CSED332 Assignment 2

Due Friday, October 6

1 Problem 1

Background: Boolean Expressions

- A Boolean expression is constructed by Boolean constants, variables of the form p_i for $i \in \mathbb{N}$, and logical operators such as `!` (negation), `&&` (and), and `||` (or). The syntax is given as follows:

$$\begin{array}{rrl} \textsf{Boolean formula} & \varphi ::= & c \mid v \mid \texttt{!}\,\varphi \mid \varphi \texttt{ \&\& } \varphi \mid \varphi \texttt{ || } \varphi \\ \textsf{Boolean constant} & c ::= & \texttt{true} \mid \texttt{false} \\ \textsf{Boolean variable} & v ::= & p_1 \mid p_2 \mid p_3 \mid \cdots \end{array}$$

- 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.
- We can *simplify* a Boolean expression into an equivalent expression by logical equivalence laws as follows (see `https://en.wikipedia.org/wiki/Logical_equivalence`):

  (1) Identity and idempotent laws

$$\varphi \texttt{ \&\& true} \equiv \varphi, \qquad \varphi \texttt{ || false} \equiv \varphi, \qquad \varphi \texttt{ \&\& } \varphi \equiv \varphi, \qquad \varphi \texttt{ || } \varphi \equiv \varphi$$

  (2) Domination and double negation laws

$$\varphi \texttt{ \&\& false} \equiv \texttt{false}, \qquad \varphi \texttt{ || true} \equiv \texttt{true}, \qquad \texttt{!}(\texttt{!}\,\varphi) \equiv \varphi$$

  (3) Negation laws

$$\varphi \texttt{ \&\& !}\,\varphi \equiv \texttt{false}, \quad \varphi \texttt{ || !}\,\varphi \equiv \texttt{true}, \quad \texttt{!(true)} \equiv \texttt{false}, \quad \texttt{!(false)} \equiv \texttt{true}$$

  (4) De Morgan's laws and

$$\texttt{!}(\varphi_1 \texttt{ \&\& } \varphi_2) \equiv \texttt{!}\,\varphi_1 \texttt{ || !}\,\varphi_2, \qquad \texttt{!}(\varphi_1 \texttt{ || } \varphi_2) \equiv \texttt{!}\,\varphi_1 \texttt{ \&\& !}\,\varphi_2$$

  (5) Absorption laws

$$\varphi_1 \texttt{ || } (\varphi_1 \texttt{ \&\& } \varphi_2) \equiv \varphi_1, \qquad \varphi_1 \texttt{ \&\& } (\varphi_1 \texttt{ || } \varphi_2) \equiv \varphi_1$$

Modifying the Gradle Build Script

- Your code need to be compiled using only Gradle in a command line for grading. Unlike the previous assignment, your `build.gradle.kts` must be modified to include extra dependencies.
- Gradle can generate coverage reports using the JaCoCo plugin, which is already included in the script `build.gradle.kts`. See how the JaCoCo plugin is added to the build.
- We provide a parser for Boolean expressions using ANTLR4 (`https://www.antlr.org`), but you must modify your `build.gradle.kts` to include ANTLR in the build.
- Add an extra dependency for ANTLR4 (version 4.13.1), and an extra argument `-visitor` to the ANTLR task. See `https://docs.gradle.org/current/userguide/antlr_plugin.html`

Implementing Operations for Boolean Expressions

- The interface `Exp` defines several operations for Boolean expressions. Implement the following methods for `Exp`'s subclasses (declared as records): `getVariables`, `evaluate`, and `simplify`.
- The method `getVariables` returns the set of integers that represent the variables in the Boolean expression. E.g. for $\varphi = (p_1 \texttt{ || } p_2) \texttt{ \&\& } (p_2 \texttt{ || ! } p_3)$, `getVariables` returns $\{1, 2, 3\}$.
- The method `evaluate` returns the truth value, given a truth assignment. E.g., given the truth assignment $\{p_1 \mapsto \textit{true},\ p_2 \mapsto \textit{false},\ p_3 \mapsto \textit{false}\}$, the method returns *true* for the above φ.
- The method `simplify` applies the above logical equivalence rules repeatedly—from the left-hand side to the right-hand side—until no more rule can be applied to simplify the Boolean expression.

Writing JUnit Test Cases

- Write at least one JUnit test case (as a separate test method) for each method of the classes Constant, Variable, Negation, Conjunction, and Disjunction, in the test class ExpTest.
- 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 gradle test.
- After executing gradle test, you can generate reports for coverage information of your unit tests using gradle jacocoTestReport. The reports will be stored in `build/reports/jacoco/test`.
- Your submitted testcases need to achieve at least 90% **statement coverage** for the classes in the package `edu.postech.csed332` (excluding the package `edu.postech.csed332.parser`).
- Each test method should test a single behavior with appropriate assertions. Do not add arbitrary code to your test method to just increase coverage.

