- To become familiar with the basic features of Lisp syntax and semantics.
- To become acquainted with the techniques of pure functional programming.
- To learn to think recursively.

NOTE:***Properly*** acknowledge (add a note and/or hyperlink and/or comment) any help or resource you used.

1) Write a set of LISP functions to implement the following operations on polynomials:

- **`Poly-add`** – to add two polynomials
- **`Poly-sub`** – to subtract two polynomials
- **`Poly-mul`** – to multiply two polynomials
- **`Poly-der`** – to compute the derivative of a polynomial
- **`Poly-val`** – to compute the value of a polynomial for a given argument
- **`Poly-prim`** – to compute the primitive(antiderivative) of a polynomial

A polynomial will be represen or each term of the polynomial.
Each sublist will have the exp fficient as the second component.
Ex: $3X^5+3X^3+7$ is represented

2)Write the LISP function:

`(deep-subst x y L)`

that returns a list *L1* formed by replacing, within the recursive-list *L*, each occurrence of the atom *x* (at any level) with the atom *y*. (You may assume that *L* is a recursive list, and that *x* and *y* are each atoms.)

```
(deep-subst 4 t '(truth is 4)) → (TRUTH IS T)
(deep-subst 4 nil '(4 is not (4))) → (NIL IS not (NIL))
(deep-subst 4 nil '(5 is not (22))) → (5 is not (22))
```

3) Write the LISP function

`(count-sub-list-occ L1 L2)`

That returns the number of occurrences of the list L1 as a sublist of L2. L1 is a sublist of L2 if all the elements of L2 can be found in sequence in L1.

For example: (1 2) is a sublistof (3 1 2)
(1 3) is not a sublist of (1 2 3)
(1 3) is not a sublist of (3 2 1)
(1 1) is a sublist of (2 3 2 3 1 1 3 2)

```
(count-sub-list-occ '(1 2) '(1 2 3))→1
(count-sub-list-occ '(1 2) '(3 2 3))→0
(count-sub-list-occ '(1 2) '(1 2 1 2 3))→2
(count-sub-list-occ '(1 1) '(1 1 1 1))→3
```

4) The following procedure illustrates a possible way of generating password for different servers:

1. Start with an easy to remember sentence. Get a letter code by taking the first letter of each word in the sentence. If the sentence contains numbers, then the entire number is selected in the code.
   - For example:
     - The sentence "How I wish how I wish you were here" generates the code HIWHIWYWH
     - The sentence "There were 12 of us this morning I am the only one this evening", generates the code TW12OUTMIATOOTE.
2. Add the first letter of the server name in front of the code and the last letter of the server name after the code to generate an intermediate password
   - For example:
     - If the server name is "My new cool business" and the code is HIWHIWYWH, the intermediate password is **M**HIWHIWYWH**S**
     - If the server name is "MacEwan" and the code is TW12OUTMIATOOTE the intermediate password is **M**TW12OUTMIATOOTE**N**
3. Consider a replacement list which specifies pairs of symbols, where each occurrence of the first symbolis to be replaced by the second symbol. Get the final password by replacing the symbols of the intermediate password according to the replacement list.
   - For example:
     - If the intermediate password is MHIWHIWYWHS and the replacement list is ((S 5) (I 1)) the final password is: MH**1**WH**1**WYWH**5**
     - If the intermediate password is TW12OUTMIATOOTE and the replacement list is ((T 7) (M Q) (O 0)) the final password is: **7**W12**0**U**7Q**IA**7007**E

Write the LISP function:

`(gen-passsn-list st-list subst-list)`

That returns a password forme **`ist`** , sentence list **`st-list`** and replacement list **`subst-`** which contains the letters of each word as lists.

For example:

The site-name "My new cool business" is represented as

`` `((m y)  (n e w) (c o o l) (b u s i n e s ``

The sentence "How I wish how I wish you were here" is repre

`` `((h o w) (i) (w i s h) (h o w) (i) (w i s h) (y o u) (w e r e) (h e r e)). ``The replacement list is a list of pairs – where the first element specifies the symbol to be replaced and the second one specifies the symbol to replace it by.

```
(gen-pass
 `((m y)  (n e w) (c o o l) (b u s i n e s s))
 `((h o w) (i) (w i s h)  (h o w)  (i) (w i s h) (y o u) (w e r e) (h e r e))
 `( (O 0) (I 1))) → '(M  H 1 W H 1 W Y W H S)
```

```
(gen-pass
 `((m a c e w a n))
`((
There)(were)(12)(of)(us)(this)(morning)(i)(am)(the)(only)(one)(this)(evening)
),
 `( (W 3) (T7))) → '(M 7 3 1 2 O U 7 M I A 7 O O 7 E N)
```

Implementation is subject to the constraint that it should illustrate pure functional programming; that is, it should not use `set`, `setf`, `setq`, or any other binding primitive. You may use `set`, etc., for the purposes of testing your functions, but not in any of the functions that form part of your solution to the problem. You may, of course, define any other subsidiary functions that you might find useful. The list of LISP built-in functions and forms that you can use is attached to this assignment..

Correctness - The programs should conform to the specifications for which it was written. It should include correct handling of special cases, error conditions, etc., except that you may assume that input will be provided in the specified format. Your file must load without error in the clisp evaluator installed on ugrad.cs.macewan.ca. There is a 50% penalty if errors are detected at loading time.

Design and Efficiency - The program should be constructed from small, coherent, independent and loosely coupled functions. Each function should access only its own parameters. The control constructs and data structures used should be those appropriate to the problem at hand. The program should not perform unnecessary steps, use extraneous variables, nor implement the algorithm in a contorted or inefficient way. Your functions must be documented – through comments and properly indented.

Marking scheme:

Item	**Mark**
1	35
2	10
3	10
4	25
Design and efficiency	**20**
TOTAL	**100**