QUIZ NUMBER 13 (TAKE-HOME)

CLISP PROGRAM

Submitted By-

Nahida Sultana Chowdhury

Student ID: 2000189256

Date:11/21/2017

Problem 1: Compute the length of a given list. Your program should handle lists of various sizes.

- **Code:**

```
(defun lengthOfList(List1)
    (cond
        ((not (listp List1)) (print "Not a list"))
        ((null List1)0)
        (t (+ 1(lengthOfList(cdr List1))))
    )
)
```

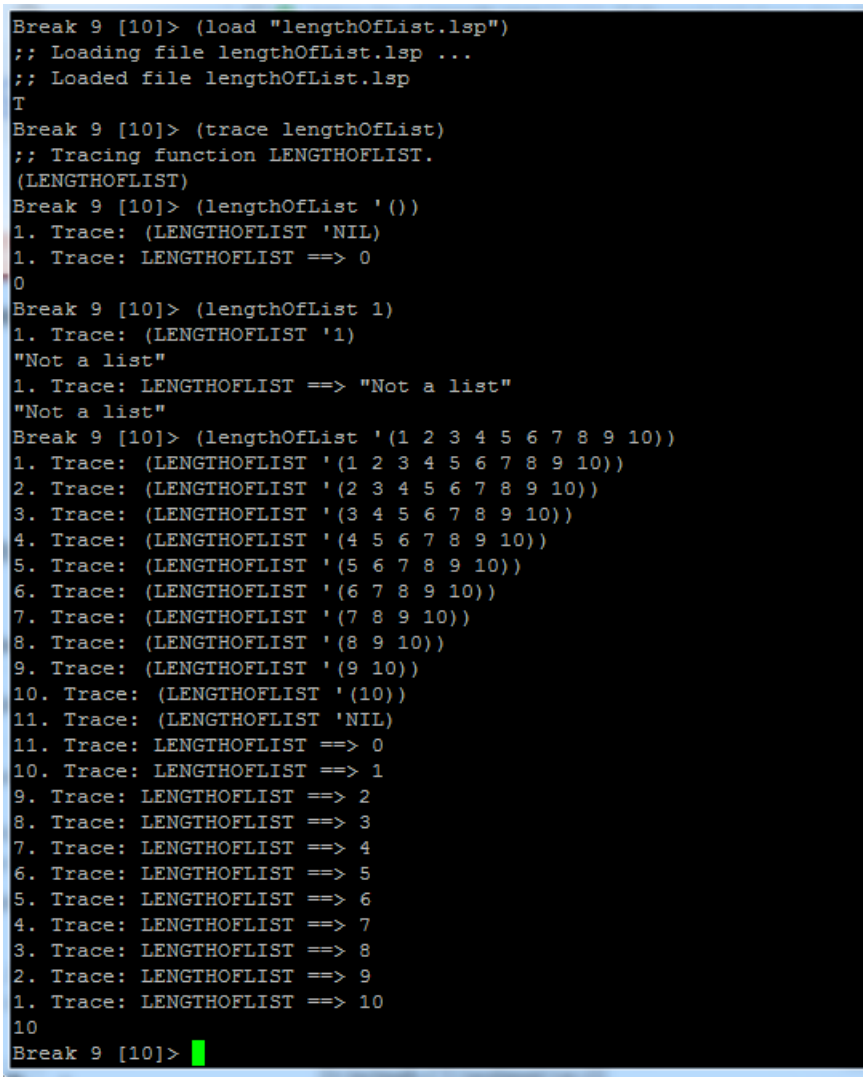
- **Screenshot (including program trace information):**

In the output screenshot three cases has been displayed:

1. When the list contains zero elements then it will return 0

2. When the input is an atom then it will display a message contains “Not a list”, to check imposed *listp* method.

3. When a list contains one or more items inside a list then it will return the total number of elements.



Problem 2: Compute the second to last element of a given list. Your program should handle lists of various sizes.

- **Code:**

```
(defun lenOfList(List1)
    (cond
        ((not (listp List1)) (print "Not a list"))
        ((null List1)0)
        (t (+ 1(lenOfList(cdr List1))))
    )
)

(defun lastElement(List1)
    (cond
        ((null List1)nil)
        ((and (= 1 (lenOfList List1))(not(listp(car List1))))(car List1))
        ((= 1 (lenOfList List1)) (lastElement(car List1)))
        (t(lastElement(cdr List1)))
    )
)

(defun secondLastElement(List1)
    (cond
        ((not (listp List1)) (princ "Not a list"))
        ((null List1)nil)
        ((= 1 (lenOfList List1))(princ "Single element"))
        ((and (= 2 (lenOfList List1))(not(listp(car List1))))(car List1))
        ((= 2 (lenOfList List1)) (lastElement(car List1)))
        (t(secondLastElement(cdr List1)))
    )
)
```