2 Problem 2

Background: Graphs and Trees

- A *undirected graph* is a pair $G = (V, E)$, where V is a set of *vertices* (also called nodes) and $E \subseteq V \times V$ is a set of edges that connects two vertices. For example, Fig. 1 shows the graph:

$$V = \{1, 2, 3, 4, 5, 6\}, \quad E = \{(1, 2), (1, 4), (2, 1), (2, 4), (3, 6), (4, 1), (4, 2), (6, 3)\}$$

- A (rooted) *tree* is a tuple $T = (V, E, \hat{v})$, where $G = (V, E)$ is a graph, $\hat{v} \in V$ is a root, and there exists exactly one path from the root to any vertex. Fig. 2 show the tree with 1 as the root:

$$V = \{1, 2, 3, 4, 5, 6\}, \quad E = \{(1, 2), (1, 4), (2, 1), (2, 3), (2, 6), (3, 2), (4, 1), (4, 5), (5, 4), (6, 2)\})$$

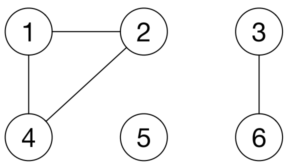

Figure 1: A graph

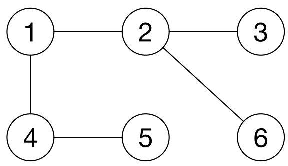

Figure 2: A tree

Abstract Interface Specifications for Graphs and Trees

- In this assignment, we consider generic abstract interfaces for graphs and trees, where vertices are represented as any elements of a given (immutable and comparable) type N:
  - Graph<N>: an interface for undirect graphs
  - Tree<N>: an interface for rooted trees, extending Graph<N>
  - MutableGraph<N>: an interface with mutable operations, extending Graph<N>
  - MutableTree<N>: an interface with mutable operations, extending Tree<N>

  Note that Graph<N> and Tree<N> do *not* contain methods for mutable operations, such as adding or removing vertices and edges. These interfaces are described in detail in the source code.

- The interfaces declare *abstract data types* for graphs and trees, which specify mathematical *abstract values* and their associated operations.
  - Abstract values of graphs are pairs $G = (V, E)$, where each vertex in V has type N.
  - Abstract values of trees are triples $T = (V, E, \hat{v})$, where $\hat{v}$ is the root.

  The implementation may use concrete values with extra information invisible to the client. This is a form of *information hiding*, one of the fundamental concepts of object-oriented programming.

- The goal is to write formal abstract specifications of these interfaces *with respect to abstract values*; namely, a class invariant of each interface, and a precondition and a postcondition of each method.
- Fill out the attached Markdown file `hw2-problem2.md`, indicating the interfaces and methods for which you need to write formal abstract specifications, including some notations and examples.[1]

Behavioral Subtypes of Graphs and Trees

- The Liskov substitution principle states that if type S is a subtype of T, then code written for objects of type T also operates correctly for objects of type S.
  - In other words, objects of type T can be substituted with objects of type S without altering any of the properties of T, such as class invariants, preconditions, postconditions, etc.
  - As shown in the class, subclassing does *not* guarantee subtyping, and trying to meet the Liskov substitution principle for subclassing is a good software design practice.
- The goal is to identify whether the abstract interfaces satisfy the Liskov substitution principle; that is, to answer the following questions:
  - is Tree<N> a subtype of Graph<N>?
  - is MutableGraph<N> a subtype of Graph<N>?
  - is MutableTree<N> a subtype of Tree<N>?
  - is MutableTree<N> a subtype of MutableGraph<N>?
- For each question, explain your reasoning *using the abstract specifications that you have defined in Problem 1.* For types S, T ∈ {Tree<N>, MutableGraph<N>, MutableTree<N>, MutableTree<N>}:
  - If S is a subtype of T, explain why S has a stronger specification than T in terms of their specifications (preconditions, postconditions, and class invariants).
  - If S is *not* a subtype of T, (i) explain which part of the specifications violate the Liskov substitution principle, and (ii) show code written for `T` that behaves differently for S.
- Similarly, fill out the attached file `hw2-problem2.md`. Note that you can write math expressions using GitLab Markdown: `https://docs.gitlab.com/ee/user/markdown.html#math`.

[1] You do *not* have to write specifications for other interfaces/methods not in `hw2-problem2.md`.

Black-box Test Cases for Graphs and Trees

- The goal of this problem is to write a high-quality test suite for the interfaces MutableGraph<N> and MutableTree<N> with respect to their specifications.
  - Because only abstract specifications are available, you will write *black-box test cases* for the interfaces, based on equivalence partitioning.
  - E.g., for the method `addVertex`(v) of MutableGraph<N>, there are two equivalence classes based on the description: v is already in the graph, or v is previously not in the graph.
- For each method, write a test method for each equivalence class in the abstract test classes in the `src/test` directory (e.g., two test methods for `addVertex`, which are already given in the code).
  - Two abstract classes are given: AbstractMutableGraphTest<N,G> for vertex type N and graph type G, and AbstractMutableTreeTest<N,T> for vertex type N and tree type T.
  - Each abstract test class contains one object (either a graph of type G or a tree of type T), and eight vertices of type N, along with some example test methods using them.

