**Software Construction Lab 11**

Github Link: <https://github.com/RyanSikandar/SC-Lab11>

**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 expressivo.parser.ExpressionLexer;

import expressivo.parser.ExpressionMainVisitor;

import expressivo.parser.ExpressionParser;

import java.util.Map;

import org.antlr.v4.runtime.CharStream;

import org.antlr.v4.runtime.CharStreams;

import org.antlr.v4.runtime.CommonTokenStream;

import org.antlr.v4.runtime.misc.ParseCancellationException;

import org.antlr.v4.runtime.tree.ParseTree;

/\*\*

 \* 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 = Value(num:double)

    //                + Variable(id:String)

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

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

    /\*\*

     \* Parse an expression.

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

     \* @return expression AST for the input, simplified as much as possible

     \* @throws IllegalArgumentException if the expression is invalid

     \*/

    // TODO: 'human-readable' error messages

    public static Expression parse(String input) {

        assert input != null && input != "";

        try {

            CharStream inputStream = CharStreams.fromString(input);

            ExpressionLexer lexer = new ExpressionLexer(inputStream);

            lexer.reportErrorsAsExceptions();

            CommonTokenStream tokens = new CommonTokenStream(lexer);

            ExpressionParser parser = new ExpressionParser(tokens);

            parser.reportErrorsAsExceptions();

            parser.setBuildParseTree(true);

            ParseTree parseTree = parser.root();

            ExpressionMainVisitor exprVisitor = new ExpressionMainVisitor();

            Expression expr = exprVisitor.visit(parseTree);

            return expr;

        } catch (ParseCancellationException e) {

            throw new IllegalArgumentException(e.getMessage());

        }

    }

    /\*\* Creates an empty expression such that Expression.parse("0").equals(emptyExpression())  \*/

    public static Expression emptyExpression() {

        return new Value(0.0);

    }

    /\*\*

     \* Appends an expression at the end of this with an addition

     \*

     \* If e equals Expression.emptyExpression(), correct to 5 decimal places,

     \* the empty expression is returned;

     \* If e equals this, an expression equivalent to

     \*      this \* 2 is returned

     \*

     \* @param e a non-null non-empty string of a valid expression

     \*          syntax

     \* @return a simplified expression equivalent to:

     \*           this + e

     \*      this and e are not modified

     \*/

    public Expression addExpr(Expression e);

    /\*\*

     \* Appends an expression at the end of this with a multiplication

     \*

     \* If e equals Expression.emptyExpression(), correct to 5 decimal places,

     \* the empty expression is returned;

     \* If e equals Expression.parse("1"), correct to 5 decimal places, this

     \* expression is returned

     \* The product of any other expression except the two above is not simplified,

     \* the resulting expression being equivalent to:

     \*      (this)\*(e)

     \* Note: This is not the case when parsing, where an expression is simplified

     \* as much as possible

     \*

     \* @param e a non-null non-empty string of a valid expression

     \*          syntax

     \* @return a new expression equivalent to:

     \*           this \* e

     \*      The returned expression is NOT simplified

     \*      this and e are not modified

     \*/

    public Expression multiplyExpr(Expression e);

    /\*\*

     \* Appends a variable at the start of this expression with an addition

     \*

     \* @param variable non-null non-empty case-sensitive string of letters, a-zA-Z

     \* @return a new expression as a result of inserting a variable at the start

     \*         of this expression with an addition.

     \*         The expression is not simplified

     \*

     \*/

    public Expression addVariable(String variable);

    /\*\*

     \* Appends a variable as a multiplicative factor to start of this expression

     \*

     \* @param variable non-null non-empty case-sensitive string of letters, a-zA-Z

     \* @return the product expression of this and variable, variable being at

     \*         the head of the expression. The expression is not simplified

     \*/

    public Expression multiplyVariable(String variable);

    /\*\*

     \* Adds a number at the start of this expression

     \*

     \* @param num nonnegative integer or floating-point number

     \* @return the result adding num at the start of this expression.

     \*      If e equals Expression.emptyExpression(), correct to 5 decimal places,

     \*      the empty expression is returned;

     \*      The expression is not simplified

     \*/

    public Expression addConstant(double num);

    /\*\*

     \* Appends a number as a multiplicative factor at the start of this expression

     \*

     \* @param num nonnegative integer or floating-point number

     \* @return the product expression where num is this expression's coefficient,

     \*         placed at the start of this expression.

