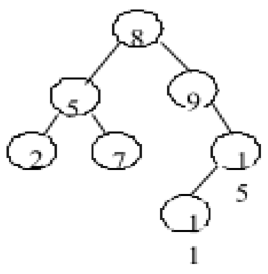
TC2006 Programming Languages
Assignment 4: Advance Programming in Racket

Team: Sebastián Saldaña A01570274, Ana Elisa Estrada A01251091 and Estefanía Charles A01283472

In this assignment you will put into practice your knowledge of recursive programming with data structures and higher-order functions in Racket. The source code must be documented internally through comments that include, at least, your student IDs and names of the team members, as well as the description of what each function calculates and the meaning of each of its formal parameters (**10 points**).

Problems on Data Structures (60 points)

1. A Binary Tree (AB) can be represented in Racket, by means of a list in the following format: (root subtree-left subtree-right). For example, if the representation of the following AB is defined as:



```
(define AB
      '(8 (5 (2 () ())
             (7 () ()))
          (9 ()
             (15 (11 () ())
                 () ))))
```

Program the following functions for binary trees:

a. The **binary-tree?** Predicate, that given a list, determines whether it is consistent with the correct list representation for binary trees.
Try with:
```
> (binary-tree? '())                              => #t
> (binary-tree? AB)                               => #t
> (binary-tree? '(a (b (() ()) ()) (d () ()))    => #f
```

```
(define (binary-tree? bin_tree)
   (cond
       ((null? bin_tree) #t)
       ((and
           (not (list? (car bin_tree)))
           (list? (cadr bin_tree))
           (list? (car (cddr bin_tree)))
       ) (and (binary-tree? (cadr bin_tree)) (binary-tree? (car (cddr bin_tree)))))
       (else #f)
))
```

b. The **get-minors** function, that given a binary tree and a value as arguments, creates a list with the node values of the tree that are smaller than the given value.

Try with:

```
> (get-minors AB 2)          => ()
> (get-minors AB 8)          => (5 2 7)
> (get-minors AB 20)         => (8 5 2 7 9 15 11)
> (get-minors '(1 (5 (2 () ()) ())(4 () (3 ()()))) 4) => (1 2 3)
```

```
(define (get-minors bt val)
   (if (null? bt)
       '()
       (if (> val (car bt))
           (make-bt-list bt)
           (get-minors (cadr bt) val)
       )
   )
)


(define (make-bt-list bt)
   (if (null? bt)
       '()
       (append (cons (car bt) (make-bt-list (cadr bt))) (make-bt-list (car (cddr bt))))
   )
)
```

c. The **longest-branch** function, that given a binary tree, returns a list with the values that are in its longest branch. In case of a tie, return the values of your first longest branch.

Try with:

```
> (longest-branch '())       => ()
> (longest-branch AB)        => (8 9 15 11)
> (longest-branch '(a (b (c () ()) ()) (d (e () ()) ()))  => (a b c)
```

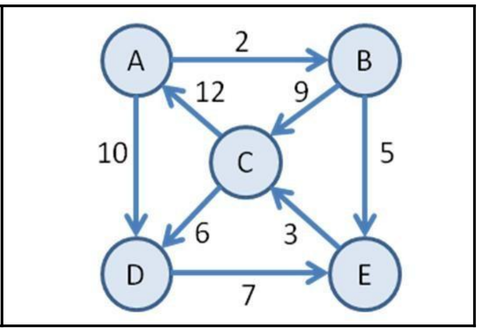
```
(define (longest-branch bin_tree)
   (if (null? bin_tree)
       '()
       (if  (>
               (length (cons (car bin_tree) (longest-branch (car (cddr bin_tree)))))
               (length (cons (car bin_tree) (longest-branch (cadr bin_tree)))))
           (cons (car bin_tree) (longest-branch (car (cddr bin_tree))))
           (cons (car bin_tree) (longest-branch (cadr bin_tree)))
       )
   )
)
```

2. A weighted directed graph can be represented in Racket by means of a list of edges. The graph representation list in this case has a record for each graph node, and this record in turn is a list with the following format:

```
(node (adjacent_node1
       arc_weight1)
       (adjacent_node2
       arc_weight2)
       ...
       (adjacent_nodeN arc_weightN)))
```

