Concordia University

Department of Computer Science and Software Engineering

SOEN 6411-AA: Comparative Study of Programming Languages

Assignment 2 on Lisp with solutions

Dr. Constantinos Constantinides, P.Eng.

constantinos.constantinides@concordia.ca

June 22, 2021

1 General information

Date posted: Monday, 24 May, 2021.

Date due: Friday 11 June, 2021, by 23:59.

Weight: 40% of the overall mark.

2 Introduction and ground rules

1. This is a team assignment. Each team should designate a leader who will submit the assignment electronically.

2. There are 10 problems in this assignment, each with an equal weight.

3. You may not seek any assistance while expecting to receive credit. You must work strictly within your team and seek no assistance for this project (from the instructor, the teaching assistants, fellow classmates and other teams or external help). Failure to do so will result in penalties or no credit.

4. All team members are expected to work relatively equally on each problem. The team leader has the responsibility to ensure that the team does not violate this rule. Failure to do so will result in penalties. In your submission, you must include only the names of those people who contributed to the assignment. Accommodating someone who did not contribute will result in penalties to each team member and a heavier penalty for the team leader.

5. If there is any problem in the team (such as lack of contribution, etc.), the team leader must contact the instructor as soon as the problem appears.

6. No late submissions will be accepted.

3 Problems

1. Consider function *consR* which creates a new list with its arguments by places an element to the right of a list, just as function *cons* creates a new list by placing an element on the left of a list. For example,

   $$consR(\langle a, b, c\rangle, d) = \langle a, b, c, d\rangle.$$

   (a) Provide a recursive computable function definition for `consR(L, e)` (in mathematical notation).

   (b) Translate the above definition into function `consr(lst elt)`. Do not use `append` or `list`.

   <u>Solution</u>:

   $$\begin{aligned} consR(L, e) = & \ if\ L = \langle\rangle\ then\ \langle e\rangle \\ & \ else\ \ cons(head(L), consR(tail(L), e)). \end{aligned}$$

```
(defun consr (lst elt)
    (if (null lst) (list elt)
       (cons (car lst) (consr (cdr lst) elt))))
```

2. Define a utility function, `fn`, that will read an argument and return a function based on the type of the argument. If the argument is a number, then the function will return `+`, otherwise if the argument is a list, then the function will return `append`.

   Define function `combine` that takes any number of arguments (note that the assumption is that all arguments are of the same type and are either numbers or lists) and will call the utility function `fn` to read in the first argument and return a function that will in turn be used to combine all arguments accordingly: If the arguments are numbers, then they will be added. If the arguments are lists, then they will be concatenated. For example:

```
> (combine 2 3 4)
9
> (combine '(a b) '(c d))
(A B C D)
```

   Solution:

```
(defun fn (x)
  (cond
    ((numberp x) #'+)
    ((listp  x) #'append)))



(defun combine (&rest args)
  (apply (fn (car args)) args))
```

3. Define function `combine-max` that takes two lists as arguments and returns a list constructed by the maximum elements after a pairwise comparison, i.e. it compares the corresponding first elements, then it compares the corresponding second elements. For example:

```
> (combine-max '(4 6 8 9 2) '(5 1))
(5 6 8 9 2)
> (combine-max '(3 4 5) '(1 2 3))
(3 4 5)
> (combine-max '() '(6 1 9))
(6 1 9)
```

Solution:

```
(defun combine-max (arg1 arg2)
  (cond ((null arg1) arg2)
        ((null arg2) arg1)
        ((< (car arg1) (car arg2)) (cons (car arg2)
                                         (combine-max (cdr arg1) (cdr arg2))))
        (t (cons (car arg1) (combine-max (cdr arg1) (cdr arg2))))))
```

4. Consider function `dist` that accepts an atom n and a non-empty list *lst*, and returns a list composed of lists of two elements, the first being n and the second being each successive element of *lst*. For example,

$$dist(a, \langle b, c, d\rangle) = \langle\langle a, b\rangle, \langle a, c\rangle, \langle a, d\rangle\rangle$$

   (a) Provide a recursive computable function for `dist`.
   (b) Unfold the definition for $dist(w, \langle x, y\rangle)$.
   (c) Define the function.
   (d) Trace the execution of function `dist` for `(dist 'a '(b c d))`.

