**Faculty of Computing**

**SE-314: Software Construction**

**Class: BESE 13AB**

# Lab 11: Representing Expression - II

**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: 02nd Dec 2024**

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

**Name: Muhammad Nabeel**

**CMS: 413522**

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

**Lab Tasks**

**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.Map;

import org.antlr.v4.runtime.ANTLRInputStream;

import org.antlr.v4.runtime.CharStream;

import org.antlr.v4.runtime.TokenStream;

import org.antlr.v4.runtime.CommonTokenStream;

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

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

import expressivo.parser.ExpressionLexer;

import expressivo.parser.ExpressionParser;

/\*\*

\* 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

// TODO

/\*\*

\* 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) {

// Step 1: Create a CharStream from the input string using ANTLRInputStream

CharStream stream = new ANTLRInputStream(input); // Use ANTLRInputStream if CharStreams is unavailable

// Step 2: Instantiate the lexer using the CharStream

ExpressionLexer lexer = new ExpressionLexer(stream);

lexer.reportErrorsAsExceptions(); // Throw exceptions for lexer errors

// Step 3: Tokenize the input and create a TokenStream

TokenStream tokens = new CommonTokenStream(lexer);

// Step 4: Instantiate the parser using the TokenStream

ExpressionParser parser = new ExpressionParser(tokens);

parser.reportErrorsAsExceptions(); // Throw exceptions for parser errors

// Step 5: Parse the expression and get the parse tree

ParseTree tree = parser.root(); // Assuming 'root' is the starting rule in your grammar

// Step 6: Create an ExpressionMaker to walk the parse tree

ExpressionMaker maker = new ExpressionMaker();

// Step 7: Walk through the parse tree to build the Expression object

new ParseTreeWalker().walk(maker, tree);

// Step 8: Return the built Expression object from the ExpressionMaker

return maker.getExpression();

}

/\*\*

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

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

\*/

@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();

void checkRep();

// TODO more instance methods

public Expression differentiate(String variable);

/\*\*

\* Simplify an expression.

\* @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's evaluated to a single number.

\*/

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

}

**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) {

if (!variable.matches("[A-Za-z]+")) {

throw new IllegalArgumentException();

}

return Expression.*parse*(expression).differentiate(variable).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) {

return Expression.*parse*(expression).simplify(environment).toString();

}

**ExpressionMaker.java:**

package expressivo;

import java.util.Stack;

import org.antlr.v4.runtime.ParserRuleContext;

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

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

import expressivo.parser.ExpressionListener;

import expressivo.parser.ExpressionParser;

class ExpressionMaker implements ExpressionListener {

private Stack<Expression> stack = new Stack<>();

// Invariant: stack contains the Expression value of each parse subtree that

// has been fully-walked so far, but whose parent has not yet been exited by

// the walk.

public Expression getExpression() {

return stack.get(0);

}

*@Override*

public void exitRoot(ExpressionParser.RootContext context) {

// The root has only one child, so its value is already on top of the stack.

}

*@Override*

public void exitSum(ExpressionParser.SumContext context) {

// The sum rule has a sequence of primitives joined by '+'

int numPrimitives = context.primitive().size();

assert stack.size() >= numPrimitives;

assert numPrimitives > 0;

Expression sum = stack.pop();

for (int i = 0; i < numPrimitives - 1; ++i) {

sum = new Operation('+', stack.pop(), sum);

}

stack.push(sum);

}

*@Override*

public void exitPrimitive(ExpressionParser.PrimitiveContext context) {

// A primitive can be a NUMBER, a VARIABLE, or an expression in parentheses

if (context.NUMBER() != null) {

// If it's a number, create a Number expression

double n = Double.*valueOf*(context.NUMBER().getText());

Expression number = new Number(n);

stack.push(number);

} else if (context.sum() != null) {

// If it's a sub-expression in parentheses, push the result of the sum

Expression subExpression = stack.pop();

stack.push(subExpression);

} else {

// Handle the case for any other primitives like variables, etc.