Where node is the identifier of a node in the graph, adjacent_node i is the identifier of a destination node and arc_weight i is the weight of the arc that joins the node with node_adjacent i. Under this format, the following graph would be represented as:

```
(define g '(
    (A (B 2) (D 10))
    (B (C 9) (E 5))
    (C (A 12) (D 6))
    (D (E 7))
    (E (C 3)) )
```



a) The **destination-nodes** function, that given a graph and a node as arguments, returns a list with the destination nodes of the egdes for the given node.
Try with:

```
> (destination-nodes g 'A) => (B D)
> (destination-nodes g 'D) => (E)
> (destination-nodes g 'F) => ()
```

```
(define (destination-nodes graph node)
   (cond
      ((null? graph) '())
      ((eq? node (caar graph)) (get-dest-nodes (cdar graph)))
      (else (destination-nodes (cdr graph) node))
   )
)


(define (get-dest-nodes adj-list)
   (if (null? adj-list)
      '()
      (cons (caar adj-list) (get-dest-nodes (cdr adj-list)))
   )
)
```

b) The **source-nodes** function, that given a graph and a node as arguments, returns a list with the nodes that have the given node as a destination in an edge.
Try with:

```
> (source-nodes g 'A) => (C)
> (source-nodes g 'C) => (B E)
> (source-nodes g 'F) => ()
```

```
(define (source-nodes graph node)
   (if (null? graph)
       '()
       (append (get-source-nodes (cdar graph) (caar graph) node) (source-nodes (cdr graph)
node))
   )
)

(define (get-source-nodes adj-list parent-node node)
   (if (null? adj-list)
       '()
       (if (eq? (caar adj-list) node)
           (cons parent-node (get-source-nodes (cdr adj-list) parent-node node))
           (get-source-nodes (cdr adj-list) parent-node node)
       )
   )
)
```

c) The **delete-arc** function, that given a graph and two node identifiers as input, returns the graph without the specified arc (if any).
Try with:

```
> (delete-arc g 'B 'C)
=> (A (B 2) (D 10))(B (E 5))(C (A 12) (D 6))(D (E 7))(E (C 3)))
> (delete-arc g 'B 'D)
=> (A (B 2) (D 10))(B (C 9) (E 5))(C (A 12) (D 6))(D (E 7))(E (C 3)))
> (delete-arc g 'D 'E)
=> (A (B 2) (D 10))(B (C 9) (E 5))(C (A 12) (D 6))(D)(E (C 3)))
> (delete-arc g 'F 'C)
=> (A (B 2) (D 10))(B (C 9) (E 5))(C (A 12) (D 6))(D (E 7))(E (C 3)))
```

```
(define (delete-arc graph node1 node2)

   (cond

       ((null? (car graph)) '())

       ((eq? (caar graph) node1)

           (cons (cons (caar graph) (remove-arc (cdar graph) node2)) (cdr graph))

       )

       (else

           (cons (car graph) (delete-arc (cdr graph) node1 node2))

       )

   )

)


(define (remove-arc adj-lst node2)

   (cond

       ((null? adj-lst) '())

       ((eq? (caar adj-lst) node2) (remove-arc (cdr adj-lst) node2))

       (else (cons (car adj-lst) (remove-arc (cdr adj-lst) node2)))

   )

)
```

Problems on Higher Order Functions (30 points)

3. **WITHOUT using explicit recursion**, implement the functions described using (if required) the higher-order primitives **map**, **apply**, and the special form **lambda**. All functions must work on matrices represented as lists of sublists, where each sublist corresponds to a row of the matrix. E.g., the list ((1 2 3)(4 5 6)) represents the matrix:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

   a. The zero-count function, that given a matrix, returns the number of zeros found in it.
      Try with:

```
> (count-zeros '())                              => 0
> (count-zeros '((0 1)(2 3)))                    => 1
> (count-zeros '((4 0 3 1)(5 1 2 1)(6 0 1 1)))   => 2
```

```
(define (count-zeros matrix)
 (length (apply append
                (map (lambda (e) (if(zero? e) '(0) '()))
                     (apply append matrix)))))
```

b. The minmax function, that given a matrix, returns a list with the smallest and largest values found in it. Assume that the matrix has at least one element.
 Try with:

```
> (minmax '((2)))                              => (2 2)
> (minmax '((0 1)(2 3)))                       => (0 3)
> (minmax '((4 0 -3 1)(5 -1 2 1)(6 0 1 1)))    => (-3 6)
```

```
(define (minmax matrix)
   (list
       (foldl (lambda (a b) (if (< a b) a b)) (car (apply append matrix)) (cdr (apply append
matrix)))
       (foldl (lambda (a b) (if (> a b) a b)) (car (apply append matrix)) (cdr (apply append
matrix)))))
```

c. The mulmat function, that given two matrices, returns the result of multiplying them. Assume that matrices shapes are correct, i.e., the number of columns of the first is equal to the number of rows of the second.
 Try with:

```
> (multmat '((1 2 3)(0 2 1)) '((4 0 3 1)(5 1 2 1)(6 0 1 1)))
          => ((32 2 10 6)(16 2 5 3))
```

```
(define (multmat matrix1 matrix2)
   (map
       (lambda (row)
           (map
               (lambda (col)
                   (apply +
                       (map * row col)))
               (apply map list matrix2)))
       matrix1)
)
```