**Faculty of Computing**

**SE-314: Software Construction**

**Class: BESE 13AB**

# Lab 10: Representing Expression

**CLO-03:** Design and develop solutions based on Software Construction principles.  
**CLO-04:** Use modern tools such as Eclipse, NetBeans etc. for software construction.

**Date: 30th Nov 2024**

**Time: 10:00 AM** **- 12:50 PM   
 02:30 PM – 04:50 PM**

**Instructor: Dr. Mehvish Rashid  
Lab Engineer: Mr. Aftab Farooq**

**Introduction:**

# Lab 10: Representing Expressions

Students will have hands-on experience of representing expressions.

Material:

https://ocw.mit.edu/ans7870/6/6.005/s16/psets/ps3/

Github link: <https://github.com/IronYR/sc-lab-10>

**Lab Tasks**

Solve problem 1 of problem set 3 listed on the link.

Define an immutable, recursive abstract data type to represent expressions as abstract syntax trees.

Your AST should be defined in the provided Expression interface (in Expression.java ) and implemented by several concrete variants, one for each kind of expression. Each variant should be defined in its own appropriately-

named .java file.Concrete syntax in the input, such as parentheses and whitespace, should not be represented at all in your AST.

## Expression

To repeat, your data type must be **immutable** and **recursive** . Follow the recipe for creating an ADT:

* + - **Spec** . Choose and specify operations. For this part of the problem set, the only operations Expression needs are creators and producers for building up an expression, plus the standard observers toString() , equals() , and hashCode() . We are strengthening the specs for these standard methods; see below.
    - **Test** . Partition and test your operations in ExpressionTest.java , including tests for toString() , equals() , and hashCode() . Note that we will not run your test cases on other implementations, just on yours.
    - **Code** . Write the rep for your Expression as a data type definition in a comment inside Expression . Implement the variant classes of your data type.

Remember to include a Javadoc comment above every class and every method you write; define abstraction functions and rep invariants, and write checkRep; and document safety from rep exposure.

## toString()

Define the toString() operation on Expression so it can output itself as a string. This string must be a valid expression as defined above. You have the freedom to decide how to format the output with whitespace and parentheses for readability, but the expression must have the same mathematical meaning.

Your toString()implementation must be recursive, and must not use instanceof . Use the @Override annotation to ensure you are overriding the toString() inherited from Object .

Remember that your tests must obey the spec. If your toString() tests expect a certain formatting of whitespace and parentheses, you must specify this formatting in your spec.

## equals() and hashCode()

Define the equals() and hashCode() operations on your AST to implement

*structural equality* .

**Structural equality** defines two expressions to be equal if:

1. the expressions contain the same variables, numbers, and operators;
2. those variables, numbers, and operators are in the same order, read left-to-right; c and they are grouped in the same way.

For example, the AST for 1 + x is *not* equal to the AST for x + 1 , but it is equal to the ASTs for 1+x , (1+x) , and (1)+(x) .

For *n* -ary groupings where *n* is greater than 2:

* Such expressions must be equal to themselves. For example, the ASTs for 3 + 4 + 5 and (3 + 4 + 5) must be equal.
* However, whether they are equal or not to different groupings with the same mathematical meaning is *not specified* , and you should choose an appropriate specification and implementation for your AST. For example, you must determine whether the ASTs for (3 + 4) + 5 and 3 + (4 + 5) are equal.

For equality of numbers, you have the freedom to choose a reasonable definition. Integers that can be represented exactly as a double should be considered equal. For example, the ASTs for x + 1 and x + 1.000 must be equal.

Remember: concrete syntax, including parentheses, should not be represented in your AST. Grouping, for example, should be reflected in the AST’s structure.

Be sure that AST instances which are considered equal according to this definition and according to equals() also satisfy observational equality .

Your equals() and hashCode() implementations must be recursive. Only

equals() can use instanceof , and hashCode() must not. Remember to use the @Override annotation.

Code:

Expression.java

/\* Copyright (c) 2015-2016 MIT 6.005 course staff, all rights reserved.

\* Redistribution of original or derived work requires permission of course staff.

\*/

package expressivo;

import java.util.Stack;

/\*\*

\* An immutable data type representing a polynomial expression of:

\* + and \*

\* nonnegative integers and floating-point numbers

\* variables (case-sensitive nonempty strings of letters)

\*

\* <p>PS3 instructions: this is a required ADT interface.

\* You MUST NOT change its name or package or the names or type signatures of existing methods.

