**CSCI 301, Summer 2019**

**Lab 1 (Racket Introductory Lab)**

**DUE: 11:59pm Monday, 7/8, Online submission**

**30 Points Total**

* This is an individual assignment. Work through the following lab.
* In this lab assignment, you will work experimentally with
  + Part I: the DrRacket **programming environment** and
  + Part 2: the DrRacket language **basics** and 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.

**Part 1 (8pts)**:

* **Part 1.1 Getting started in DrRacket**

Start up DrRacket, which is the programming environment that we’ll be using. If this is the first time you're starting up DrRacket, you'll see a warning about no language being specified. You can ignore it, and all will work; alternatively, you can go to the Languages menu for DrRacket, select "Choose Language," and make sure the first option is selected, which says "Start your program with #lang..."

You will see a pair of windows. The bottom window says "Welcome to DrRacket." That is the **interactions/execution** window. This is where you can test statements to see what they will do. Try it out - in the interactions window, type

(+ 3 5)

This should add 3 to 5. In Racket, + is a function. To call a function in Racket, you place the name of the function and its arguments, separated by spaces, inside parentheses. This takes a little getting used to! In most programming languages, function calls look something like this:

function(arg1, arg2, arg3)

In Racket, function calls look like this:

(function arg1 arg2 arg3)

Change your program in some way. For example, re-type it as

(+ 3 7)

Experiment with hitting the Esc key followed by either the p or the n keys, which allow you to move backwards and forwards through your command history.

To run a program in DrRacket, either (1) type it into the edit (upper) window and then press Run (upper right), or type it into the interactions window and follow it with a new line (enter).

* **Part 1.2 Basic Racket Primitives**

1. In the interactions window, enter the following

(car '(apple orange pineapple))

(cdr '(apple orange pineapple))

(car (apple orange pineapple))

What do car and cdr do? Why is the single quote necessary? What does it do? That last line of the above should cause an error: why?

**Answer**:

|  |
| --- |
| car returns the first element in a pair, and cdr returns the second part / the remainder of the pair. The single quote is necessary to declare the type of the pair. It allows us to write the list as an expression. The last line is causing an error because we are not declaring that (apple orange pineapple) is a list, and the car function cannot fetch the first element of something that isn’t a list. |

1. Write sequences of cars and cdrs that will pick the symbol ‘pear out of the following expressions:

(apple orange pear grapefruit)

(((apple) (orange) (pear) (grapefruit)))

(apple (orange) ((pear)) (((grapefruit))))

**Answer**:

|  |
| --- |
| (caddr '(apple orange pear grapefruit))  (car (caddar '(((apple) (orange) (pear) (grapefruit)))))  (car (caaddr '(apple (orange) ((pear)) (((grapefruit)))))) |

1. Execute the following statements. What do the functions cons, append, and list do?

;; This is a comment, by the way!

(cons 'x '(1 2))

(cons '(1 5) '(2 3))

(append '(1) '(2 3))

(append '(1 5) '(2 3))

(list '1 '2 '3 '(4 5))

**Answer**:

|  |
| --- |
| These functions construct lists based off of the parameters given to them. |

1. Execute the following code. What do length and reverse do?

(length '(plato socrates aristotle))

(reverse '(plato socrates aristotle))

(member 'socrates '(plato socrates aristotle))

(member 'raphael '(plato socrates aristotle))

**Answer**:

|  |
| --- |
| Length returns the number of elements in the list and reverse returns the list but with the elements in the reverse order. |

* **Part 1.3: Saving your code**

Entering your code interactively is fun, but not a good idea for creating large programs. A better way to go is to write your code, save it, then run it. Here's how to do it.

1. Start typing in some Racket code in the **definitions** window at the top of the screen. Make sure that the first line says:

#lang racket

Use any of the above examples that you wish. When finished, save your program by going to the File menu, and choosing Save Definitions.

