**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**

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

# Lab 11: Representing Expressions-II

**Introduction:**

Students will have hands-on experience of representing expressions.

Material:

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

**Lab Tasks**

Solve problem 2, problem 3 & problem 4 listed on the link.

**Problem 2:**we will create the parser that turns a string into an Expression and implement Expression.Parse()

**Problem 2: Parsing Expressions :**

Now we will create the parser that takes a string and produces an Expression value from it. The entry point for your parser should be Expression.parse() , whose spec is provided in the starting code.

Examples of valid inputs:

3 + 2.4

3 \* x + 2.4

3 \* (x + 2.4)

((3 + 4) \* x \* x)

foo + bar+baz

(2\*x )+ ( y\*x )

4 + 3 \* x + 2 \* x \* x + 1 \* x \* x \* (((x)))

Examples of invalid inputs:

3 \*

( 3

3 x

Examples of optional inputs:

2 - 3

(3 \* x) ^ 2

6.02e23

You may consider these inputs invalid, or you may choose to support additional features (new operators or number representations) in the input. However, *your system may not produce an output with a new feature unless that feature appeared in its input*. This way, a client who knows about your extensions can trigger them, but clients who don’t know won’t encounter them unexpectedly.

**2.1 Write a grammar**

Write an ANTLR grammar for polynomial expressions as described in the overview. A starting ANTLR grammar file can be found in src/expressivo/parser/Expression.g4 . This starting grammar recognizes sums of integers, and ignores spaces.

The file Configuration.g4 contains some common boilerplate that is imported into Expression.g4 . You should not edit Configuration.g4 .

See the reading on [parser generators](https://web.mit.edu/6.005/www/sp16/classes/18-parser-generators/)for more information about ANTLR, including links to documentation.

In accordance with the rules provided , the following grammar rules could be incorporated for better parsing of expression:

Expression.g4

|  |
| --- |
| grammar Expression;  import Configuration;  /\*  \* Start rule: describes the entire input, including EOF.  \*/  root  : expr EOF // Start rule  ;  /\*  \* Nonterminal rules (parser rules): handle different operations and precedence.  \*/  expr  : expr '+' term # AddExpr // Handles addition: left-associative  | term # TermExpr // Single term (base case)  ;  term  : term '\*' factor # MulExpr // Handles multiplication: higher precedence  | factor # FactorExpr  ;  factor  : '(' expr ')' # ParenExpr // Parentheses for grouping  | NUMBER # NumberExpr // Numeric values  | VARIABLE # VariableExpr // Variables (case-sensitive)  ;  /\*  \* Terminal rules (lexer rules): define tokens.  \*/  NUMBER  : [0-9]+ ('.' [0-9]+)? // Matches integers and floating-point numbers  ;  VARIABLE  : [a-zA-Z]+ // Matches variable names (letters only)  ;  /\*  \* Ignored tokens: spaces and tabs.  \*/  SPACES  : [ \t\r\n]+ -> skip // Skips whitespace characters  ; |

**After compiling it using antlr4, we get the java version to be associated with our project:**

**A screenshot of a computer

Description automatically generated**

**Files generated:**

**A screenshot of a computer

Description automatically generated**

**2.2 Implement Expression.parse()**

Implement Expression.parse() by following the recipe:

* **Spec**. The spec for this method is given, but you may strengthen it if you want to make it easier to test.
* **Test**. Write tests for Expression.parse() and put them in ExpressionTest.java . Note that we will not run your tests on any implementations other than yours.
* **Code**. Implement Expression.parse() so that it calls the parser generated by your ANTLR grammar. The reading on [parser generators](https://ocw.mit.edu/ans7870/6/6.005/s16/classes/18-parser-generators/)discusses how to call the parser and construct an abstract syntax tree from it, including code examples.

A general note on precision: you are only required to handle nonnegative decimal numbers in the range of the [double](https://docs.oracle.com/javase/8/docs/api/?java/lang/Double.html)type.  
  
