1. Lisp recursive programs

1. 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.
2. 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).
3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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).
11. 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.

12. 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)
```

13. 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.

14. 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.

15. Write a recursive function, `(defun intersection (set1 set2) ... )`, that computes the set intersection of set1 and set2. Use the member function – `(member item list)`, it returns the sublist beginning at the item, if item is in the list and returns nil otherwise.

16. 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).

17. 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.

18. 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))))))
```

19. Write a recursive Lisp function **flatten(alist)** to return all the atoms, except nil, at all levels in **alist** as a single level list while retaining their order. The following are examples.

```
(flatten '(A (B (C D) E) (F G))) ⇒ (A B C D E F G)
(flatten '(1 () 2)) ⇒ (1 2)
(flatten '(1 () (()) ( 2 () 3))) ⇒ (1 2 3)
```

Assume the input is error free. You may write support functions, although you get lower evaluation if you do. 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.

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

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

22. Define a function to merge sort a numeric list

23. 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)

24. Write a recursive function, `insert-nth` that inserts `item` as the `n`'th element into 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. You are permitted to and will need to write recursive support functions.

    Precondition: n ≥ 1 ^ n ≤ 1+length(shortest list at any level in `list`)

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

25. 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
    ```

26. 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., except for `append`, and the following functions `prefix` and `suffix`. You must use `prefix` and `suffix` in your definition.

    (prefix 3 (10 20 30 40 50)) → (10 20 30)
    (suffix 3 (10 20 30 40 50)) → (30 40 50) Next time define 3 as the position not length.

    Use `car`, `cdr` and `cons` for the basic list operations, and `cond` for conditional statements. You are permitted to and will need to write recursive support functions.

    Precondition: n ≥ 1

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

27. You are given the following two functions, which you do not have to implement, where # is the length function, and list member counting begins at 1.

    ```
    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 sub-lists of each of the lists in `list-of-lists`. The definition is to have no explicit recursion. The definition of a sublist is the following.

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

    Example: (sublist-all 2 4 '((a b c d e) (a b c) (a) ((a) (b) (c) (d)))
    ⇒ ((b c d) (b c) nil ((b) (c) (d)))

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

28. Write a function, FUNSEARCH, using only functionals (no recursion) to return the first item of every sublist in a list of lists where the sublist contains a particular member, otherwise return the sublist. You may use lambda expressions and Lisp's member function. The following is an example.

    ```
    theList = ((1 2 3 x) (4 5 x 6) (7 8 9) (10 11) (x 13 14 15))
    ```

```
(funsearch 'x theList) ==> (1 4 (7 8 9) (10 11) x)
```

29. Write a functional program, `compress(list1 list2)`, (no explicit recursion) that uses a lambda function to produces the sum of the pair-wise subtraction of the smaller numbers from larger numbers.

    Example `(compress '(10 20 30 40) '(5 21 33 39))` → 10

30. In Lisp write a functional program, `replace`, (no recursion) that uses a lambda function to replace with nil every item at the top level of a list that is a list.

```
(replace '(1 () 2 nil 3 (a b) 4 (a (b) c) 5))
     ⇒ (1 nil 2 nil 3 nil 4 nil 5)
```

31. Write a recursive program, `deeper-atoms`, that embeds each atom, except nil, in a list containing only the atom. Use only the basic Lisp operators: car, cdr, cons, cond, null and atom.

    Examples: `(a)` → `((a))` `(())` → `(())` `(a () b)` → `((a) () (b))`
    `(a b (c (b d)) (e))` → `((a) (b) ((c) ((b) (d))) ((e)))`

31. Write a recursive program, `lift-atoms`, that lifts each atom, except nil, that is in a list containing only the atom up one level. This is the inverse of part B. Use only the basic Lisp operators: car, cdr, cons, cond, null and atom.

    Examples: `((a))` → `(a)` `(())` → `(())` `((a) () (b))` → `(a () b)`
    `((a) (b) ((c) ((b) (d))) ((e)))` → `(a b (c (b d)) (e))`

32. Write a recursive Lisp definition for the mapcar function that can have an arbitrary number of input lists. The mapcar functions stops as soon as any one of the input lists becomes empty. 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. You may use support functions.

33. Write a recursive function, `removeContig(theList)` that reduces all instances of a contiguous sequence of identical items to one instance at all levels of `theList`. You cannot use any implicitly recursive function, such as `mapcar`, `length`, etc. Use `cond, car`, `cdr` and `cons` for the basic list operations. You are permitted to write recursive support functions.

    Precondition: `theList` is a list

```
(defun removeContig (theList)  ;; You supply the rest
```

34. Write a recursive Lisp function **nodups**, which takes one argument, a list (alist), and returns a list with any consecutive identical top-level items removed.

