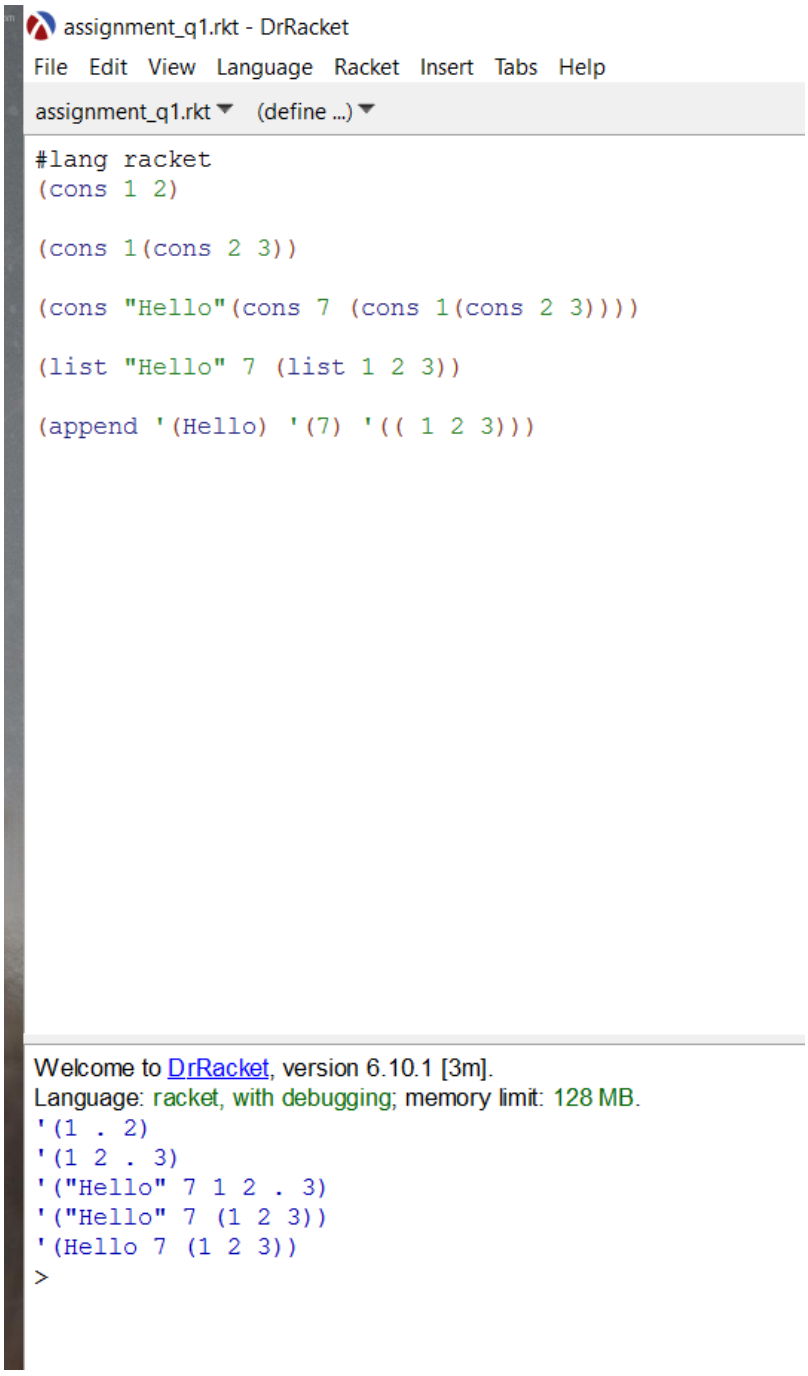
Cormac Buckley 15534413 CT331 Assignment 2

Github: https://github.com/CormacBuckley/ct331_assignment2

Q1.



B)

Cons is the constructor for all pairs. Append is a procedure that uses cons to make a list with all the elements of the argument lists left to right. As I discovered in my assignment, cons accepts strings whereas append does not.

Q2.

