**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.

**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.*

**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!

**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.

**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!  
  
**Problem 4: Simplification:**

The simplification operation takes an expression and an environment (a mapping of variables to values). It substitutes the values for those variables into the expression, and attempts to simplfy the substituted polynomial as much as it can.

The set of variables in the environment is allowed to be different than the set of variables actually found in the 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.

The only required simplification is that if the substituted polynomial is a constant expression, with no variables remaining, then simplification must reduce it to a single number, with no operators remaining either. Simplification for substituted polynomials that still contain variables is underdetermined, left to the implementer’s discretion. You may strengthen this spec if you wish to require particular simplifications in other cases.

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.

For example, the following are correct output for simplified expressions:

**x\*x\*x**with environment **x=5 , y=10 , z=20**  
125

**x\*x\*x + y\*y\*y**with environment **y=10**  
x\*x\*x+10\*10\*10

**x\*x\*x + y\*y\*y**with environment **y=10**  
x\*x\*x+1000

**1+2\*3+8\*0.5**with empty environment  
11.000

Incorrect simplified expressions:

**x\*x\*y**with environment **x=1 , y=3**  
1\*1\*3 *not a single number*

**x\*x\*x**with environment **x=2**  
(8) *includes parentheses*

**x\*x\*x**with empty environment  
x^3 *uses unexpected operator*

Optional simplified expressions:

**x\*x\*y + y\*(1+x)**with environment **x=2**  
7\*y

**x\*x\*x + x\*x\*x**with empty environment  
2\*x\*x\*x

**4.1 Add an operation to Expression**

You should implement simplification 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 (perhaps by calling recursive helper methods). It must not use instanceof , nor any equivalent operation you have defined that checks the type of a variant class.

You may find it useful to add more operations to Expression to help you implement the simplify operation. Spec/test/code them using the same recipe, and make them recursive as well where appropriate. Your helper operations should not simply be a variation on using instanceof to test for a variant class.

**4.2 Commands.simplify()**

In order to connect your simplify operation to the user interface, we need to implement the Commands.simplify() 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 simplify() and put them in CommandsTest.java . These tests will likely be very similar to the tests you used for your lower-level simplify operation, but they should use Strings instead of Expression objects. Note that we will not run your tests on any implementations other than yours.
* **Code**. Implement simplify() . This should be straightforward: simply parsing the expression, calling your simplify operation, and converting it back to a string.

**4.3 Run the console interface**

We’ve now implemented the !simplify command in the console interface. Run Main and try using it in the Console view.

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

**- 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.)