    For example:

    (nodups '(a a a b c c d)) should return (a b c d)

    (nodups '(a a b a c c)) should return (a b a c)

    Note: for this question, you may use only the Lisp functions defun, cond, null, car, cdr, cons, and equal. You may use combinations of car and cdr such as caddr.

35. Write a recursive Lisp function **dups,** which takes two arguments, a list (alist) and an integer (N), and returns a list with every top-level item in alist duplicated N times. Assume N ≥ 1.

    For example:

    (dups '(a b c) 3) should return (a a a b b b c c c)

    (dups '(a ( ) (a)) 2) should return (a a nil nil (a) (a))

    Note: for this question, you may use only the Lisp functions defun, cond, null, car, cdr, cons, append, list, +, - and =. You may use combinations of car and cdr such as caddr.

36. Define a recursive Lisp function, `dup`, which checks whether its argument is a list containing two successive elements at the top level that are equal.

```
(dup '(A B B C) ⇒ t
(dup '(A (B) B C) ⇒ nil
```

2. Lisp macros

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 Lisp macro `mycase` that translates the following macro call as shown. Assume the input will be error free. The input lists can be any length. Use standard Lisp functionals. If you need support functions, your answer should have only non-recursive support functions.

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

   translates to the following

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

2. 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))`

3. Complete the macro definition, without using backquote, of `our-if` that translates the following macro call

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

   ```
   (defmacro our-if   ;; complete the parameters and body
   ```

4. 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

5. 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))`

   Write a macro `select` that returns a form, which if executed, returns the sublist of items from a list such that, if you apply a given function to a selected item, the result bears the correct relationship to a given value.

6. The following are examples showing the result of executing the form returned by the macro.

   (select item from '(10 20 25 15 30 12 23 5) if 1+ (item) > 20) ➔ (20 25 30 23)
   (select item from '(10 20 25 15 30 12 23 5) if 1+ (item) > 21) ➔ (25 30 23)
   (select item from '(10 20 25 15 30 12 23 5) if 1- (item) < 20) ➔ (10 20 15 12 5)
   (select item from '(10 20 25 15 30 12 23 5) if 1- (item) < 19) ➔ (10 15 12 5)

   Use the back quote style to write your macro. Do not explicitly use recursion. Do not use support functions, use lambda instead. Note that there are no nils in the result.

7. Define the following Lisp macro. Use lambda instead of support functions. If possible avoid explicit recursion. define without using backquote and define using backquote.

```
(select item from (v-1 e-1) … (v-p e-p)) →
    (cond ((equal item v-1) e-1) ... ((equal item v-p) e-p))
```

8. Write a Lisp macro `rearrange` that translates the following macro call as shown. Assume the input will be error free. The input lists can be any length. There is a functional solution. Do not use explicit recursion. If you need support functions, your answer should have only non-recursive support functions. You are required to have comments clearly explaining, **in detail**, how your macro produces the translation.

```
(rearrange (A1 A2 … An) (B1 B2 … Bn) … (Last1 Last2 … Lastn) )
```

translates to the following

```
(A1 B1 … Last1 A2 B2 … Last2 … An Bn … Lastn)
```

9. Complete the macro definition of `my-if` that translates the following macro calls.

```
(my-if a then (p1 p2 ...) (s1 s2 ...))
      translates into  (cond (a (cond (p1 s1) (p2 s2) ... )))

(my-if a then (p1 p2 ...) (s1 s2 ...) else c)
      translates into  (cond (a (cond (p1 s1) (p2 s2) ... )) (t c))
```

(defmacro my-if ;; complete the parameters and body

3. Functional Programs

1. Using the following function `isMember`, write a functional program `intersection` to compute the intersection of two sets.

```
(defun isMember (anItem theList)
(cond ((member anItem theList) (list anItem))
      (t nil)))


(defun intersection (set_A set_B) ;; complete the definition …
```

2. Give the above definition of intersection using a lambda expression to replace the call to isMember.
3. Write a functional program, `checktype1`, that given an input list, returns a corresponding list of T or nil depending upon whether or not an item at the top level of the list is an atom (t) or a list (nil). The empty list is to be considered as a list y the program. Do not use a support function. Use a lamda expressions. You may not use listp

   Example `(checktype1 '(a b () (c) d) → (t t nil nil t)`

4. Write a functional program, `checktype2`, that given an input list which is a list of lists, returns a list of lists where each sublist is a corresponding sublist of t or nil depending upon whether or not the second level items are an atom (t) or list (nil).). The empty list is to be considered a list the program. You may not use listp but you may use a support function to get the simplest program.

