Q1: let, let*, letrec, let!!

What do each of the following evaluate to? If they result in an error, explain why:

a.
```
(define y 7)
(let ([x 5] [y 6])
    (+ x y))
```

b.
```
(let ([a (lambda (x y) (if (equal? x y) 1 0))])
  (a 1 2))
```

c.
```
(let* ([a (lambda (x y) (if (equal? x x) (+ x 1) (a (- x 1))))])
  (a 1))
```

d.
```
(letrec ([a (lambda (x) (if (equal? x x) (+ x 1) (a (- x 1))))])
  (a 1))
```

e.
```
(let* ([a 12] [b 3] [c (+ a b)])
  (let ([a 3])
    (+ a c)))
```

f.
```
(define (f a b) (if (a (+ b b) (* b b)))
(let ([x (f #t 2)])
  (f #f x))
```

Q2: I like curry … and currying

Write a Racket macro my-let that supports:

(my-let ([key value] …) (expressions)…)

For Example:

(my-let () (printf “cse”)) => “cse”
(my-let ([x 10] [y 100]) (+ x y)) => 110
(my-let ([x 10] [y 100] [z 1]) (if (equals x y) z (+ x y))) => 110

Hint: lambda function is your friend

Q3: Haskell One-liners

Write Haskell functions and their type definition for the following questions!
All of them should just take one line! (Well, ok, with type definitions they’ll take two lines, but the function itself should be implemented in only one line.)

a. Write a function xylist that takes an integer x and an integer y and returns an infinite list whose first element is x and whose nth element equals its n-1th element + y.

b. Write a function revstr that takes a list of strings and returns a list where each string is in reverse order; so revstr [“Spyro”, “the”, “Dragon”] should return [“orypS”, “eht”, “nogarD”]. Make this function pointfree.

c. Write a function toonegative that takes a list of Integers and returns a list of all the integers >= 0 in the input list, in their original order. Make this function pointfree.

Q4: May the 4ths be with you

For the Following infinite lists, what will be the first 4 outputs other than those provided?

```
a = 1 : 2 : map (\x-> if x `mod` 2 == 0  then x + 1  else  x - 1)  a
```

```
b = zip a [1, 3..]
```

Q5: Datatypes FTW

Consider the following datatype:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
        deriving (Show,Read)
```

a. Consider a function “search” that takes a Tree and an element and returns True if that element is in the Tree. What’s the most general type of this function?

b. Write the search function. Assume the Tree is sorted – that is, if you have a Tree instance (Node x L R), every Node in L is less than x and every Node in R is greater than x – and ensure your function is tail recursive. Does this assumption change the type of the function? If so, write the new type of search above your function definition.

Q6: Macros are quite silly

Suppose we have a structure representing a key value pair:

(struct kvpair (key value) #:mutable #:transparent)

a. Write a silly racket macro called silly-find that takes in a key k and an **arbitrary number** of kvpairs and returns the first value v that k is paired with in the kvpairs, or ‘() if no values exist. The following examples illustrate the syntax:

(silly-find key: 5 pairs: (kvpair 1 2) (kvpair 3 4)) (Returns ‘())
(silly-find key: 5 pairs: (kvpair 5 “jello”)) (Returns “jello”)

b. Write a function map-pairs that takes in a function and a list of kvpairs and **mutates** the pairs such that the new value for each pair is the function applied to their original values.

Q7: Our 8 tentacled friend

Write a case for the OCTOPUS eval function to handle or. You can use a helper function if needed. Your code should have OCTOPUS handle or exactly as in Racket: it can take 0 or more arguments, and does short-circuit evaluation.

For example:

    (or #f (+ 10 10) 3 #t)

evaluates to 20 (Be sure you only evaluate (+ 10 10) one time!)

Here is the header for the new case:
eval (OctoList (OctoSymbol "or" : args)) env = …..