     \*      - If e equals Expression.emptyExpression(), correct to 5 decimal places,

     \*        the empty expression is returned;

     \*      - If e equals Expression.parse("1"), correct to 5 decimal places, this

     \*        expression is returned

     \*      The expression is simplified

     \*/

    public Expression appendCoefficient(double num);

    /\*\*

     \* Substitutes a variable in this expression with a number

     \*

     \* The set of variables in the environment can contain variables not

     \* in this expression:

     \*  - Any variables in the expression but not the environment

     \*    remain as variables in the substituted polynomial.

     \*  - Any variables in the environment but not the expression are simply ignored.

     \* If the substituted polynomial is a constant expression, with no variables remaining,

     \* then simplification reduces it to a single number, with no operators remaining.

     \*

     \* @param environment maps variables to values.  Variables are required to be case-sensitive nonempty

     \*         strings of letters.  The set of variables in environment is allowed to be different than the

     \*         set of variables actually found in expression.  Values must be nonnegative numbers.

     \* @return an expression equal to the input, but after substituting every variable v that appears in both

     \*         the expression and the environment with its value, environment.get(v).  If there are no

     \*         variables left in this expression after substitution, it must be evaluated to a single number.

     \*/

    public Expression substitute(Map<String,Double> environment);

    /\*\*

     \* Produces an expression with the derivative of this expression

     \* with respect to an input variable

     \*

     \* @param variable non-null non-empty case-sensitive string of letters, a-zA-Z

     \* @return the derivative of this expression with respect

     \*         to variable. The returned expression is equal to the derivative,

     \*         simplified as much as possible.

     \*/

    public Expression differentiate(String variable);

    /\*\*

     \* Returns a string representation of this expression

     \*

     \* The string returned is such that:

     \*   - for additions, exactly one space exists between

     \*     operand and the operator:

     \*          operand + operand

     \*   - for multiplications, no space exists between operands

     \*     and the operator, and operands are inside parentheses:

     \*          (factor)\*(factor)

     \*     Factors of products are grouped from left to right by default:

     \*          x\*x\*x -> ((x)\*(x))\*(x)

     \* Numbers in the string are truncated and correct to 5 decimal places

     \*

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

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

     \*/

    @Override public String toString();

    /\*\*

     \* Checks if an object is equal to this addition expression

     \* Two expressions are equal if and only if:

     \*   - The expressions contain the same variables, numbers, and operators;

     \*   - those variables, numbers, and operators are in the same order, read left-to-right;

     \*   - and they are grouped in the same way.

     \* Two sums are equal if having different groupings with

     \* the same mathematical meaning. For example,

     \*     (3 + 4) + 5 and 3 + (4 + 5) are equal.

     \* However, two products are NOT equal if they have different groupings regardless

     \* of mathematical meaning. For example:

     \*     x\*(2\*y) is not equal to (x\*2)\*y

     \* @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();

}

**Commands.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.Map;

/\*\*

 \* String-based commands provided by the expression system.

 \*

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

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

 \* You MUST NOT add fields, constructors, or instance methods.

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

 \*/

public class Commands {

    /\*\*

     \* Differentiate an expression with respect to a variable.

     \* @param expression the expression to differentiate

     \* @param variable the variable to differentiate by, a case-sensitive nonempty string of letters.

     \* @return expression's derivative with respect to variable.  Must be a valid expression equal

     \*         to the derivative, but doesn't need to be in simplest or canonical form.

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

     \*/

    public static String differentiate(String expression, String variable) {

        assert expression != null && expression != "";

        assert variable != null && variable != "";

        Expression expr = Expression.parse(expression);

        Expression deriv = expr.differentiate(variable);

        return deriv.toString();

    }

    /\*\*

     \* Simplify an expression.

     \* @param expression the expression to simplify

     \* @param environment maps variables to values.  Variables are required to be case-sensitive nonempty

     \*         strings of letters.  The set of variables in environment is allowed to be different than the

     \*         set of variables actually found in expression.  Values must be nonnegative numbers.

     \* @return an expression equal to the input, but after substituting every variable v that appears in both

     \*         the expression and the environment with its value, environment.get(v).  If there are no

     \*         variables left in this expression after substitution, it must be evaluated to a single number.

     \*         Additional simplifications to the expression may be done at the implementor's discretion.

     \* @throws IllegalArgumentException if the expression is invalid

     \*/

    public static String simplify(String expression, Map<String,Double> environment) {

        assert expression != null && expression != "";

        assert environment != null;

        Expression expr = Expression.parse(expression);

        Expression simpExpr = expr.substitute(environment);

        return simpExpr.toString();

    }

}

**CommandsTest.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 java.util.HashMap;

import java.util.Map;

import org.junit.Test;

/\*\*

 \* Tests for the static methods of Commands.

