Lab 1

Introduction to Racket

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

Part 1: Get started in DrRacket

1. Start DrRacket, which is the programming environment that we'll be using. If this is the first time that 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..."
   ```

2. 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 first and then 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_name(arg1, arg2, arg3)
   ```

   In Racket, function calls look like this:

   ```
   (function_name 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.

3. 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 2: Basic Racket Primitives

1. In the interactions window, enter the following:

   ```
   (car '(11 12 13 14))
   (car '(a b c d))
   (cdr '(11 12 13 14))
   (cdr '(a b c d))
   (car (11 12 13 14))
   (cdr (a b c d))
   ```

   What do `car` and `cdr` do? That last line of the above should cause an error. Explain why the single quote necessary `and` what it does.

Car returns the first element of a list, and cdr returns the rest of the list excluding the first element. The single quote is necessary to prevent evaluation of the list; otherwise lang would interpret it as a function.

2. Write sequences of `car`s and `cdr`s that will pick the symbol `'x` out of the following expressions:

```
(x y z m)
(y x z m)
(y z m x)
((y)(x)(z)(m))   ((y
z) (m x))
```

`(car (cdr (cdr (cdr ‘(x y z m)))))`
`(car (cdr (cdr ‘(y x z m))))`
`(car (cdr (cdr ‘(y z m x))))`
`(car (cdr ‘(x)))`
`(car (cdr (cdr ‘(m x))))`

3. Execute the following statements. Write down the return value for each statement and explain what each of the functions `cons`, `append`, and `list` does.

```
 ;; This is a comment, by the way!
(cons 3 '(1 2))
(cons '(1 5) '(2 3))   (list
3 '(1 2))
(list '(1 5) '(2 3))
(append '(1) '(2 3))
(append '(1 5) '(2 3))
(cons 'x '(1 2))
(list 1 2 3 '(4 5))
(cons '1 '2 '3 '(4 5))
```

Cons: function constructs a new list by prepending an element to an existing list
Append: combines two lists together
List: creates a new list with the provided elements
(cons 3 ‘(1 2)): returns value: `(3 1 2) `
(cons '(1 5) '(2 3)): `((1 5) 2 3)`
(list 3 '(1 2)): `(3 (1 2))`
(list '(1 5) '(2 3)): `((1 5) (2 3))`
(append '(1) '(2 3)): `(1 2 3)`
(append '(1 5) '(2 3)): `(1 5 2 3)`
(cons 'x '(1 2)): `(x 1 2)`
(list 1 2 3 '(4 5)): `(1 2 3(4 5))`
(cons '1 '2 '3 '(4 5)): returns an error

4. Execute the following code. Write down the return value for each statement and explain what each of the functions `length`, `reverse` and `member` does.

```
(length '(a b c))
(reverse '(a b c))
(member 'a '(a b c))
(member 'b '(a b c))
(member 'c '(a b c))
(member 'd '(a b c))
```

Length: returns the number of elements in a list
Reverse: returns a new list with the elements of the original list in reverse order

member: checks if an element is a “member” of a list , returning the rest of the list starting from that element if it is found
(length '(a b c)): 3
(reverse '(a b c)): (c b a)
(member 'a '(a b c)): (a b c)
(member 'b '(a b c)): (b c)
(member 'c '(a b c)): (c)
(member 'd '(a b c)): #f

Part 3: Save 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.

2. 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 4: Conditionals

Racket has several different predicates for testing equality.

1. Try this code:

   ```
   (equal? '(hi there) '(hi there))
   (eqv? '(hi there) '(hi there))
   (= '(hi there) '(hi there)) ;; yes, this will give an error
   (equal? '(hi there) '(bye now))
   (eqv? '(hi there) '(bye now))
   (equal? 3 3)
   (eqv? 3 3)
   (= 3 3)
   (equal? 3 (+ 2 1))
   (eqv? 3 (+ 2 1))
   (= 3 (+ 2 1))
   (equal? 3 3.0)
   (eqv? 3 3.0)
   (= 3 3.0)
   (equal? 3 (/ 6 2))
   (eqv? 3 (/ 6 2))
   (= 3 (/ 6 2))
   (equal? -1/2 -0.5)
   ```

```
(eqv? -1/2 -0.5)
(= -1/2 -0.5)
```

Write down the return value for each of the statement and explain the difference among `equal?`, `eqv?`, and `=`.

Equal: checks if the structure and the contents of two lists is the same

eqv: compares two objects in memory

=: compares numberical equality

(equal? '(hi there) '(hi there)) : #t

(eqv? '(hi there) '(hi there)): #f

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

(equal? '(hi there) '(bye now)): #f

(eqv? '(hi there) '(bye now)): #f

(equal? 3 3): #t

(eqv? 3 3): #t

(= 3 3): #t

(equal? 3 (+ 2 1)): #t

(eqv? 3 (+ 2 1)): #t

(= 3 (+ 2 1)): #t

(equal? 3 3.0): #f

(eqv? 3 3.0): #f

(= 3 3.0): #t

(equal? 3 (/ 6 2)): #t

(eqv? 3 (/ 6 2)): #t

(= 3 (/ 6 2)): #t

(equal? -1/2 -0.5): #t

(eqv? -1/2 -0.5): #f

(= -1/2 -0.5): #t

2. 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

3. Enter the following code:

```
(cond ((equal? 16 3) (+ 3 8))
 ((equal? 16 8) 12)
(else (* 6 3)))
```

Write the return value for the above code. If replace all of the 16's in the above code with 8, what is the return value? What about replace the 16’s with 3? What does the `cond` function do?

```
 (cond ((equal? 16 3) (+ 3 8)): #f
   ((equal? 16 8) 12): #t

(else (* 6 3)))
```

The output does not change if you replace all the 16s with 8s, but with all 16s replaced with 3s, the output is 11.

Part 5: Define 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 is an anonymous *function*. In this case, you have defined an anonymous function that takes a parameter `x` and adds 1 to it. Functions are also called *procedures*. Without parameter(s), the above function cannot do much. It can be used this way by given arguments:

   ```
   ((lambda (x)
    (+ x 1)) 3)
   ```

   Run this, and give the result.

   4

2. Define named functions:

   ```
   (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. Run this, and give the result.
   Run the following, and give the result. Does it serve the same purpose as the previous one?

   ```
   (define (add-one x)
    (+ x 1))
   ```

   Yes

3. Add the following lines in the same file and run it. Given the result.

   ```
   (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.

This functionally adds 1 to 5, resulting in 6.

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

```
(define a 2);;binding a variable to a value
(define b 3)
(define c 4)
(define (strange x)
 (let ((a 1) (b 2))
(+ x a b))))
```

After executing this code, what are the values of a, b, and c? What is the return value when you make the call `(strange 4)`? Explain your answers.

A: 2
B: 3
C:4
When you call strange 4, you are adding 4 to the local (not global) variables of a(+1) and b (+2).
4+1=5+2=7, so calling strange 4 returns the value of 7.

Submission

Please submit your answers to Part 2, 4, and 5 in a PDF file on Canvas.