- **Screenshot (including program trace information):**

In the output screenshot five cases has been displayed:

1. When the input is an atom then it will display a message contains “Not a list”, to check imposed *listp* method.

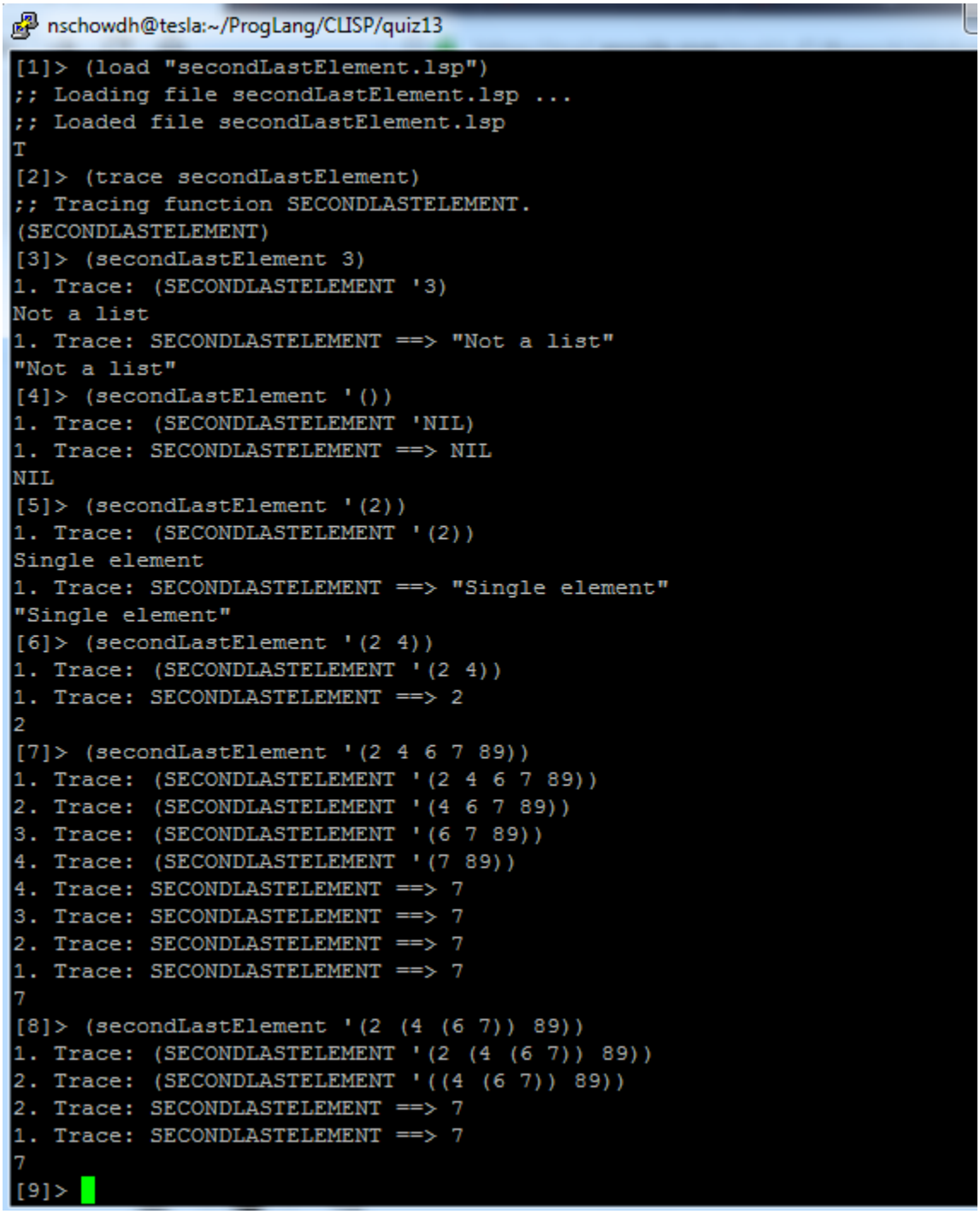
2. When the list contains zero elements then it will return *NIL*

3. When the list contains exactly one element then will return a message contains “Single element”.

4. When the list contains exactly two elements then will return the head of the list.

5. When the list contains more than two items then will call *secondLastElement* recursively and when it will reach up to length of list 2 then will return the head element of that remain list.