 \*/

public class CommandsTest {

    // Testing strategy

    //   Partitions for differentiate: Expression x String -> Expression:

    //    expression: contains one variable,

    //              multiple variables,

    //              one operator,

    //              multiple operators.

    //     where operators are addition and multiplication

    //     include expressions having groups

    //    variable: exists in expression,

    //              doesn't exist in expression

    //

    //   Partitions for simplify: Expression x Map -> Expression

    //     expression: contains no variable in map,

    //              contains one variable in map,

    //              contains all the variables in map,

    //              contains multiple variables in map

    @Test(expected=AssertionError.class)

    public void testAssertionsEnabled() {

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

    }

    // Tests for Command.differentiate()

    @Test

    // covers one operator, addition,

    //        multiple variables,

    //        variable exists in expression

    public void testDifferentiate\_Addition() {

        String inputExpr = "x + y + x";

        Expression expectedExpr = Expression.parse(inputExpr).differentiate("x");

        String expectedString = expectedExpr.toString();

        String outputExpr = Commands.differentiate(inputExpr, "x");

        assertNotEquals("Expected non-null string",

                null, outputExpr);

        assertNotEquals("Expected non-empty string",

                "", outputExpr);

        assertEquals("Expected derived expression",

                expectedString, outputExpr);

        assertEquals("Expected a valid expression",

                expectedExpr, Expression.parse(outputExpr));

    }

    @Test

    // covers one operator, addition,

    //        multiple variables,

    //        variable doesnt exist in expression

    public void testDifferentiate\_AddVarNotExist() {

        String inputExpr = "x + y + x";

        Expression expectedExpr = Expression.parse(inputExpr).differentiate("foo");

        String expectedString = expectedExpr.toString();

        String outputExpr = Commands.differentiate(inputExpr, "foo");

        assertNotEquals("Expected non-null string",

                null, outputExpr);

        assertNotEquals("Expected non-empty string",

                "", outputExpr);

        assertEquals("Expected derived expression",

                expectedString, outputExpr);

        assertEquals("Expected a valid expression",

                expectedExpr, Expression.parse(outputExpr));

    }

    @Test

    // covers multiple operators,

    //        multiple variables,

    //        variable exists in expression

    public void testDifferentiate\_AddMult() {

        String inputExpr = "x\*y + x";

        Expression expectedExpr = Expression.parse(inputExpr).differentiate("x");

        String expectedString = expectedExpr.toString();

        String outputExpr = Commands.differentiate(inputExpr, "x");

        assertNotEquals("Expected non-null string",

                null, outputExpr);

        assertNotEquals("Expected non-empty string",

                "", outputExpr);

        assertEquals("Expected derived expression",

                expectedString, outputExpr);

        assertEquals("Expected a valid expression",

                expectedExpr, Expression.parse(outputExpr));

    }

    @Test

    // covers one operator, multiplication,

    //        multiple variables,

    //        variable exists in expression

    public void testDifferentiate\_Mult() {

        String inputExpr = "x \* (x \* y)";

        Expression expectedExpr = Expression.parse(inputExpr).differentiate("x");

        String expectedString = expectedExpr.toString();

        String outputExpr = Commands.differentiate(inputExpr, "x");

        assertNotEquals("Expected non-null string",

                null, outputExpr);

        assertNotEquals("Expected non-empty string",

                "", outputExpr);

        assertEquals("Expected derived expression",

                expectedString, outputExpr);

        assertEquals("Expected a valid expression",

                expectedExpr, Expression.parse(outputExpr));

    }

    @Test

    // covers one operator, multiplication,

    //        multiple variables,

    //        variable doesnt exist in expression

    public void testDifferentiate\_MultVarNotExist() {

        String inputExpr = "x\*(y + x)";

        Expression expectedExpr = Expression.parse(inputExpr).differentiate("foo");

        String expectedString = expectedExpr.toString();

        String outputExpr = Commands.differentiate(inputExpr, "foo");

        assertNotEquals("Expected non-null string",

                null, outputExpr);

        assertNotEquals("Expected non-empty string",

                "", outputExpr);

        assertEquals("Expected derived expression",

                expectedString, outputExpr);

        assertEquals("Expected a valid expression",

                expectedExpr, Expression.parse(outputExpr));

