**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 SAAD UMER |
| **CLASS** | BESE 13A |
| **CMS** | 408485 |

**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.

|  |
| --- |
| **GRAMMER** |
| grammar Expression;  import Configuration;  // Root rule  root : expr EOF;  // Expressions  expr : term ('+' term)\* ; // Addition/Subtraction  term : factor ('\*' factor)\* ; // Multiplication  factor : NUMBER | VARIABLE | '(' expr ')' ; // Numbers, Variables, Parentheses  // Terminals  NUMBER : [0-9]+ ('.' [0-9]+)? ; // Integer or Decimal  VARIABLE : [a-zA-Z]+ ; // Variable names  // Whitespace  SPACES : [ \t\r\n]+ -> skip ; |

**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 AND SPECS** |
| package expressivo;  import org.antlr.v4.runtime.\*;  import org.antlr.v4.runtime.tree.\*;  import expressivo.parser.ExpressionLexer;  import expressivo.parser.ExpressionParser;  /\*\*  \* An immutable data type representing a polynomial expression.  \*/  public class Expression {  private final String representation;  private Expression(String representation) {  this.representation = representation;  }  @Override  public String toString() {  return representation;  }  @Override  public boolean equals(Object obj) {  if (!(obj instanceof Expression)) return false;  return this.representation.equals(((Expression) obj).representation);  }  @Override  public int hashCode() {  return representation.hashCode();  }  /\*\*  \* Parse an expression string into an Expression object.  \*  \* @param input expression to parse, as defined in the PS3 handout.  \* @return an Expression object representing the input  \* @throws IllegalArgumentException if the expression is invalid  \*/  public static Expression parse(String input) {  try {  // Set up ANTLR lexer and parser  ANTLRInputStream inputStream = new ANTLRInputStream(input);  ExpressionLexer lexer = new ExpressionLexer(inputStream);  lexer.removeErrorListeners();  lexer.addErrorListener(new BaseErrorListener() {  @Override  public void syntaxError(Recognizer<?, ?> recognizer, Object offendingSymbol,  int line, int charPositionInLine, String msg,  RecognitionException e) {  throw new IllegalArgumentException("Invalid expression: " + msg);  }  });  CommonTokenStream tokens = new CommonTokenStream(lexer);  ExpressionParser parser = new ExpressionParser(tokens);  parser.removeErrorListeners();  parser.addErrorListener(new BaseErrorListener() {  @Override  public void syntaxError(Recognizer<?, ?> recognizer, Object offendingSymbol,  int line, int charPositionInLine, String msg,  RecognitionException e) {  throw new IllegalArgumentException("Invalid expression: " + msg);  }  });  // Parse and build the Abstract Syntax Tree (AST)  ParseTree tree = parser.root();  // Convert the AST to an Expression object  return new Expression(tree.getText());  } catch (IllegalArgumentException e) {  throw e;  } catch (Exception e) {  throw new IllegalArgumentException("Failed to parse expression.", e);  }  }  } |

