Programming Assignment #1
Racket

CS-4337 – Organization of Programming Languages

(Due: 11:59PM July 13, 2015)

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, unless specified otherwise. 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 auxiliary functions that are called by your primary function. In some cases below you may be restricted to which auxiliary functions you may use from the Racket standard library.

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

Your submission should consist of a Racket source code file that is archived using a zip utility. This may be accomplished by defining your functions in the top pane of the Dr. Racket IDE and then using "Save Definitions". Both your Racket source file and zip file should be named using your NetID. Example: `cid021000.rkt` and `cid021000.zip`. 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. my-reverse

Define *your own* Racket function that duplicates the functionality of **`reverse`** from the standard library. You may not use either the built-in **`reverse`** or **`append`** as auxiliary functions. Your implementation must use recursion.

Input: A list of elements of any data type, potentially heterogenous.

Output: A new list of the original elements in reverse order.

Example:

```
> (my-reverse '(2 6 3 7))
'(7 3 6 2)

> (my-reverse '(3 2.71 "foo" 5))
'(5 "foo" 2.71 3)

> (my-reverse '(7))
'(7)

> (my-reverse '((1 2) 3 (4 5))
'((4 5) 3 (1 2))
```

2. 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 an auxiliary function. Your implementation must use recursion.

Input: A function name (of a function that takes a single argument) and a list of elements of the same data type compatible with the function.

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

Example:

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

> (my-map sub1 '(6 4 8 3))
'(5 3 7 2)

> (my-map sqr '(5 7))
'(25 49)
```

3. function-3

Define a function that takes a function as an argument and passes the number 3 to that function.

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

Output: The value returned by applying the named function to the number 3.

Example:

```
> (function-3 sqrt)
1.7320508075688772

> (function-3 sqr)
9

> (function-3 add1)
4
```

4. zipper

Define a function takes two lists as arguments and returns a single list of pairs (i.e. two-items lists. The the first pair should be the both first items from the respective lists. The second pairs should be the second items from the respective lists, and so on. If one 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:

```
> (zipper '(1 2 3 4) '(a b c d))
'((1 a) (2 b) (3 c) (4 d))

> (zipper '(1 2 3) '(4 9 5 7))
'((1 4) (2 9) (3 5))

> (zipper '(3 5 6) '("one" 6.18 #t "two"))
'((3 "one")(5 6.18)(6 #t))

> (zipper '(5) '())
'()
```

5. segregate

Define a function that takes a list of integers as an argument and returns a list containing two sublists, the first sublist containing the even numbers from the original list and the second sublist containing the odd numbers from the original lists. You may not use the built-in functions **`filter`**, **`even?`**, or **`odd?`** as auxiliary functions. However, you may define your own auxiliary functions that duplicate built-in functions. Your implementation must be recursive.

Input: A lists of integers.

Output: A new list with two sublists. The first sublist contains the even numbers from the original list and the second sublist contains the odd numbers.

Example:

```
> (segregate '(7 2 3 5 8)
'((2 8) (7 3 5))

> (segregate '(3 -5 8 16 99))
'((8 16) (3 -5 99))

> (segregate '())
'(() ())

> (segregate '(4 8 2))
'(() (4 8 2))
```

6. is-member?

Define a function that takes two arguments, a list and a single value. Your function should return true (`#t`) if the value 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 auxiliary functions. Your implementation must be recursive. Note: **`is-member`** behaves slightly differnt than the built-in **`member`** in the case of its return value types.

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

Output: A boolean value that indicates whether the input atom 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
```

7. my-sorted?

Define a function that takes a list as an argument. It should return a boolean (i.e. #t or #f) indicating whether the list is sorted in ascending order. You may not use the built-in **`sorted?`** function or any other built-in sort test functions. Data type check: If a list argument contains a heterogenous mix of strings and numbers, your function should display a warning message and terminate. Your implementation must use recursion.

Input: A list of elements of homogenous data type, *either* numbers *or* strings.

Output: A boolean value that indicates whether the elements of the list are sorted in strictly increasing order.

Example:

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

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

> (my-sorted? '("alpha" "beta" "gamma"))
#t

> (my-sorted? '("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 an auxiliary 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. You may not use the built-in **`flatten`** function as an auxiliary function. 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. threshold

Define a function that takes three arguments: a comparison operator, a number (the threshold), and list of numbers. It should return a new list that has the same numbers as the input list, but with only the elements that meet the threshold criteria. You may not use the built-in functions **`filter`** or **`map`** as an auxiliary function. Your implementation must be recursive.

Input: a comparison operator, a number (the threshold), and list of numbers.

Output: A new list of numbers that contains only the numbers from the original list that meet the comparison and threshold criteria.

Example:

```
> (threshold < 6 '(3 6.2 7 2 9 5.3 1))
'(3 2 5.3 1)

> (threshold <= 4'(1 2 3 4 5))
'(1 2 3 4)

> (threshold > 5.5 '(4 8 5 6 7))
`(8 6 7)

> (threshold = 2 '(8 3 5 7))
'()
```

10. 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 an auxiliary 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 **`display`**, **`write`**, **`print`**, or **`printf`** the string "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) 2)
9

> (my-list-ref '(4 7 9) 3)
index out of bounds
```

OPTIONAL BONUS. 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 an auxiliary function. However, you *may* use your own **`my-reverse`** function as an auxiliary 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)
```