Programming Assignment #1
Racket

CS-4337 – Orgranization of Programming Languages

(Due: March 19, 2014)

Define and test the functions described below. In doing the assignment, you may assume that the inputs to your functions have the correct 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 required input types. Your implementation of each function must be spelled *exactly* as it is in the description.

You should also create test input cases that include at least *three* different test inputs for each function.

Your submission should consist of two files: (1) the required function definitions, including comments and any necessary auxiliary functions. All function definitions should be submitted in a single .rkt file. This may be accomplished by defining your functions in the top pane of the Dr. Racket IDE and then using "Save Definitions", and (2) a transcript of your tests submitted as a single .rkt file. This may be accompished by saving the contents of your bottom pane using the "Save Interactions" option. Your two files should be combined using zip, gzip, or tar and named with the convention <netid>_hw1.<archive>. Example: `cid021000.zip`. Upload this file to eLearning.

As mentioned above, you may define auxiliary functions that are called by your main function. For example, the standard mergsort algorithm uses both "split" and "merge" auxiliary functions. In some cases below you may be restricted to which auxiliary functions you may use from the Racket standard library.

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 **`reverse`** as an auxiliary function.

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

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

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

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

2. my-map

Define *your own* Racket function that duplicates the the functionality of **`map`** from the standard library. You may not use **`map`** as an auxiliary function.

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.

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

> (my-map double '(6 4 8 3))
'(12 8 16 6)
```

3. function-3

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

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

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

```
> (function-3 sqrt)
1.7320508075688772

> (function-3 square)
9

> (function-3 add-one)
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.

> **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 repsectively, etc. If one list is longer than the other, extra elements of the longer list are ignored.

```
> (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.

> **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.

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

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

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

6. in-list?

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.

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.

```
> (in-list? 6 '(4 8 6 2 1))
#t

> (in-list? 7 '(4 8 6 2 1))
#f

> (in-list "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.

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.

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

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

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

> (sorted? '("john" "zack" "bob"))
#f
```

8. flatten

Define a function that takes a list containing one or more sublists as an argument. Sublists 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.

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.

```
> (flatten '(1))
'(1)

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

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

9. threshold

Define a function that takes a list of numbers and a number (the threshold). It should return a new list that has the same numbers as the input list, but with all elements strictly greater than the threshold number removed.

Input: A list of numbers and a single atomic number.

Output: A new list of numbers that contains only the numbers from the original list that are strictly "less than" (<) the threshold number.

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

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

> (threshold '(4 8 5 6 7) 6.1)
'(4 5 6)

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

10. value-at

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. You may not use the built in `list-ref` function.

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 fuction should return the string "index out of bounds".

```
> (value-at '(4 7 9) 0)
4

> (value-at '(4 7 9) 1)
7

> (value-at '(4 7 9) 2)
9

> (value-at '(4 7 9) 3)
index out of bounds
```

11. rev-sort

Define a function that takes a list of numbers as an argument and returns reverse sorted list of the items in the first list. You may not use either the **`reverse`** function from the Racket standard library or **`my-reverse`** from problem #1. The reverse order sort should be implemented into your sorting algorithm (not by an ascending sort and then reversing). You may implement **`rev-sort`** using any standard sorting algorithm, e.g. mergesort, bubblesort, quicksort, etc.

Input: A list of integers.

Output: A new list of integers composed of the same integers as the input list, but sorted in descending order.

```
(rev-sort '(4 3 6 7 2 9 5 1))
'(9 7 6 5 4 3 2 1)

(rev-sort '(3 2 1))
'(3 2 1)

(rev-sort '(23))
'(23)
```