CSCI 301, Winter 2020

Lab 2 (Racket Programming: Recursion)

DUE: 11:59pm Wednesday, 1/29, Online submission

25 Points Total

• This is an **individual** assignment. Work through the following lab.

• In this lab assignment, you will work experimentally with the DrRacket language on **recursions**. • Keep in mind that in addition to this lab, there are Racket and Scheme resources linked in the syllabus if you need help.

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.

Answer:

This function reverses the order of the elements in the list by appending the first element of the list onto the end of a list missing the first element for each element in the list.

2) As you may have noticed, there is no `return` statement here. Explain how the return value is determined in the above function.

Answer:

The entire function is written as one big expression that can be evaluated. Whatever that expression evaluates to is what is returned

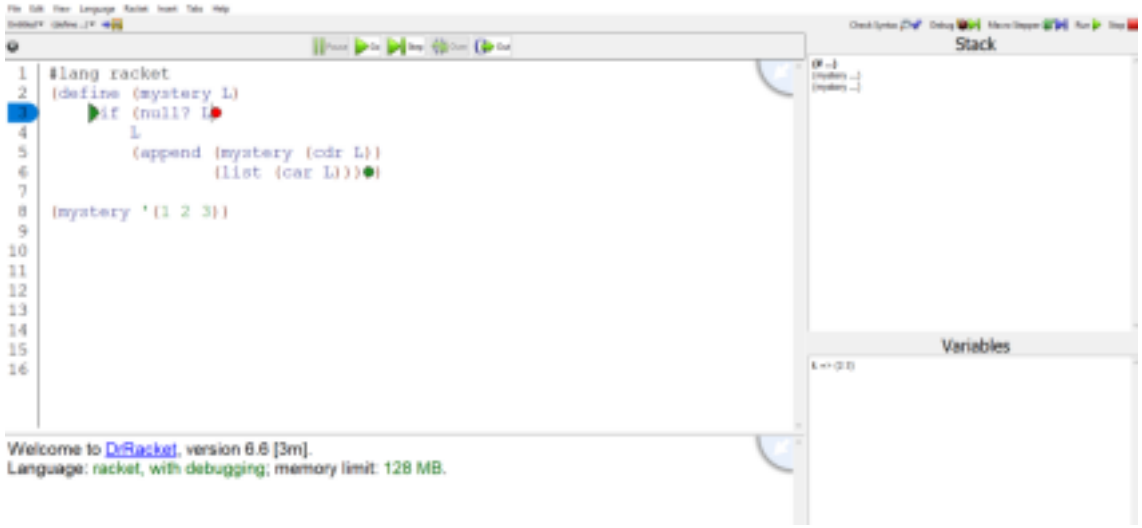
2. To watch your program in action with the debugger, click the “Debug” button instead of the “Run” button. Then rerun your program by typing (in the definitions window)

```
(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 as 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?

Answer:

begin evaluates each expression in it in order, but only returns the value of the last expression. Displayln displays the list in the console.

4. Write a **recursive** function `(gen-list start end)`. This function will generate a list of consecutive integers, from `start` to `end`. (If `start` > `end` then an empty list is generated. See how to compare two numbers: https://docs.racket-lang.org/heresy/math.html).

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

5. Write a recursive function named `sum` that adds up all the elements in a list. For example:

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

Do something reasonable if the list is empty.

6. Write a recursive function, `retrieve-first-n`, that returns a list of the first n elements in a list.

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. [An exception: if you wish, you may use the functions <, >, <=, or >=.]

7. Write a recursive function `pair-sum?` that takes an integer sequence as generated by the `gen list` function in exercise 4 above. This function tests whether any two adjacent values in the given list 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.
```

You must use recursion, and not iteration. You may not use side-effects (e.g. set!).

To turn this assignment in:

1. Exercise 1 and 3: submit a copy of the document with your answers to exercise 1 and 3 on Canvas.
2. Exercise 4 to 7: The solutions will be turned in by posting a single Racket program (lab02. rkt) containing a definition of all the functions specified, (including **gen-list**, etc.).