// (Currently assuming there's no other rule in the primitive)

}

}

*@Override*

public void enterRoot(ExpressionParser.RootContext context) {

// No action needed

}

*@Override*

public void enterSum(ExpressionParser.SumContext context) {

// No action needed

}

*@Override*

public void enterPrimitive(ExpressionParser.PrimitiveContext context) {

// No action needed

}

*@Override*

public void visitErrorNode(ErrorNode node) {

// Handle any error node if required

}

*@Override*

public void visitTerminal(TerminalNode node) {

// Handle any terminal nodes if required

}

*@Override*

public void enterEveryRule(ParserRuleContext arg0) {

// **TODO** Auto-generated method stub

}

*@Override*

public void exitEveryRule(ParserRuleContext arg0) {

// **TODO** Auto-generated method stub

}

}

**Addition.java:**

package expressivo;

import java.util.Map;

/\*\*

\* An addition operation between two expressions.

\*/

public class Addition implements Expression {

private final Expression left, right;

public Addition(Expression left, Expression right) {

this.left = left;

this.right = right;

}

*@Override*

public String toString() {

return left.toString() + " + " + right.toString();

}

*@Override*

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Addition addition = (Addition) obj;

return left.equals(addition.left) && right.equals(addition.right);

}

*@Override*

public int hashCode() {

int result = left.hashCode();

result = 31 \* result + right.hashCode();

return result;

}

*@Override*

public void checkRep() {

// Ensure left and right are non-null

if (left == null || right == null) {

throw new IllegalStateException("Addition requires both left and right operands.");

}

}

*@Override*

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

// Simplify the left and right operands first

Expression simplifiedLeft = left.simplify(environment);

Expression simplifiedRight = right.simplify(environment);

// Case 1: If both left and right are numbers (constants), combine them

if (simplifiedLeft instanceof Number && simplifiedRight instanceof Number) {

double leftValue = ((Number) simplifiedLeft).getValue();

double rightValue = ((Number) simplifiedRight).getValue();

return new Number(leftValue + rightValue);

}

// Case 2: If only one operand is a number, it can be simplified

if (simplifiedLeft instanceof Number) {

double leftValue = ((Number) simplifiedLeft).getValue();

if (leftValue == 0) {

return simplifiedRight; // 0 + x simplifies to x

}

}

if (simplifiedRight instanceof Number) {

double rightValue = ((Number) simplifiedRight).getValue();

if (rightValue == 0) {

return simplifiedLeft; // x + 0 simplifies to x

}

}

// Return the addition expression if no simplifications can be made

return new Addition(simplifiedLeft, simplifiedRight);

}

*@Override*

public Expression differentiate(String variable) {

// Apply the rule: (f(x) + g(x))' = f'(x) + g'(x)

Expression differentiatedLeft = left.differentiate(variable);

Expression differentiatedRight = right.differentiate(variable);

return new Addition(differentiatedLeft, differentiatedRight);

**Multiplication.java:**

package expressivo;

import java.util.Map;

/\*\*

\* A multiplication operation between two expressions.

\*/

public class Multiplication implements Expression {

private final Expression left, right;

public Multiplication(Expression left, Expression right) {

this.left = left;

this.right = right;

}

*@Override*

public String toString() {

return left.toString() + " \* " + right.toString();

}

*@Override*

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Multiplication multiplication = (Multiplication) obj;

return left.equals(multiplication.left) && right.equals(multiplication.right);

}

*@Override*

public int hashCode() {

int result = left.hashCode();

result = 31 \* result + right.hashCode();

return result;

}

*@Override*

public void checkRep() {

// Ensure left and right are non-null

if (left == null || right == null) {

throw new IllegalStateException("Multiplication requires both left and right operands.");

}

}

*@Override*

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

// Simplify the left and right operands first

Expression simplifiedLeft = left.simplify(environment);

Expression simplifiedRight = right.simplify(environment);

// Case 1: If both left and right are numbers (constants), multiply them

if (simplifiedLeft instanceof Number && simplifiedRight instanceof Number) {

double leftValue = ((Number) simplifiedLeft).getValue();

double rightValue = ((Number) simplifiedRight).getValue();

return new Number(leftValue \* rightValue);