    }

    @Test

    // covers multiple operators,

    //        multiple variables,

    //        variable exists in expression,

    //        grouping

    public void testDifferentiate\_MultAdd() {

        String inputExpr = "x\*(y + x)";

        Expression expectedExpr = Expression.parse(inputExpr).differentiate("x");

        String expectedString = expectedExpr.toString();

        String outputExpr = Commands.differentiate(inputExpr, "x");

        assertNotEquals("Expected non-null string",

                null, outputExpr);

        assertNotEquals("Expected non-empty string",

                "", outputExpr);

        assertEquals("Expected derived expression",

                expectedString, outputExpr);

        assertEquals("Expected a valid expression",

                expectedExpr, Expression.parse(outputExpr));

    }

    // Tests for Commands.simplify()

    @Test

    // covers expression contains no variable in map

    public void testSimplify\_NotExist() {

        String expr = "m\*x + c";

        Map<String, Double> env = new HashMap<>();

        env.put("PI", 3.142);

        env.put("radius", 12.0);

        String actual = Commands.simplify(expr, env);

        String expected = Expression.parse(expr).toString();

        assertNotNull("Expected non-null string expression", actual);

        assertNotEquals("Expected non-empty string", "", actual);

        assertEquals("Expected unchanged string", expected, actual);

    }

    @Test

    // covers one variable in expression and map

    public void testSimplify\_OneVar() {

        String expr = "PI\*diameter";

        Map<String, Double> env = new HashMap<>();

        env.put("PI", 3.142);

        env.put("radius", 12.0);

        String actual = Commands.simplify(expr, env);

        String expected = Expression.parse("3.142\*diameter").toString();

        assertNotNull("Expected non-null string expression", actual);

        assertNotEquals("Expected non-empty string", "", actual);

        assertEquals("Expected variable substituted", expected, actual);

    }

    @Test

    // covers all variables in expression are in map

    public void testSimplify\_AllVars() {

        String expr = "PI \* (radius + radius)";

        Map<String, Double> env = new HashMap<>();

        env.put("PI", 3.142);

        env.put("radius", 12.0);

        String actual = Commands.simplify(expr, env);

        String expected = String.valueOf(3.142 \* (12.0 + 12.0));

        assertNotNull("Expected non-null string expression", actual);

        assertNotEquals("Expected non-empty string", "", actual);

        assertEquals("Expected all variables substituted and simplified",

                expected, actual);

    }

    @Test

    // covers multiple variables in both expression and map

    public void testSimplify\_MultipleVars() {

        String expr = "0.5\*length\*width + PI\*(length\*0.5)";

        Map<String, Double> env = new HashMap<>();

        env.put("PI", 3.142);

        env.put("length", 4.0);

        env.put("width", 2.0);

        String actual = Commands.simplify(expr, env);

        String expected = Expression.parse("(0.5\*4\*2) + (3.142\*4\*0.5)").toString();

        assertNotNull("Expected non-null string expression", actual);

        assertNotEquals("Expected non-empty string", "", actual);

        assertEquals("Expected variables substituted and simplified",

                expected, actual);

    }

}

**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 java.util.HashMap;

import java.util.Map;

import org.junit.Test;

/\*\*

 \* Tests for the concrete variants of Expression

 \*/

public class ExpressionTest {

    // Testing Strategy:

    //  Partition for Expression.parse: String -> Expression

    //   For operators + and \*, the input string to contain:

    //     one operator,

    //     multiple operators of the same type,

    //     multiple operators of different types.

