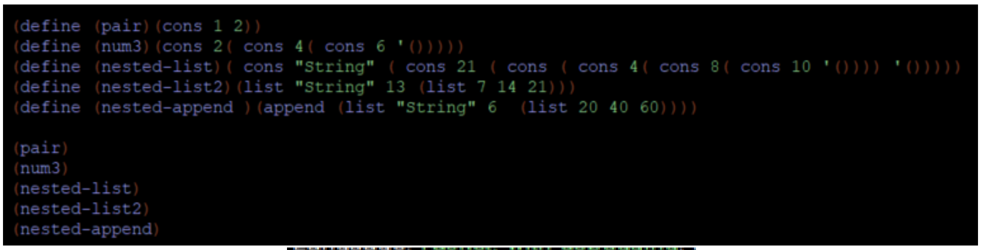
CT331 Assignment 2

Brian Moyles – 21333461

Question 1

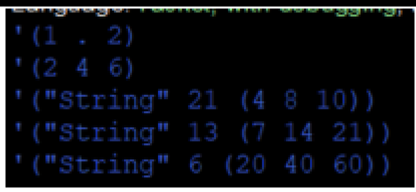
- (define (pair)(cons 1 2))
- (define (num3)(cons 2(cons 4(cons 6 '()))))
- (define (nested-list)(cons "String" (cons 21 (cons (cons 4(cons 8(cons 10 '()))) '()))))
- (define (nested-list2)(list "String" 13 (list 7 14 21)))
- (define (nested-append)(append (list "String" 6 (list 20 40 60))))

<u>Output</u>



```
(define (pair)(cons 1 2))
(define (num3)(cons 2( cons 4( cons 6 '()))))
(define (nested-list)( cons "String" ( cons 21 ( cons ( cons 4( cons 8( cons 10 '()))) '()))))
(define (nested-list2)(list "String" 13 (list 7 14 21)))
(define (nested-append )(append (list "String" 6  (list 20 40 60))))

(pair)
(num3)
(nested-list)
(nested-list2)
(nested-append)
```



```
'(1 . 2)
'(2 4 6)
'("String" 21 (4 8 10))
'("String" 13 (7 14 21))
'("String" 6 (20 40 60))
```

<u>Results</u>

- The cons function is used to pair to elements together the first element is the car and the rest of the list is the cdr.
- You can also do a series of cons operations, as seen in the second line of code, to link pairs together.
- The third task is the same as the previous one, except different data types are used. In this one we use a string, a number and a nested list of numbers.
- Task 4 uses the list function instead of the cons function. It stores different data types in a list Together. A list is created with 3 numbers and that is added to a list with a string and another number.
- The final task uses the append function is used to concatenate 3 separate lists together. 3 separate lists are defined inside the append function, which then combines these into a single list.

<u>Difference between Cons, Append and List</u>

- Cons is mainly used to construct pairs between elements. It is also used to add elements to the front of the list as a new car by forming another pair.
- List does not create pairs and instead creates the list with the specified elements with less restrictions
- Append combines lists into a bigger, single list by using existing lists and doesn’t create new elements