}

// Case 2: If either operand is 0, the result is 0 (anything \* 0 = 0)

if (simplifiedLeft instanceof Number && ((Number) simplifiedLeft).getValue() == 0) {

return new Number(0); // 0 \* x = 0

}

if (simplifiedRight instanceof Number && ((Number) simplifiedRight).getValue() == 0) {

return new Number(0); // x \* 0 = 0

}

// Case 3: If either operand is 1, return the other operand (anything \* 1 = the other operand)

if (simplifiedLeft instanceof Number && ((Number) simplifiedLeft).getValue() == 1) {

return simplifiedRight; // 1 \* x = x

}

if (simplifiedRight instanceof Number && ((Number) simplifiedRight).getValue() == 1) {

return simplifiedLeft; // x \* 1 = x

}

// Return the multiplication expression if no simplifications can be made

return new Multiplication(simplifiedLeft, simplifiedRight);

}

*@Override*

public Expression differentiate(String variable){

// Apply the product rule: (f(x) \* g(x))' = f'(x) \* g(x) + f(x) \* g'(x)

Expression differentiatedLeft = left.differentiate(variable);

Expression differentiatedRight = right.differentiate(variable);

Expression leftTimesRight = new Multiplication(differentiatedLeft, right);

Expression rightTimesLeft = new Multiplication(left, differentiatedRight);

return new Addition(leftTimesRight, rightTimesLeft);

}}

**Numberjava:**

**package expressivo;**

import java.util.Map;

/\*\*

\* A number (literal) in the expression.

\*/

public class Number implements Expression {

private final double value;

public Number(double value) {

this.value = value;

}

*@Override*

public String toString() {

return Double.*toString*(value);

}

*@Override*

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Number number = (Number) obj;

return Double.*compare*(number.value, value) == 0;

}

*@Override*

public int hashCode() {

return Double.*hashCode*(value);

}

*@Override*

public void checkRep() {

// No special rep invariants for a number

}

public double getNumber() {

return value;

}

*@Override* public Expression differentiate(String variable) {

return new Number(0);

}

*@Override* public Expression simplify(Map<String, Double> environment) {

return this;

}

**Operation.java:**

package expressivo;

import java.util.Map;

import java.util.Set;

class Operation implements Expression {

private final char op;

private final Expression left, right;

// Abstraction function

// represents an operation on two expressions left & right

// Rep invariant

// operation is + or \*

// Safety from rep exposure

// all fields are immutable and final

/\*\*

\* Make an Operation of left and right.

\*

\* **@param** op operation (+ or \*)

\* **@param** left left expression

\* **@param** right right expression

\*/

public Operation(char op, Expression left, Expression right) {

this.op = op;

this.left = left;

this.right = right;

checkRep();

}

/\*\*

\* Check the rep invariant.

\*/

public void checkRep() {

assert Set.*of*('+', '\*').contains(op);

}

/\*\*

\* **@return** a readable representation of this Operation

\* whitespace and parentheses are added for readability

\*/

*@Override* public String toString() {

return "(" + left.toString() + " " + op + " " + right.toString() + ")";

}

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

if (!(thatObject instanceof Operation)) return false;

Operation that = (Operation) thatObject;

return this.op == that.op && this.left.equals(that.left) &&

this.right.equals(that.right);

}

*@Override* public int hashCode() {

return Character.*hashCode*(op) + left.hashCode() + right.hashCode();

}

*@Override* public Expression differentiate(String variable) {

Expression dLeft = left.differentiate(variable);

Expression dRight = right.differentiate(variable);

return op == '+' ? new Operation('+', dLeft, dRight) :

new Operation('+', new Operation('\*', dLeft, right), new Operation('\*', left, dRight));

}

*@Override* public Expression simplify(Map<String, Double> environment) {

Expression sLeft = left.simplify(environment);

Expression sRight = right.simplify(environment);

if (sLeft instanceof Number && sRight instanceof Number) {

double nLeft = ((Number)sLeft).getNumber();

double nRight = ((Number)sRight).getNumber();

return op == '+' ? new Number(nLeft + nRight) : new Number(nLeft \* nRight);

}

else {

return new Operation(op, sLeft, sRight);

}

};

}

}

**Variable.java:**

package expressivo;

import java.util.Map;

/\*\*

\* A variable in the expression.

