CSED232 Assignment 5

Due Friday, May 2

Background: Arithmetic Expressions and Polynomial Expansion

- We consider mathematical expressions of the following syntax, where C denotes numeric constants, and x_i denotes variables indexed by positive integers $i > 0$:

$$\text{Arithmetic expression} \quad E ::= \quad C \quad | \quad x_i \quad | \quad -E \quad | \quad E + E \quad | \quad E * E$$

- Each expression can be evaluated given an *environment* that assigns a numeric value to each variable x_i appearing in the expression.
- An arithmetic expression can be *expanded* into a *canonical polynomial form*: a sum of monomials, where each monomial is a product of variables (each raised to a positive integer power), multiplied by a constant coefficient (see `https://en.wikipedia.org/wiki/Polynomial_expansion`).

Interfaces for Expressions and Polynomials

- The codebase provides immutable data structures to represent arithmetic expressions and their canonical polynomial forms. All interfaces and record types described below are *immutable*.
- `AExp<T>` is an interface representing expressions over a numeric type `T`, and includes subtypes `Const<T>` (constant), `Var<T>` (variable), `Neg<T>` (negation) `Add<T>` and `Mul<T>`.
- `Polynomial<T>` is an interface representing polynomials, and includes subtypes `PolyZero<T>` (zero), `Monomial<T>` (monomial), and `PolySum<T>`: a sum of monomials with coefficients
- `Monomial<T>` is an interface (and a subtype of `Polynomial<T>`) representing terms, and includes subtypes `PolyOne<T>` (one) and `PolyTerm<T>` (a product of variables with integer exponents).
- A polynomial is typically represented using `PolySum`, which maintains a map from monomials to their numeric coefficients. A monomial is defined using immutable variable-exponent maps.
- You will use the above immutable data structures to implement operations such as evaluation, variable extraction, and symbolic polynomial expansion.

Problems 1: Implementing an Expression Evaluator

- `AExpEvaluator<T>` declares the evaluation and expansion of arithmetic expressions of type `AExp<T>`. You must implement the following methods:
  - `getVariables(AExp<T> expr)`: returns a `Set<Integer>` of all variable indices appearing in `expr`. For example, for $x_1 + (x_2 * x_3)$, it returns $\{1, 2, 3\}$.
  - `evaluate(AExp<T> expr, Map<Integer,T> env)`: evaluates `expr` using the given variable assignment `env`.
  - `expand(AExp<T> expr)`: returns a `Polynomial<T>` representing the canonical expansion of the input expression.
- It is highly recommended to use *record pattern matching* to implement the above methods. See the provided implementation of `toPrettyString(AExp<T> expr)` for reference.
- To implement the `expand` method, you will use the methods `negatePoly`, `polySum`, and `polyProduct` from `PolynomialEvaluator<T>`, which are parts of Problem 2.

Problems 2: Implementing a Polynomial Evaluator

- `PolynomialEvaluator<T>` declares operations on canonical polynomials of type `Polynomial<T>`. You must implement the following methods:
  - `getVariables(Polynomial<T> poly)`: returns the set of all variable indices in `poly`.
  - `evaluate(Polynomial<T> poly, Map<Integer,T> env)`: evaluates `poly` under `env`.
  - `toPrettyString(Polynomial<T> poly)`: returns a canonical string representation of the polynomial, where terms and variables are arranged in the predefined order.
  - `polySum(Polynomial<T> p, Polynomial<T> q)`: returns the sum of `p + q`.
  - `polyProduct(Polynomial<T> p, Polynomial<T> q)`: returns the product of `p * q`.
- Again, record pattern matching will be helpful when implementing these methods. Refer to the provided implementation of `negatePoly`.
- You are encouraged to define *private* helper methods to simplify logic and improve code readability. Remember that you should try to write clean and readable code as much as possible.

Problem 3: Writing JUnit Test Cases

- For each public method in `AExpEvaluator` and `PolynomialEvaluator`, write at least one JUnit test case in `AExpEvaluatorTest` and `PolynomialEvaluatorTest`, respectively.
- Each test case should cover one logical scenario and include meaningful assertions. Test both typical and edge cases, such as constants, zero expressions, and empty or singleton polynomials.
- 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 test coverage (measured by JaCoCo) should be at least 90% **branch coverage** for each evaluator class (`AExpEvaluator` and `PolynomialEvaluator`).

General Instruction

- The `src/main` directory contains the skeleton code. You should implement all classes and methods marked with *TODO*. Before writing code, carefully read the descriptions in the source files.
- The `src/test` directory contains test cases. Use JaCoCo to find out the coverage achieved by your tests. Upload the JaCoCo report in CSV (`build/reports/jacoco/test/jacocoTestReport.csv`).
- Do not modify the existing interface, class names, or the signatures of public methods. However, you may add additional private fields and methods if needed.

Turning in

1. We use Github Classroom to manage homework. Click on the following link, and accept the assignment: `https://classroom.github.com/a/QIDCU5bT`
2. Your assignment repository should now be created. Clone the repository, complete your homework (including the JaCoCo coverage report), and push the changes to GitHub before the deadline.
3. The JaCoCo coverage report in CSV should be uploaded to the root directory of your repository (i.e., the same directory where `build.gradle.kts` is located).

Java Reference

- Java Language Specification: `https://docs.oracle.com/javase/specs/`
- Learn Java: `https://dev.java/learn/`
- Core Java, Volume I: Fundamentals 13th by Cay S. Horstmann, Pearson, 2024 (available online at the POSTECH digital library `http://library.postech.ac.kr`)