    //   include inputs with grouping

    //

    //  Partition for addExpr: Expression x Expression -> Expression

    //    an empty expression,

    //    contains multiple variables,

    //    input expression as a subset of this expression,

    //    input expression equals this

    //

    //  Partition for multiplyExpr: Expression x Expression -> Expression

    //    an empty expression,

    //    expression as the value 1,

    //    contains multiple variables

    //    input expression as a subset of this expression

    //    input expression equals this

    //

    //  Partition for addVariable: Expression x String -> Expression

    //    this: empty expression,

    //          doesn't contain input variable,

    //          contains input variable

    //

    //  Partition for multiplyVariable: Expression x String -> Expression

    //    this: empty expression,

    //          expression as a single value 1,

    //          doesn't contain input variable,

    //          contains input variable

    //

    //  Partition for addConstant: Expression x double -> Expression

    //    this: empty expression,

    //          expression contains the value

    //    value: 0, > 0

    //

    //  Partition for appendCoefficient: Expression x double -> Expression

    //    this: empty expression,

    //          expression as the value 1,

    //    coefficient: 0, 1, > 1

    //

    //  Partition for substitute: Expression x Map -> Expression

    //    variables in the expression but not in the input string,

    //    variables in the input string but not in the expression,

    //    one variable in both the expression and the input string,

    //    multiple variables in both

    //

    //  Partition for differentiate: Expression x String -> Expression

    //    this: empty expression,

    //          expression as a single value,

    //          does not contain input variable,

    //          contains multiple instances of variable

    //

    //  Partition for toString: Expression -> String

    //    this: empty expression,

    //          contains multiple variables and values

    //

    //  Partition for equals: Expression x Expression -> boolean

    //    reflexive, symmetric and transitive equality

    //    include tests for numbers correct to 5 decimal places

    //

    //  Partition for hashCode: Expression -> int

    //    include equal expressions having equal values equal correct

    //    to 5 decimal places

    final Expression empty = Expression.emptyExpression();

    final Expression one = Expression.parse("1.000009");

    final Expression expr = Expression.parse("x\*y + x + 0.5");

    final Expression expr1 = Expression.parse("x\*y\*0.5");

    @Test(expected=AssertionError.class)

    public void testAssertionsEnabled() {

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

    }

    //Tests for Expression.parse()

    @Test

    // covers single addition expression

    public void testParse\_SingleAddExpr() {

        String input = "2 + x";

        Expression result = Expression.parse(input);

        assertNotNull("Expected non-null expression", result);

        assertEquals("Expected correct parse", input, result.toString());

    }

    @Test

    // covers multiple additions

    public void testParse\_MultipleAddExprs() {

        String input = "(x + x + y)";

        Expression result = Expression.parse(input);

        String actual = result.toString();

        String expected = "x + x + y";

        assertNotNull("Expected non-null expression", result);

        assertEquals("Expected correct parse", expected, actual);

    }

    @Test

    // covers single multiplication expression

    public void testParse\_SingleMult() {

        String input = "x\*y";

        Expression result = Expression.parse(input);

        String actual = result.toString();

        String expected = "(x)\*(y)";

        assertNotNull("Expected non-null expression", result);

        assertEquals("Expected correct parse", expected, actual);

    }

    @Test

    // covers multiple multiplication expressions

    public void testParse\_MultipleMults() {

        String input = "x\*2\*x";

        Expression result = Expression.parse(input);

        String actual = result.toString();

        String expected = "((x)\*(2))\*(x)";

        assertNotNull("Expected non-null expression", result);

        assertEquals("Expected correct parse", expected, actual);

    }

    @Test

    // covers addition, multiplication and grouping

    public void testParse\_AddExprMult() {

        String input = "(x + (2.12)\*(x))\*(y)";

        Expression result = Expression.parse(input);

        String actual = result.toString();

        assertNotNull("Expected non-null expression", result);

        assertEquals("Expected correct parse", input, actual);

    }

    // Tests for addExpr()

    @Test

    // covers empty expression

    public void testAddExpr\_Empty() {

        Expression actual1 = empty.addExpr(expr);

        Expression actual2 = expr1.addExpr(empty);

        assertEquals("Expected expression + 0 = expression",

                expr, actual1);

        assertEquals("Expected expression + 0 = expression",

                expr1, actual2);

    }

    @Test

    // covers

    public void testAddExpr\_Subset() {

        Expression subset = Expression.parse("x + 0.5");

        Expression actual = expr.addExpr(subset);

        Expression expected = Expression.parse(expr.toString() + "+" + subset.toString());

        assertEquals("Expected expression added at the end",

                expected, actual);

    }

    @Test

    // covers input equals this

    //        input as a subset

    public void testAddExpr\_EqualsThis() {

        Expression actual = expr.addExpr(expr);

        Expression expected = Expression.parse("(x\*y + x)\*2 + 1");

        assertEquals("Expected simplified expression",

                expected, actual);

    }

    // Tests for multiplyExpr()

    @Test

    // covers empty expression

    public void testMultiplyExpr\_Empty() {

        Expression actual1 = empty.multiplyExpr(expr);

        Expression actual2 = expr.multiplyExpr(empty);

        assertEquals("Expected 0\*expression = 0",

                empty, actual1);

        assertEquals("Expected expression\*0 = 0",

                empty, actual2);

    }

    @Test

    // covers expression as the value 1

    public void testMultiplyExpr\_One() {

      Expression actual1 = one.multiplyExpr(expr);