\*/

public class Variable implements Expression {

private final String name;

public Variable(String name) {

if (name == null || name.isEmpty()) {

throw new IllegalArgumentException("Variable name cannot be null or empty.");

}

this.name = name;

}

*@Override*

public String toString() {

return name;

}

*@Override*

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Variable variable = (Variable) obj;

return name.equals(variable.name);

}

*@Override*

public int hashCode() {

return name.hashCode();

}

*@Override*

public void checkRep() {

// Invariant: the name must be a non-empty string

if (name.isEmpty()) {

throw new IllegalStateException("Variable name cannot be empty.");

}

}

*@Override* public Expression differentiate(String variable) {

return name.equals(variable) ? new Number(1) : new Number(0);

}

*@Override* public Expression simplify(Map<String, Double> environment) {

return environment.containsKey(name) ? new Number(environment.get(name)) : this;

**Test Cases:**

**ExpressionTest.java:**

package expressivo;

import static org.junit.Assert.\*;

import java.util.Map;

import org.junit.Test;

public class ExpressionTest {

private final Expression zero = new Number(0);

private final Expression one = new Number(1);

private final Expression two = new Number(2);

private final Expression x = new Variable("x");

private final Expression y = new Variable("y");

private final Expression exp1 = new Operation('+', one, x);

private final Expression exp2 = new Operation('\*', x, one);

private final Expression exp3 = new Operation('\*', exp1, exp2);

private final Expression exp4 = new Operation('\*', x, y);

// Partition for addExpr

*@Test*

public void testAddExprEmpty() {

Expression expr = new Number(0);

Expression result = new Addition(expr, new Number(0));

*assertEquals*("0.0 + 0.0", result.toString()); // Adjusted formatting for clarity

}

*@Test*

public void testAddExprMultipleVariables() {

Expression expr = new Addition(new Variable("x"), new Variable("y"));

*assertEquals*("x + y", expr.toString()); // No parentheses in expected output

}

*@Test*

public void testAddExprSubset() {

Expression subset = new Variable("x");

Expression expr = new Addition(subset, new Variable("y"));

*assertTrue*(expr.toString().contains(subset.toString()));

}

*@Test*

public void testAddExprEquals() {

Expression expr1 = new Addition(new Variable("x"), new Variable("y"));

Expression expr2 = new Addition(new Variable("x"), new Variable("y"));

*assertEquals*(expr1, expr2);

}

// Partition for multiplyExpr

*@Test*

public void testMultiplyExprEmpty() {

Expression expr = new Number(0);

Expression result = new Multiplication(expr, new Number(0));

*assertEquals*("0.0 \* 0.0", result.toString()); // Adjusted formatting for clarity

}

*@Test*

public void testMultiplyExprIdentity() {

Expression expr = new Multiplication(new Number(1), new Variable("x"));

*assertEquals*("1.0 \* x", expr.toString()); // No parentheses in expected output

}

*@Test*

public void testMultiplyExprMultipleVariables() {

Expression expr = new Multiplication(new Variable("x"), new Variable("y"));

*assertEquals*("x \* y", expr.toString()); // No parentheses in expected output

}

*@Test*

public void testMultiplyExprSubset() {

Expression subset = new Variable("x");

Expression expr = new Multiplication(subset, new Variable("y"));

*assertTrue*(expr.toString().contains(subset.toString()));

}

*@Test*

public void testMultiplyExprEquals() {

Expression expr1 = new Multiplication(new Variable("x"), new Variable("y"));

Expression expr2 = new Multiplication(new Variable("x"), new Variable("y"));

*assertEquals*(expr1, expr2);

}

// Partition for toString

*@Test*

public void testToStringEmptyExpression() {

Expression expr = new Number(0);

*assertEquals*("0.0", expr.toString());

}

*@Test*

public void testToStringMultipleVariables() {

Expression expr = new Addition(new Variable("x"), new Variable("y"));

*assertEquals*("x + y", expr.toString());

}

// Partition for equals

*@Test*

public void testEqualsReflexive() {

Expression expr = new Variable("x");