**TEST RUNS**



**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.

|  |
| --- |
| CODE AND SPECS |
| package expressivo;  import org.antlr.v4.runtime.\*;  import org.antlr.v4.runtime.tree.\*;  import expressivo.parser.ExpressionLexer;  import expressivo.parser.ExpressionParser;  /\*\*  \* An immutable data type representing a polynomial expression.  \*/  public class Expression {  private final String representation;  private Expression(String representation) {  this.representation = representation;  }  @Override  public String toString() {  return representation;  }  @Override  public boolean equals(Object obj) {  if (!(obj instanceof Expression)) return false;  return this.representation.equals(((Expression) obj).representation);  }  @Override  public int hashCode() {  return representation.hashCode();  }  /\*\*  \* Parse an expression string into an Expression object.  \*  \* @param input expression to parse.  \* @return an Expression object representing the input  \* @throws IllegalArgumentException if the expression is invalid  \*/  public static Expression parse(String input) {  try {  ANTLRInputStream inputStream = new ANTLRInputStream(input);  ExpressionLexer lexer = new ExpressionLexer(inputStream);  CommonTokenStream tokens = new CommonTokenStream(lexer);  ExpressionParser parser = new ExpressionParser(tokens);  ParseTree tree = parser.root();  return new Expression(tree.getText());  } catch (Exception e) {  throw new IllegalArgumentException("Failed to parse expression.", e);  }  }  /\*\*  \* Compute the derivative of this expression with respect to the given variable.  \*  \* @param variable the variable to differentiate with respect to.  \* @return a new Expression representing the derivative.  \*/  public Expression differentiate(String variable) {  // Recursive differentiation logic  return differentiateRecursive(this.representation, variable);  }  private Expression differentiateRecursive(String expr, String variable) {  // Base cases  if (expr.equals(variable)) {  return new Expression("1");  }  if (expr.matches("\\d+")) { // Constant  return new Expression("0");  }  // Handle operations: "+", "\*"  if (expr.contains("+")) {  String[] terms = expr.split("\\+");  String differentiatedTerms = "";  for (String term : terms) {  Expression differentiatedTerm = differentiateRecursive(term.trim(), variable);  differentiatedTerms += (differentiatedTerms.isEmpty() ? "" : " + ") + differentiatedTerm;  }  return new Expression(differentiatedTerms);  } else if (expr.contains("\*")) {  String[] factors = expr.split("\\\*");  if (factors.length == 2) {  String u = factors[0].trim();  String v = factors[1].trim();  Expression du = differentiateRecursive(u, variable);  Expression dv = differentiateRecursive(v, variable);  return new Expression(  String.format("(%s \* %s) + (%s \* %s)", du, v, u, dv)  );  }  }  // If none of the above cases match, return 0 as a fallback  return new Expression("0");  }  } |

|  |
| --- |
| **TEST CASES** |
| package expressivo;  import static org.junit.Assert.\*;  import org.junit.Test;  public class ExpressionTest {  @Test  public void testDifferentiateSimpleExpressions() {  Expression expr = Expression.parse("x");  assertEquals(Expression.parse("1"), expr.differentiate("x"));  Expression constantExpr = Expression.parse("5");  assertEquals(Expression.parse("0"), constantExpr.differentiate("x"));  }  @Test  public void testDifferentiateComplexExpressions() {  Expression expr = Expression.parse("x \* x");  assertEquals(Expression.parse("(1 \* x) + (x \* 1)"), expr.differentiate("x"));  Expression sumExpr = Expression.parse("x + x \* x");  assertEquals(Expression.parse("1 + (1 \* x) + (x \* 1)"), sumExpr.differentiate("x"));  }  @Test  public void testDifferentiateInvalidVariable() {  Expression expr = Expression.parse("x \* x");  assertEquals(Expression.parse("0"), expr.differentiate("y"));  }  } |

**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.

|  |
| --- |
| **CODE AND SPECS** |
| package expressivo;  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) {  try {  // Parse the input expression  Expression expr = Expression.*parse*(expression);    // Differentiate the parsed expression  Expression derivative = expr.differentiate(variable);    // Convert the derivative back to a string  return derivative.toString();  } catch (IllegalArgumentException e) {  throw new IllegalArgumentException("Invalid expression or variable: " + e.getMessage(), e);  }  }  } |
| **TEST CASES** |
| package expressivo;  import static org.junit.Assert.\*;  import org.junit.Test;  public class CommandsTest {  @Test  public void testDifferentiateSimpleExpressions() {  assertEquals("1", Commands.differentiate("x", "x"));  assertEquals("0", Commands.differentiate("5", "x"));  }  @Test  public void testDifferentiateMultiplication() {  assertEquals("(1 \* x) + (x \* 1)", Commands.differentiate("x \* x", "x"));  assertEquals("(1 \* y) + (x \* 0)", Commands.differentiate("x \* y", "x"));  }  @Test  public void testDifferentiateAddition() {  assertEquals("1 + 1", Commands.differentiate("x + x", "x"));  assertEquals("0 + 1", Commands.differentiate("5 + x", "x"));  }  @Test  public void testDifferentiateComplexExpressions() {  assertEquals("(1 \* x) + (x \* 1) + 1", Commands.differentiate("x \* x + x", "x"));  assertEquals("1 + (1 \* x) + (x \* 1)", Commands.differentiate("x + x \* x", "x"));  }  @Test  public void testDifferentiateInvalidVariable() {  assertEquals("0", Commands.differentiate("x \* x", "y"));  }  @Test(expected = IllegalArgumentException.class)  public void testDifferentiateInvalidExpression() {  Commands.differentiate("x \*", "x");  }  } |

