

---

## CSED332 ASSIGNMENT 2

Due Monday, September 27

---

### Objectives

- Unit testing and coverage
- Maven (adding dependencies to pom.xml)
- Learn Java programming language

### Maven and Testing

- Your code need to be compiled using only Maven in a command line for grading. Unlike the previous assignment, your pom.xml must be modified to include extra dependencies.
- Maven can generate coverage reports using JaCoCo plugin. We have already added this plugin in pom.xml (see the provided pom.xml and understand how the plugin is added to the build).
- To run your tests with JaCoCo and produce code coverage results, you can go to where your pom.xml is and execute the following commands: `mvn test` and `mvn jacoco:report`.
- The command `mvn jacoco:report` generates human readable reports of coverage information of your unit tests. The reports will be stored in the `target/site/jacoco` directory.
- You MUST ensure that your tests pass on your code. You will get overall 0 points if your submitted code and tests do not work with `mvn test`.

### Problem 1

- The goal of this problem is to create a library management application with unit tests, to evaluate quality of these tests, and to automate build.
- You will create a simple model for this library application, including books and collections (see the skeleton code for more details).
  - Books are organized in collections.
  - A book can be added or removed from a collection.
  - A collection contains a various number of books, and can also contain other sub-collections.

For example, a collection **Computer Science** can contain a series of books about computer science in general and other sub-collections such as **Operating System**, **Software Verification**, etc.

- Books, collections, and libraries can be exported to or imported from a file in the JSON format (see <https://www.json.org>).
  - Do not reinvent the wheel. You may want to use a Java library for JSON, such as `org.json` (<https://github.com/stleary/JSON-java>), or any other JSON libraries of your choice.
  - You must modify your pom.xml to include extra dependencies for using extra libraries. *Your code may not be graded, if you just download such libraries, not using Maven.*
- You will have to write at least one (separate) JUnit test for each method of the classes `Book`, `Collection`, and `Library` in the corresponding test classes.
  - Each test method should test a single behavior with appropriate assertions. *Do not write a single method that checks multiple behaviors.*
  - Your submitted tests need to achieve at least 80% **statement coverage** (instruction coverage for JaCoCo). *Do not add arbitrary code to your test method to just increase coverage.*

## Problem 2

- The goal of this problem is to implement several functions for Boolean expressions. The syntax of Boolean expressions is given as follows:

Boolean formula  $\varphi ::= c \mid v \mid !\varphi \mid \varphi \ \&\& \ \varphi \mid \varphi \ || \ \varphi$   
 Boolean constant  $c ::= \text{true} \mid \text{false}$   
 Boolean variable  $v ::= p_1 \mid p_2 \mid p_3 \mid \dots$

A Boolean expression is constructed by constants (true and false), variables of the form  $p_i$  for natural number  $i > 0$ , and logical operators such as  $!$  (negation),  $\&\&$  (and), and  $||$  (or).

- We provide a parser for Boolean expressions using ANTLR4 (<https://www.antlr.org>), but you must modify your `pom.xml` to include extra dependencies for ANTLR4 (`antlr4-runtime`).
- A Boolean expression  $e$  *evaluates* to a truth value (either *true* or *false*), given a *truth assignment* that assign a truth value to each Boolean variable in the expression  $e$ .
  - For example, the expression  $(p_1 \ || \ p_2) \ \&\& \ (p_2 \ || \ ! \ p_3)$  evaluates to *true*, given the truth assignment  $\{p_1 \mapsto \text{true}, p_2 \mapsto \text{false}, p_3 \mapsto \text{false}\}$ .
- We can *simplify* a Boolean expression into an equivalent expression by logical equivalence laws as follows (see [https://en.wikipedia.org/wiki/Logical\\_equivalence](https://en.wikipedia.org/wiki/Logical_equivalence)):

(1) Identity and idempotent laws

$$\varphi \ \&\& \ \text{true} \equiv \varphi, \quad \varphi \ || \ \text{false} \equiv \varphi, \quad \varphi \ \&\& \ \varphi \equiv \varphi, \quad \varphi \ || \ \varphi \equiv \varphi$$

(2) Domination and negation laws

$$\varphi \ \&\& \ \text{false} \equiv \text{false}, \quad \varphi \ || \ \text{true} \equiv \text{true}, \quad \varphi \ \&\& \ !\varphi \equiv \text{false}, \quad \varphi \ || \ !\varphi \equiv \text{true}$$

