Lisp Recursive Programming Exercises

1. Write a Lisp macro `mycase` that translates the following macro call. Assume the input will be error free. The input lists can be any length. You must document your solution.

   ```
   (mycase (C1 C2 ... Cn) (P1 P2 ... Pn))
   ```

   translates to the following

   ```
   (mycond (C1 P1) (C2 P2) ... (Cn Pn))
   ```

2. Write a recursive function, `(defun nth (pos list) ???),` that returns the n'th item from a list. Assume the list has at least n items. `(nth 1 aList)` is to return the first item in aList.

3. Write a recursive function, `(defun index (???) ???),` that returns a matrix element – A[I1,I2,...,In] – the element at I1 in the first dimension, I2 in the second dimension, etc. A call of the form `(index array I1 I2 ... In)` would be used. Assume caller will not have out of bound indices. There is no fixed size for the number of dimensions of the matrix. Use the function `nth` from Question 2. Assume index value 1 is the first item in the corresponding dimension. Do not use length, last, butlast, etc.., stick to first (or car) and rest (or cdr).

4. Write a macro function `our-if` that translates the following macro calls.

   `(our-if a then b)` translates into `(cond (a b))`
   `(our-if a then b else c)` translates into `(cond (a b) (t c))`

5. Write your own recursive version `myMaplist` of the maplist function. If possible, do not define additional functions but it is better to have them with a correct and commented function than have an incorrect function. Maplist, which can have any number of lists as arguments, terminates when one of the input lists becomes empty. A function with a fixed number of arguments is not acceptable.

6. How would multi-dimensional matrices be implemented in Lisp? Define the operation 'index' which has an array and an index list as parameters; the function is to return the indexed array element.

7. Write simple lisp functions such as the following. Take into account lists which are too short.
   - (remove-first '(a b c ...)) -> (b c ...) --- remove the first item from the list.
   - (remove-second '(a b c ...)) -> (a c ...) -- remove the second item from the list.
   - (insert-as-second 'b '(a c ...)) -> (a b c ...) --- insert as the second element.

10. Write a recursive function, `(defun nth (???) ???)`, that returns the n'th item from a list. Assume the list has at least n items. `(nth 1 aList)` is to return the first item in aList.

11. Write a recursive function, `(defun diagOf(theMatrix) ...)` to return the diagonal of a square matrix. Assume the input is error free. You may write support functions. Do not use global variables. Do not use let, prog and similar features to introduce local variables; use only parameters to functions as local variables.

12. Write a your own recursive version myMaplist of the `maplist` function. If possible, do not define additional functions but it is better to have them with a correct and commented function than have an incorrect function. Hints: Recall the functions some and every. Maplist terminates when one of the input lists becomes nil.

13. Write a recursive version of reverse

    ```
    (defun myrev (theList) ... )
    ```

using LAST and BUTLAST. Check it by reversing a list twice to see if it equals the original. Only the top level is reversed; so reversing (A B (C D)) produces ((C D) B A). Use only functions from Chapters 1 to 6 inclusive.

14. Write a recursive version of reverse using LAST and BUTLAST where every level of a list is reversed. For example, reversing (A B (C D)) produces ((D C) B A).

15. The function READ reads the next s-expression from the input and returns it. Experiment by typing an s-expression after entering `(setq x (read))` and then check the value of x. Experiment with `(setq x (cons (read) (read))` as well. Write the following function. to read a sequence of s-expressions as defined in the following. Assume correct input.

```
 (defun create-symbol-and-prop () ... )

Input ::= aLispSymbol aValue PropertyList ;
PropertyList ::=  ( "endp" , aPropName aPropValue PropertyList) ;
aLispSymbol ::= is a Lisp symbol
aValue ::= any s-expression
aPropName ::= is the name of a property
aPropValue ::= any s-expression
```

Example Input

```
vertex (3.0 4.0) colour black change (penny 3 dime 4 looney 6)
endp
```

The result of the function is to create the global variable "vertex" assign it the value "(3.0 4.0)" and give it the property "colour" with value "black", and the property "change" with the value "(penny 3 dime 4 looney 6)".

Use recursion to process the property list data. Except for READ do not use material beyond Chapter 7. Use PUTPROP as defined in exercise 1 chapter 7 (page 121). You must define your own function.

Verify your function using SYMBOL-PLIST and GET.

16. Do exercise 4 in Wilensky Chapter 8 (page 140). For the solution I'm looking do not write support functions. Instead use a LAMBDA form (Chapter 9). Hints: Consider the following function which uses the keyword "&rest" (only thing from Chapter 12) to gather a sequence of parameters into a single list.

```
(defun first-of-each (&rest sequence-of-lists)
 (mapcar 'car sequence-of-lists))

 (first-of-each '(1 2 3) '(a b c) '(d e f))
 (1 a d)
```

17. Do exercise 5 in Wilensky Chapter 12 (page 220). The only thing used from that chapter is the &rest keyword (see exercise 3 above). Otherwise all you need is material from Chapter 8 and earlier. Write a fully recursive version. Do not use functionals.