\* You may, however, add additional methods, or strengthen the specs of existing methods.

\* Declare concrete variants of Expression in their own Java source files.

\*/

public interface Expression {

// Datatype definition

// Expression = Add(left:Expression, right:Expression) +

// Multiplication(left:Expression, right:Expression) +

// Number(number:double) +

/\*\*

\* Parse an expression.

\* **@param** input expression to parse, as defined in the PS3 handout.

\* **@return** expression AST for the input

\* **@throws** IllegalArgumentException if the expression is invalid

\*/

public static Expression parse(String input) {

throw new RuntimeException("unimplemented");

}

/\*\*

\* **@return** a parsable representation of this expression, such that

\* for all e:Expression, e.equals(Expression.parse(e.toString())).

\* For a Expression is not a primitive, subExpression will be surround by

\* parentheses for grouping, and whitespace will be place before and after operator.

\* Subexpression will be group from right to left.

\* If Expression is a number, output an equivalent number, accurate to at least 4 decimal places,

\*/

*@Override*

public String toString();

/\*\*

\* **@param** thatObject any object

\* **@return** true if and only if this and thatObject are structurally-equal

\* Expressions, as defined in the PS3 handout.

\*/

*@Override*

public boolean equals(Object thatObject);

/\*\*

\* **@return** hash code value consistent with the equals() definition of structural

\* equality, such that for all e1,e2:Expression,

\* e1.equals(e2) implies e1.hashCode() == e2.hashCode()

\*/

*@Override*

public int hashCode();

/\*\*

\* **@return** true is this Expression is a primitive element

\*/

public boolean isPrimitive();

/\*\*

\* **@param** that Expression to be add

\* **@return** a new Expression represent add this Expression with other Expression

\*/

public static Expression add(Expression exp1, Expression exp2) {

return new Addition(exp1, exp2);

}

/\*\*

\* **@param** that Expression to be multiplication

\* **@return** a new Expression represent multiplication this Expression with other Expression

\*/

public static Expression multiplication(Expression exp1, Expression exp2) {

return new Multiplication(exp1, exp2);

}

}

Addition.java

package expressivo;

public class Addition implements Expression {

/\*\*

\* Abstraction function:

\* AF(exp1, exp2) = exp1 + exp2

\* Rep invariant:

\* exp1 and exp2 keep its rep invariant

\* Safety from rep exposure:

\* all fields are private and final

\*/

private final Expression exp1, exp2;

public Addition(Expression exp1, Expression exp2) {

this.exp1 = exp1;

this.exp2 = exp2;

}

*@Override* public String toString() {

String left = exp1.isPrimitive() ? exp1.toString() :

"(" + exp1.toString() + ")";

String right = exp2.isPrimitive() ? exp2.toString() :

"(" + exp2.toString() + ")";

return left + "+" + right;

}

*@Override* public boolean equals(Object thatObject) {

if (!(thatObject instanceof Addition)) {

return false;

}

Addition that = (Addition) thatObject;

return exp1.equals(that.exp1) && exp2.equals(that.exp2);

}

*@Override* public int hashCode() {

return (exp1.hashCode() + exp2.hashCode()) % 1000;

}

public boolean isPrimitive() {

return false;

}

}

Multiplication.java

package expressivo;

public class Multiplication implements Expression {

/\*\*

\* Abstraction function:

\* AF(exp1, exp2) = exp1 \* exp2

\* Rep invariant:

\* exp1 and exp2 keep its rep invariant

\* Safety from rep exposure:

\* all fields are private and final

\*/

private final Expression exp1, exp2;

public Multiplication(Expression exp1, Expression exp2) {

this.exp1 = exp1;

this.exp2 = exp2;

}

*@Override* public String toString() {

String left = exp1.isPrimitive() ? exp1.toString() :

"(" + exp1.toString() + ")";

String right = exp2.isPrimitive() ? exp2.toString() :

"(" + exp2.toString() + ")";

return left + "\*" + right;

}

*@Override* public boolean equals(Object thatObject) {

if (!(thatObject instanceof Multiplication)) {

return false;

}

Multiplication that = (Multiplication) thatObject;

return exp1.equals(that.exp1) && exp2.equals(that.exp2);

}

*@Override* public int hashCode() {

return (exp1.hashCode() \* exp2.hashCode()) % 1000;

}

public boolean isPrimitive() {

return false;

}

}

Number.java

package expressivo;

public class Number implements Expression {

private final double value;

// Abstraction function:

// represents the value of this number

// Rep invariant:

// value >= 0

// Safety from rep exposure:

// value is final and immutable

/\*\*

\* Create a Number instance.

