

# Assignment 1 – Cathal Lawlor – 21325456

## Question 1

```
#lang racket
(display "\nAssignment 2\nQuestion 1\n\n")

;;A cons pair of two numbers
(cons 1 2)

;; A list of 3 numbers, using only the cons function.
(cons 2 (cons 4 (cons 7 empty)))

;;A list containing a string, a number and a nested list of three numbers, using only the
;;cons function.
(cons "This is a string!"
      (cons 5
            (cons (cons 1 (cons 2 (cons 3 empty))) empty)
              )
    )

;;A list containing a string, a number and a nested list of three numbers, using only the
;;list function
(list "This is a string!" 5 (list 1 2 3))
;;A list containing a string, a number and a nested list of three numbers, using only the
;;append function.
(append '("This is a string!") '(5) '((1 2 3)))
```

Assignment 2  
Question 1

A cons pair of two numbers.  
'(1 . 2)

A list of 3 numbers, using only the cons function.  
'(2 4 7)

;A list containing a string, a number and a nested list of three numbers.  
'("This is a string!" 5 (1 2 3))

A list containing a string, a number and a nested list of three numbers - list  
'("This is a string!" 5 (1 2 3))

A list containing a string, a number and a nested list of three numbers - append.  
'("This is a string!" 5 (1 2 3))

1. The cons function creates a cons pair, which is not always a 'list' as we see here, due to the two elements being delimited by a '.' as it is only a cons pair, and not a 'list' which is actually a cons pair, where the second element is actually a 'list' or empty.
2. We create a list using cons, of three numbers, where firstly we create a list with just one element, having 3 and empty in the innermost cons. From there we make a two-element list by making a cons pair, with 2 and the earlier created one element list. We again create a three-element list by making a cons pair, with 1 and the two-element list we have.
3. We use the cons to make a one-element list containing 3, and then pair that list with 2 and 1. Then we use cons to pair that three-element list to empty, generating a 'list', where we pair that sub-list (1,2,3) with 5. From there finally we use cons to pair a string to that list (5, (1,2,3)) containing the number and the sub-list.
4. We use the list function to put 3 elements in a list, being a string ("This is a string"), a number (5), and then a sub-list (1,2,3), created by running (list 1 2 3) inside the function.
5. First, we create three lists using '()' of the string, the number and then a list of three numbers. Then, we use append which can only accept lists as an argument, to append the three lists together.

The difference between cons, list, and append is that cons creates a new pair, list creates a new list, and append concatenates lists. cons is used to build up pairs and lists, while list is used to create new lists from scratch. append is used to combine existing lists into a single list.

## Question 2

```
#lang racket
(provide ins_beg)
(provide ins_end)
(provide cout_top_level)
(provide count_instances)
(provide tail_count_instances)
(provide tail_count_instances_helper)
(provide count_instances_deep)

;; ins_beg: Scheme function that takes an element and a list and returns a new list with
the element added to the beginning of the original list.
(define (ins_beg el lst)
  (cons el lst)) ;;creating new list containing the new element at the start using cons

;; ins_end: Scheme function that takes an element and a list and returns a new list with
the element added to the end of the original list.
(define (ins_end el lst)
  (append lst (cons el empty))) ;;Assuming lst is a list, and appending a new list
(created using cons) with el to the end of lst

;; cout_top_level: Scheme function that takes a list and returns the number of top-level
items in the list (i.e., items that are not in a sublist).
(define (cout_top_level lst)
  (if (empty? lst) ;; empty? is a built-in function to 0 when true - if the list is empty.
      0
      (+ 1 (cout_top_level (cdr lst)))) ;;if not empty, return 1 + counting the rest of the
list - recursively
  )
)

;;D. Write a non-tail recursive function called count_instances that counts the number
;;of times an item occurs in a list of items (you may assume all items are atomic).
(define (count_instances item lst)
  (cond
    [(null? lst) 0] ;; Base case - list is empty / finished counting
    [(equal? (car lst) item) (+ 1 (count_instances item (cdr lst)))] ;;If the first
element in list matches return incremented by 1 + recursively count the remainder of the
list
    [else (count_instances(cdr lst))] ;;Otherwise, continue with the rest of the list.
  )
)

;; E. Write a tail-recursive version of part D solution called count_instances_tr
(define (tail_count_instances item lst cnt)
  (cond
    [(null? lst) cnt] ;; Base case - list is empty, finished counting, return count

    ;; if first element is equals to the item
    [(eq? (car lst) item)
     ;; increment 1 + recurse remainder of the list
     (tail_count_instances item (cdr lst) (+ cnt 1))
    ]
  )
)
```

```

]

;; else - first element isn't equal to the item
[else
  ;; recursively count the remainder of the list - count is the same
  (tail_count_instances item (cdr lst) cnt)
]
)
)

;; Tail recursive function helper for count of a given item in a list - assumption all
items are atomic
(define (tail_count_instances_helper item lst)
  ;; Calling the function with starter count - 0
  (tail_count_instances item lst 0)
)

;; F. Write a Scheme function named count_instances_deep that counts the number of
;; times an item occurs in a list of items; note that the list may contain sub-lists and
you
;; should also count occurrences in those lists.
(define (count_instances_deep item lst)
  (cond
    [(null? lst) ;; Base case - list is empty / finished counting
     0]

    ;; If the first element in the list is a list, recurse through the list, and then
recurse through the remainder of the main list
    ;; returning the sum of the two results
    [(pair? (car lst))
     (+ (count_instances_deep item (car lst)) (count_instances_deep item (cdr lst)))]

    ;; If the first element in list matches return incremented by 1 + recursively count
the remainder of the list
    [(equal? (car lst) item)
     (+ 1 (count_instances_deep item (cdr lst)))]

    [else (count_instances_deep item (cdr lst))] ;; Otherwise, continue with the rest of
the list.
  )
)