Question 2

#lang racket
(provide ins_beg)

```
(define (ins_beg el lst)
  (cons el lst))

(provide ins_end)
(define (ins_end el lst)
 (append lst(list el)))

(provide count_top_level)
(define (count_top_level lst)
  (cond
    ((null? lst)0)
    ((not (pair? lst)) 1)
    (else
     (+ (count_top_level (car lst)) (count_top_level (cdr lst))))))

(provide count_instances)
(define (count_instances el lst)
  (cond
    ((null? lst) 0)
    ((equal? el (car lst)) (+ 1 (count_instances el (cdr lst))))
    (else
     (count_instances el (cdr lst)))))

(provide count_instances_tr)
(define (count_instances_tr el lst)
  (define (c_i_tr el lst count)
    (cond
      ((null? lst) count)
      ((equal? el (car lst)) (c_i_tr el (cdr lst) (+ count 1)))
      (else
       (c_i_tr el (cdr lst) count))))
  (c_i_tr el lst 0))

(provide count_instances_deep)
(define (count_instances_deep el lst)
  (cond
    ((null? lst)0)
    ((pair? (car lst))
    (+ (count_instances_deep el (car lst))
       (count_instances_deep el (cdr lst))))
    ((equal? el (car lst))
     (+ 1 (count_instances_deep el (cdr lst))))
           (else
            (count_instances_deep el (cdr lst)))))

(display "ins beg function: \n")
(ins_beg 'a '(b c d))
(ins_beg '(a b) '(b c d))

(display "ins end function: \n")
(ins_end 'a '(b c d))
(ins_end '(a b) '(b c d))

(display "Count Top Level: \n")
(count_top_level '(1 2 (3 4) 5 (6 (7)) 8))

(display "Count Instances: \n")
(count_instances 2 '(1 2 3 2 4 2))

(display "Count Instances Tail Recursion: \n")
(count_instances_tr 2 '(1 2 3 2 4 2))

(display "Count Instances Deep: \n")
(count_instances_deep '2 '(1 2 3 (2 (4 2) 5) 2 6))
```

Output

```
ins beg function:
'(a b c d)
'((a b) b c d)
ins end function:
'(b c d a)
'(b c d (a b))
Count Top Level:
8
Count Instances:
3
Count Instances Tail Recursion:
3
Count Instances Deep:
4
```

Question 3

```
#lang racket
; Binary Tree Initialized
(define binaryTree '(10(6(4()())(8()()))(16(12()())(20()()))))

;Part A

(define (partA tree)
  (cond
    ((null? tree) '()) ; Return nothing if the tree is empty
    (else
     (let ((leftSubtree (cadr tree)) ; store the left Subtree in a variable
           (value (car tree)) ; Make a variable to get the values in the tree
           (rightSubtree (caddr tree))) ; store the right subtree in a variable
       (begin ; recursively go through the subtrees and print them out in order
         (partA leftSubtree)
         (display value)(display "-")
         (partA rightSubtree))))))

(partA binaryTree)

;Part B

(define (partB tree el)
  (cond
    ((null? tree) #f) ; Return #f if the tree is empty
    (else
     (let ((leftSubtree (cadr tree)) ; store the left Subtree in a variable
           (value (car tree)) ; Make a variable to get the values in the tree
           (rightSubtree (caddr tree))) ; store the right subtree in a variable
       (let
         ((leftEl (partB leftSubtree el)) ; Check the left subtree's elements
          (rightEl (partB rightSubtree el))) ; Check the right subtree's elements
         (if (or leftEl (equal? value el) rightEl) ; Check if element is present in tree
             #t ; return #t if it is
             #f)))))) ; Else return #f if it is not present

; 2 tests to confirm it works: 1 true + 1 false
(partB binaryTree 8)
(partB binaryTree 11)

;Part C

(define (partC tree el)
  (cond
    ((null? tree) (list el '() '()))
    (else
     (let ((leftSubtree (cadr tree)) ; store the left Subtree in a variable
```

```
        (value (car tree)) ; Make a variable to get the values in the tree
        (rightSubtree (caddr tree)))  ; store the right subtree in a variable
      (cond
       ((< el value) ; If element is less than value
        (list value (partC leftSubtree el) rightSubtree)) ; Insert into left subtree
       ((> el value) ; If element is greater than value
        (list value leftSubtree (partC rightSubtree el))) ; Insert into right subtree
       (else (list value leftSubtree rightSubtree))))))

(partC binaryTree 5)
```