Implementing Graphs

- In this problem, we will implement a direct graph using an *adjacency list representation*[2]
  - You must use the following representation provided in the class AdjacencyListGraph<N>, a (sorted) map from vertices to the (sorted) set of their adjacent vertices.

```
private final @NotNull SortedMap<V, SortedSet<V>> adjMap;
```

  - For example, the graph in Fig. 1 is represented as the sorted map[3]

$$\{1 \mapsto \{2,4\}, 2 \mapsto \{1,4\}, 3 \mapsto \{6\}, 4 \mapsto \{1,2\}, 5 \mapsto \emptyset, 6 \mapsto \{3\}\}$$

- Implement the class AdjacencyListGraph<N>, which is a subclass of MutableGraph<N>, using this representation of undirected graphs.
  - What are an abstract function and a class invariant for AdjacencyListGraph<N>? Document the abstraction function and class invariant (as comments in the source code).
  - Implement the method `checkInv` that checks your class invariant. The method `toRepr` provides a string representation for abstract values.

Implementing Trees

- In this problem, we will write two different implementations of a tree using different representations.
  - DelegateTree<N> uses an instance of MutableGraph<N> to implement its functionality.[4]
  - ParentPointerTree<N> uses *pointers to parent vertices*[5] to represent a rooted tree.
- Implement DelegateTree<N> and ParentPointerTree<N>. Both are subclasses of MutableTree<N> *with the same specification* but with different representations.
  - What are an abstract function and a class invariant of these classes? Document the abstraction function and class invariant for each class (as comments in the source code).
  - Implement the method `checkInv` that checks your class invariant for each class. Similarly, the method `toRepr` provides a string representation for abstract values.
  - You may find that the provided method `findReachableVertices` is useful for implementing some methods declared in MutableGraph<N>.

[2] https://en.wikipedia.org/wiki/Adjacency_list
[3] https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/SortedMap.html
[4] https://en.wikipedia.org/wiki/Delegation_(object-oriented_programming)
[5] https://en.wikipedia.org/wiki/Parent_pointer_tree

Writing White-box Test Cases

- Whenever you write a method, check whether your implementation passes your black-box test cases that you have written, following test-driven development practice.
  - The test class StringAdjacencyMutableGraphTest extends your AbstractMutableGraphTest. It contains `setUp()` to initialize abstract graphs and vertices for black-box test cases.
  - There are two test classes that extend your AbstractMutableTreeTest, along with appropriate `setUp()`: IntegerDelegateMutableTreeTest and DoubleParentPointerMutableTreeTest.
  - You may write more *white-box test cases* to these test classes to achieve more code coverage for AdjacencyListGraph<N>, DelegateTree<N> and ParentPointerTree<N>, if needed.
- Your submitted tests need to achieve at least 80% **branch coverage**. Your black-box test cases should already give high coverage, but you may add more white-box test cases if needed.
  - Your black-box test cases will be graded according to whether they clearly describe different scenarios from the specifications using equivalence partitioning.
  - *Do not add arbitrary code to your test method to just increase coverage.* In particular, this will severely affect your scores for black-box test cases.
  - The abstract test classes should only depend on abstract interfaces, namely, MutableGraph<N> and MutableTree<N>; importing concrete implementations is not allowed.
- After executing gradle test, you can generate reports for coverage information of your unit tests using gradle jacocoTestReport. The reports will be stored in `build/reports/jacoco/test`.

3 General Instruction

- Download the attached file `homework2.zip`, which contains two directores `hw2-problem1` and `hw2-problem2`. Each of them can be imported as a separate project into IntelliJ IDEA.
- 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 (`build/reports/jacoco/test/jacocoTestReport.csv`).
- Do not modify the existing interfaces, the class names, and the signatures of the public methods and checkInv(). You can add more private methods if you want.
  - For Problem 2, we use *fixed* representations for AdjacencyListGraph<N>, DelegateTree<N>, and ParentPointerTree<N>. *You cannot add even private member variables to these classes.*

Turning in

1. Create a private project with name `homework2` in `https://csed332.postech.ac.kr`. Commit your changes in your `homework2` project, and push them to the remote repository.
2. The JaCoCo coverage reports in CSV will be uploaded to the directory `homework2` as follows: `homework2/jacoco1.csv` for Problem 1 and `homework2/jacoco2.csv` for Problem 2
3. 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 17 Fundamentals 3nd by Kishori Sharan and Adam L. Davis, Apress, 2022 (available online at the POSTECH digital library `http://library.postech.ac.kr`)
- Gradle User Manual: `https://docs.gradle.org/current/userguide/userguide.html`