CSCI 350: Structures of Programming Language
Functional Programming Assignment

Due January 27th, 3 PM (Firm)

- Submit your assignment at http://www.networks.howard.edu/lij/courses/submit_hw.html
- Double check your submission. (You may submit multiple times with the latest one considered for grading.) No correction will be allowed after deadline even if wrong files are submitted.
- Create a project on Github and commit your code periodically. Make your project private initially and make it public at the deadline. (To create a private repository for free, sign up with your .edu email address.)
- Materials to be submitted (Zip all the materials and submit the zip file)
  a. A *plain text file* named fp.rkt that include ONLY the code of the functions with comments. DO NOT include the code that you use to test the functions.
  b. A PDF file that contains the following information:
     - A URL of your project on Github.
     - A screenshot of the Github web page that shows the summary commit history of your project. The commit history is expected to show the progress of your development over time. There must be at least 5 commits for the program(s) of each problem. While you are developing, at least one commit must be done every two hours. More is better.
     - The details of each commit, which can be obtained from Github web site. Each commit should contain non-trivial modifications. If there are much more than 5 commits for the program(s) of a problem, select five significant commits that are spread evenly over time.
- You can only use the functions appearing in the slides and `number?`. The use of other functions will result in significant loss of credits at the discretion of the instructor.
- The required functions must be named exactly the same as described by the questions. However, you are free to use any names for other functions that help the definition of the required ones.
- Grading
  No credits will be given and the submission will not be evaluated further if any of the following is true.
  - Any of the required files has a wrong format.
  - Any of the required files is missing.
  - The code file has more code than what is required.
  - The commit history does not satisfy the requirements.
  - The commit history shows suspicious practice.

  After that, programs will be judged according to the following criteria.
  a. Indentation: 5%
  b. Comments in code: 10%
  c. Correction of programs: 85% (judged by the outcome of testing cases)

1. (25 pts) Write a function `(reverse-general L)`. `L` is a list. The result of the function is the reversed version of `L`. Every single sub-list in L should be reversed as well. For example, the result of `(reverse-general '(a b (c (d e)) f)` should be `(f ((e d) c) b a)`.

L	Result
()	()
(a b c)	(c b a)
(a b ())	(() b a)
((a b c))	((c b a))
((a b c) (d e f))	((f e d) (c b a))
(a (b c) ((d e) f) g)	(g (f (e d)) (c b) a)
(1 (2 3) (4 (a (b (c d)))))	(((((d c) b) a) 4) (3 2) 1)

2. (25 pts) Write a function `(sum-up-numbers-simple L)`. `L` is a list, which may contain as elements numbers and non-numbers. The result of the function is the sum of the numbers *not* in nested lists in `L`. If there are no such numbers, the result is zero. For example, the result of `(sum-up-numbers-simple '(a b 1 2 c 3 d))` should be `6`.
   Test cases:

L	Result
()	0
(100 200)	300
(a b c)	0
(100 a)	100
(a 100)	100
(a 100 b 200 c 300 d)	600
(())	0
((100))	0
(100 (200))	100
(a 100 b (200) c 300 d)	400

3. (25 pts) Write a function `(sum-up-numbers-general L)`. `L` is a list, which may contain as elements numbers and non-numbers. The result of the function is the sum of *all* the numbers (including those in nested lists) in `L`. If there are no such numbers, the result is zero. For example, the result of `(sum-up-numbers-general '(a b 1 (2 c (3)) d))` should be `6`.
   Test cases:

L	Result
()	0
(100)	100
(100 200)	300
(a)	0
(a 100 b 200 c 300 d)	600
(())	0
((100))	100
(100 (200))	300
(a 100 b (200) c 300 d)	600

`(a 100 ((b ((200) c)) 300 d))`	`600`

4. (25 pts) Write a function `(min-above-min L1 L2)`. `L1` and `L2` are both simple lists, which *do not* contain nested lists. Both lists may have non-numeric elements. The result of the function is the minimum of the numbers in `L1` that are larger than the smallest number in `L2`. If there is no number in `L2`, all the numbers in `L1` should be used to calculate the minimum. If there is no number in `L1` larger than the smallest number in `L2`, the result is false (`#F`). For example, the result of `(min-above-min '(2 a 1 3) '(b 5 3 1))` should be `2`.
Test cases:

L1	L2	Result
`()`	`(a 100 b 200 c 300 d)`	`#F`
`(100)`	`()`	`100`
`(a 200 b 100 c 300 d)`	`()`	`100`
`(a)`	`()`	`#F`
`(a)`	`(a 200 b 300 c 100 d)`	`#F`
`(a b c)`	`(a 200 b 300 c 100 d)`	`#F`
`(a 200)`	`(a 200 b 300 c 100 d)`	`200`
`(a 100)`	`(a 200 b 300 c 100 d)`	`#F`
`(100 200 300)`	`(300 100 200)`	`200`
`(a 300 b 100 c 200 d)`	`(a 200 b 300 c 100 d)`	`200`