Programming Languages
Homework 4
Due Wednesday, February 5th at 2 AM

In this Homework assignment, we will be writing in LISP. If you need help on how to run LISP on Openlab, refer to this document here.

In writing the following LISP functions, you may use only these primitive functions:

- defun
- cond
- cons
- car
- cdr
- operators +, -, <, and >
- null
- eq
- listp
- atom
- symbolp

You may also write and use auxiliary functions, but you must submit these along with your solution. HINT: some functions require subfunctions.

NOTE: You MUST use recursion instead of looping. DO NOT assume the lists are SIMPLE unless it is explicitly specified in the problem statement.

EFFICIENCY NOTE: Do not traverse the whole list more than once. For example, do not flatten the list before processing it. Please provide sample runs that adequately test the functionality of each function.

We will start by writing some built-in functions, but we will add “my_” (or “my-”) to the front of the name. I tried to put these in order of difficulty (IMO). The first two are for practice and not worth any points, but please show them working correctly in your report.

1. **[8]** Define a function **my-length** that takes one parameter, a list L, and returns the number of top-level elements in L. Examples:
   ```
   ○ (my-length nil)                --> 0
   ○ (my-length '(b (a b c))        --> 2
   ○ (my-length '(a (((b))) c))     --> 3
   ○ (my-length '(a b c))           --> 3
   ```

2. **[10]** Define a function **my-memq** that takes two parameters, a symbol A and a list of symbols L, and returns the list starting where the symbol bound to A was found. It returns nil otherwise. Examples:

```
○ (my-memq 'a nil)             --> nil
○ (my-memq 'b '(a b c))        --> (b c)
○ (my-memq 'd '(a b c d e f g)) --> (d e f g)
○ (my-memq 'd '(a b c d))      --> (d)
○ (my-memq 'd '(a b c))        --> nil
```

3. **[8]** Define a function **my-append** that takes two parameters, a list L1 and a list L2, and returns the result of appending the two lists together. You must not call append.

```
○ (my-append '(a b c) '(d e f))
  --> (a b c d e f)
○ (my-append '((a) (b) (c)) '((d) (e) (f)))
  --> ((a) (b) (c) (d) (e) (f))
○ (my-append nil '(d e f))
  --> (d e f)
○ (my-append '(a b c) nil)
  --> (a b c)
```

4. **[6]** Define the function **my-attach** which takes an object O and a list L and returns the list L with O added to the end. Examples:

```
○ (my-attach 'a nil)           --> (a)
○ (my-attach 'd '(a b c))      --> (a b c d)
○ (my-attach '(a) '(b c))      --> (b c (a))
```

5. **[10]** Define the function **my-assoc** that takes an atom A and a list L and returns the association pair for A. L is of the form ((key1 . value1)(key2 . value2) … (keyn . valuen)) Examples:

```
○ (my-assoc 'a nil)                     --> nil
○ (my-assoc 'a '((a . b)(c e f)(b)))    --> (a . b)
○ (my-assoc 'c '((a . b)(c e f)(b)))    --> (c e f)
○ (my-assoc 'b '((a . b)(c e f)(b)))    --> (b)
○ (my-assoc 'f '((a . b)(c e f)(b)))    --> nil
```

6. **[6]** Define the function **freq** that takes a symbol A and a list L and counts the occurrence of symbol A found anywhere in L. Examples:

```
○ (freq  'c '((a c) c e))      --> 2
○ (freq  'f '(((s) o ) d))     --> 0
○ (freq  'f '(((f) f) f f))    --> 4
```

7. **[6]** Define the function **mapping** that takes 2 arguments - a list L, and an integer value val. Every element of the list L is a list of two atoms - key and object. (e.g. L <-- ((35 kim) (67 clinton) (45 emma))) The function returns a list of objects whose key is less than val. Examples:

   -
     ```
     (mapping '((35 kim) (67 clinton) (45 emma))  40)
     --> (kim)
     ```

   -
     ```
     (mapping '((24 a) (15 b) (56 c) (19 d)) 26)
     --> (a b d)
     ```

   -
     ```
     (mapping '((90 a) (80 b) (70 c))  40)
     --> nil
     ```

8. **[10]** Define a function **my-last** that takes two parameters, a symbol A and a list of symbols L, and returns the list starting where the last occurrence of symbol A is in L. It returns nil only if A is not in the list. Examples:

   -
     ```
     (my-last 'a '(a b c a b c a b c d e f g))
     -->    (a b c d e f g)
     ```

   -
     ```
     (my-last 'b '(a b c a b c a b c d e f g))
     -->    (b c d e f g)
     ```

   -
     ```
     (my-last 'c '(a b c a b c a b c d e f g))
     -->    (c d e f g)
     ```

   -
     ```
     (my-last 'g '(a b c a b c a b c d e f g))
     -->    (g)
     ```

   -
     ```
     (my-last 'h '(a b c a b c a b c d e f g))
     -->    nil
     ```

9. **[8]** Define the function **my-reverse** that takes a list L and returns the reverse of L. Examples:

   -
     ```
     (my-reverse nil)                  --> nil
     ```

   -
     ```
     (my-reverse '(a))                 --> (a)
     ```

   -
     ```
     (my-reverse '(1 2 3 4 5))         --> (5 4 3 2 1)
     ```

   -
     ```
     (my-reverse '((1 2 3) 4 ((5 6)))) --> (((5 6)) 4 (1 2 3))
     ```

10. **[8]** Define the function **is-pattern?** that takes two SIMPLE lists pat and str and returns the sublist of str which starts with the pat if pat is a substring of str. Otherwise, it returns nil. Examples:
    - `(is-pattern? '(a b s) '(c d b a s))`
      `--> nil`
    - `(is-pattern? '(c a c) '(b a j a c a c t u s))`
      `--> (c a c t u s)`
    - `(is-pattern? nil '(a n y l i s t))`
      `--> nil`
    - `(is-pattern? '(l i s p) nil)`
      `--> nil`

11. **[8]** Define the function **first-atom** that takes a list L and returns the first atom of L. Examples:
    - `(first-atom nil)                  --> nil`
    - `(first-atom '((2 (1) 4) 6))       --> 2`
    - `(first-atom '((((s)) o )))        --> s`
    - `(first-atom '(1 (((2)) 3 4)))     --> 1`

12. **[10]** Define a function **find-all** that takes a symbol A and a list L and finds and returns the first symbol following each occurrence of A in L, or nil if A does not occur in L. Note that A may occur nested within L, possibly as the last element of a sublist. You may assume that there is always a symbol occurring afterwards. Examples:
    - `(find-all 'a nil)                 --> nil`
    - `(find-all 'a '(b a c a e))        --> (c e)`
    - `(find-all 'a '(b d c e))          --> nil`
    - `(find-all 'a '(b (a a) c))        --> (a c)`
    - `(find-all 'a '((b a) ((c a b))))  --> (c b)`

Submit one file:

- **submit the file hw4.l that contains all your function definitions inside on Gradescope so we may run them through MOSS and the autograder. The Lisp compiler that I will be using to run these tests will be sbcl, so please take that into consideration while testing and submitting.**