CT331 Assignment 2

Student name: Paulius Zabinskas
Student ID: 20120267

Question 1

(A & B):

- A cons pair of two numbers.
  - (A): (print (cons 1 2))
  - (B): Result is a pair of two numbers
  - (Output): (1 . 2)

- A list of 3 numbers, using only the cons function.
  - (A): (print (cons 1 (cons 2 (cons 3 '()))))
  - (B): Result is a list of three numbers.
  - (Output):

- A list containing a string, a number and a nested list of three numbers, using only the cons function.
  - (A): (print (cons "hello" (cons 42 (cons (cons 1 (cons 2 (cons 3 '()))) '()))))
  - (B): Result is a list with a nested list inside.
  - (Output): ("hello" 42 (1 2 3))

- A list containing a string, a number and a nested list of three numbers, using only the list function.
  - (A): (print (list "hello" 42 (list 1 2 3)))
  - (B): Result is a a list with a nested list, similar to previous output
  - (Output): ("hello" 42 (1 2 3))

- A list containing a string, a number and a nested list of three numbers, using only the append function.
  - (A): (print (append (list "hello") (list 42) (list (list 1 2 3))))
  - (B): Result is a a list with a nested list, similar to previous two outputs
  - (Output): ("hello" 42 (1 2 3))

(B):

(cons) is a low-level function used for constructing pairs, and it is the fundamental building block for lists in Scheme.
(list) is a high-level function that provides an easy way to create lists out of multiple elements, improving readability.
(append) is a high-level function that provides an easy way to create lists and / or add new elements to a list and provides readability.

Code for Question 1:

```
(print (cons 1 2))
(print (cons 1 (cons 2 (cons 3 '()))))
(print (cons "hello" (cons 42 (cons (cons 1 (cons 2 (cons 3 '()))) '()))))
(print (list "hello" 42 (list 1 2 3)))
(print (append (list "hello") (list 42) (list (list 1 2 3))))
```

Question 2:

(A):

```
(define (ins_beg element lst)
  (cons element lst))

(print (ins_beg 'a '(b c d)))
(print (ins_beg '(a b) '(b c d)))

Output:
(A B C D)
((A B) B C D)
```

(B):

```
(define (ins_end element lst)
   (append lst (list element)))

(print (ins_end 'a '(b c d)))
(print (ins_end '(a b) '(b c d)))

Output:
((B C D) . A)
((B C D) A B)
```

(C):

```
(define (count_top_level lst)
  (if (null? lst)
      0
      (+ 1 (count_top_level (cdr lst)))))


Output:
```

```
3
```

(D):

```
(define (count_instances item lst)
  (if (null? lst)
      0
      (+ (if (equal? item (car lst)) 1 0)
         (count_instances item (cdr lst)))))

(print (count_instances 'a  '(a b c a)))
Output:
1
```

(E):

```
(define (count_instances_tr item lst)
  (define (helper item lst count)
    (if (null? lst)
        count
        (helper item (cdr lst) (if (equal? item (car lst)) (+ count 1) count))))
  (helper item lst 0))

(print (count_instances_tr 'a '(a b c)))
Output:
1
```

(F):

```
(define (count_instances_deep item lst)
  (if (null? lst)
      0
      (+ (if (list? (car lst))
             (count_instances_deep item (car lst))
             (if (equal? item (car lst)) 1 0))
         (count_instances_deep item (cdr lst)))))

(print (count_instances_deep 'a '(a b c'(a b c))))

Output:
2
```

Question 3:

(A):

```
;; Define a BST node: ( left-child value right-child)
  (define (make-bst-node left value right)
    (list left value right))

;; Create individual nodes
  (define node-1 (make-bst-node  '() 1 '()))
  (define node-3 (make-bst-node  node-1 3 '()))
  (define node-5 (make-bst-node '() 5 '()))

;; Construct the BST
  (define bst-root (make-bst-node  node-3 4 node-5))

;; Define display BST function
  (define (display-bst-inorder bst)
    (cond
      ((null? bst) '())  ; If the tree is empty, return an empty list.
      (else (append
              (display-bst-inorder (car bst))
              (list (cadr bst))
              (display-bst-inorder (caddr bst))
            ))))

;; Call function to display previously defined BST
(display-bst-inorder bst-root)

Output:
'(1 3 4 5)
```

(B):

```
(define (is-item-in-bst? item bst)
    (cond
      ((null? bst) #f)                         ; If the tree is empty, item is not found.
      ((equal? item (cadr bst)) #t)            ; Item found at the current node.
      ((< item (cadr bst)) (is-item-in-bst? item (car bst)))   ; Search left subtree.

      (else (is-item-in-bst? item (caddr bst)))  ; Search right subtree.
    ))
```

```
Output:
(is-item-in-bst? 3 bst-root)
#t
(is-item-in-bst? 2 bst-root)
#f
```

(C):

```
(define (insert-bst item bst)
  (cond
    ;; If the spot is empty, insert the item here.
    ((null? bst) (make-bst-node '() item '()))

    ;; If the item is found, return the tree as is.
    ((equal? item (cadr bst)) bst)

    ;; If the item is less, recurse on the left subtree.
    ((< item (cadr bst)) (make-bst-node (insert-bst item (car bst)) (cadr bst)
(caddr bst)))

    ;; Otherwise, recurse on the right subtree.
    (else (make-bst-node (car bst) (cadr bst) (insert-bst item (caddr bst)))))
  )

(define new-bst (insert-bst 2 bst-root))
(display-bst-inorder new-bst)

Output:
'(1 2 3 4 5)
```

(D):

(E):

(F):