Programming Languages
CSCI-GA.2110.001 Fall 2015

Scheme Assignment
Due Sunday, October 18

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 difference
;;               between k and n.
;; Base Case:  if k = n (i.e. if the size of the problem is 0), then
;;             the result is the list containing only n.
;; Hypothesis: Assume (fromTo (+ k 1) n) returns the list of integers
;;             from k+1 to 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 the comment showing the recursive reasoning.

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 the comment showing the recursive reasoning.

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 the comment showing the recursive reasoning. (Hint: Use removeMults).

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).

5. Define a function `(maxDepth L)` 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 the comment showing the recursive reasoning.

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 the comment showing the recursive reasoning.

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 the comment showing the recursive reasoning.