File Edit View Language Racket Insert Tabs Help

assignment_q2.rkt (define ...)

```
#lang racket

(provide ins_beg)
(provide ins_end)
(provide count_top_level)
(provide count_instances)
(provide count_instances_tr)
(provide count_instances_deep)

;A
(define (ins_beg el lst)
  (
   display "ins_beg Running\n")

  (cons el lst)
  )

;B
(define (ins_end el lst)
  (
   display "ins_end Running\n")

  (cons lst (list el))
  )

;C
(define (count_top_level list)
  (if (null? list)
      0
      (+ 1 (count_top_level (cdr list)))
  )
)

;D
(define (count_instances el lst)
   (cond ((null? lst)0)
   ((equal? el (car lst))(+ 1(count_instances el (cdr lst))))
    ((count_instances el (cdr lst)))
   )
  )
```

```
;E
(define (count_instances_tr el lst) ;This is the main funciton
  (tinst el 0 lst) ;Main funciton calls this helper funciton to do the 'grunt work' It gets the element to look for, runing total and list
 )

(define (tinst el total lst);This is the helper function
   (cond ((null? lst)0);Check for null
   ((equal? el (car lst))(+ 1 total(tinst el total (cdr lst))));If match found increment the total and recurse in the helper.
    ((tinst el total (cdr lst)));;Recurse the helper
    ;Since the function is being called on known values (el, total and list) there is very little overhead to the recursion.
    )
  )


;F
(define (count_instances_deep el lst); Main Function
  (cond [(empty? lst) 0]; Check empty
        [(list? (car lst)) (+ (count_instances_deep el (car lst)) (count_instances_deep el (cdr lst)))]
        ; if car is a list, recurse inside that car and add the result to the recurse of the rest of the original list(cdr)
        [(equal? el (car lst)) (+ 1 (count_instances_deep el (cdr lst)))]; same code as previous question to increment total
        [else (count_instances_deep el (cdr lst))]))); recurse rest of list
```

```
Welcome to DrRacket, version 6.10.1 [3m].
Language: racket, with debugging; memory limit: 128 MB.
> (ins_beg 1 '(2 3))
ins_beg Running
'(1 2 3)
> (ins_end 1 '(2 3))
ins_end Running
'((2 3) 1)
> (count_top_level '(1 2 3 (4 5 (6))))
4
> (count_instances 3 (1 2 3 3 3 4 5))
application: not a procedure;
 expected a procedure that can be applied to arguments
  given: 1
  arguments...:
> (count_instances 3 '(1 2 3 3 3 4 5))
3
> (count_instances_tr 3 '(1 2 3 3 3 4 5))
3
> (count_instances_deep 3 '(1 2 (3 (3) 3) 4 5))
3
>
```

Q3.

```
#lang racket

(provide tree)
(provide to_Sort)
(provide left_child)
(provide right_child)
(provide val)
(provide sortTree)
(provide present)
(provide addItem)
(provide add_list)
(provide higher_order_add_list)
(provide tree_sort)
(provide higher_order_tree_sort)
(provide higher_order_addItem)

(define tree '((((() 3 ()) 8 (() 11 ())) 19 ((() 25 ()) 29 (() 52 ())))))
(define to_Sort '(2 11 105 66 4 19 47 33))

(define (left_child bst)
  (car bst))

(define (right_child bst)
  (caddr bst))

(define (val bst)
  (cadr bst))
;A
(define (sortTree bst);sort left then sort right
 (begin(cond [(not (empty?(left_child bst))) (sortTree (left_child bst))])
   (printf "~a " (val bst));
```

```
  (cadr bst))
;A
(define (sortTree bst);sort left then sort right
 (begin(cond [(not (empty?(left_child bst))) (sortTree (left_child bst))])
   (printf "~a " (val bst));
   (cond [(not (empty?(right_child bst))) (sortTree (right_child bst))])))

;B
(define (present el bst)
(cond
  [(empty? bst) #f]
  [(equal? el (val bst)) #t]
  [(< el (val bst)) (present el (left_child bst))]
  [else (present el (right_child bst))]
  )
  )

;C
(define (addItem el bst); Check val - go left or right - recurse till null and insert
  (higher_order_addItem el bst <)
  )


;D
(define (add_list lst bst)
  (if (empty? lst) bst
      (add_list (cdr lst) (addItem (car lst) bst))))

(define (higher_order_add_list lst bst left)
  (if (empty? lst) bst
      (higher_order_add_list (cdr lst) (higher_order_addItem (car lst) bst left) left)))



;E
(define (tree_sort lst)
  (sortTree (add_list lst '())))

(define (higher_order_tree_sort lst orderFunc)
  (sortTree (higher_order_add_list lst '() orderFunc)))



;F
(define (higher_order_addItem item bst left)
  (cond [(empty? bst) (list '() item '())]
        [(equal? item (val bst)) bst]
        [(left item (val bst))
         (list (higher_order_addItem item (left_child bst) left) (val bst) (right_child bst))]
        [else (list (left_child bst) (val bst) (higher_order_addItem item (right_child bst) left))]))
```

```
(define (ascending_last_digit a b)
  (< (remainder a 10) (remainder b 10)))

(display "display_sorted:\n")
(sortTree tree)

(display "present_in_tree:\n")
(present 19 tree)
(present 208 tree)

(display "addItem\n")
(addItem 12 tree)

(display "add_list:\n")
(add_list '(4 19 88 99 65) tree)

(display "tree_sort:\n")
(tree_sort to_Sort)

(display "higher_order_tree_sort:\n")
(display "ASCENDING:\n")
(higher_order_tree_sort to_Sort <)
(display "\nDESCENDING:\n")
(higher_order_tree_sort to_Sort >)
(display "\nASCENDING BASED ON LAST DIGIT:\n")
(higher_order_tree_sort to_Sort ascending_last_digit)
```