1. Run your program by clicking on the clicking on the Run button, or by using the combination Ctrl-T.

You should generally use this approach for entering and running Racket code, but entering code directly into the interactions window is good for testing out quick ideas.

* **Part 1.4: Conditionals**

Racket has a number of different predicates for testing equality.

1. Try this code:

(equal? '(hi there) '(hi there))

(equal? '(hi there) '(bye now))

(equal? 3 3)

(equal? 3 (+ 2 1))

(equal? 3 3.0)

(equal? 3 (/ 6 2))

(equal? -1/2 -0.5)

(eqv? '(hi there) '(hi there))

(eqv? '(hi there) '(bye now))

(eqv? 3 3)

(eqv? 3 (+ 2 1))

(eqv? 3 3.0)

(eqv? 3 (/ 6 2))

(eqv? -1/2 -0.5)

(= 3 3)

(= 3 (+ 2 1))

(= 3 3.0)

(= 3 (/ 6 2))

(= -1/2 -0.5)

(= '(hi there) '(hi there)) ;; yes, this will give an error

What kind of responses do you get? How are equal?, eqv?, and = different?

**Answer**:

|  |
| --- |
| I get boolean responses from these statements. ‘equal?’ checks if the two parameters have the same value and same precision, eqv? Checks if they are the same object, = checks if the two elements have the same value but does not check for precision like equal?. |

1. Enter the following code:

(if (equal? 8 3)

9

10)

Modify the condition following if to get a different value to return.

[Note that Racket pays no attention whatsoever to how you indent your code. The above indenting is stylistically useful to see what the "if" function is doing.] For textual conventions in Racket programming, refer to <https://docs.racket-lang.org/style/Textual_Matters.html>

1. Enter the following code:

(cond ((equal? 16 3) (+ 3 8))

((equal? 16 8) 12)

(else (\* 6 3)))

Try to replace all of the 16's in the above code with some other value to change the return value? What does the cond function do?

**Answer**:

|  |
| --- |
| The cond function allows us to have a control structure, we can use conditionals discussed in the previous parts of this task to create if, else if, and else statements which allows us to control what value is returned by the cond function. |

* **Part 1.5: Defining functions**

1. In a new file, enter in the following code, save it, then run it.

(lambda (x)

(+ x 1))

What does Racket return? The lambda function returns a *function* without a name. In this case, you have defined a function that takes a parameter and adds 1 to it. Functions are also called *procedures*.

1. A function without a name is useless, as it is immediately garbage collected. Try this instead:

(define add-one

(lambda (x)

(+ x 1)))

Save this code to a file called “add-one.rkt”, run it, then type (add-one 5) in the interactions window. The define statement created a *pointer*, called add-one, which points to the function you just created.

1. Try this out:

(define another-add-one add-one)

(another-add-one 5)

At the pointer level, what is happening here? Draw a picture in the answer box indicating what is happening.

**Answer**:

|  |
| --- |
| The pointer is a way to reference the procedure created by the lambda function to avoiding it being picked up by the trash collector. By using the pointer we put the function on the stack which will allow it to be runned    Another-add-one  Stack  <----  Procedure created by lambda function |

1. You can declare "local" variables in Racket via the use of the let function. For example, try the following code:

(define a 5)

(define b 6)

(define c 7)

(define strange

(lambda (x)

(let ((a 1) (b 2))

(+ x a b))))

After executing this code, what are the values of a, b, and c? What about what you get when you make the call (strange 3)? Why?

**Answer**:

|  |
| --- |
| A = 1 and b = 2 c = 7. when you call (strange 3) you get 6. This is because the function strange has local variables a and b and uses those values instead of the global variables defined above the function. This function returns the sum of x and the two local variables a and b |
|  |

**Part 2 (22pts)**:

1. Enter and load the following function.

(define mystery

(lambda (L)

(if (null? L)

L

(append (mystery (cdr L))

(list (car L))))))

1. What does this function do? Why?

**Answer**:

|  |
| --- |
| This function checks if the list parameter is null, if it is not it will create a copy of the list provided but the elements are in the reverse order. |

1. As you may have noticed, there is no return statement here. Why? How is the return value determined in the above function?