<u>Solution</u>:

$$\begin{aligned} dist(a, \langle b, c, d\rangle) &= \langle\langle a, b\rangle, \langle a, c\rangle, \langle a, d\rangle\rangle \\ &= \langle\langle a, b\rangle, dist(a, \langle c, d\rangle)\rangle. \end{aligned}$$

We can therefore provide the following computable function definition:

$$\begin{aligned} dist(x, \langle\rangle) &= \langle\rangle, \\ dist(x, \langle L\rangle) &= cons(\langle x, head(L)\rangle, dist(x,\ tail(L))). \end{aligned}$$

$$\begin{aligned} dist(w, \langle x, y\rangle) &= cons(\langle w, x\rangle, dist(x, \langle y\rangle)) \\ &= cons(\langle w, x\rangle, cons(\langle w, y\rangle, dist(x, \langle\rangle))) \\ &= \langle\langle w, x\rangle, \langle w, y\rangle\rangle. \end{aligned}$$

```
(defun dist (n lst)
  (if (null lst)
      nil
      (cons (list n (car lst)) (dist n (cdr lst) ))))
```

```
(dist 'a '(b c d)) = (cons (list 'a 'b) dist ('a '(c d)))
                   = (cons '(a b) (cons ((list 'a 'c) dist ('a '(d)))))
                   = (cons '(a b) (cons '(a c) (cons (list 'a 'd) '()))))
                   = (cons '(a b) (cons '(a c) (cons '(a d) '())))
                   = ((a b) (a c) (a d))
```

5. Define function `rem-if-dupl` that receives a list as its argument, and proceeds to return its argument having removed all its repetitive elements. Consider the following tests:

```
; Test:

(print (rem-if-dupl nil))
(print (rem-if-dupl '(1 1)))
(print (rem-if-dupl '(1 2 2 3 4)))
(print (rem-if-dupl '(1 2 3 4)))
(print (rem-if-dupl '(1 2 3 4 2)))

; Output:

; NIL
; NIL
; (1 3 4)
; (1 2 3 4)
; (1 3 4)
```

<u>Solution</u>:

```
(defun remove-all(l a)
    (cond ((null l) nil)
          ((equal (car l) a) (remove-all (cdr l) a))
          (t (cons (car l) (remove-all (cdr l) a)))))

(defun rem-if-dupl (l)
(cond ((null l) nil)
((member (car l) (cdr l))  ; key idea: is car in cdr?
(rem-if-dupl (remove-all (cdr l) (car l)))) ; this handles the last example
```

```
; calling rem-if-dupl is probably wrong (see below)
(t (cons (car l) (rem-if-dupl(cdr l))))))
```

6. Define function `oseq` that receives an integer `n` and returns a list that contains a sequence of odd numbers less than `n`, starting from `1`. Consider the following tests:

```
; Test:
(print (oseq 1))
(print (oseq 2))
(print (oseq 10))
(print (oseq 11))

; Output:
; NIL
; (1)
; (1 3 5 7 9)
; (1 3 5 7 9)
```

Solution:

```
; *recursive solution*:

(defun oseq (n)
(cond ((not (numberp n)) nil)   ; if not present, OK
((< n 2) nil)  ; (oseq 1) is NIL, to handle first case
        ((= n 2) '(1))
((oddp n) (oseq (- n 1)))
(t (append (oseq (- n 2)) (list (- n 1))))))
```

```
; iterative solution 1:

(defun oseq (n)
    (progn
        (setq result nil)
        (setq i 1)
        (loop
             (if (>= i n) (return result))
             (setq result (append result (list i)))
             (setq i (+ i 2)))))

; iterative solution 2:

(defun oseq (n)
(cond ((not (numberp n)) nil)
((< n 2) nil)
         (t
(let ((result))
                  (dotimes (i (/ (- n 1) 2) result)
                      (setf result (append result (list (+ 1 (* 2 i))))))))))
```

7. Define function `filter` that takes two arguments, a) a non-empty list of integers, and b) a positive integer, and produces a list whose elements are those elements of the first argument that are larger than the second argument. For example:

```
> (filter '5 3)
NIL
> (filter '() 5)
NIL
> (filter '(7 9 11) '(2))
NIL
> (filter '(3 4 5) '0)
NIL
> (filter '(3 4 5) '2.5)
NIL
> (filter '(3 4 5) '0)
NIL
> (filter '(5 9 3 2 11) '7)
(9 11)
```

Trace the execution of the function for `(filter '(12 9 3 2 7) '4)`.