**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.

|  |
| --- |
| **CODE AND SPECS** |
| Expression.simplify()  package expressivo;  import org.antlr.v4.runtime.\*;  import org.antlr.v4.runtime.tree.\*;  import expressivo.parser.ExpressionLexer;  import expressivo.parser.ExpressionParser;  /\*\*  \* An immutable data type representing a polynomial expression.  \*/  public class Expression {  private final String representation;  private Expression(String representation) {  this.representation = representation;  }  *@Override*  public String toString() {  return representation;  }  *@Override*  public boolean equals(Object obj) {  if (!(obj instanceof Expression)) return false;  return this.representation.equals(((Expression) obj).representation);  }  *@Override*  public int hashCode() {  return representation.hashCode();  }  /\*\*  \* Parse an expression string into an Expression object.  \*  \* **@param** input expression to parse.  \* **@return** an Expression object representing the input  \* **@throws** IllegalArgumentException if the expression is invalid  \*/  public static Expression parse(String input) {  try {  ANTLRInputStream inputStream = new ANTLRInputStream(input);  ExpressionLexer lexer = new ExpressionLexer(inputStream);  CommonTokenStream tokens = new CommonTokenStream(lexer);  ExpressionParser parser = new ExpressionParser(tokens);  ParseTree tree = parser.root();  return new Expression(tree.getText());  } catch (Exception e) {  throw new IllegalArgumentException("Failed to parse expression.", e);  }  }  /\*\*  \* Compute the derivative of this expression with respect to the given variable.  \*  \* **@param** variable the variable to differentiate with respect to.  \* **@return** a new Expression representing the derivative.  \*/  public Expression differentiate(String variable) {  // Recursive differentiation logic  return differentiateRecursive(this.representation, variable);  }  private Expression differentiateRecursive(String expr, String variable) {  // Base cases  if (expr.equals(variable)) {  return new Expression("1");  }  if (expr.matches("\\d+")) { // Constant  return new Expression("0");  }  // Handle operations: "+", "\*"  if (expr.contains("+")) {  String[] terms = expr.split("\\+");  String differentiatedTerms = "";  for (String term : terms) {  Expression differentiatedTerm = differentiateRecursive(term.trim(), variable);  differentiatedTerms += (differentiatedTerms.isEmpty() ? "" : " + ") + differentiatedTerm;  }  return new Expression(differentiatedTerms);  } else if (expr.contains("\*")) {  String[] factors = expr.split("\\\*");  if (factors.length == 2) {  String u = factors[0].trim();  String v = factors[1].trim();  Expression du = differentiateRecursive(u, variable);  Expression dv = differentiateRecursive(v, variable);  return new Expression(  String.*format*("(%s \* %s) + (%s \* %s)", du, v, u, dv)  );  }  }  // If none of the above cases match, return 0 as a fallback  return new Expression("0");  }  public Expression simplify(Map<String, Double> environment) {  return simplifyRecursive(this.representation, environment);  }  private Expression simplifyRecursive(String expr, Map<String, Double> environment) {  // Base case: a variable  if (expr.matches("[a-zA-Z]+")) {  if (environment.containsKey(expr)) {  return new Expression(Double.*toString*(environment.get(expr)));  } else {  return new Expression(expr);  }  }  // Base case: a constant number  if (expr.matches("\\d+(\\.\\d+)?")) {  return new Expression(expr);  }  // Handle operations: "+", "\*"  if (expr.contains("+")) {  String[] terms = expr.split("\\+");  double sum = 0.0;  String simplifiedTerms = "";  boolean isConstant = true;  for (String term : terms) {  Expression simplifiedTerm = simplifyRecursive(term.trim(), environment);  if (simplifiedTerm.representation.matches("\\d+(\\.\\d+)?")) {  sum += Double.*parseDouble*(simplifiedTerm.representation);  } else {  isConstant = false;  simplifiedTerms += (simplifiedTerms.isEmpty() ? "" : " + ") + simplifiedTerm;  }  }  if (isConstant) {  return new Expression(Double.*toString*(sum));  } else {  return new Expression((sum > 0 ? sum + " + " : "") + simplifiedTerms);  }  } else if (expr.contains("\*")) {  String[] factors = expr.split("\\\*");  double product = 1.0;  String simplifiedFactors = "";  boolean isConstant = true;  for (String factor : factors) {  Expression simplifiedFactor = simplifyRecursive(factor.trim(), environment);  if (simplifiedFactor.representation.matches("\\d+(\\.\\d+)?")) {  product \*= Double.*parseDouble*(simplifiedFactor.representation);  } else {  isConstant = false;  simplifiedFactors += (simplifiedFactors.isEmpty() ? "" : " \* ") + simplifiedFactor;  }  }  if (isConstant) {  return new Expression(Double.*toString*(product));  } else {  return new Expression((product != 1.0 ? product + " \* " : "") + simplifiedFactors);  }  }  return new Expression(expr);  }  } |
| **TEST CASES** |
| ExpressionTest.java  package expressivo;  import static org.junit.Assert.\*;  import org.junit.Test;  import java.util.HashMap;  import java.util.Map;  public class ExpressionTest {  @Test  public void testSimplifySimpleExpressions() {  Expression expr = Expression.parse("x");  Map<String, Double> env = new HashMap<>();  env.put("x", 5.0);  assertEquals(Expression.parse("5.0"), expr.simplify(env));  }  @Test  public void testSimplifyConstants() {  Expression expr = Expression.parse("5");  Map<String, Double> env = new HashMap<>();  assertEquals(Expression.parse("5"), expr.simplify(env));  }  @Test  public void testSimplifyComplexExpressions() {  Expression expr = Expression.parse("x + y");  Map<String, Double> env = new HashMap<>();  env.put("x", 2.0);  assertEquals(Expression.parse("2.0 + y"), expr.simplify(env));  }  } |

