can email me if you get stuck, but don't wait until the last minute.

Put all your code into single file, `assignment.scm`, and upload it to the course web site.

Additionally every recursive function that you write must be preceded by a comment that shows your reasoning about the recursion. For example, the comment before the solution to problem 1 below might be:

```
;;; Base Case:  L is empty, return 0
;;; Assumption: (count-numbers M) returns a count of the numbers in M, for
;;;             any list M smaller than L (including (car L) and (cdr L)).
;;; Step: If (car L) is a list, then return the sum of the count of the
;;;       numbers in (car L) and the count of the numbers in (cdr L).
;;;       If (car L) is a number, return 1 plus the count of the numbers in
;;;       (cdr L). Otherwise, return the count of the numbers in (cdr L).
```

1. Write a function `(count-numbers L)` that returns a count of the numbers found anywhere within `L`, including in nested lists. For example,

   ```
   > (count-numbers '(11 (22 (a 33 44) 55) (6 (b 7 8 (9 c) 100 d))))
   10
   ```

   Note that `(list? x)` returns true if `x` is a list, false otherwise. Also, `(number? x)` returns true if `x` is a number, false otherwise. Your function should be roughly 5 lines. If it is much more than that, you are doing something wrong.

2. Define a function `(insert x L)` where `x` is a number and `L` is a sorted list of numbers in increasing order, that returns a new sorted list containing `x` and all the elements `L`. That is, `x` appears in the right place in the new list. For example,

   ```
   > (insert 5 '(1 2 3 4 6 7 8))
   (1 2 3 4 5 6 7 8)
   ```

   Your function should be roughly 4 lines.

3. Define a function `(insert-all L M)`, where `L` is list of numbers and M is a sorted list of numbers, that returns a sorted list containing all the elements of `L` and all the elements of `M`. For example,

```
> (insert-all '(2 4) '(1 3 5))
(1 2 3 4 5)
> (insert-all '(3 6 1 5 2 7 4) '())
(1 2 3 4 5 6 7)
```

Of course, your `insert-all` function should call `insert`. Note that when the second parameter is the empty list, `insert-all` performs a kind of insertion sort. Your function should be roughly 3 lines.

4. Define the function `(sort L)`, where `L` is an unsorted list of numbers, that returns a new list containing the elements of `L` in sorted order. `sort` must perform an insertion sort, such as one you implemented above, but it cannot call any external function that you've defined. That is, `sort` cannot call any user-defined function defined outside `sort`, such as `insert-all`, but of course can define nested functions using `letrec`. For example,

   ```
   > (sort '(3 6 1 5 2 7 4))
   (1 2 3 4 5 6 7)
   ```

   Your function should be roughly 9 lines or less (including the nested function(s)).

5. In Scheme, there is no difference between `7` and `'7`. That is, quoting a number returns the number. However, there is a big difference between `+`, the addition operator, and `'+`, the symbol whose name is the plus symbol. It's the same distinction between `x`, a reference to the variable named x, and `'x`, the symbol whose name is x.

   Write a simple function, `(translate op)`, where `op` is the symbol `'+`, `'-`, `'*`, or `'/`, that returns the corresponding operator. For example,

   ```
   >  (translate '-)   ;; should return the - operator
   #<procedure:->
   > ((translate '*) 3 4)   ;; calling the operator that translate returns.
   12
   ```

   To test equality of symbols, you can use the `(eq?  x y)` function, which returns true if `x` and `y` are the same symbol, false otherwise. For example, `(eq?  'a (car '(a b c)))` returns true. Your function should be roughly 5 lines or less (and it doesn't have to be recursive).

6. Write the function `(postfix-eval exp)`, where `exp` represents an expression in postfix notation, that returns the result of evaluating `exp`. For this assignment, a postfix expression can only be of the following form:

   - A number, or
   - a list of the form `(`*`arg1 arg2 op`*`)`, where *arg1* and *arg2* are postfix expressions and *op* is an operator symbol. Note that the list must contain exactly three elements (some of which may be lists, of course) and represents the application of the operator *op* to the operands *arg1* and *arg2*.

   For example,

```
>   ;; evaluating the postfix equivalent of  (16 * 12) / ((2 + 6) * (9 - 1))
(postfix-eval '((16 12 *) ((2 6 +) (9 1 -) *) /))
3
```

You should use your `translate` function, above. Hint: Do not try to rearrange the postfix expression, just evaluate it directly. Your function should be roughly 5 lines.

7. Suppose you are using a list whose elements are distinct to represent a set (which, unlike a list, has no order). Define the function `(powerset L)`, where `L` is such a list representing a set, that returns the power set of `L`, i.e. the set of all subsets of `L`. Of couse, each subset is also represented by a list. When considered as sets, the list `(2 3)` is the same as the list `(3 2)`, so both of these lists cannot appear in the same list representing a larger set (such as a power set). For example,

```
> (powerset '(1 2 3))
(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))   ;; the order doesn't matter here
>  (powerset '(1 2 3 4))
(() (4) (3) (3 4) (2) (2 4) (2 3) (2 3 4) (1) (1 4) (1 3)
(1 3 4) (1 2) (1 2 4) (1 2 3) (1 2 3 4))
```

As an aid to your recursive thinking, here are the first two parts:

```
;;; Base Case:  L is empty, return the set containing the empty
;;;             set, i.e. '(()).
;;; Assumption: (powerset M) returns the powerset of M, for any set M
;;;             smaller than L (including (cdr L)).
;;; Step: HINT: Think about the relationship between (powerset L) and
;;;      (powerset (cdr L)).
```

Your function should be roughly 5 lines.