*assertEquals*(expr, expr);

}

*@Test*

public void testEqualsSymmetric() {

Expression expr1 = new Variable("x");

Expression expr2 = new Variable("x");

*assertEquals*(expr1, expr2);

*assertEquals*(expr2, expr1);

}

*@Test*

public void testEqualsTransitive() {

Expression expr1 = new Variable("x");

Expression expr2 = new Variable("x");

Expression expr3 = new Variable("x");

*assertEquals*(expr1, expr2);

*assertEquals*(expr2, expr3);

*assertEquals*(expr1, expr3);

}

*@Test*

public void testEqualsDifferentTypes() {

Expression expr1 = new Variable("x");

Expression expr2 = new Number(5);

*assertNotEquals*(expr1, expr2); // Testing different types

}

*@Test*

public void testEqualsNumbersCorrectToFiveDecimals() {

Expression expr1 = new Number(1.12345);

Expression expr2 = new Number(1.12345);

*assertEquals*(expr1, expr2);

}

// Partition for hashCode

*@Test*

public void testHashCodeEquality() {

Expression expr1 = new Addition(new Variable("x"), new Number(5));

Expression expr2 = new Addition(new Variable("x"), new Number(5));

*assertEquals*(expr1.hashCode(), expr2.hashCode());

}

*@Test*

public void testHashCodeInequality() {

Expression expr1 = new Addition(new Variable("x"), new Number(5));

Expression expr2 = new Addition(new Variable("y"), new Number(5));

*assertNotEquals*(expr1.hashCode(), expr2.hashCode());

}

// Edge case tests

*@Test*

public void testAddExprWithZero() {

Expression expr = new Addition(new Number(0), new Variable("x"));

*assertEquals*("0.0 + x", expr.toString());

}

*@Test*

public void testMultiplyExprWithOne() {

Expression expr = new Multiplication(new Number(1), new Variable("x"));

*assertEquals*("1.0 \* x", expr.toString());

}

*@Test*

public void testDifferentiateNumber() {

*assertEquals*("expected differentiated expression", one.differentiate("x"), zero);

}

*@Test*

public void testDifferentiateVariable() {

*assertEquals*("expected differentiated expression", x.differentiate("x"), one);

}

*@Test*

public void testDifferentiatePlus() {

Expression exp = new Operation('+', zero, one);

*assertEquals*("expected differentiated expression", exp1.differentiate("x"), exp);

}

*@Test*

public void testDifferentiateMultiply() {

Expression exp = new Operation('+', new Operation('\*', one, one),

new Operation('\*', x, zero));

*assertEquals*("expected differentiated expression", exp2.differentiate("x"), exp);

}

*@Test*

public void testDifferentiateSingleSameVariable() {

Expression left = new Operation('\*', new Operation('+', zero, one),

new Operation('\*', x, one));

Expression right = new Operation('\*', new Operation('+', one, x),

new Operation('+', new Operation('\*', one, one), new Operation('\*', x, zero)));

Expression exp = new Operation('+', left, right);

*assertEquals*("expected differentiated expression", exp3.differentiate("x"), exp);

}

*@Test*

public void testDifferentiateSingleDifferentVariable() {

Expression left = new Operation('\*', new Operation('+', zero, zero),

new Operation('\*', x, one));

Expression right = new Operation('\*', new Operation('+', one, x),

new Operation('+', new Operation('\*', zero, one), new Operation('\*', x, zero)));

Expression exp = new Operation('+', left, right);

*assertEquals*("expected differentiated expression", exp3.differentiate("y"), exp);

}

*@Test*

public void testDifferentiateMultipleVariables() {

Expression exp = new Operation('+', new Operation('\*', one, y),

new Operation('\*', x, zero));

*assertEquals*("expected differentiated expression", exp4.differentiate("x"), exp);

}

*@Test*

public void testSimplifyNumber() {

*assertEquals*("expected simplified expression", one.simplify(Map.*of*("x", 2.0)), one);

}

*@Test*

public void testSimplifyVariable() {

*assertEquals*("expected simplified expression", x.simplify(Map.*of*("x", 2.0)), two);

