COMP 348 Assignment 2 LISP

Name: Hualin Bai ID:40053833

Question 1:

```
(defun triangle (n)
     (if(integerp n)
        (loop for i from 1 to n
            do(loop for j from 1 to i
                  do(write j)
              )
            (write-line "")
        )
        (if (floatp n)
           (print "decimal numbers are not valid input, please enter an integer")
            (if(stringp n)
             (print "strings are not valid input, please enter an integer")
             (print "not valid input, please enter an integer!")
            )
        )
      )

)

```

```
(triangle 4)
(triangle 5)
(triangle 2.5)
(triangle "A")
(triangle 1/3)
```

```
1
12
123
1234
1
12
123
1234
12345

"decimal numbers are not valid input, please enter an integer"
"strings are not valid input, please enter an integer"
"not valid input, please enter an integer!"
```

Question 2:

(1) Using anonymous function (lambda)

```
;using anonymous function (lambda)
(defun getdistance(x1 y1 x2 y2)
    (print(
    (lambda (x1 y1 x2 y2) (sqrt(+(*(- x1 x2)(- x1 x2))(*(- y1 y2)(- y1 y2))))) x1 y1 x2 y2))
)


(getdistance 2 1 2 3)
(getdistance 2 1 2 4)
(print((lambda (x1 y1 x2 y2) (sqrt(+(*(- x1 x2)(- x1 x2))(*(- y1 y2)(- y1 y2))))) 2 1 2 3))
```

```
$clisp main.lisp

2
3
2
```

(2) Without applying the anonymous function.

```
;without applying the anonymous function

(defun getdistance2 (x1 y1 x2 y2)

    (let( (x (- x1 x2))  (y (- y1 y2))  )

        (let( (x (* x x)) (y (* y y)) )

            (print (sqrt(+ x y)))

        )
    )
)

(getdistance2 2 1 2 3)

(getdistance2 5 1 2 3)
```

```
$clisp main.lisp

2
3.6055512
```

Which method is more efficient in terms of memory allocation?

Actually, using anonymous function (lambda) is more efficient, since an anonymous function only has one line of code, which simply apply the anonymous function to the input elements. Unlike functions defined with defun, anonymous functions are not stored in memory. Contrarily, getdistance2() use the let syntatic which need more memory allocation to create some new variables.

Question 3:

Method 1:

```
(defun findchar(lst)

      (cond
        ((null lst) nil)

        ( (and (atom (car lst))  (symbolp (car lst)) )

          (cons (car lst) (findchar(cdr lst)) )

        )
        ((listp (car lst)) (remove-duplicates(append (findchar(car lst)) (findchar(cdr lst )) )) )

        (t (findchar(cdr lst)) )

      )
)


(print(findchar '((z f) (b a 5 3.5) 6 (7) (a) c ) ))

(print(findchar '( (n) 2 (6 h 7.8) (w f) (n) (c) n) ))

```

```
$clisp main.lisp

(Z F B A C)
(H W F C N)
```

Method 2:

```
(defun findchar(lst)
   (setq lst2
        (cond
            ((null lst) nil)
            ( (and (atom (car lst))  (symbolp (car lst)) )
                (cons (car lst)  (findchar(cdr lst)) )
            )

        ((listp (car lst)) (append (findchar(car lst)) (findchar(cdr lst ))))
        (t (findchar(cdr lst)) )
        )
    )

  ;use member function
   (setq lst3 '())
   (dolist (a lst2)
      (if(member a (cdr(member a lst2)))
         (setq lst2 (cdr(member a lst2)))
         (setq lst3 (append lst3 (list a)))
      )

   )
(return-from findchar lst3)
)

(print(findchar '((z f) (b a 5 3.5) 6 (7) (a) c ) ))
(print(findchar '( (n) 2 (6 h 7.8) (w f) (n) (c) n) ))