Solution:

```
( defun filter (lst el)
  ( cond
  (( not ( listp lst )) nil )
  (( null lst ) ())
  (( not ( atom el )) nil )
  (( or (<= el 0) (not ( integerp el ))) nil )
  (( <= ( car lst ) el) ( filter ( cdr lst ) el ))
  (t ( cons (car lst ) ( filter ( cdr lst ) el )))))
```

8. Define function `is-bst` to check whether a binary tree is a Binary Search Tree. A Binary Search Tree is a tree in which all the nodes follow the below-mentioned properties:

   - The left sub-tree of a node has a key less than or equal to its parent node's key.
   - The right sub-tree of a node has a key greater than to its parent node's key.

   For example:

```
(print "YES")
(print (is-bst '() ))
(print (is-bst '(1 (0)) ))
(print (is-bst '(1 (0) (3)) ))
(print "NO")
(print (is-bst '(1 (2)) ))
(print (is-bst '(1 (0) (3 (4))) ))
```

   <u>Solution</u>:

```
(defun leftmost (tt)
    (cond ((null tt) nil)
        ((null (car (cdr tt))) (car tt))
        (t (leftmost (car (cdr tt)))))
)

(defun rightmost (tt)
    (cond ((null tt) nil)
        ((null (car (cddr tt))) (car tt))
        (t (rightmost (car (cddr tt)))))
)
```

```
(defun is-bst (tt)
    (cond ((null tt) t)
        ((not (is-bst (car (cdr tt)))) nil)
        ((not (is-bst (car (cddr tt)))) nil)
        ((and
          (or (null (car (cdr tt))) (< (rightmost (car (cdr tt))) (car tt)))
          (or (null (car (cddr tt))) (> (leftmost (car (cddr tt))) (car tt)))) t)
    )
)
```

9. Consider a binary tree. No assumption is made whether or not the tree is balanced.

   (a) Provide a recursive definition of a procedure to calculate the height of the tree.

   (b) Define function `height` that receives a list representing a binary tree, and proceeds to calculate the height of the tree.

   Solution:

   To calculate the height, we proceed as follows:

   - Base case: If tree is empty, or if the tree has one node (root) then return 0.
   - Recursive case: Add one to the maximum of the heights of the left and right subtrees.

```
(defun height (l)
(cond
((null l) 0)
(T (+ 1
           (let ((n1 (height (car (cdr l))))
            (n2 (height (car (cdr (cdr l))))))
            (if (> n1 n2) n1 n2))))
))
```

```
( defun tree-height ( tree )
  (if (or ( null tree )
         (and ( null (car ( cdr tree )))
              ( null (car ( cdr ( cdr tree ))))))
      0
  (+ 1 ( take-max ( tree-height (car ( cdr tree )))
  ( tree-height (car ( cdr ( cdr tree ))))))))
```

```
( defun take-max (n1 n2)
 (if (> n1 n2)
    n1
    n2 ))
```

10. Define a function that accepts a non-empty binary tree as an argument and returns a list of nodes that represents the post-order traversal of the tree.

    Solution:

    ;; This is pre-order. Must be modified.

```
; Validate if given parameter is a binary tree or not
(defun is-tree(tree)
 (cond
  ((null tree) NIL)
  ((< (list-length tree) 3) NIL)
  (t)))

;return left subtree of a binary tree
(defun left-subtree(tree)
 (cond
  ((null tree) NIL)
  ((not (listp tree)) NIL)
  (t (car (cdr tree)))))



(defun right-subtree(tree)
  (cond
((null tree) NIL)
((not (listp tree)) NIL)
(t (car (cdr (cdr tree))))))



;perform postorder traverse
  (defun postorder(tree)
  (if (not (is-tree tree)) NIL
```

```
(append
  (postorder (left-subtree tree))
  (postorder (right-subtree tree))
  (list (car tree)))))
```

4 What to submit

You must submit a zip file containing the following two files:

1. File `README.txt` that contains the names and id's of all contributing members of your team.

2. File `functions.lisp` that contains all functions.

3. File `assignment.txt` which contains all non-coding components of the problems.

Name the zip file after your team e.g. `team1.zip` and submit it at the Electronic Assignment Submission portal

(`https://fis.encs.concordia.ca/eas`)

under **Programming Assignment 2**.

END OF ASSIGNMENT
