CSED332 Assignment 5

Due Friday, October 22

Objectives

- Learn Visitor and Decorator patterns

Simple Arithmetic Expressions

- We consider simple arithmetic expressions with unknown variables. The syntax is as follows:

$$\begin{array}{rrcl} \textsf{Expression} & E & ::= & E + E \mid E - E \mid E * E \mid E \,/\, E \mid E \,\hat{}\, E \mid v \mid \mathit{number} \mid (\, E \,) \\ \textsf{Variable} & v & ::= & x_1 \mid x_2 \mid x_3 \mid \cdots \end{array}$$

- An expression is constructed by double-precision floating point *numbers*, variables of the form x_i for natural number $i > 0$, and arithmetic operators such as $+$, $-$, $*$, $/$, and ^. For example:

```
5.0 + 1.0 * 2.0,   x1 ^ x2 + 2.0 * x1,   (x4 ^ 7.26 / x2 - x2) ^ x3 / x4
```

- Exp is an abstract base class for expressions, and has the subclasses PlusExp, MinusExp, MultiplyExp, DivideExp, ExponentiationExp, VariableExp, NumberExp, etc., as depicted in Figure 1.

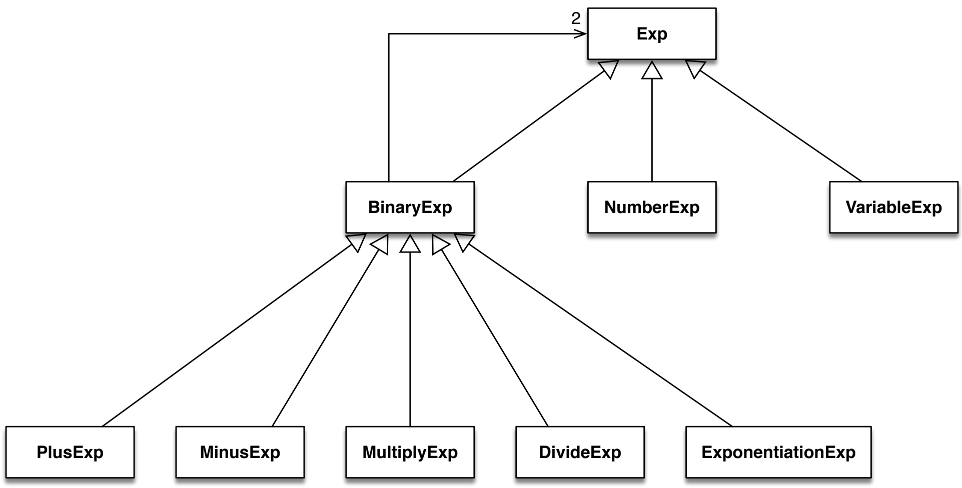

Figure 1: A class hierarchy for expressions

- We provide a parser for simple arithmetic expressions as the static method `parseExp` of Exp, which returns an instance of Exp, given a string representation of an arithmetic expression.

```
public static Exp parseExp(@NotNull String str);
```

For example, `Exp.parseExp("1.0 + 2.0 * x1")` returns an instance of Exp that represents

$$\textsf{Plus}(\textsf{NumberExp}(1.0), \textsf{MultiplyExp}(\textsf{NumberExp}(2.0), \textsf{VariableExp}(1)))$$

- In this assignment, you will implement various operations for Exp using the *visitor* design pattern, and the variations of these operations using the *decorator* design pattern.

Problem 1: Visitor Pattern

1. ExpVisitor<T>

   - The interface ExpVisitor<T>, which is currently empty, is a base interface for visitors of Exp. Write the `visit` methods of ExpVisitor<T> in `edu.postech.csed332.homework5`.
   - One method should be defined for each *leaf* subclass of Exp (i.e., all subclasses except for BinaryExp). The type parameter `T` is defined for the return type of a visit method.
   - Using the `visit` methods in ExpVisitor<T>, implement the `accept` method for each leaf subclass of Exp. Note that these `accept` methods should *not* be written in the superclasses.