```

```
$clisp main.lisp

(Z F B A C)
(H W F C N)
```

Question 4:

```lisp
;Write a Lisp function f-l-swap that swaps the first and last elements of a list.
(defun f-l-swap(lst)
    ;set variables for exchange elements
    (setq lst2 lst)
    (setq f_elt (car lst2))
    ;remove the first element from list2
    (if(null (cdr lst2))
        (setq lst2 nil)
        (setq lst2 (cdr lst2))
    )
    ;get the last element of old list
    ;consider list with 1 and 2 elements situations
    (if (or(null lst)(null (cdr lst)))
        lst
        (loop
            (when (null (cdr lst)) (return))
            ;return the list with only last element
            (setq lst (cdr lst))
        )
    )
   ;swap the last and first elements
   ; reconstruct lst
   (if(or(null lst2)(null (cdr lst2)))
      lst
      ;add middle elemets of list
      (loop
         (setq lst(append lst (list (car lst2))))
         (setq lst2 (cdr lst2))
         (when (null (cdr lst2)) (return))
      )
    )
    ;add the first element
    ;consider special cases, such as nil,(f),(f g)
    (if(or(null f_elt)(null lst2))
        lst
        (setq lst (append lst (list f_elt)))
    )
)

(print(f-l-swap '((a d) f 10 w h)) )
(print(f-l-swap '(g 6 p 10 m)) )
;some special cases
(print(f-l-swap '(f g h 5)) )
(print(f-l-swap '(f g h)) )
(print(f-l-swap '(f g)) )
(print(f-l-swap '(f)) )
(print(f-l-swap '()) )
```

```
$clisp main.lisp

(H F 10 W (A D))
(M 6 P 10 G)
(5 G H F)
(H G F)
(G F)
(F)
NIL
```

Question 5:

```lisp
;is_bst: check if the given binary tree is a binary search tree.
;use inorder nodes traversal the given binary tree to store in a list,
;then compare if the elements of list is ascending sort, which is a binary tree.
(defun is-bst(new_tree)
       ;store inorder-traverse(tree) in a list, then check if it is sorted by ascending.
   (let* ( (lst-tree (inorder-traverse new_tree))
           (isBST (apply #'< lst-tree)) )
          ;show the list of the binary tree
          (write "show the list of inorder traversal, and compare whether it is ascending sorted.")
                 (print lst-tree)
         ;compare the boolean value of isBST
         (if(null isBST)
            (print "It is not a binary search tree!")
            (print "It is a binary search tree!")
         )
          ;return boolean value of isBST
          isBST
    )
);defun

(defun inorder-traverse(old_tree)
       (cond
          ;base case: just a node, no left and right childs of this node.
           ((and(null(car(cdr old_tree)))
                (null(car(cdr(cdr old_tree)))))
                ;add node to the list
                (list (car old_tree))
            )
           ;recursive call: have a right child, no left child
           ((null(car(cdr old_tree)))
             ;since right > node, then put it after node.
             ;such as (node, right)
             (append (list(car old_tree)) (inorder-traverse(car(cdr(cdr old_tree)))) )
           )
          ;recursive call: have a left child, no right child
          ((null (car(cdr(cdr old_tree))))
          ;since left < node, then put it before node.
          ;such as (left, node)
             (append (inorder-traverse(car(cdr old_tree))) (list(car old_tree)) )
          )
          ;recursive call: have both left and right child
          ;put right child after node, while put left child before node.
          ;such as (left, node, right)
          (t (append (inorder-traverse(car(cdr old_tree))) (list (car old_tree))
                                                  (inorder-traverse(car(cdr(cdr old_tree)))) )))
);inorder


(is-bst '(8 (3 (1 () () ) (6 (4 () ()) (7 () ()) ) (10 (()) (14 (13) () )))) )
```

```
$clisp main.lisp
"show the list of inorder travers
(1 3 4 6 7 8)
"It is a binary search tree!"
```