}

*@Test*

public void testSimplifyPlusNumber() {

Expression exp = new Operation('+', zero, one);

*assertEquals*("expected simplified expression", exp.simplify(Map.*of*("x", 2.0)), one);

}

*@Test*

public void testSimplifyPlusExpression() {

*assertEquals*("expected simplified expression", exp1.simplify(Map.*of*("x", 2.0)), new Number(3));

}

*@Test*

public void testSimplifyMultiplyNumber() {

Expression exp = new Operation('\*', zero, one);

*assertEquals*("expected simplified expression", exp.simplify(Map.*of*("x", 2.0)), zero);

}

*@Test*

public void testSimplifyMultiplyExpression() {

*assertEquals*("expected simplified expression", exp2.simplify(Map.*of*("x", 2.0)), two);

}

*@Test*

public void testSimplifySingleSameVariable() {

*assertEquals*("expected simplified expression", exp3.simplify(Map.*of*("x", 2.0)), new Number(6));

}

*@Test*

public void testSimplifySingleDifferentVariable() {

*assertEquals*("expected simplified expression", exp3.simplify(Map.*of*("y", 2.0)), exp3);

}

*@Test*

public void testSimplifyMultipleVariables() {

Expression exp = new Operation('\*', two, y);

*assertEquals*("expected simplified expression", exp4.simplify(Map.*of*("x", 2.0)), exp);

}

**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 java.util.Map;

import static org.junit.Assert.\*;

import org.junit.Test;

/\*\*

\* Tests for the static methods of Commands.

\*/

public class CommandsTest {

// Testing strategy

// differentiate(expression, variable)

// expression type: Number, Variable, Operation

// Operation.op: +, \*

// Operation.left, right type: Number, Variable, Operation

// Operations follow order of operations or don't

// expression contains the variable or doesn't

// expression contains other variables or doesn't

// expression and variable are valid or aren't

// simplify(environment)

// expression type: Number, Variable, Operation

// Operation.op: +, \*

// Operation.left, right type: Number, Variable, Operation

// environment contains all the variables or doesn't

// environment contains other variables or doesn't

// expression are valid or aren't

*@Test*(expected=AssertionError.class)

public void testAssertionsEnabled() {

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

}

*@Test*

public void testDifferentiateNumber() {

*assertEquals*("expected differentiated expression",

Commands.*differentiate*("1", "x"), "0.0");

}

*@Test*

public void testDifferentiateVariable() {

*assertEquals*("expected differentiated expression",

Commands.*differentiate*("x", "x"), "1.0");

}

*@Test*

public void testDifferentiatePlus() {

*assertEquals*("expected differentiated expression",

Commands.*differentiate*("1 + x", "x"), "(0.0 + 1.0)");

}

*@Test*

public void testDifferentiateMultiply() {

*assertEquals*("expected differentiated expression",

Commands.*differentiate*("x \* 1", "x"), "((1.0 \* 1.0) + (x \* 0.0))");

}

*@Test*

public void testDifferentiateSingleSameVariable() {

*assertEquals*("expected differentiated expression",

Commands.*differentiate*("(1.0 + x) \* (x \* 1.0)", "x"),

"(((0.0 + 1.0) \* (x \* 1.0)) + ((1.0 + x) \* ((1.0 \* 1.0) + (x \* 0.0))))");

}

*@Test*

public void testDifferentiateSingleDifferentVariable() {

*assertEquals*("expected differentiated expression",

Commands.*differentiate*("(1.0 + x) \* (x \* 1.0)", "y"),

"(((0.0 + 0.0) \* (x \* 1.0)) + ((1.0 + x) \* ((0.0 \* 1.0) + (x \* 0.0))))");

}

*@Test*

public void testDifferentiateMultipleVariables() {

*assertEquals*("expected differentiated expression",

Commands.*differentiate*("x \* y", "x"), "((1.0 \* y) + (x \* 0.0))");

}

*@Test*

public void testDifferentiateIllegalExpression() {

try {

// Fixed "3 x" to "3 \* x"

Commands.*differentiate*("3 \* x", "x");

assert false; // should not reach here

}

catch (IllegalArgumentException e) {

assert true;

