Project #2 – Recursive Functions in Racket

Learning objectives

- Design and implement recursive solutions to programming problems
- Implement a program in a functional programming language
- Use functional programming concepts to design concise and elegant code

Overview

The object of this assignment is to develop recursive programming skills by writing a few basic Racket functions operating on lists and sets. Before writing a function, you should try to determine how the problem can be decomposed into smaller subproblems.

Grading

Points: 40

The grade will be based upon the number of functions successfully implemented, as follows:

Number of correctly-implemented functions	Grade (assuming no errors)
7	A
6	B
5	C
4	D

To obtain full credit, each function must be robust and free of errors, including edge cases. Within the range of possible points for each grade, the specific score will be determined by programming style. That is, a submission with 7 fully-functional and correct functions, but no comments and poor programming style will earn the minimum A grade: 35 out of 40 points, while a well-commented submission with good programming style will earn the maximum A grade: 40 out of 40 points.

Programming style

In order to obtain the target grade your code must be:

- Readable -- Your code should:
    - Use readable variable and parameter names;
    - Use inline comments for each major block ;
    - Include header comments for each function describing the parameters (and their data types) any outputs or return values, any side effects, and the general purpose of the function; and
    - Be well organized, with *appropriate indentation, newlines, and whitespace*.
- Robust -- For this assignment it will not be necessary to check the *types* of the input parameters. In other words, if your function expects two integers, you do not have to worry about what will happen if it is passed two strings instead. However, you should be able to handle any input of the correct type. In other words, a function expecting an integer should function correctly for *any* integer. A function expecting a list should work correctly for *any* list, even the empty list.
- Correct -- Your functions must be named *exactly* as described below, and must expect *exactly* the parameters described below. I will check your functions with my own, automated main program, so you must follow the specifications to get full credit. If your recursive function needs additional parameters besides those listed below, you must use a recursive helper function.

  Note: Your functions may be globally defined, but any helper functions should be defined locally using `letrec`.

The functions

The functions that you may choose to implement include the following:

- **`(set-equal? set1 set2)`** -- Expects two lists, each of which represents a set of integers. Returns `#t` (that is, the boolean value for TRUE in Racket) if the sets are equal, `#f` otherwise. Two sets are equal if they contain the same members, ignoring the order. You may assume that the sets contain no duplicate values, and no recursive subsets. Example: `(set-equal? '(1 2 3 4) '(4 2 1 3))` should return `#t`.
- **`(nested-set-equal? set1 set2)`** -- Identical to the previous problem, but the sets can contain other sets. Example: `(nested-set-equal? '(1 2 (3 4 5)) '(2 (4 3 5) 1))` should return `#t`.
- **`(union set1 set2)`** -- Expects two lists, each of which represents a set of integers. Returns the union of the two sets. Recall that the union of two sets contains every element from either of the two sets, with no repetitions. Example: `(union '(1 2 3 4) '(2 3 4 5))` should return `(1 2 3 4 5)`. *The order of the returned set does not matter.*
- **`(intersection set1 set2)`** -- Expects two lists, each of which represents a set of integers. Returns the intersection of the two sets. Recall that the intersection of two sets contains every element that is present in both sets, with no repetitions. Example: `(intersection '(1 2 3 4) '(2 3 4 5))` should return `(2 3 4)`. *The order of the returned set does not matter.*
- **`(mergesort lst)`** -- Expects a flat (no sub-lists) list of integers. Sorts the integers using the merge sort algorithm and returns the resulting sorted list. If you do not recall the merge sort algorithm, it will be discussed briefly in class. You should also be able to find it easily on the web or in any data structures or algorithms textbook. Example: `(mergesort '(3 1 2 7 9))` should return `(1 2 3 7 9)`.

  *(Note: you may **not** use the mergesort defined in the library, or any other pre-written code. The point is to write your own.)*

- **`(powerset lst)`** -- Expects a flat (no sub-lists) list representing a set of of integers. Returns a list of lists representing the powerset of the input set. *(Recall that the the powerset of a set is the set of all subsets of the original set).* Example: `(powerset '(1 3 5))` should return `(() (1) (3) (5) (1 3) (1 5) (3 5) (1 3 5))`.

- **`(nested-reduce lst)`** -- Expects a list of integers, which may include sublists (nested lists). Returns a version of the list that has no duplicate integers, and no duplicate sublists. **Order matters** in the sublists: the sublist (1 2 3) is not equal to (3 2 1). Example: `(nested-reduce '(1 3 (2 5) (2 5) (2 5 (2 5) (2 5)) 3 7 1))` should return `(1 3 (2 5) (2 5 (2 5)) 7)`.

Submitting your code

While you may need to write a main program to test your code, you should submit **only your function definitions (including any helper functions)**. Place this code in a file called **Project2.rkt** and submit it via the Pilot Dropbox. **Write your name (commented by ;) on the first line Immediately following the line #lang racket.**