Result

```
$clisp main.lisp
"show the list of inorder traversal, and compare whether it is ascending sorted"
(1 3 4 6 7 8)
"It is a binary search tree!"
```

Question 5 : (method 2 use lables)

```
;is_bst: check if the given binary tree is a binary search tree.
;use inorder nodes traversal the given binary tree to store in a list,
;then compare if the elements of list is ascending sort, which is a binary tree.
(defun is-bst(new_tree)
    (labels ( (inorder-traverse(old_tree)
                    (cond
                        ;base case: just a node, no left and right childs of this node.
                        ((and(null(car(cdr old_tree)))
                          (null(car(cdr(cdr old_tree)))))
                         ;add node to the list
                          (list (car old_tree))
                        )
                        ;recursive call: have a right child, no left child
                        ((null(car(cdr old_tree)))
                         ;since right > node, then put it after node.
                         ;such as (node, right)
                         (append (list(car old_tree)) (inorder-traverse(car(cdr(cdr old_tree)))) )
                        )
                        ;recursive call: have a left child, no right child
                        ((null (car(cdr(cdr old_tree))))
                          ;since left < node, then put it before node.
                          ;such as (left, node)
                          (append (inorder-traverse(car(cdr old_tree))) (list(car old_tree)) )
                        )
                        ;recursive call: have both left and right child
                        ;put right child after node, while put left child before node.
                        ;such as (left, node, right)
                        (t (append (inorder-traverse(car(cdr old_tree))) (list (car old_tree))
                                   (inorder-traverse(car(cdr(cdr old_tree)))) )) ))
            )
        ;store inorder-traverse(tree) in a list, then check if it is sorted by ascending.
        (let* ( (lst-tree (inorder-traverse new_tree))
                (isBST (apply #'< lst-tree)) )
            ;show the list of the binary tree
            (write "show the list of inorder traversal, and compare whether it is ascending sorted.")
            (print lst-tree)

            ;compare the boolean value of isBST
            (if(null isBST)
               (print "It is not a binary search tree!")
               (print "It is a binary search tree!")
            )
            ;return boolean value of isBST
            isBST
        )
    );labels
);defun

(is-bst '(8 (3 (1 () () ) (6 (4 () ()) (7 () ()) ) (10 (()) (14 (13) () )))) )
```

Question 6:

```
(defun sin-cos-comp(x n)
    (cond
        ((or (stringp x) (stringp n)) (print "String is not correct value!"))
        ((floatp n) (print "Decimal is not correct value!"))
        ((< n 0) (print "n can't be negative!"))
        ((and (oddp n)(or(>= x 10)(<= x -10))) (print " x is not in range (-10,10) when n is odd. "))
        ;calculate sin(x)
        ((and(< x 10)(> -10)(oddp n))
            (if(= n 1)(print x)
               (let((sum 0)(index 0))
               ;such as n is 5, i is[1,3,5] with step 2, index is [0,1,2]
               ;if index is even, then plus+, else minus-
                   (loop for i from 1 to (+ n 1)

                         do(if(evenp index)
                              (setq  sum (+ sum (/ (cal-power x i) (factorial i))))
                              (setq  sum (- sum (/ (cal-power x i) (factorial i))))
                         )
                         (setq i (+ i 1))
                         (setq index (+ index 1))

                   )
                   (print sum)
               )
            )
        )
        ;calculate cos(x)
        ((and(numberp x)(evenp n))
            ;n is 0, cos(x) is 1.
            (if(= n 0)(print 1)
               (let((sum 0)(index 0))
               ;such as n is 4, i is[0,2,4] with step 2,index is [0,1,2]
               ;if index is even, then plus+, else minus-
                   (loop for i from 0 to (+ n 1)
                         do(if(evenp index)
                              (setq  sum (+ sum (/ (cal-power x i) (factorial i))))
                              (setq  sum (- sum (/ (cal-power x i) (factorial i))))
                         )
                         (setq i (+ i 1))
                         (setq index (+ index 1))
                   )
                   (print sum)
               )
            )
        )

    )
)
```

```
49
50  (defun factorial(num)
51      (if (zerop num)1
52          (* num (factorial (- num 1)))
53      )
54  )
55
56  (defun cal-power(num1 exp)
57     (if(zerop exp)1
58        (* num1 (cal-power num1 (- exp 1))))
59  )
60
61  ;test cases
62  (sin-cos-comp 2 "10")
63  (sin-cos-comp 2 2.5)
64  (sin-cos-comp 2 -1)
65  (sin-cos-comp 10 1)
66  (sin-cos-comp 3 1)
67  (sin-cos-comp 3 3)
68  (sin-cos-comp 3 5)
69  (sin-cos-comp 3 0)
70  (sin-cos-comp 3 6)
71
```