      Expression actual2 = expr.multiplyExpr(one);

      assertEquals("Expected 1\*expression = expression",

              expr, actual1);

      assertEquals("Expected expression\*1 = expression",

              expr, actual2);

    }

    @Test

    // covers expression contains multiple variables,

    //        input as a subset

    public void testMultiplyExpr\_MultipleVars() {

        Expression subset = Expression.parse("x + 0.500009");

        Expression actual = expr.multiplyExpr(subset);

        String expected = "((x)\*(y) + x + 0.5)\*(x + 0.5)";

        assertEquals("Expected non-simplified expression \* subset",

                expected, actual.toString());

    }

    @Test

    // covers expression contains multiple variables,

    //        input as a subset

    //        input equals this

    public void testMultiplyExpr\_Equals() {

        Expression actual = expr.multiplyExpr(expr);

        String expected = "(" + expr.toString() + ")" + "\*(" + expr.toString() + ")";

        assertEquals("Expected non-simplified expression \* subset",

                expected, actual.toString());

    }

    @Test

    // covers empty expression

    public void testAddVariable\_Empty() {

        Expression actual = empty.addVariable("x");

        String expected = "x";

        assertEquals("Expected expression + 0 = expression",

                expected, actual.toString());

    }

    @Test

    // covers input variable does not exist in expression

    public void testAddVariable\_NotExist() {

        Expression actual = expr.addVariable("foo");

        Expression expected = Expression.parse("foo + " + expr.toString());

        assertEquals("Expected variable added at the start",

                expected, actual);

    }

    @Test

    // covers contains input variable

    public void testAddVariable\_Exists() {

        Expression actual1 = expr.addVariable("y");

        Expression actual2 = Expression.parse("x\*y").addVariable("x");

        Expression actual3 = expr.addVariable("x");

        Expression expected1 = Expression.parse("y + " + expr.toString());

        Expression expected2 = Expression.parse("x + x\*y");

        Expression expected3 = Expression.parse("x + " + expr.toString());

        assertEquals("Expected variable added at the start",

                expected1, actual1);

        assertEquals("Expected variable added at the start",

                expected2, actual2);

        assertNotEquals("Expected expression not simplified",

                expected3, actual3);

    }

    //Tests for addConstant()

    @Test

    // covers empty expression,

    //        value > 0

    public void testAddConstant\_Empty() {

        Expression actual = empty.addConstant(1);

        assertEquals("Expected 1 + 0 = 1",

                "1", actual.toString());

    }

    @Test

    // covers non-empty expression,

    //        value = 0

    public void testAddConstant\_ValZero() {

        Expression actual1 = expr.addConstant(0);

        Expression actual2 = expr1.addConstant(0);

        assertEquals("Expected 0 + expression = expression",

                expr, actual1);

        assertEquals("Expected 0 + expression = expression",

                expr1, actual2);

    }

    @Test

    // covers non-empty expression,

    //        value > 0

    public void testAddConstant\_Expr() {

        Expression actual1 = expr.addConstant(12.000009);

        Expression actual2 = expr1.addConstant(3.142);

        Expression actual3 = actual2.addConstant(12);

        Expression expected1 = Expression.parse(12 + "+" + expr.toString());

        Expression expected2 = Expression.parse(3.142 + "+" + expr1.toString());

        Expression expected3 = Expression.parse(12 + "+" + actual2.toString());

        assertEquals("Expected constant added at the start",

                expected1, actual1);

        assertEquals("Expected constant added at the start",

                expected2, actual2);

        assertNotEquals("Expected constant added at the start, expression not simplified",

                expected3, actual3);

    }

    // Tests for appendCoefficient()

    @Test

    // covers coefficient = 0

    public void testAppendCoefficient\_CoeffZero() {

        Expression actual1 = expr.appendCoefficient(0);

        Expression actual2 = expr1.appendCoefficient(0);

        assertEquals("Expected 0 \* expression = 0",

                empty, actual1);

        assertEquals("Expected 0 \* expression = 0",

                empty, actual2);

    }

    @Test

    // covers coefficient = 1

    public void testAppendCoefficient\_CoeffOne() {

        Expression actual1 = expr.appendCoefficient(1);

        Expression actual2 = expr1.appendCoefficient(1);

        assertEquals("Expected 1 \* expression = expression",

                expr, actual1);

        assertEquals("Expected 1 \* expression = expression",

                expr1, actual2);

    }

    @Test

    // covers empty expression

    public void testAppendCoefficient\_Empty() {

        Expression actual1 = empty.appendCoefficient(12.2);