**Answer**:

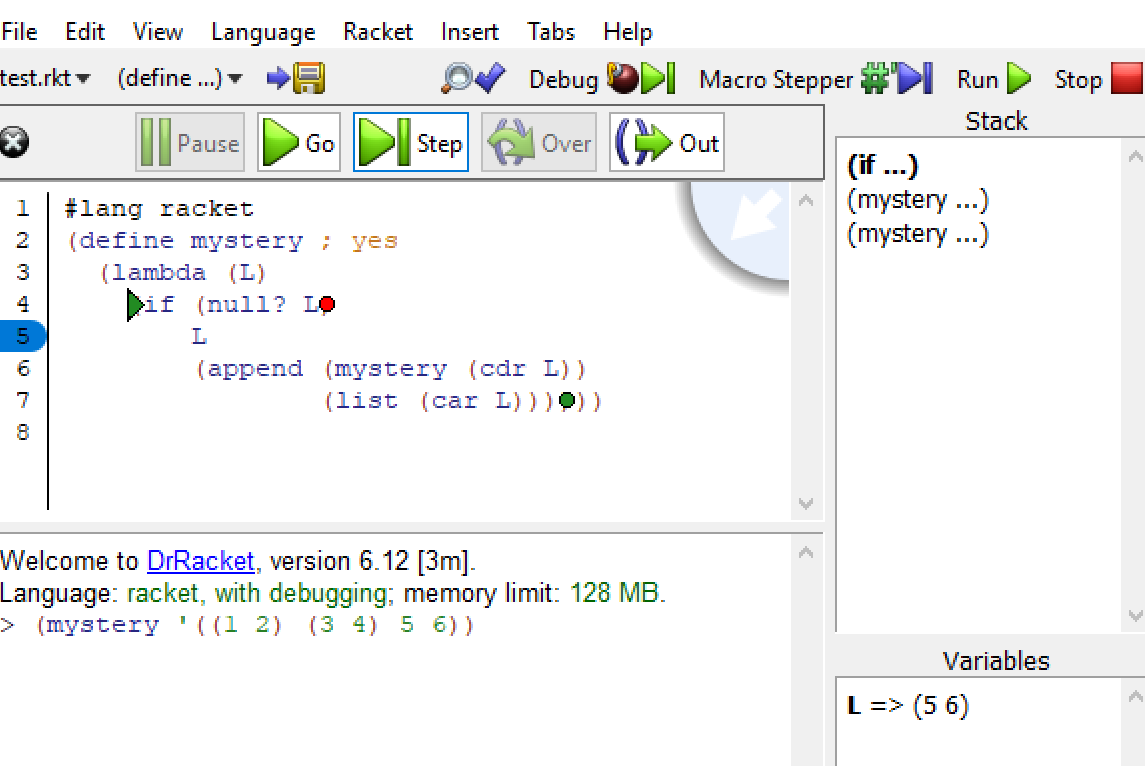
|  |
| --- |
| The result is the value that was found last, this function is putting the value on the stack and the last value on the stack when the function finished running is it’s result. |

1. 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 interactions window)

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

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>

1. Modify the program as follows:

(define mystery

(lambda (L)

(if (null? L)

L

(begin

(displayln L)

(append (mystery (cdr L))

(list (car L)))))))

What does begin do? What does displayln do?

**Answer**:

|  |
| --- |
|  |

1. 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.) For example:

(gen-list 1 5) ---> (1 2 3 4 5)

1. 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.

1. 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 >=.]

1. 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. Part 1, and Part 2 Exercise 1 and 3: submit a copy of the document with your answers to exercise 1 and 3 on Canvas.
2. Part 2 Exercise 4 to 7: The solutions will be turned in by posting a single Racket program (lab01. rkt) containing a definition of all the functions specified, (including **gen-list**, etc.).