Programming Assignment #1
Functional Programming Racket

CS-4337 Organization of Programming Languages

Define and test the functions described below. In doing the assignment, you may assume that the inputs to your functions have the correct data type. This means that you need not check the inputs for validity (i.e. type checking). However, your function ***must*** behave properly on all instances of valid input data types. You may define helper functions that are called by your primary function. In some cases below you may be restricted to which helper functions you may use from the Racket standard library.

You must add an export statement at the top of your code, e.g.

```
(provide (all-defined-out))
```

so that your functions may be imported and called by another script.

Your implementation of each function must also have its function name spelled *exactly* as it is in the description.

Any helper functions must be embedded inside the primary function to encapsulate them against being called outside of context.

All functions must use *explicit* lambdas for their arguments. Syntactic sugar of having implied lambdas is not permitted.

Your submission should consist of a single Racket source code file. This may be accomplished by defining your functions in the top pane of the Dr. Racket IDE and then using "Save Definitions". Your Racket source file should be named using your NetID. Example: `cid021000.rkt`. Upload this file to eLearning.

You may find Racket language help here:

- http://docs.racket-lang.org/reference/
- http://docs.racket-lang.org/guide/

1. divisible-by-x?

Define *your own* Racket function that takes an integer as an argument and returns a function that indicates whether its integer argument is evenly divisible by the first. You do not have to perform data type checking on the input. You may assume that the input is an integer.

Input: An integer.

Output: A function.

Example:

```
> ((divisible-by-x? 3) 12)
#t

> ((divisible-by-x? 7) 20)
#f

> (define div-by-5 (divisible-by-x? 5))
> (div-by-5 15)
#t
```

2. function-4

Define a function that takes a function as an argument and passes the number 4 to that function. The function argument must be able to accept a single integer as its argument.

Input: A named function which takes a single number as an argument.

Output: The value returned by applying the named function to the number 4.

Example:

```
> (function-4 sqrt)
2

> (function-4 add1)
5

> (function-4 (lambda (x) (+ x 7)))
11
```

3. my-map

Define *your own* Racket function that duplicates the the functionality of **map** from the standard library. You *may not* use the built-in **map** function as a helper function. You *may not* use any of the the built-in **fold** functions as a helper functions, i.e. *Your implementation must be recursive*.

Input: The input to **my-map** is a function that takes a function and a homogeneous list of elements of the same data type compatible with the function. Note: the function argument can be named or anonymous (lambda).

Output: A new list of the original elements with the same procedure applied to each.

Example:

```
> (my-map sqrt '(9 25 81 49))
'(3 5 9 7)

> (my-map add1 '(6 4 8 3))
'(7 5 9 4)

> (my-map (lambda (n) (* n n)) '(5 7))
'(25 49)

> (my-map even? '(2 5 7 12))
'(#t #f #f #t)
```

4. pair-up

Define a function takes two lists as arguments and returns a single list of pairs (i.e. two element sublists). The the first pair should be the both first elements from the respective lists. The second pairs should be the second elements from the respective lists, and so on. If one input list is longer than the other, extra elements of the longer list are ignored. *Your implementation must be recursive*.

Input: Two lists of elements of any type, potentially heterogenous. The two lists do not have to be the same length.

Output: A new list whose elements are each two-element sublists. The first sublist is composed of the first elements from two input lists respectively, the second sublist is composed of the second elements form the two input lists respectively, etc. If one list is longer than the other, extra elements of the longer list are ignored.

Example:

```
> (pair-up '(1 2 3 4) '(a b c d))
'((1 a) (2 b) (3 c) (4 d))

> (pair-up '(1 2 3) '(4 9 5 7))
'((1 4) (2 9) (3 5))

> (pair-up '(3 5 6) '("one" 6.18 #t "two"))
'((3 "one")(5 6.18)(6 #t))

> (pair-up '(5) '())
'()
```

5. classify

Define a function that takes a procedure that executes a Boolean test on an atomic value and a list of elements as arguments. It should returns a list containing exactly two sublists. You *may not* use the built-in **`filter`** function as a helper function. You *may* define your own helper functions. *Your implementation must be recursive*.

