Programming Languages
CSCI-GA.2110.001 Fall 2017

Scheme Assignment
Due Sunday, October 22

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.

Additionally, each 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:

```
;; (fromTo k n)  returns the list of integers from k to n. The size
;;               of the problem can be seen as the number of integers
;;               between k and n, inclusive.
;; Base Case:  if k > n (i.e. if the size of the problem is 0), then
;;             the result is the empty list.
;; Hypothesis: Assume (fromTo (+ k 1) n) returns the list of integers
;;             from k+1 to n, since the size of that problem is one
;;             less than the size of the orignal problem, (fromTo k n).
;; Recursive step: (fromTo k n) = (cons k (FromTo (+ k 1) n)
```

1. Define the function `(fromTo k n)` that returns the list of integers from k to n, inclusive. For example, `(fromTo 3 8)` should return the list `(3 4 5 6 7 8)`. Include a comment showing the recursive reasoning. This function should be roughly 3 lines long (depending on where you put the line breaks), so if your function is much longer, it's probably incorrect.

2. Define the function `(removeMults m L)` that returns a list containing all the elements of `L` that are not multiples of `m`. For example, `(removeMults 3 '(2 3 4 5 6 7 8 9 10))` should return `(2 4 5 7 8 10)`. Note that the `modulo` operator in Scheme, `(modulo i j)` returns the remainder of dividing `i` by `j`. Include a comment showing the recursive reasoning. This function should be roughly 4 lines long.

3. Define the function `(removeAllMults L)` which, given a list `L` containing integers in strictly increasing order, returns a list containing those elements of `L` that are not multiples of each other. For example, `(removeAllMults '(3 4 6 7 8 10 12 15 20 22 23))` returns the list `(3 4 7 10 22 23)`. Include a comment showing the recursive reasoning. (Hint: Use `removeMults`, above). This function is roughly 3 lines long.

4. Define the function `(primes n)` that computes the list of all primes less than or equal to `n`. For example, `(primes 30)` should return `(2 3 5 7 11 13 17 19 23 29)`. (Hint: use some of the functions you've already defined). This function should be no longer than 2 lines. It does not need to be recursive, so you do not need to write a comment showing recursive reasoning.

5. Define a function `(maxDepth L)`, where `L` is a list, that returns the maximum nesting depth of any element within `L`, such that the topmost elements are at depth 0. Since `L` can contain lists, which themselves have nested lists, `maxDepth` will need to traverse down through the nesting to figure out the maximum depth. For example, `(maxdepth '(1 2 3))` should return 0, since there is no nesting, and

```
(maxdepth '((0 1) (2 (3 (4 5 (6 (7 8) 9) 10) 11 12) 13) (14 15)))
```

should return 5, since the elements 7 and 8 are at nesting depth 5. Include a comment showing the recursive reasoning. This function should be roughly 7 lines long.

Please be sure not to repeat calls to functions unnecessarily. For example, if you call `maxdepth` on a list, don't call it again on the same list. Instead, use a `LET` to save the result the first time for reuse. Otherwise, this function may end up exponential in complexity rather than linear.

6. Define a function `(prefix exp)` which transforms an infix arithmetic expression `exp` into prefix notation. An infix arithmetic expression has arithmetic operators between operands, whereas in prefix notation, the operator precedes the operands. In this case, `exp` can either be an atom (number or symbol) or a list containing at least three elements: the first operand, the operator, and the second operand. Each operand can itself be an expression (atom or list). Furthermore, the infix expression can contain more than three elements, e.g. `'(3 + 4 * 5 + 6)`, in which case the operators should be considered to be right associative and all of the same precedence. The result should be either an atom or a list containing exactly three elements: operator, first operand, and second operand. Note that your `prefix` function should <u>not</u> evaluate the infix expression, it should just transform the expression to prefix form. Here are some examples of the result of calling `prefix`.

```
> (prefix 3)
3
> (prefix '(3 + 4))
(+ 3 4)
> (prefix '((3 + 4) * 5))
(* (+ 3 4) 5)
> (prefix '(3 + 4 * 5 - 6))
(+ 3 (* 4 (- 5 6)))
> (prefix '((3 * 4) + (5 - 6) * 7))
(+ (* 3 4) (* (- 5 6) 7))
```

Include a comment showing the recursive reasoning. This function should be roughly 5 lines.

7. Define a function `(composition fns)` takes a list of functions `fns` and returns a function that is the composition of the functions in `fns`. That is, if `fns` contains the functions f, g, and h, then `(composition fns)` should return a function that is defined to be $f \circ (g \circ h)$. Recall that $(p \circ q)(x) = p(q(x))$. For example, given

```
(define f (composition (list (lambda (x) (+ x 1)) (lambda (x) (* x 2)))))
```

the call `(f 3)` should return 7. You can assume that `fns` contains at least one function. Include a comment showing the recursive reasoning. This code should be between 3 and 6 lines (depending on your solution).

8. The next three functions implement a functional form of bubble sort. In an imperative language, bubble sort takes an array A of size N and sorts it according to the following code (written in C):

```
for(i = N-1; i >= 0; i--)
  for(j = 0; j < i; j++)
    if (A[j] > A[j+1]) {
      temp = A[j];  // swap
      A[j] = A[j+1];
      A[j+1] = temp;
    }
```

   That is, what bubble sort does in the first pass over the array – by just comparing neighbors and swapping them as necessary – is move ("bubble up") the largest number to the Nth (last) position in the array. In the second pass over the array, it moves the largest remaining number to the N-1th position in the array, etc.

   The first function for you to write in Scheme is `(bubble-to-nth L N)`, where `L` is a list of numbers and `N` is an integer. The result should be a list containing all the elements of `L`, except that the largest element among the first `N` elements of `L` is now the `N`th element of the resulting list, and the elements after the `N`th element are left in their original order. Only neighboring elements in a list should be compared to each other (since this is will be used for bubble sort). For example,

```
> (bubble-to-nth  '(1 6 2 3 5 4 8 0) 3) ;; bubble the largest number among the
                                        ;; first 3 elements to the 3rd element
(1 2 6 3 5 4 8 0)
> (bubble-to-nth  '(1 6 2 3 5 4 8 0) 4) ;; bubble among the first 4 elements
(1 2 3 6 5 4 8 0)
> (bubble-to-nth  '(1 6 2 3 5 4 8 0) 5) ;; bubble among the first 5 elements
(1 2 3 5 6 4 8 0)
> (bubble-to-nth  '(1 6 2 3 5 4 8 0) 6)
(1 2 3 5 4 6 8 0)
> (bubble-to-nth  '(1 6 2 3 5 4 8 0) 7)
(1 2 3 5 4 6 8 0)
> (bubble-to-nth  '(1 6 2 3 5 4 8 0) 8)
(1 2 3 5 4 6 0 8)
```

   Be sure to write out your recursive reasoning. This function should be roughly 7 lines or less.

9. Write the function `(b-s L N)`, where `L` is a list of numbers and `N` is an integer, that returns the a list containing the elements of `L` in their original order except that the first `N` elements are in sorted order. This function should call `bubble-to-nth` above. Be sure to write out your recursive thinking. Your code should be roughly four lines.

10. Define the function `(bubble-sort L)`, which calls `b-s` above to return a list of the elements of `L` in sorted order. This does not need to be recursive, so there is no need to write out any recursive reasoning. It should be at most two lines.