Programming Languages
CSCI-GA.2110.001 Spring 2017

Scheme Assignment
Due Monday, March 20

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.

Important: In a comment preceding each recursive function, you must write out your recursive thinking. The comment should describe the base case, the assumption, and the recursive step. You don't need to do this for non-recursive functions, of course. For example, for the first question, my recursive reasoning would be:

```
;;; Base Case: L contains one element, return the first element of L, (car L).
;;; Assumption: find-min works on (cdr L), returning the smallest element
;;;             of (cdr L).
;;; Step: Let x be the result of calling (find-min (cdr L)).  If (car L) is
;;;       less than x, return (car L). Otherwise, return x.
```

1. Write a function `(find-min L)`, where `L` is a list of numbers, that returns the smallest element of `L`. You can assume that `L` has at least one element (no error checking required). For example,

   ```
   > (find-min '(4 1 7 2 9 10))
   1
   ```

   Be sure to have at most one recursive call to `find-min` within the body of the function. Remember to show your recursive reasoning. Your code should be roughly 4 lines.

2. Write a function `(find-min-rest L)`, where `L` is a list of numbers, that returns a list of two things:

   - The smallest element of `L`, and
   - A list containing all the elements of `L` except for the smallest element of `L`.

   For example,

   ```
   > (find-min-rest '(4 1 7 2 9 10))
   (1 (4 2 7 9 10))
   ```

   You should not call your `find-min` function above. That is, the only function (other than the built-in functions `null?`, `car`, `cdr`, `cadr`, and `list`) that `find-min-rest` should call is itself (and only once in the body of the function). You can assume that `L` has at least one element. Hint: The base case is:

   ```
   ;;; Base Case: If L has one element, return a list containing (car L) and '().
   ```

   Also, consider using a form of `LET` to save the result of calling `find-min-rest` recursively. Your function should be roughly 8 lines (or less). Don't forget to write down your recursive reasoning.

3. Write a single function `(sort L)`, where `L` is a list of numbers, that uses a form of selection sort to return a list containing the elements of `L` sorted in increasing order. The selection sort function can be described as returning a list whose first element is the smallest element of the original list and whose subsequent elements result from performing selection sort on the other elements of the original list. You should use your `find-min-rest` to implement `sort`. For example,

```
> (sort '(3 4 1 2 6 9))
(1 2 3 4 6 9)
```

Be sure to only call `find-min-rest` and `sort` once in the body of the sort function. Your function should be roughly 4 lines. Don't forget to write down your recursive reasoning.

4. Define a function `(sum-list L)`, where `L` is a list containing numbers and nested lists, that adds up all the numbers found in `L` at any depth of nesting. All the atoms within `L` are numbers, so you don't have to worry about finding symbols, etc., inside of `L`. For example,

```
> (sum-list '(2 3 (4 5) (6 (7 8)) 9))
44
```

Other than the built-in functions, the only function that `sum-list` should call is itself (which it can call several times). Your code should be roughly 5 lines. Don't forget to write down your recursive reasoning.

5. Write the function `(map2 f L1 L2)`, where `f` is a function of two parameters and `L1` and `L2` are lists. `map2` should apply `f` to the corresponding elements of `L1` and `L2`, returning a list of the results. For example,

```
> (map2 (lambda (x y) (+ x y)) '(1 2 3 4 5) '(10 20 30 40 50))
(11 22 33 44 55)
```

You can assume the two lists are the same length. Do not use the built-in `map` function. Your code should be roughly 3 lines. Don't forget to write down your recursive reasoning.

6. Write the function `(nums-from n m)`, where `n` and `m` are integers, that returns the list of all integers from `n` to `m`, inclusive. For example,

```
> (nums-from 5 15)
(5 6 7 8 9 10 11 12 13 14 15)
```

Your code should be roughly 3 lines. Don't forget to write down your recursive reasoning.

7. Write the function `(remove-mults n L)`, where `n` is an integer and `L` is a list of integers, that returns the list of all elements of `L` that are <u>not</u> a multiple of `n`. For example,

```
> (remove-mults 3 '(1 2 3 4 5 6 7 8 9))  ;; removes multiples of 3
(1 2 4 5 7 8)
> (remove-mults 2 (nums-from 2 20))  ;; removes multiples of 2
(3 5 7 9 11 13 15 17 19)
```

You can use the built-in function `(modulo x y)` which returns the remainder of `x` divided by `y` (so the result is 0 if `x` is a multiple of `y`). Your code should be roughly 4 lines. Don't forget to write down your recursive reasoning.

8. Write the function `(sieve L)`, where `L` is a list of integers, that returns the list of elements of `L` that are not multiples of each other. For example,

```
> (sieve '(4 5 6 7 8 9 10 11 12 13 14 15 16 17))
(4 5 6 7 9 11 13 17)
```

`sieve` should call `remove-mults`. Note: You can assume that the elements of `L` are sorted in increasing order. Your code should be roughly 3 or 4 lines. Don't forget to write down your recursive reasoning.

9. Write the function `(primes n)`, where `n` is an integer, that returns a list of all the prime numbers less than or equal to `n`. For example:

```
(primes 30)
(2 3 5 7 11 13 17 19 23 29)
```

Hint: `primes` should not be recursive, it should just call `sieve` and `nums-from`. It should be one or two lines. No need to provide the recursive reasoning, since it is not recursive.

10. Write a function `(gen-fn-list n)` that returns a list of `n` functions, such that the first function in the list takes a parameter x and returns x+1, the second function takes a parameter x and returns x+2, the third functon takes a list and returns x+3, etc. For example,

```
> (define fs (gen-fn-list 3))
> ((car fs) 10)
11
> ((cadr fs) 10)
12
> (map (lambda (f) (f 10)) (gen-fn-list 6))
(11 12 13 14 15 16)
```

`gen-fn-list` should not call any external functions, other than `cons` and arithmetic operators. You can define a nested function using `letrec`, though, if you want. Your code should be roughly 5 lines. If you define a recursive fuction, be sure to write down your recursive reasoning.