Result

```
$clisp main.lisp

"String is not correct value!"
"Decimal is not correct value!"
"n can't be negative!"
" x is not in range (-10,10) when n is odd. "
3
-3/2
21/40
1
-91/80
```

Question 7:

a) An iterative approach

```
;pellnumbers for iterative
(defun pellnumbers(n)
    (let((lst nil))
        (loop for i from 0 to n
            do(setq lst (append lst (list(pell-iterative i)) )))
        (print lst)
    )
)
;an iterative approach
(defun pell-iterative(n)
    (let((x 0)(y 1)(sum 0))
        (if(or(zerop n)(= n 1))
            (setq sum n)
            (loop for j from 2 to n
                do(setq sum (+ (* 2 (+ y)) x))
                (setq x y)
                (setq y sum)
            )
        )
        sum
    )
)
```

b) A recursive approach

```
;an recursive approach
(defun pell-recursive(n)
    (cond
        ((or(zerop n)(= n 1)) n)
        (t (+ (* 2 (pell-recursive (- n 1))) (pell-recursive (- n 2))) )
    )
)
;pellnumbers for recursive
(defun pellnumbers2(n)
    (let((lst nil))
        (loop for i from 0 to n
            do(setq lst (append lst (list(pell-recursive i)) )))
        (print lst)
    )
)
```

Outputs of test cases for both cases:

```
43  ;test cases
44  (pellnumbers 6)
45  (pellnumbers 12)
46  (pellnumbers 20)
47  (pellnumbers 50)
48
49  (pellnumbers2 6)
50  (pellnumbers2 12)
51  (pellnumbers2 20)
```

Result

```
$clisp main.lisp

(0 1 2 5 12 29 70)
(0 1 2 5 12 29 70 169 408 985 2378 5741 13860)
(0 1 2 5 12 29 70 169 408 985 2378 5741 13860 33461 80782 195025 470832 1136689
 2744210 6625109 15994428)
(0 1 2 5 12 29 70 169 408 985 2378 5741 13860 33461 80782 195025 470832 1136689
 2744210 6625109 15994428 38613965 93222358 225058681 543339720 1311738121
 3166815962 7645370045 18457556052 44560482149 107578520350 259717522849
 627013566048 1513744654945 3654502875938 8822750406821 21300003689580
 51422757785981 124145519261542 299713796309065 723573111879672
 1746860020068409 4217293152016490 10181446324101389 24580185800219268
 59341817924539925 143263821649299118 345869461223138161 835002744095575440
 2015874949414289041 4866752642924153522)
(0 1 2 5 12 29 70)
(0 1 2 5 12 29 70 169 408 985 2378 5741 13860)
(0 1 2 5 12 29 70 169 408 985 2378 5741 13860 33461 80782 195025 470832 1136689
 2744210 6625109 15994428)
```

Recursive approach (pellnumbers2 50) can't get result, since the recursion slows down the execution of the program, I think that the time complexity is O(2^n).
However, the iterative has O(n) time complexity.