Assignment 1

In this assignment, you have to accomplish two tasks. Always try to write understandable and well structured code, use meaningful names, and maybe refactor your code when you solved the task.

The bonus is calculated by weighting each assignment sheet the same, even if the points on the assignement sheets differ. That is, the percentage of points that you get each assignment is what counts towards the final bonus.

Task 1: Scala Warm-Up (5+5 Points)

The first task is mainly about learning Scala, in order to get familiar with its list implementation and recursion.

In this task, you need to implement two functions `flatten` and `reverse`.

The function `flatten` converts a list of lists *xss* into a list *xs*, which is formed by the elements of *xss*. For example, `List(List(1), List(2, 3), List(5))` is converted to `List(1, 2, 3, 5)`.

The function `reverse` takes a list `List(x1, ... ,xn)` and computes the reversed list `List(xn, ..., x1)`. You can use list construction (`List(...), ::, Nil`), data extraction with head and tail, and pattern matching (`case head :: tail => ...`). The method `List.isEmpty`, which tests whether a list is empty, may also be helpful.

Do not use any non-trivial, built-in members of `List`, like `List.reverse` or other methods for appending lists such as `:+, ++, :::`, etc . Instead, you should implement them yourself using recursion. Your code should be minimal and concise. Check out the Scala API on List to learn about how to work with lists.

Task 2: Boolean Expressions (5+4 Points)

In the second task you will implement an interpreter and a preprocessor for boolean expressions.

Initially, you should implement an interpreter for a small boolean expression language (BE) which should support boolean literals, conjunction, disjunction, and negation. You should have a look at the Pattern Matching and Case Classes features of Scala while working at this task.

Next, you should implement a preprocessor to add more features to the boolean expression language without modifying the interpreter. The preprocessor should implement implications and biimplications by transforming a given BE tree. The resulting tree must be free of any Imp or BiImp expressions.

Please have a look on the provided test cases for the expected definition of implication and biimplication.