```

## Question 2 continued.

Part A

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

---

Part B

```
> (ins_end 'a '(b c d))  
(ins_end '(a b) '(b c d))  
'(b c d a)  
'(b c d (a b))
```

---

Part C

```
(define my-list '(1 2 3 2 4 2 5))  
(display (count_top_level my-list))  
7
```

---

Part D

```
> (define my-list '(1 2 3 2 4 2 5))  
(define item-to-count 2)  
  
(display (count_instances item-to-count my-list))  
3  
>
```

---

Part E

```
> (define my-list '(1 2 3 2 4 2 5))  
(define item-to-count 2)  
  
(display (count_instances_tr_helper item-to-count my-list))  
3
```

---

Part F

```
> (define my-list '(1 (2 (1 3)) 4 (1 (5 (1 6))) 7))  
  
(define item-to-count 1)  
  
(display (count_instances_deep item-to-count my-list))  
4
```

### Question 3

**Part A**

```
> (define bst '(5 (3 (2 () ()) (4 () ())) (8 (7 () ()) (9 () ())))
(display_in_order bst)
2
3
4
5
7
8
9
```

---

**Part B**

```
> ;; Define a binary search tree
(define bst '(5 (3 (2 () ()) (4 () ())) (8 (7 () ()) (9 () ())))

(display "Testing tree_search:\n")

;; Test cases
(display "Is 4 present in the tree? ")
(if (tree_search 4 bst) (display "#t") (display "#f"))
(newline)

(display "Is 6 present in the tree? ")
(if (tree_search 6 bst) (display "#t") (display "#f"))
(newline)

(display "Is 7 present in the tree? ")
(if (tree_search 7 bst) (display "#t") (display "#f"))
(newline)
Testing tree_search:
Is 4 present in the tree? #t
Is 6 present in the tree? #f
Is 7 present in the tree? #t
```

---

**Part C**

```
> (define empty-tree '()) ; Define an empty tree

;; Insert items into the tree
(define tree1 (tree_insert 5 empty-tree))
(define tree2 (tree_insert 3 tree1))
(define tree3 (tree_insert 8 tree2))
(define tree4 (tree_insert 2 tree3))
(define tree5 (tree_insert 4 tree4))
(define tree6 (tree_insert 7 tree5))
(define tree7 (tree_insert 9 tree6))

;; Define a function to display the binary search tree in sorted order
(define (display_tree_in_order bst)
  (cond
    [(null? bst) '()]
    [else
     (append (display_tree_in_order (cadr bst))
             (list (car bst))
             (display_tree_in_order (caddr bst))))])

(display "Original binary search tree (empty):\n")
(display (display_tree_in_order empty-tree))
(newline)

(display "Binary search tree after inserting elements:\n")
(for-each (lambda (item) (display item) (newline)) (display_tree_in_order tree7))

Original binary search tree (empty):
()
Binary search tree after inserting elements:
2
3
4
5
7
8
9
```

Note, all other parts there, I couldn't fully implement tests for them. They do/should work.