6. When the list contains list inside in it then use another method to find the last element of inner list is named *lastElement* and it call itself recursively until the list length become 1 and return the head of the list. As for list with one element the tail is nil.


nschowdh@tesla:~/ProgLang/CLISP/quiz13
[1]> (load "secondLastElement.lsp")
;; Loading file secondLastElement.lsp ...
;; Loaded file secondLastElement.lsp
T
[2]> (trace secondLastElement)
;; Tracing function SECONDLASTELEMENT.
(SECONDLASTELEMENT)
[3]> (secondLastElement 3)
1. Trace: (SECONDLASTELEMENT '3)
Not a list
1. Trace: SECONDLASTELEMENT ==> "Not a list"
"Not a list"
[4]> (secondLastElement '())
1. Trace: (SECONDLASTELEMENT 'NIL)
1. Trace: SECONDLASTELEMENT ==> NIL
NIL
[5]> (secondLastElement '(2))
1. Trace: (SECONDLASTELEMENT '(2))
Single element
1. Trace: SECONDLASTELEMENT ==> "Single element"
"Single element"
[6]> (secondLastElement '(2 4))
1. Trace: (SECONDLASTELEMENT '(2 4))
1. Trace: SECONDLASTELEMENT ==> 2
2
[7]> (secondLastElement '(2 4 6 7 89))
1. Trace: (SECONDLASTELEMENT '(2 4 6 7 89))
2. Trace: (SECONDLASTELEMENT '(4 6 7 89))
3. Trace: (SECONDLASTELEMENT '(6 7 89))
4. Trace: (SECONDLASTELEMENT '(7 89))
4. Trace: SECONDLASTELEMENT ==> 7
3. Trace: SECONDLASTELEMENT ==> 7
2. Trace: SECONDLASTELEMENT ==> 7
1. Trace: SECONDLASTELEMENT ==> 7
7
[8]> (secondLastElement '(2 (4 (6 7)) 89))
1. Trace: (SECONDLASTELEMENT '(2 (4 (6 7)) 89))
2. Trace: (SECONDLASTELEMENT '((4 (6 7)) 89))
2. Trace: SECONDLASTELEMENT ==> 7
1. Trace: SECONDLASTELEMENT ==> 7
7
[9]>

Problem 3: Take two lists, L1 and L2, of same lengths and will return a merged list, L3, such that L3 is obtained by carrying out the perfect shuffle operation on L1 and L2. Your program should handle lists of various sizes. (Note: A perfect shuffle on two lists, [1, 2] and [3, 4], will result in [1, 3, 2, 4].)

Code:

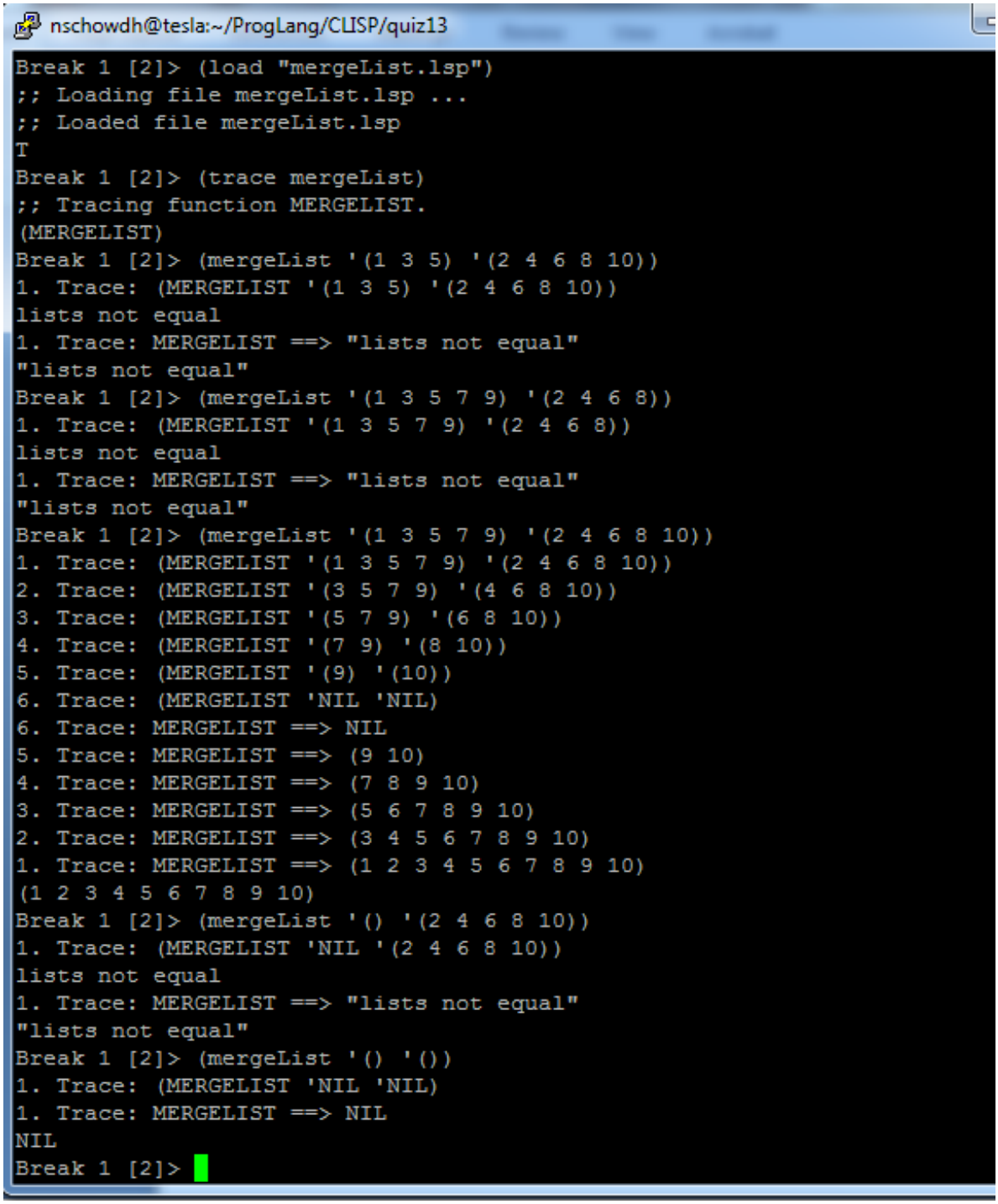
```
(defun lengthOfList(List1)
    (cond
        ((not (listp List1)) (print "Not a list"))
        ((null List1)0)
        (t (+ 1(lengthOfList(cdr List1))))
    )
)

(defun mergeList (List1 List2)
        (cond
         ((not (= (lengthOfList List1) (lengthOfList List2)))(princ "lists not equal"))
         ((null List1)nil)
         (t (cons (car List1)(cons (car List2) (mergeList (cdr List1) (cdr List2))))))
)
```