        Expression actual2 = empty.appendCoefficient(1);

        assertEquals("Expected value \* 0 = 0",

                empty, actual1);

        assertEquals("Expected value \* 0 = 0",

                empty, actual2);

    }

    @Test

    // covers expression = 1

    public void testAppendCoefficient\_One() {

        Expression actual1 = one.appendCoefficient(12.2);

        Expression actual2 = one.appendCoefficient(1);

        assertEquals("Expected value \* 1 = value",

                "12.2", actual1.toString());

        assertEquals("Expected value \* 1 = value",

                "1", actual2.toString());

    }

    @Test

    public void testAppendCoefficient\_Expr() {

        Expression actual1 = expr.appendCoefficient(2);

        Expression actual2 = expr1.appendCoefficient(3.142);

        Expression expected1 = Expression.parse("(2)\*(" + expr.toString() + ")");

        Expression expected2 = Expression.parse("(3.142)\*(" + expr1.toString() + ")");

        assertEquals("Expected value multiplied at the start and expression simplified",

                expected1, actual1);

        assertEquals("Expected value multiplied at the start and expression simplified",

                expected2, actual2);

    }

    // Tests for substitute()

    @Test

    // covers expression contains no variable in map

    public void testSubstitute\_NotExist() {

        Expression expr = Expression.parse("m\*x + c");

        Map<String, Double> env = new HashMap<>();

        env.put("PI", 3.142);

        env.put("radius", 12.0);

        Expression actual = expr.substitute(env);

        Expression expected = expr;

        assertNotNull("Expected non-null string expression", actual);

        assertNotEquals("Expected non-empty string", "", actual);

        assertEquals("Expected unchanged string", expected, actual);

    }

    @Test

    // covers one variable in expression and map

    public void testSubstitute\_OneVar() {

        Expression expr = Expression.parse("PI\*diameter");

        Map<String, Double> env = new HashMap<>();

        env.put("PI", 3.142);

        env.put("radius", 12.0);

        Expression actual = expr.substitute(env);

        Expression expected = Expression.parse("3.142\*diameter");

        assertNotNull("Expected non-null string expression", actual);

        assertNotEquals("Expected non-empty string", "", actual);

        assertEquals("Expected variable substituted", expected, actual);

    }

    @Test

    // covers all variables in expression are in map

    public void testSubstitute\_AllVars() {

        String exprString = "PI \* (radius + radius)";

        Expression expr = Expression.parse(exprString);

        Map<String, Double> env = new HashMap<>();

        env.put("PI", 3.142);

        env.put("radius", 12.0);

        Expression actual = expr.substitute(env);

        String expected = String.valueOf(3.142 \* (12.0 + 12.0));

        assertNotNull("Expected non-null string expression", actual);

        assertNotEquals("Expected non-empty string", "", actual);

        assertEquals("Expected all variables substituted and simplified",

                expected, actual.toString());

    }

    @Test

    // covers multiple variables in both expression and map

    public void testSubstitute\_MultipleVars() {

        String exprString = "0.5\*length\*width + PI\*(length\*0.5)";

        Expression expr = Expression.parse(exprString);

        Map<String, Double> env = new HashMap<>();

        env.put("PI", 3.142);

        env.put("length", 4.0);

        env.put("width", 2.0);

        Expression actual = expr.substitute(env);

        Expression expected = Expression.parse("(0.5\*4\*2) + (3.142\*4\*0.5)");

        assertNotNull("Expected non-null string expression", actual);

        assertNotEquals("Expected non-empty string", "", actual);

        assertEquals("Expected variables substituted and simplified",

                expected, actual);

    }

    // Tests for differentiate()

    @Test

    // covers one operator, addition,

    //        multiple variables,

    //        variable exists in expression

    public void testDifferentiate\_Addition() {

        Expression expr = Expression.parse("x + y + x");

        Expression actual = expr.differentiate("x");

        String expected = "2";

        assertNotNull("Expected non-null expression", actual);

        assertEquals("Expected simplified derived expression",

                expected, actual.toString());

    }

    @Test

    // covers one operator, addition,

    //        multiple variables,

    //        variable doesnt exist in expression

    public void testDifferentiate\_AddVarNotExist() {

        String inputExpr = "x + y + x";

        Expression expr = Expression.parse(inputExpr);

        Expression actual = expr.differentiate("foo");

        String expected = "0";

        assertNotNull("Expected non-null Expression", actual);

        assertEquals("Expected simplified derived expression",

                expected, actual.toString());