\*

\* **@param** value nonnegative value

\*/

public Number(double value) {

this.value = value;

checkRep();

}

/\*\*

\* Returns the value of this number.

\*

\* **@return** the number's value

\*/

public double getValue() {

return value;

}

/\*\*

\* Checks the rep invariant.

\*/

private void checkRep() {

assert value >= 0;

}

*@Override*

public String toString() {

return String.*valueOf*(value);

}

*@Override*

public boolean equals(Object obj) {

if (!(obj instanceof Number)) return false;

Number other = (Number) obj;

return this.value == other.value;

}

*@Override*

public int hashCode() {

return Double.*hashCode*(value);

}

public boolean isPrimitive() {;

return true;

}

}

ExpressionTest.java

/\* Copyright (c) 2015-2016 MIT 6.005 course staff, all rights reserved.

\* Redistribution of original or derived work requires permission of course staff.

\*/

package expressivo;

import static org.junit.Assert.\*;

import org.junit.Test;

/\*\*

\* Tests for the Expression abstract data type.

\*/

public class ExpressionTest {

// Testing strategy

// toString():

// - test primitive numbers

// - test simple addition

// - test simple multiplication

// - test nested expressions with addition and multiplication

// - test complex expressions with multiple operations

//

// equals():

// - test same numbers are equal

// - test different numbers are not equal

// - test same expressions with same structure are equal

// - test same expressions with different structure are not equal

// - test with null and different types

//

// hashCode():

// - test same expressions have same hashcode

// - test different numbers have different hashcodes

// - test different expressions have different hashcodes

// - test complex expressions maintain consistency

*@Test*(expected=AssertionError.class)

public void testAssertionsEnabled() {

assert false; // make sure assertions are enabled with VM argument: -ea

}

// toString() tests

*@Test*

public void testToStringNumber() {

Expression num = new Number(42.0);

*assertEquals*("42.0", num.toString());

}

*@Test*

public void testToStringSimpleAddition() {

Expression sum = Expression.*add*(new Number(1.0), new Number(2.0));

*assertEquals*("1.0+2.0", sum.toString());

}

*@Test*

public void testToStringSimpleMultiplication() {

Expression product = Expression.*multiplication*(new Number(2.0), new Number(3.0));

*assertEquals*("2.0\*3.0", product.toString());

}

*@Test*

public void testToStringNestedExpression() {

Expression inner = Expression.*multiplication*(new Number(2.0), new Number(3.0));

Expression outer = Expression.*add*(new Number(1.0), inner);

*assertEquals*("1.0+(2.0\*3.0)", outer.toString());

}

*@Test*

public void testToStringComplexExpression() {

Expression e1 = Expression.*multiplication*(new Number(2.0), new Number(3.0));

Expression e2 = Expression.*multiplication*(new Number(4.0), new Number(5.0));

Expression sum = Expression.*add*(e1, e2);

*assertEquals*("(2.0\*3.0)+(4.0\*5.0)", sum.toString());

}

// equals() tests

*@Test*

public void testEqualsSameNumbers() {

Expression num1 = new Number(42.0);

Expression num2 = new Number(42.0);

*assertTrue*(num1.equals(num2));

}

*@Test*

public void testEqualsDifferentNumbers() {

Expression num1 = new Number(42.0);

Expression num2 = new Number(24.0);

*assertFalse*(num1.equals(num2));

}

*@Test*

public void testEqualsSameStructure() {

Expression e1 = Expression.*add*(new Number(1.0), new Number(2.0));

Expression e2 = Expression.*add*(new Number(1.0), new Number(2.0));

*assertTrue*(e1.equals(e2));

}

*@Test*

public void testEqualsDifferentStructure() {

Expression e1 = Expression.*add*(new Number(1.0), new Number(2.0));

Expression e2 = Expression.*multiplication*(new Number(1.0), new Number(2.0));

*assertFalse*(e1.equals(e2));

}

*@Test*

public void testEqualsNull() {

Expression num = new Number(42.0);

*assertFalse*(num.equals(null));

*assertFalse*(num.equals("42.0"));

}

// hashCode() tests

*@Test*

public void testHashCodeConsistency() {

Expression e1 = Expression.*add*(new Number(1.0), new Number(2.0));

Expression e2 = Expression.*add*(new Number(1.0), new Number(2.0));

*assertEquals*(e1.hashCode(), e2.hashCode());