18. Construct a macro definition, foreach, to do the following.

Form 1 applies a function to each item in a list that satisfies a given predicate (test) and returns nil otherwise.

```
(foreach (item in list) (apply func) (when (test)))
  ==> (mapcar '(lambda (item)
                 (cond ((test item)
                        (funcall func item)))) list)
```

Form 2 like form1 except all nil values are removed.

```
(foreach (item in list) (save func) (when (test)))
     ==> (apply 'append
          (mapcar '(lambda (item)
                     (cond ((test item)
                            (list (funcall func item))))) list))
```

Example 1: assuming b1 = (11 20 33 40 55 60)

```
(foreach (number in b1) (apply '1+) (when (evenp)))
```

when executed gives `(nil 21 nil 41 nil 61)`

Example 2:

```
(foreach (number in b1) (save '1+) (when (evenp)))
```

when executed gives `(21 41 61)`

19. Do a variation of exercise 7 in Wilensky, Chapter 6 (page 110). Do only the recursive version. Make sure you "sub-splice" every occurrence of the second parameter.

20. Write a recursive function, `(defun myinter (list1 list2) ... )`, that computes the set intersection of list1 and list2. Use the member function.

21. Define your versions of the functions some (call it mysome) and every (call it myevery) in exercise 7 in Wilensky Chapter 8 (page 140-141).

22. Modify the fully recursive definition of sub-splice from above that accepts the keyword :everywhere (Chapter 12). If the argument is non-nil, then your function will do sub-splice everywhere in the input list. If the argument is nil, then your function will do sub-splice only at the top level of the list.

23. Write a macro that expands (select smallInt from aList) into (selector aList) where selector is one of the following. For all other values of smallInt return NIL.

smallInt	**selector**
1	first
2	second
3	third
4	fourth

24. The following program countRemove removes all instances of the item from the list. Complete the program so it returns as its first value the modified list and as its second value a count of the number of replacements.

```
(defun countRemove (item list)
(cond ((atom list) list)
      ((equal (first list) item) (countRemove item (rest list)))
        (t (cons (first list) (countRemove item (rest list))))))
```

25. Write **one** macro function `cfunc` that translates the following macro calls.

    `(cfunc fname (parm))` translates into `(function fname (parm))`
    `(cfunc fname (parm) int)` translates into `(int function fname (parm))`

26. Write a recursive function, `(defun flatten(theList) ...)` to return all the atoms in the theList as a single level list while retaining their order.
    For example `(A (B (C D) E) (F G))` becomes `(A B C D E F G).`
    Assume the input is error free. You may write support functions. Do not use global variables. Do not use let, prog and similar features to introduce local variables; use only parameters to functions as local variables.

27. Program insert sort and bubble sort (with two values the returned list and whether a swap was done).

28. Define a function to merge two sorted numeric lists.

29. Define a function to merge sort a numeric list

30. Define functions for the prefix, suffix and sublist of a list.
    - by index position: prefix first n, suffix last n, sublist lowerBound to upperBound inclusive
    - boolean to return true if list_1 is a prefix, suffix or sublist of list_2 (compare with Prolog)

31. Write a recursive function, `remove-nth` that removes the `n`’th element from every `list` at all levels. Counting begins at 1. You cannot use any implicitly recursive function, such as `mapcar`, `length`, etc. Use `car`, `cdr` and `cons` for the basic list operations and use `cond` for conditional expressions. You are permitted to and will need to write recursive support functions.

    Precondition: n ≥ 1.

    ```
    (defun remove-nth (n list)  ;; You supply the rest
    ```

32. Using the following two functions, that you do not have to implement,

    ```
    prefix(p, list) = list(1 .. min(p, #list))
    suffix(q, list) = list(max(1, q) .. #list)
    ```

    write a functional program `sublist-all(p, q, list-of-lists)` that returns a list of the sublists of each of the lists in `list-of-lists`. The definition is have no explicit recursion, including within any lambda functions. The definition of a sublist is the following.

    ```
    sublist(p, q, list) = list(max(1,p) .. min(q, #list))
    ```

    Example: `(sublist-all 2 3 ‘((1 2 3 4) (1 2) (1) ((1) (2) (3) (4))`
    □ `((2 3) (2) nil ((2) (3)))`

    ```
    (defun sublist-all (p, q, list-of-lists)  ;; You supply the rest
    ```