    }

    @Test

    // covers multiple operators,

    //        multiple variables,

    //        variable exists in expression

    public void testDifferentiate\_AddMult() {

        String inputExpr = "x\*y + x";

        Expression expr = Expression.parse(inputExpr);

        Expression actual = expr.differentiate("x");

        String expected = "y + 1";

        assertNotNull("Expected non-null Expression", actual);

        assertEquals("Expected simplified derived expression",

                expected, actual.toString());

    }

    @Test

    // covers one operator, multiplication,

    //        multiple variables,

    //        variable exists in expression

    public void testDifferentiate\_Mult() {

        String inputExpr = "x \* (x \* y)";

        Expression expr = Expression.parse(inputExpr);

        Expression actual = expr.differentiate("x");

        Expression expected = Expression.parse("x\*y + x\*y");

        assertNotNull("Expected non-null Expression", actual);

        assertEquals("Expected simplified derived expression",

                expected, actual);

    }

    // Tests for toString()

    @Test

    // covers empty expression

    public void testToString\_Empty() {

        Expression empty = Expression.parse("0.000009");

       assertEquals("Expected empty string to be 0",

               "0", empty.toString());

    }

    @Test

    // covers contains multiple variables and values

    public void testToString() {

        String actual1 = expr.toString();

        String actual2 = expr1.toString();

        String actual3 = Expression.parse("x\*(x\*y)").toString();

        String expected1 = "(x)\*(y) + x + 0.5";

        String expected2 = "((x)\*(y))\*(0.5)";

        String expected3 = "(x)\*((x)\*(y))";

        assertEquals("Expected correct spacing and grouping",

                expected1, actual1);

        assertEquals("Expected correct default grouping",

                expected2, actual2);

        assertEquals("Expected correct grouping",

                expected3, actual3);

    }

    // Tests for equals()

    @Test

    // covers reflexive equality

    public void testEquals\_Reflexive() {

        assertEquals("Expected expression equal to itself",

                empty, empty);

        assertEquals("Expected expression equal to itself",

                one, one);

        assertEquals("Expected expression equal to itself",

                expr, expr);

        assertEquals("Expected expression equal to itself",

                expr1, expr1);

    }

    @Test

    // covers symmetric equality

    public void testEquals\_Symmetric() {

        Expression exprEqual = Expression.parse("x\*y + x + 0.500009");

        Expression expr1Equal = Expression.parse("x\*y\*0.500002");

        if (expr.equals(exprEqual)) {

            assertTrue("Expected symmetric equality", exprEqual.equals(expr));

        } else {

            fail("Expected expressions to be equal");

        }

        if (expr1.equals(expr1Equal)) {

            assertTrue("Expected symmetric equality", expr1Equal.equals(expr1));

        } else {

            fail("Expected expressions to be equal");

        }

    }

    @Test

    // covers transitive equality

    public void testEquals\_Transitive() {

        Expression exprA = expr;

        Expression exprB = Expression.parse("x\*y + x + 0.500009");

        Expression exprC = Expression.parse("x\*y + x + 0.500002");

        Expression expr1A = expr1;

        Expression expr1B = Expression.parse("x\*y\*0.500002");

        Expression expr1C = Expression.parse("x\*y\*0.500009");

        if (exprA.equals(exprC) && exprA.equals(exprB)) {

            assertTrue("Expected transitive equality", exprC.equals(exprB));

            assertTrue("Expected transitive equality", exprB.equals(exprC));

        } else {

            fail("Expected expressions to be equal");

        }

        if (expr1A.equals(expr1C) && expr1A.equals(expr1B)) {

            assertTrue("Expected transitive equality", expr1C.equals(expr1B));

            assertTrue("Expected transitive equality", expr1B.equals(expr1C));

        } else {

            fail("Expected expressions to be equal");

        }

    }

    // Test for hashCode()

    @Test

    public void testHashCode() {

        Expression exprA = expr;

        Expression exprB = Expression.parse("x\*y + x + 0.500009");

        Expression exprC = Expression.parse("x\*y + x + 0.500002");

        Expression expr1A = expr1;

        Expression expr1B = Expression.parse("x\*y\*0.500002");

        Expression expr1C = Expression.parse("x\*y\*0.500009");

        assertEquals("Expected equal objects to have the same hash code",

                exprA.hashCode() == exprB.hashCode(),

                exprB.hashCode() == exprC.hashCode());

        assertEquals("Expected equal objects to have the same hash code",

                expr1A.hashCode() == expr1B.hashCode(),

                expr1B.hashCode() == expr1C.hashCode());

    }

}