Lab 2

Recursion in Racket

This is an individual assignment. In this lab assignment, you will work experimentally with the DrRacket programming environment and the DrRacket language.

Part I (3 points per question for Q1 and Q3)

1. Enter and load the following function.

```
(define (mystery L)
    (if (null? L)
     L
    (append (mystery (cdr L))
            (list (car L)))))
```

1) Run this function on the following lists

```
(mystery '(1 2 3))
(mystery '((1 2) (3 4) 5 6))
```

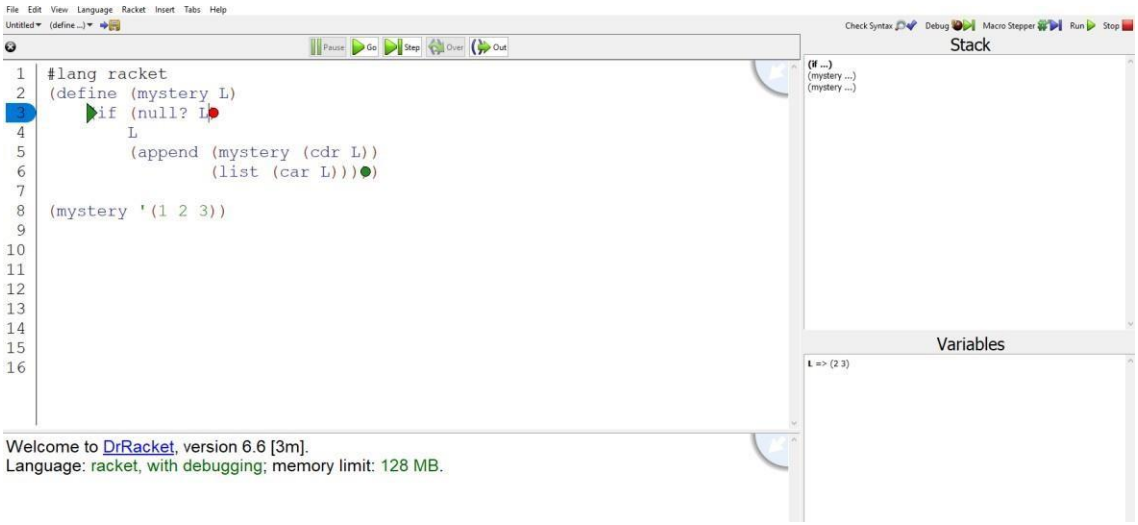
What does this function do? Explain the logic of the function.
effectively reverses any list that is passed to it, whether the elements are atoms or sublists.

2) As you may have noticed, there is no `return` statement here. Explain how the return value is determined in the above function.
In racket, the return value of `mystery` is determined by the result of the `if` expression, as the last evaluated expression is returned automatically.

2. To watch your program in action with the debugger, click the “Debug” button instead of the “Run” button. Then rerun your program by typing (in the definitions window)

```
(mystery '(1 2 3))
```

A series of debugging buttons will appear at the top of the definitions window. Click "Step" repeatedly, and watch the pointer move through your code. Also watch the gray bar to the far left of the debugging buttons. DrRacket will show you the return values for functions when they are called. You can also hover your mouse over variables (hover the mouse over the variable L, for example), and DrRacket will show you those variable values to the right of the debugging buttons. You can also see the stack of function calls and variable values to the right.



You will note a green arrow and circle in the body of mystery. These represent the expression that is currently being evaluated. You should also note the red circle. That represents a breakpoint.

You can use the "Go" button to resume execution of your program. More instructions on debugging can be found in the Racket documentation:

https://docs.racket-lang.org/drracket/debugger.html

3. Modify the program from Question 1 to the follows:

```
(define (mystery L) (if (null? L) L
    (begin
      (displayln L)
     (append (mystery (cdr L))
              (list (car L))))))
```

What does `begin` do? What does `displayln` do?
begin is used inside the if clause to do two things 1) display the current list 2) proceed with the recursive call and list appending. Displayln augments the output to append a newline at the end.

Part II (3 points per question for Q4 to Q7)

4. Write a recursive function named `gen-list` that takes two integer parameters `(gen-list start end)`. This function will generate a list of consecutive integers, from `start` to `end`.

For example: `(gen-list 1 5) ---> (1 2 3 4 5)`

If `start` > `end` then an empty list is generated. See how to compare two numbers:
https://docs.racket-lang.org/heresy/math.html

```
(define (gen-list start end)
  (if (> start end)
    '()
    (cons start (gen-list (+ start 1) end))))
```

5. Write a recursive function named `sum` that takes one list parameter. This function will add up all the elements in the given list and return the result.

   For example:

   ```
   (sum '(4 5 0 1)) ---> 10
   (sum (gen-list 1 5)) ---> 15
   ```

   When the list argument is empty, 0 can be returned.

```
(define (sum lst)
 (if (null? lst)
    0
    (+ (car lst)
       (sum (cdr lst)))))
```

6. Write a recursive function named `retrieve-first-n` that takes two parameters, an integer *n* and a list. The function shall return a list of the first *n* elements in a list.

   For example:

   ```
   > (retrieve-first-n 3 '(a b c d e f g h i)) (a b c)
   ```

   Your code should do something appropriate if n is too big or too small (negative). It doesn't matter to me precisely what it does under these circumstances, so long as it does something reasonable (doesn't crash or return complete nonsense).

   Your function should not use any other Racket functions than those which have been introduced in this lab and lab 1. (You may use the functions <, >, <=, or >=).

```
(define (retrieve-first-n n lst)
 (cond
   [(or (< n 1) (null? lst)) `()]
   [else (cons (car lst)
               (retrieve-first-n (- n 1) (cdr lst)))]))
```

7. Write a recursive function named `pair-sum?` that takes two parameters, a list and an integer `val`. This function tests whether there are any two adjacent values in the given list that sum to the given `val`.

   For example:

```
(pair-sum? '(1 2 3) 3) ---> #t since 1+2=3. Similarly,
(pair-sum? (gen-list 1 100) 1000) ---> #f since no two adjacent integers
in the range 1 to 100 can sum to 1000.
```

The list argument can be generated using the `gen-list` function your coded for Question 4.

```
(define (pair-sum? lst val)
 (cond
[(or (null? lst) (null? (cdr lst))) #f]
[(= (+ (car lst) (cadr lst)) val) #t]
[else (pair-sum? (cdr lst) val)]))
```

Submission

Part I: Submit a PDF file with your answers to questions 1 and 3 on Canvas.

Part II: Prepare a single Racket program file (`lab2.rkt`) containing definitions of all the requested functions. Please make sure to use the requested function names and use comments to explain the parts that are hard to understand.
Submit the file on Canvas.