***Hint****: use reportErrorsAsExceptions to change how the lexer or parser handles errors.*

CODE:

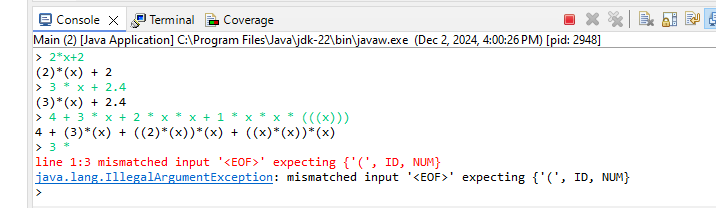
|  |
| --- |
| /\* Copyright (c) 2015-2016 MIT 6.005 course staff, all rights reserved.  \* Redistribution of original or derived work requires permission of course staff.  \*/  package expressivo;  import expressivo.parser.ExpressionLexer;  import expressivo.parser.ExpressionMainVisitor;  import expressivo.parser.ExpressionParser;  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();  } |

**2.3 Run the console interface**

Now that Expression values can be both parsed from strings with parse() , and converted back to strings with toString() , you can try entering expressions into the console interface.

Run Main . In Eclipse, the Console view will allow you to type expressions and see the result. Try some of the expressions from the top of this handout.

**Commit to Git.**Once you’re happy with your solution to this problem, commit to your repo!



TEST CASES:

A screenshot of a computer program

Description automatically generated

**Problems 3-4:**we will add new Expression operations for differentiation and simplification, and implement Commands.differentiate() and Commands.simplify() .

**Problem 3: Differentiation :**

The symbolic differentiation operation takes an expression and a variable, and produces an expression with the derivative of the input with respect to the variable. The result does not need to be simplified.

For example, the following are correct derivatives :

**x\*x\*x**with respect to **x**  
3 \* x \* x

**x\*x\*x**with respect to **x**  
x\*x + (x + x)\*x

**x\*x\*x**with respect to **x**  
( ( x\*x )\*1 )+( ( ( x\*1 )+( 1\*x ) )\*x )+( 0 )

Incorrect derivatives:

**y\*y\*y**with respect to **y**  
3\*y^2 *uses unexpected operator*

**y\*y\*y**with respect to **y**  
0 *d/dx, should be d/dy*

To implement your recursive differentiation operation, use these rules:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

where *c*is a constant or variable other than the variable we are differentiating with respect to (in this case *x*), and *u*and *v*can be anything, including *x*.  
  
*Note : If the output is an expression, your system may output an equivalent expression, including variations in spacing, parentheses, simplification, and number representation.*

*If a number, your system may output an equivalent number, accurate to at least 4 decimal places.*

**3.1. Add an operation to Expression**

You should implement differentiation as a method on your Expression datatype, defined recursively. The signature and specification of the method are up to you to design. Follow the recipe:

* **Spec**. Define your operation in Expression and write a spec.
* **Test**. Put your tests in ExpressionTest.java . Note that we will not run your test cases on other implementations, just on yours.
* **Code**. The implementation must be recursive. It must not use instanceof , nor any equivalent operation you have defined that checks the type of a variant.

**3.2 Implement Commands.differentiate()**

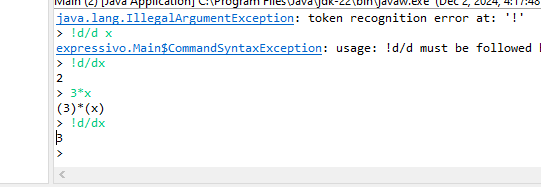
In order to connect your differentiation operation to the user interface, we need to implement the Commands.differentiate() method.

* **Spec**. The spec for this operation is given, but you may strengthen it if you want to make it easier to test.
* **Test**. Write tests for differentiate() and put them in CommandsTest.java . These tests will likely be very similar to the tests you used for your lower-level differentiation operation, but they must use Strings instead of Expression objects. Note that we will not run your tests on any implementations other than yours.
* **Code**. Implement differentiate() . This should be straightforward: simply parsing the expression, calling your differentation operation, and converting it back to a string.

|  |
| --- |
| /\* 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();  }    } |

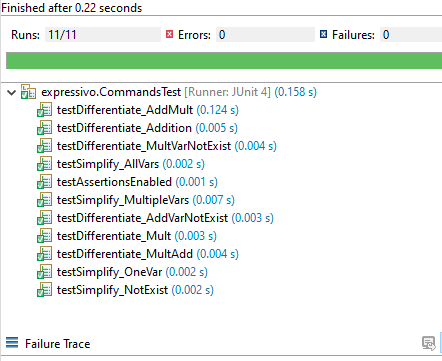
**3.3 Run the console interface**

We’ve now implemented the !d/d command in the console interface. Run Main and try some derivatives in the Console view.



**Commit to Git.**Once you’re happy with your solution to this problem, commit to your repo!

**TEST CASE:**



**- 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.)