;Part D

```
(define (partD tree lst)
  (if (null? lst) ; Check if the list is empty
      tree ; return the tree as it is as no elements to insert
  (partD (partD-insert tree (car lst)) (cdr lst)))) ; List not empty, recursively call function to insert elements in the correct place

; Function used in part C to add element to a tree
; Used recursively to add each list element one by one
(define (partD-insert tree el)
  (cond
    ((null? tree) (list el '() '())) ; If the tree is empty, add the element and give it subtrees
    (else
     (let ((leftSubtree (cadr tree)) ; store the left Subtree in a variable
           (value (car tree)) ; Make a variable to get the values in the tree
           (rightSubtree (caddr tree)))  ; store the right subtree in a variable
       (cond
         ((< el value) ; If element is less than value
          (list value (partD-insert leftSubtree el) rightSubtree)) ; Insert into left subtree
         ((> el value) ; If element is greater than value
          (list value leftSubtree (partD-insert rightSubtree el)))  ; Insert into right subtree
         (else tree))))))

(partD binaryTree '(7 13 18))
```

;Part E

```
; Function used to add element to a tree
; Used recursively to add each list element one by one
(define (partE-insert tree el)
  (cond
    ((null? tree) (list el '() '())) ; If the tree is empty, add the element and give it subtrees
    (else
     (let ((leftSubtree (cadr tree)) ; store the left Subtree in a variable
           (value (car tree)) ; Make a variable to get the values in the tree
           (rightSubtree (caddr tree)))  ; store the right subtree in a variable
       (cond
         ((< el value) ; If element is less than value
          (list value (partE-insert leftSubtree el) rightSubtree)) ; Insert into left subtree
         ((> el value) ; If element is greater than value
          (list value leftSubtree (partE-insert rightSubtree el)))  ; Insert into right subtree
         (else tree))))))

; Insert From list into the tree
(define (partE tree lst)
  (if (null? lst) ; Check if the list is empty
      tree ; return the tree as it is as no elements to insert
  (partE (partE-insert tree (car lst)) (cdr lst)))) ; List not empty, recursively call function to insert elements in the correct place

; Traversal Algortihm
(define (traversal-algorithm tree)
  (cond
    ((null? tree) '())
    (else
     (append (traversal-algorithm (cadr tree))
```

```
        (list (car tree))
        (traversal-algorithm (caddr tree))))))

(traversal-algorithm binaryTree)

(define (custom-comparison a b)
  (cond
    ((< a b) #t) ; Return true if a is less than b
    ((> a b) #f) ; Return false if a is greater than b
    ((< (modulo a 10) (modulo b 10)) #t))) ; Return true if last digit of a is less than last digit of b
```

;Part F

```
; Ascending
(define (less a b)
  (< a b))
; Descending
(define (greater a b)
  (> a b))
; Ascending Last digit
(define (asc-last-digit a b)
  (< (modulo a 10) (modulo b 10)))
; Descending Last Digit
(define (des-last-digit a b)
  (> (modulo a 10) (modulo b 10)))

(define (partF lst order)
  (define (insert orderedList el order)
    (cond
      ((null? orderedList) (list el))
      ((order el (car orderedList)) (cons el orderedList))
      (else
       (cons (car orderedList) (insert (cdr orderedList) el order)))))

  (define (insert-lots inp order)
    (cond
      ((null? inp) '()) ; If input is null return an empty list
      (else
        (insert (insert-lots (cdr inp) order) (car inp) order))))
  (insert-lots lst order))

(display "Ascending Order: ")
(display (partF '(4 7 2 9 5) less))
(newline)

(display "Descending Order: ")
(display (partF '(4 7 2 9 5) greater))
(newline)

(display "Ascending based on Last Digit: ")
(display (partF '(14 37 22 95 5) asc-last-digit))
(newline)

(display "Descending based on Last Digit: ")
(display (partF '(14 37 22 95 5) des-last-digit))
(newline)
```

<u>Output</u>

```
4-6-8-10-12-16-20-'()
#t
#f
'(10 (6 (4 () (5 () ())) (8 () ())) (16 (12 () ()) (20 () ())))
'(10 (6 (4 () ()) (8 (7 () ()) ())) (16 (12 () (13 () ())) (20 (18 () ()) ())))
'(4 6 8 10 12 16 20)
Ascending Order: (2 4 5 7 9)
Descending Order: (9 7 5 4 2)
Ascending based on Last Digit: (22 14 5 95 37)
Descending based on Last Digit: (37 5 95 14 22)
```