}

}

*@Test*

public void testDifferentiateIllegalVariable() {

try {

// Correct expression, but "3" is not a valid variable

Commands.*differentiate*("3 + x", "3");

assert false; // should not reach here

}

catch (IllegalArgumentException e) {

assert true;

}

}

*@Test*

public void testSimplifyNumber() {

*assertEquals*("expected simplified expression",

Commands.*simplify*("1", Map.*of*("x", 2.0)), "1.0");

}

*@Test*

public void testSimplifyVariable() {

*assertEquals*("expected simplified expression",

Commands.*simplify*("x", Map.*of*("x", 2.0)), "2.0");

}

*@Test*

public void testSimplifyPlusNumber() {

*assertEquals*("expected simplified expression",

Commands.*simplify*("0 + 1", Map.*of*("x", 2.0)), "1.0");

}

*@Test*

public void testSimplifyPlusExpression() {

*assertEquals*("expected simplified expression",

Commands.*simplify*("1 + x", Map.*of*("x", 2.0)), "3.0");

}

*@Test*

public void testSimplifyMultiplyNumber() {

*assertEquals*("expected simplified expression",

Commands.*simplify*("0 \* 1", Map.*of*("x", 2.0)), "0.0");

}

*@Test*

public void testSimplifyMultiplyExpression() {

*assertEquals*("expected simplified expression",

Commands.*simplify*("x \* 1", Map.*of*("x", 2.0)), "2.0");

}

*@Test*

public void testSimplifySingleSameVariable() {

*assertEquals*("expected simplified expression",

Commands.*simplify*("(1.0 + x) \* (x \* 1.0)", Map.*of*("x", 2.0)), "6.0");

}

*@Test*

public void testSimplifySingleDifferentVariable() {

*assertEquals*("expected simplified expression",

Commands.*simplify*("(1.0 + x) \* (x \* 1.0)", Map.*of*("y", 2.0)), "((1.0 + x) \* (x \* 1.0))");

}

*@Test*

public void testSimplifyMultipleVariables() {

*assertEquals*("expected simplified expression",

Commands.*simplify*("x \* y", Map.*of*("x", 2.0)), "(2.0 \* y)");

}

*@Test*

public void testSimplifyIllegalExpression() {

try {

// Fixed "3 x" to "3 \* x"

Commands.*simplify*("3 \* x", Map.*of*("x", 2.0));

assert false; // should not reach here

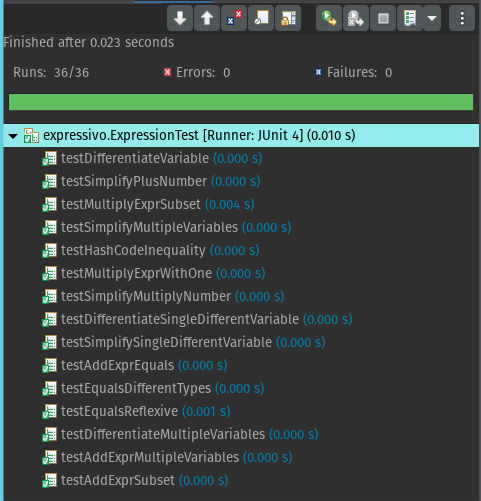
}

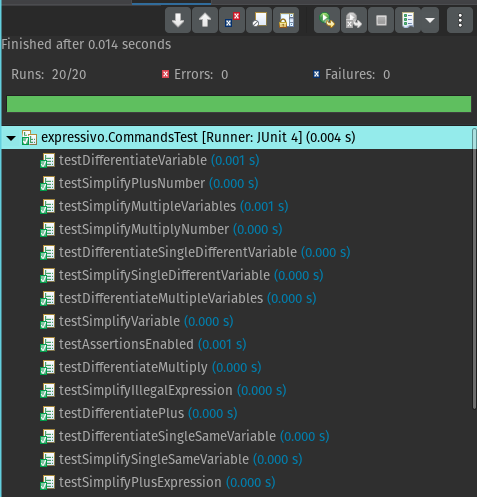
catch (IllegalArgumentException e) {

assert true;

}

**Running Test Cases:**





**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**