(3) De Morgan's laws

$$!(\varphi_1 \ \&\& \ \varphi_2) \equiv !\varphi_1 \ || \ !\varphi_2, \quad !(\varphi_1 \ || \ \varphi_2) \equiv !\varphi_1 \ \&\& \ !\varphi_2$$

(4) Absorption laws

$$\varphi_1 \ || \ (\varphi_1 \ \&\& \ \varphi_2) \equiv \varphi_1, \quad \varphi_1 \ \&\& \ (\varphi_1 \ || \ \varphi_2) \equiv \varphi_1$$

(5) Double negation law

$$!(\ !\varphi) \equiv \varphi$$

(6) Distributive laws

$$\begin{aligned} \varphi_1 \ || \ (\varphi_2 \ \&\& \ \varphi_3) &\equiv (\varphi_1 \ || \ \varphi_2) \ \&\& \ (\varphi_1 \ || \ \varphi_3), \\ \varphi_1 \ \&\& \ (\varphi_2 \ || \ \varphi_3) &\equiv (\varphi_1 \ \&\& \ \varphi_2) \ || \ (\varphi_1 \ \&\& \ \varphi_3) \end{aligned}$$

To simplify Boolean expressions, these rules are applied repeatedly—from the left-hand side to the right-hand side for each rule—in order from (1) to (6), with respect to the associativity and commutativity of  $\&\&$  and  $||$ ,<sup>1</sup> until no more rule can be applied. For example:

- $p_1 \ \&\& \ (p_2 \ \&\& \ ! \ p_1)$  can be simplified into false.
- $(p_1 \ \&\& \ \text{true}) \ \&\& \ (p_2 \ \&\& \ !( \ ! \ p_1 \ \&\& \ ! \ p_2))$  can be simplified into  $p_1 \ \&\& \ p_2$ .
- Similarly, you will have to write at least one (separate) JUnit test for each method of the classes `Constant`, `Variable`, `Negation`, `Conjunction`, and `Disjunction`, in the test class `ExpTest`.
  - Each test method should test a single behavior with appropriate assertions. *Do not write a single method to check multiple behaviors.*
  - Your submitted tests need to achieve at least 80% **branch coverage**. *Do not add arbitrary code to your test method to just increase coverage.*

<sup>1</sup>Hint: Conjunction and Disjunction can have a list of subexpressions of an arbitrary length. Even though `ExpParser` generates expressions with at most two subexpressions, you can freely transform them to “flattened” structures. To simplify an expression, first flatten the parsed expression of type `Exp` using the associativity of  $\&\&$  and  $||$ , and “search” for its subexpressions to apply the equivalence rules with respect to the associativity and commutativity of  $\&\&$  and  $||$ .

## General Instruction

- Download the attached file `homework2.zip`, which contains two directories `problem1` and `problem2`. Each of them can be imported as a separate project into IntelliJ IDEA.
- Initially, `mvn test` will fail, because no dependencies are declared in `pom.xml`. Modify your `pom.xml` to include required libraries, including JUnit5 (`org.junit.jupiter`), ANTLR4 (`antlr4-runtime`), etc.
- The `src/main` directory contains the skeleton code. You should implement all the methods marked with *TODO*. Before writing code, read the description in the source code carefully.
- The `src/test` directory contains test classes. Use JaCoCo to find out how much coverage your tests have. Upload the JaCoCo report in CSV format from `target/site/jacoco/jacoco.csv`.
- As usual, do not modify the existing interfaces, the class names, and the signatures of the public methods. You can add more methods or classes if you want.

## Turning in

1. Create a private project with name `homework2` in <https://csed332.postech.ac.kr>, and clone the project on your machine.
2. Commit your changes in your `homework2` project that includes two directories `problem1` and `problem2`, including JaCoCo coverage reports, and push them to the remote repository.
3. The JaCoCo coverage report for each problem, generated by `mvn jacoco:report`, will be uploaded to the directory `homework2` as follows:
  - `homework2/jacoco1.csv` for Problem 1
  - `homework2/jacoco2.csv` for Problem 2
4. Tag your project with “submitted” and submit your homework. We will use the tagged version of your project for grading.

## Reference

- Java Language Specification: <https://docs.oracle.com/javase/specs/>
- Beginning Java 9 Fundamentals 2nd by Kishori Sharan, Apress, 2017 (available online at the POSTECH digital library <http://library.postech.ac.kr>)
- Maven Getting Started Tutorial: <https://maven.apache.org/guides/getting-started/>