**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.

|  |
| --- |
| CODE AND SPECS |
| Commands.simplify()  package expressivo;  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) {  try {  // Parse the input expression  Expression expr = Expression.parse(expression);    // Differentiate the parsed expression  Expression derivative = expr.differentiate(variable);    // Convert the derivative back to a string  return derivative.toString();  } catch (IllegalArgumentException e) {  throw new IllegalArgumentException("Invalid expression or variable: " + e.getMessage(), e);  }  }  public static String simplify(String expression, Map<String, Double> environment) {  try {  Expression expr = Expression.parse(expression);  Expression simplifiedExpr = expr.simplify(environment);  return simplifiedExpr.toString();  } catch (IllegalArgumentException e) {  throw new IllegalArgumentException("Invalid expression: " + e.getMessage(), e);  }  }  }  } |
| TEST CASES |
| CommandTest.java  package expressivo;  import static org.junit.Assert.\*;  import org.junit.Test;  import java.util.HashMap;  import java.util.Map;  public class ExpressionTest {  @Test  public void testSimplifySimpleExpressions() {  Expression expr = Expression.parse("x");  Map<String, Double> env = new HashMap<>();  env.put("x", 5.0);  assertEquals(Expression.parse("5.0"), expr.simplify(env));  }  @Test  public void testSimplifyConstants() {  Expression expr = Expression.parse("5");  Map<String, Double> env = new HashMap<>();  assertEquals(Expression.parse("5"), expr.simplify(env));  }  @Test  public void testSimplifyComplexExpressions() {  Expression expr = Expression.parse("x + y");  Map<String, Double> env = new HashMap<>();  env.put("x", 2.0);  assertEquals(Expression.parse("2.0 + y"), expr.simplify(env));  }  } |

## 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.)