DIT725 V14, Assignment 1: Logic and Recursion

The assignment should be done in groups (6 people) and the solution be submitted in GUL. The deadline of the assigment can be found in the GUL pages. A `.zip` file and a `.java` file accompanies this problem description, containing code to get you started.

The assignment consists of two problems, both related to recursion (Lecture 2). The first one is about logic (Lecture 1, mainly), but also involves recursion.

1 Tautology Checker

Your task is to write a tautology checker for arbitrary propositional formulas. Given a formula, check whether its true no matter what the truth values of the atoms (propositional variables) are. First familiarize yourself with the source code of the following classes:

`Formula`	Parent class for different type of formulas.
`Atom`	Atomic formulae (propositional variables).
`Neg`	Negated formula.
`Conn`	Binary connectives: `AND, OR, IMPLIES, IFF`.
`Bin`	Compound formula with binary connective.

`Formula`s have a method `String toString()` which prints the formula with parentheses for disambiguation and a method `boolean eval(boolean[] valuation)` which returns the truth value given that the propositional variables are valuated according to `valuation`. If and only if `valuation[i] == true`, then atom number `i` is true.

You are also given a *parser* (ParseFormula.java), a method to turn a formula `String` into a `Formula` tree. If you are curious how the parser works, study the source code. In any case, read `ParserTest.java` to get an example how to use the parser API. Here is the relevant part:

```
// Create a parser for String s.
FormulaParser parser = new FormulaParser(s);
// Parse all of s into formula p.
Formula       p      = parser.parse();
// After parsing, number of atoms is known.
int           n      = parser.numberOfAtoms();
```

By running `ParserTest.main` you can test the parser as follows.

```
$ java -jar formula.jar (A|B)->C !(A&!A) A1<->(A2&(A2->A1))
Testing parser for propositional formulae...

Parsed formula ((A ∨ B) ⇒ C).
It has 3 different propositional variables.
Its main connective is binary.

Parsed formula ¬(A ∧ ¬A).
It has 1 different propositional variables.
It is a negated formula.

Parsed formula (A1 ⇔ (A2 ∧ (A2 ⇒ A1))).
It has 2 different propositional variables.
Its main connective is binary.
```

In Eclipse, use the run configuration menu to set parameters for the execution. You can have several run configurations for the same executable to run with different input. NOTE: If the fancy logical operators in source code and output look odd when using Eclipse, you need to set text file encoding to UTF-8 in the project properties (right click the project in the package explorer and choose properties).
Your tasks:

1. Using the classes provided with this exercise, write a class `Tautology` which parses each command line argument as formula and states whether it is a tautology. The output should be like this:

   ```
   $ java Tautology 'A|!A' 'A&!A' '(A->B)->(B->A)' '(A&(A->B))->B'
   (A ∨ ¬A) is a tautology
   (A ∧ ¬A) isn't a tautology
   ((A ⇒ B) ⇒ (B ⇒ A)) isn't a tautology
   ((A ∧ (A ⇒ B)) ⇒ B) is a tautology
   ```

2. Write a PDF that explains in detail how your tautology checker works. Add to the text:

   (a) 10 good examples of tautologies.

   (b) 10 examples of formulas that are not tautologies.

   Make sure your tautology checker gets all of them right.

Submit your file `Tautology.java` file and the `.pdf` file.

Challenge (optional): Can you modify your program to also recognize contradictions and answer tautology/contradiction/neither for each formula?

2 Flood Filler

Flood filling in an image is to start from a given point and change all pixels reachable from that point to a given new color. The reachable pixels are those which we can get to by moving one pixel at a time without passing any pixel with an unallowed color. Each move can be either left, right, up or down. In drawing applications, flood filling often has a paint pouring bucket as symbol.

You are provided with a file `FloodFiller.java` which loads a named image file, turns it black and white and displays it in a window. Your job is to add functionality to this program so that whenever the user clicks on the image, flood filling takes place from that point. Flood filling should take place in black areas. If you click on a white pixel, nothing should happen. The program should fill the black areas you click on with a color, which should be neither black nor white.

Solve the problem using recursion. You should submit your `FloodFiller.java` file and a `.pdf` file describing in details how your flood filler works.

Note: You might get a `StackOverflowError`, especially for larger images. In this case, try running Java with a larger stack size. For instance, the following command calls Java with a stack size of 4 MB.

```
$ java -Xss4m FloodFiller images/nyc.jpg
```

In Eclipse, the `-Xss4m` option is added under VM Arguments in run configuration. A `StackOverflowError` could also be caused by an error in your program, like a non-terminating recursion. If the error persists even for very large stack sizes, rewrite your program.

Challenge (optional): Keeping code short and to the point is usually a good thing. It's possible to implement the flood-filling method in 10 lines of code (without obscuring the algorithm). Can you do it?

Challenge (optional): Can you make a non-recursive version of the flood filler? Hint: Use your own stack (a LinkedList for instance) instead of the program stack, replace every recursive call with adding a point to the stack.