   Example `(checktype2 '((a) (b () c) (() (d) ((()))) (e))`

   `→ ( ((t) (t t) (nil nil) nil) (t))`

5. Define a functional function `sigma2` with no explicit recursion that generalizes `sigma` to two integer indices. It should be able to compute the following expression. Note that the function `term` has two parameters i and j. You cannot use `sigma` because `sigma` accepts a `term` with only one argument. You cannot use any explicitly recursive functions. Assume + is a binary operator. You may assume

the following functionals are available – allpairs, bu, curry, comp, compl, distl, distr, filter, genlist, mapcar, maplist, range, reduce, rev, trans.

$$sigma2(term,k,l,m,n) = \sum_{i=k}^{l} \sum_{j=m}^{n} term(i,j)$$

6. **A** Define the following function to generate the first n terms of the sine expansion. where x is a real number. Use lambda-functions where necessary. Do not define support functions. You are given the `factorial`, !, function. Use `(genlist length next start)` to get the numerators. Use `(range from to)` to get the denominators. Then combine the two lists.

   $$\text{sine}(x, n) = x - x^3/3! + x^5/5! - x^7/7! + \ldots + x^{2n-1}/(2n-1)!$$

   ```
   (defun sine-terms(x n) ;;  You supply the rest
   ```

   B Your genlist expression in part A should use a lambda function. Define a support function equivalent to your lambda function. Rewrite your genlist expression to use the support function instead of the lambda function.

7. Define a functional program, with no explicit recursion, that produces the list of the sum of the integers i..max, for i = 1 .. max. Do not use lambda-functions.

   ```
   (fun-sum 3) → (6 5 3) ≡   ((+ 1 2 3) (+ 2 3) (+ 3))
   (fun-sum 4) → (10 9 7 4) ≡   ((+ 1 2 3 4) (+ 2 3 4) (+ 3 4) (+ 4))

   (defun fun-sum (max)  ;;  You supply the rest
   ```

8. A palindrome is a list that reads the same backward and forward. For instance (a b c b a) is a palindrome, but (a b c a a) is not. Define, with no explicit recursion, the following function that returns t if the input list is a palindrome; otherwise it returns nil.

   ```
   (defun is_palindrome (the_list) ;; you supply the rest
   ```

9. An arithmetic progression is a sequence of the form r, r + s, r + 2*s, … , r + n*s, for some integer r and some positive integer s. For example (2, 5, 8, 11) is an arithmetic progression with r = 2, s = 3 and n = 3. By definition, if the length of the sequence is less than three, its elements form an arithmetic progression.

   Define, with no explicit recursion, the following function that returns t if the sequence of integers is an arithmetic progression; otherwise it returns nil Use cond for conditional statements. You may define support functions using functionals with no explicit recursion. You may find the functions (butlast 10 20 30 40 50) → (10 20 30 40), and length useful.

   ```
   (defun is_arithmetic_progression (int_seq)   ;; You supply the rest
   ```

10. Define a functional program, with no explicit recursion, that produces the list of the partial factorial values `max!/i!`, for i = 0 .. max-1. Do not use lambda-functions. You may assume factorial is available.

    ```
    (part-factorial 3) → (6 6 3)
    (part-factorial 4) → (24 24 12 4)

    (defun part-factorial (max)  ;;  You supply the rest
    ```

11. Write a recursive function **interleave(list-1 list-2)** that returns a list in which the elements of list-1 are interleaved with list-2. The following are examples.

    ```
    (interleave '(a (c d) e) '(1 2 3 4 5)) → (a 1 (c d) 2 e 3 4 5)
    (interleave '(1 2 3 4 5) '(a (c d) e)) → (1 a 2 (c d) 3 e 4 5)
    ```

12. Consider the following Lisp function.

    ```
    (defun mystery (s)
     (cond ((null s) 1)
           ((atom s) 0)
    ```

```
            (t (max (+ mystery (first s)) 1)
                    (mystery (rest s))))
    ))
```

Explain in general what the function mystery returns.

4. Miscellaneous Expressions

1. Assume the following forms have been typed into the Lisp interpreter and evaluated.

   (defun a (x) (values (list x) x))

   (setq a '(a b))

   (defun b (x) `(x ,x))

   (setq b (cdr a))

   (setq c (car a))

   (setq d c)

   (setq e ((lambda (x) (list x)) d))

2. What will the following forms evaluate to?
   1. (cons c (car a))
   2. (cons e b)
   3. (eval a)
   4. (let ((a b) (y a)) (append a y))
   5. (multiple-value-list (a a))
   6. (b c)
   7. (set (car a) (cdr a))
   8. (setf (car a) (cdr a))

3. The following forms are entered into the Lisp interpreter and evaluated.

   (defun f1 (v1) (f2 v1))
   (defun f2 (v2) (1+ v1))

   Under an environment with static scoping what do the following forms evaluate to?

   1. (f1 6)
   2. (setq v1 10)
      (f1 6)
   3. For 2 above what does the environment look like in f2?

   Under an environment with dynamic scoping what do the following forms evaluate to?

   1. (f1 6)
   2. (setq v1 10)
      (f1 6)
   3. For 2 above what does the environment look like in f2?