}

*@Test*

public void testHashCodeDifferentNumbers() {

Expression num1 = new Number(42.0);

Expression num2 = new Number(24.0);

*assertNotEquals*(num1.hashCode(), num2.hashCode());

}

*@Test*

public void testHashCodeDifferentExpressions() {

Expression e1 = Expression.*add*(new Number(1.0), new Number(2.0));

Expression e2 = Expression.*multiplication*(new Number(1.0), new Number(2.0));

*assertNotEquals*(e1.hashCode(), e2.hashCode());

}

*@Test*

public void testHashCodeComplexExpression() {

Expression e1 = Expression.*add*(

Expression.*multiplication*(new Number(2.0), new Number(3.0)),

Expression.*multiplication*(new Number(4.0), new Number(5.0))

);

Expression e2 = Expression.*add*(

Expression.*multiplication*(new Number(2.0), new Number(3.0)),

Expression.*multiplication*(new Number(4.0), new Number(5.0))

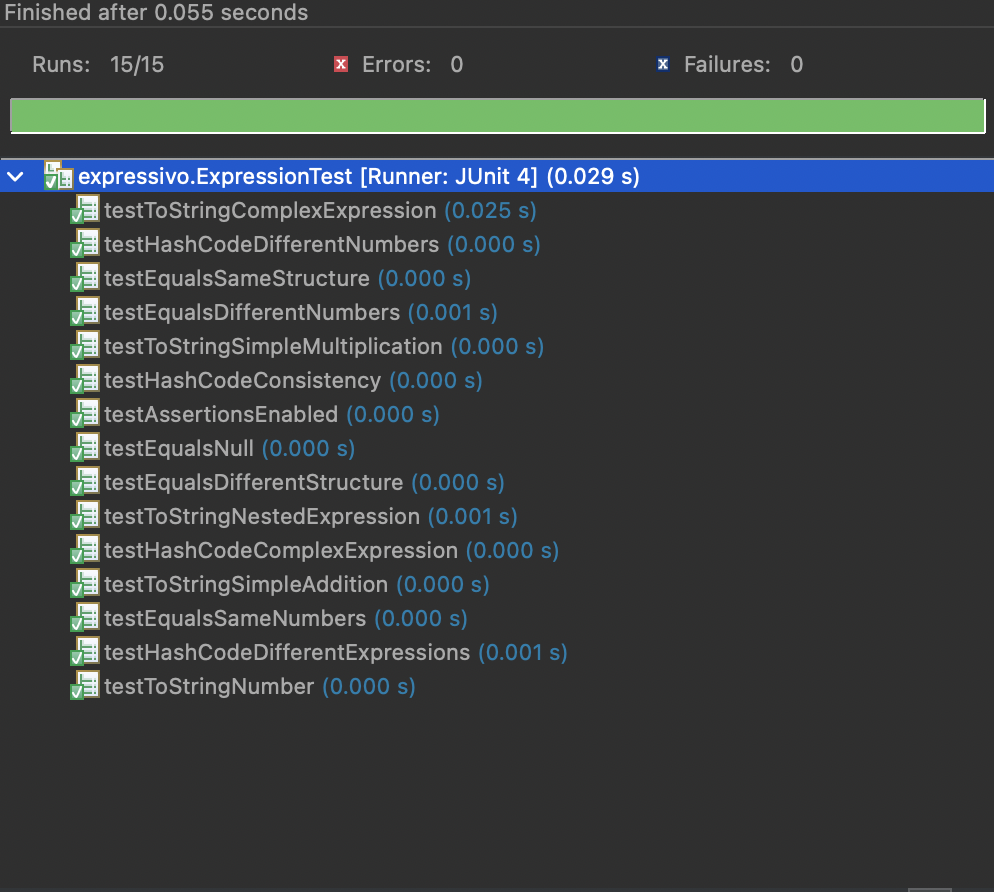
);

*assertEquals*(e1.hashCode(), e2.hashCode());

}

}

Screenshots:



Github link: <https://github.com/IronYR/sc-lab-10>

**- Push Your Code on GitHub  
- Add Git Link in Document.**

**Source Code: Zip your source code and upload one file (Including Git link) on LMS as well.**

**Solution**

## Deliverables

Compile a single word document by filling in the solution part and submit this Word file on LMS. In case of any problems with submissions on LMS, submit your Lab assignments by emailing it to [aftab.farooq@seecs.edu.pk.](mailto:aftab.farooq@seecs.edu.pk.)