> **Input**: Two arguments… (1) a function that takes a single element and returns a Boolean, and (2) a list of elements whose element types are compatible with the single element from the first argument.
>
> **Output**: A new list with two sublists. The first sublist contains the elements from the original list that return true (**`#t`**) and the second sublist contains the elements from the original list that return false (**`#f`**).
>
> **Example**:

```
> (classify even? '(7 2 3 5 8))
'((2 8) (7 3 5))

> (classify integer? '(3.0 -5.2 8 16 99.7))
'((3.0 8 16) (-5.2 99.7))

> (classify real? '())
'(() ())
```

6. is-member?

Define a function that takes two arguments, a list and an expression, which may be atomic or a list. Your function should return true (**`#t`**) if the element is a member of the list and false (`#f`) if it does not. You may not use the built-in **`member`** or **`element`** functions as helper functions. *Your implementation must be recursive*.

> **Input**: A single item of any data type and a (potentially heterogenous) list of elements of any data type.
>
> **Output**: A boolean value that indicates whether the input item is a member of the input list.
>
> **Example**:

```
> (is-member? 6 '(4 8 6 2 1))
#t

> (is-member? 7 '(4 8 6 2 1))
#f

> (is-member? "foo" '(4 5 #f "foo" a))
#t

> (is-member? '(3 4)  '(4 5 #f "foo" (3 4)))
#t
```

7. is-sorted?

Define a function that takes two arguments—a comparison function and a list. It should return a boolean (i.e. #t or #f) indicating whether the list is sorted according to the comparison function. You may not use the built-in **`sorted?`** function. *Your implementation must be recursive*.

Input: A comparison function and a list of elements whose values are compatible with the comparison function.

Output: A boolean value that indicates whether the elements of the list are sorted according to the comparison function.

Example:

```
> (is-sorted? < '(2 5 6 9 11 34))
#t

> (is-sorted? < '(7 25 4 15 11 34))
#f

> (is-sorted? string<? '("alpha" "beta" "gamma"))
#t

> (is-sorted? string<? '("john" "zack" "bob"))
#f
```

8. my-flatten

Define your own Racket function that duplicates the the functionality of **`flatten`** from the standard library. *You may not use the built-in* **`flatten`** *function* as a helper function. It should take a list containing zero or more sublists as an argument. Each sublist may contain an arbitrary level of nesting. It should return a single list containing all of the items from all nested levels with *no* sublists. *Your implementation must be recursive*.

Input: A single list which may contain an arbitrary number of elements and sublists, each sublists may also contain an arbitrary number of elements and sublists, nested to an any depth.

Output: A new single-level list which contains all of the atomic elements from the input list.

Example:

```
> (my-flatten '(1))
'(1)

> (my-flatten '((1 2) 3))
'(1 2 3)

> (my-flatten '(((4 3) 6)((7 2 9)(5 1))))
'(4 3 6 7 2 9 5 1)
```

9. my-list-ref

Define your own Racket function that duplicates the the functionality of **`list-ref`** from the standard library. You may not use the built-in **`list-ref`** function as a helper function.

Define a function that takes a list and an integer. The function should return the list element at the integer number (first list position is index "0"). If the integer is larger than the index of the last list member, it should display an "index out of bounds" message. *Your implementation must be recursive*.

Input: A list of elements of any data type, potentially heterogenous, and a single integer.

Output: A single element from the original list that is at the "index" indicated by the integer. The first list position is position "0", the second list position is "1", etc. If the integer is greater than the number of list elements, the function should throw an error (using the **`error`** function) with the message string "ERROR: Index out of bounds".

Example:

```
> (my-list-ref '(4 7 9) 0)
4
> (my-list-ref '(4 7 9) 1)
7
> (my-list-ref '(4 7 9) 3)
ERROR: Index out of bounds
```

10. deep-reverse

Define a function similar to the built-in **`reverse`** function, except that it acts recursively, reversing the order the members of any nested sublists. You may *not* use the built-in **`reverse`** function as a helper function. However, you *may* use your own **`my-reverse`** function as a helper function.

Input: A single list which may contain an arbitrary number of elements and sublists, each sublists may also contain an arbitrary number of elements and sublists, nested to an any depth.

Output: A new list which contains all elements in reverse order, as well as recursively reverse order all members of sublists.

Example:

```
> (deep-reverse '(((4 3) 6)((7 2 9)(5 1))))
'(((1 5) (9 2 7)) (6 (3 4)))

> (deep-reverse '((1 2) 3))
'(3 (2 1))

> (deep-reverse '((4 5)))
'((5 4))

> (deep-reverse '(3 6 9 12))
'(12 9 6 3)
```