Screenshot (including program trace information):
In the output screenshot three cases has been displayed:

1. When the List1 contains 3 elements and List2 contains 5 elements then it will return terminate with a message that “lists are not equal”

2. When the List1 contains 5 elements and List2 contains 4 elements then it will return terminate with a message that “lists are not equal”

3. When both List1 and List2 contain equal number of elements then the merge take place which conduct the shuffle operation between List1 and List2 and return the resulted list.
4. When both list contain zero elements it will return *NIL*.


nschowdh@tesla:~/ProgLang/CLISP/quiz13
Break 1 [2]> (load "mergeList.lsp")
;; Loading file mergeList.lsp ...
;; Loaded file mergeList.lsp
T
Break 1 [2]> (trace mergeList)
;; Tracing function MERGELIST.
(MERGELIST)
Break 1 [2]> (mergeList '(1 3 5) '(2 4 6 8 10))
1. Trace: (MERGELIST '(1 3 5) '(2 4 6 8 10))
lists not equal
1. Trace: MERGELIST ==> "lists not equal"
"lists not equal"
Break 1 [2]> (mergeList '(1 3 5 7 9) '(2 4 6 8))
1. Trace: (MERGELIST '(1 3 5 7 9) '(2 4 6 8))
lists not equal
1. Trace: MERGELIST ==> "lists not equal"
"lists not equal"
Break 1 [2]> (mergeList '(1 3 5 7 9) '(2 4 6 8 10))
1. Trace: (MERGELIST '(1 3 5 7 9) '(2 4 6 8 10))
2. Trace: (MERGELIST '(3 5 7 9) '(4 6 8 10))
3. Trace: (MERGELIST '(5 7 9) '(6 8 10))
4. Trace: (MERGELIST '(7 9) '(8 10))
5. Trace: (MERGELIST '(9) '(10))
6. Trace: (MERGELIST 'NIL 'NIL)
6. Trace: MERGELIST ==> NIL
5. Trace: MERGELIST ==> (9 10)
4. Trace: MERGELIST ==> (7 8 9 10)
3. Trace: MERGELIST ==> (5 6 7 8 9 10)
2. Trace: MERGELIST ==> (3 4 5 6 7 8 9 10)
1. Trace: MERGELIST ==> (1 2 3 4 5 6 7 8 9 10)
(1 2 3 4 5 6 7 8 9 10)
Break 1 [2]> (mergeList '() '(2 4 6 8 10))
1. Trace: (MERGELIST 'NIL '(2 4 6 8 10))
lists not equal
1. Trace: MERGELIST ==> "lists not equal"
"lists not equal"
Break 1 [2]> (mergeList '() '())
1. Trace: (MERGELIST 'NIL 'NIL)
1. Trace: MERGELIST ==> NIL
NIL
Break 1 [2]>