2. ToStringVisitor

   - ToStringVisitor is used to implement `Exp.toString` for the string representation of an Exp object. Write the `visit` methods of ToStringVisitor in `edu.postech.csed332.homework5`.
   - You may use `Double.toString` to obtain the string representation of double-precision floating point numbers in the expression.
   - Note that the string representation must *always* be parsed as an equivalent expression by `Exp.parseExp`. That is, the following test must pass for any expression `exp`:

   ```
   Exp g = Exp.parseExp(exp.toString());
   assertEquals(exp.toString(), g.toString());
   ```

3. EvaluationVisitor

   - EvaluatorVisitor is used to implement `Exp.eval` that returns the value of the expression, given a valuation of the variables. Implement EvaluationVisitor.
   - A valuation is a map $i \mapsto n$ of type Map<Integer,Double> that assigns to each variable x_i a number n. For example, the map $\{1 \mapsto 4.0,\ 2 \mapsto 0.1\}$ assigns 4.0 to x_1 and 0.1 to x_2.
   - For example, consider the arithmetic expression "`x1 ^ x2 + 2.0 * x1`". Given the valuation $\{x_1 \mapsto 3.0,\ x_2 \mapsto 1.0\}$, the method `eval` returns 9.0.

4. EquivalenceVisitor

   - EquivalenceVisitor is used to implement the method `Exp.equiv` that checks if this expression is syntactically the same as the `other` expression. Implement EquivalenceVisitor.
   - `Exp.equiv` returns `true` if and only if a given expression (as an argument) represents exactly the same expression as the current expression. For example, the following test should pass:

   ```
   Exp e1 = Exp.parseExp("1.0 + 2.0 * x1 + x1");
   Exp e2 = Exp.parseExp("1.0 + 2.0 * x1 + x1");
   assertTrue(e1.equiv(e2));
   ```

   - EquivalenceVisitor needs to keep track of the structures of two expressions (e.g., e_1 and e_2). This can be done by defining *internal states* (extra member variables) of EquivalenceVisitor.

Problem 2: Decorator Pattern

1. ExpDecorator

   - ExpDecorator, which is currently empty, is a base class for decorators of Exp. Implement the class ExpDecorator in `edu.postech.csed332.homework5`.
   - An ExpDecorator object wraps an original Exp object that is given as a member variable of ExpDecorator, and transfers every operation to the wrapped object.
   - In this assignment, concrete decorators for Exp (see below) will be implemented as a subclass of ExpDecorator. In this way, multiple decorators can be stacked on top of each other.

2. PrettyPrintExpDecorator

   - In PrettyPrintExpDecorator, each double-precision value will be written in decimal format. For example, PrettyPrintExpDecorator.toString returns `1234567890123`, not `1.234567890123E12`.
   - Implements the `toString` method of PrettyPrintExpDecorator. To obtain a decimal-format string of a double-precision number, you can use java.math.BigDecimal.
   - *Hint:* this can be easily implemented by defining an anonymous subclass of ToStringVisitor (see `https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html`).

3. DefaultValueExpDecorator

   - In DefaultValueExpDecorator, a variable has a given default value when its valuation is not provided for evaluation. Implement the `eval` method of DefaultValueExpDecorator.
   - Consider the expression "`x1 ^ x2 + 2.0 * x1`" and the default value 1.0. Given the valuation $\{x_1 \mapsto 3.0\}$, the method `eval` returns 9.0.

4. RenamingEquivDecorator

   - In RenamingEquivDecorator, the expression is equivalent to another expression *up to renaming*. Implement the `equiv` method of RenamingEquivDecorator.
   - E.g., "`(x1 + x2) * x3 + 1.0 * x1`" is equivalent to "`(x3 + x4) * x2 + 1.0 * x3`" up to renaming, by one-to-one renaming mapping $\{x_1 \mapsto x_3,\ x_2 \mapsto x_4,\ x_3 \mapsto x_2\}$.
   - But "`(x1 + x2) * x3 + 1.0 * x1`" is *not* equivalent to "`(x3 + x3) * x2 + 1.0 * x3`" up to renaming, since the renaming mapping $\{x_1 \mapsto x_3,\ x_2 \mapsto x_3,\ x_3 \mapsto x_2\}$ is not one-to-one.

General Instruction

- Your code need to be compiled using only Maven in a command line for grading. You MUST ensure that your tests pass on your code using mvn test.
- 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 provides a number of test cases to check whether your implementation is OK. You can execute all the test cases by running `mvn test`.
  - ExpTest includes several simple JUnit test cases that must pass to earn points.
  - ExpRandomTest implements automated random testing, using JUnit QuickCheck.
- As usual, do not modify the existing interfaces, the class names, and the signatures of the public methods, *unless otherwise stated*. You can add private methods or member variables if you want.

Turning in

1. Create a private project with name `homework5` in `https://csed332.postech.ac.kr`, and clone the project on your machine.
2. Commit your changes in your `homework5` project, and push them to the remote repository.
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 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/`