Equation Simplification Assignment

Details:

Language used: Java (JDK 1.8)

Developed Using: Intellij IDE

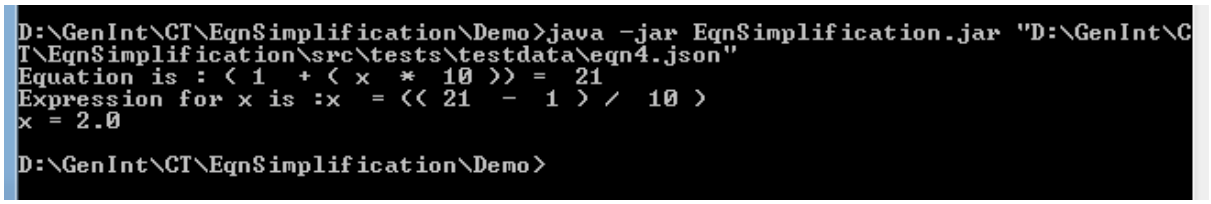
Json Parser: Used https://code.google.com/archive/p/json-simple/

Unit Testing: Used JUnit4 for writing integration tests

Demo:

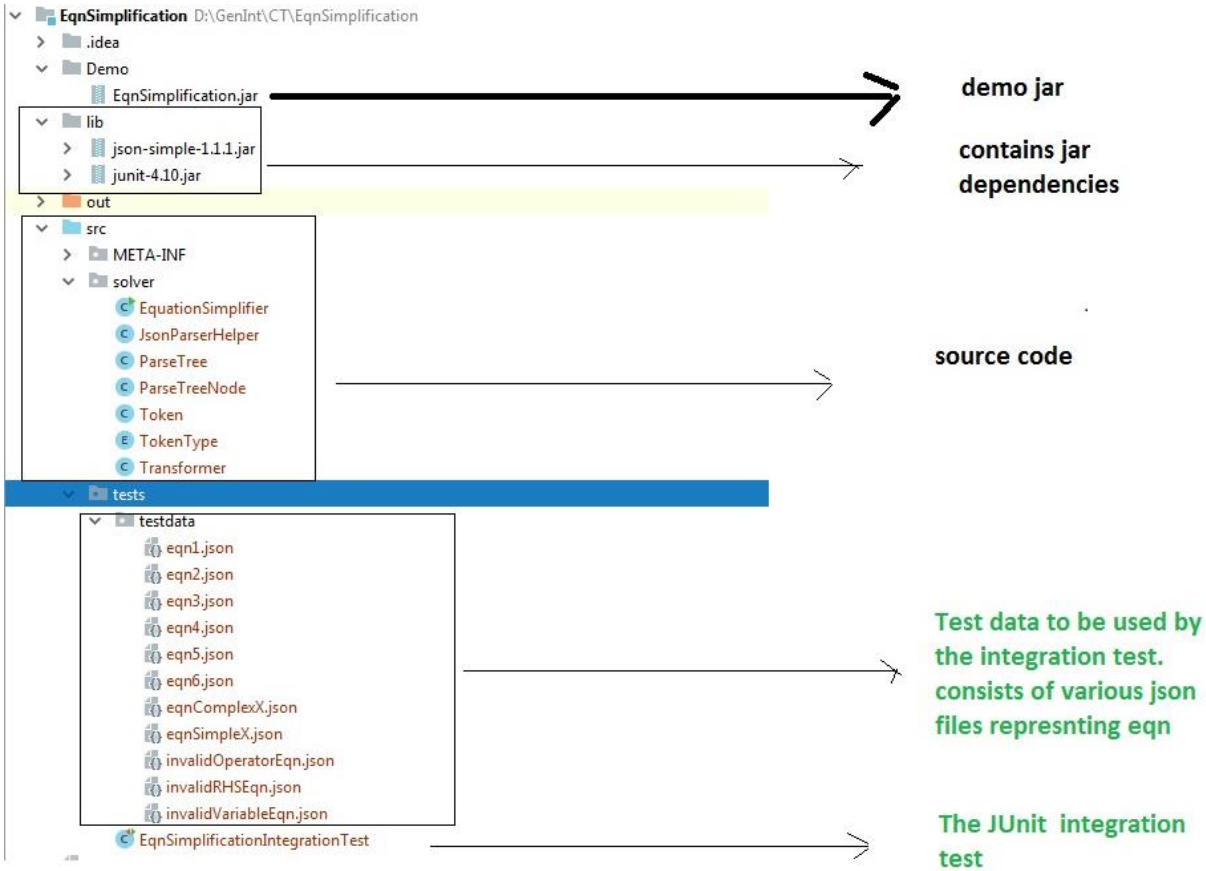
A jar EqnSimplification.jar has been provided in Demo folder. You need to invoke the jar as below from the command line assuming you have java installed (JDK 1.8) giving the path of the json file.

This will automatically parse the equation from the json file. Pretty print the equation.Find the expression for x. Then it will also find the value for the variable x



If the json file does not adhere to the rules of the assignment i.e if the eqn is not properly formed, the application will give appropriate error.

Code Structure:



High Level Design:

Main Classes

Class	Responsibility
EquationSimplifier	This is the entry point of the application. * This class exposes high level API for reading the equation from json file, * Reading the equation is a parse tree, * Getting printable format for the equation, * Transforming the parse tree to have X on one side, * Finding the value of x
JsonParserHelper	This class encapsulates the parsing mechanism of the json file representing the equation * Right now it use the library json simple parser(defined in json-simple-1.1.1.jar). In future it may use * a different json parser but the client code will not change
ParseTree	/** * This class is used for the internal representation of the equation, * The underlying object model is a binary tree whose root will be an * and the left and right subtree will be operation or constants or va * For eg:1 + (x * 10) = 21 will be represented as * * = * / \ * + 21 * / \ * 1 * * / \ * x 10 */
ParseTreeNode	* This class is a binary tree data structure * used for representing expression of the form * exp1 op exp2 * where op can be add , subtract , multiply , divide , and equal * and exp1 and exp2 can be * another operation, * Or a fixed number, * Or a variable reference
Token	* This class is used for representing a token of the equation * For example for the equation: * 1 + (x * 10) = 21 * the tokens are 1, +, x, *, 10, = , 21 * For 1, the token type is CONSTANT and its rep is "1" * For +, the token type is OPERATOR_ADD and its rep is "+" * For x, the token type is VARIABLE and its rep is "x" * For *, the token type is OPERATOR_MULTIPLY and its rep is "*" * For 10, the token type is CONSTANT and its rep is "10" * For =, the token type is OPERATOR_EQUAL and its rep is "11" * For 21, the token type is CONSTANT and its rep is "21"

TokenType	* This class is used for classifying the tokens of the equation * The types are * OPERATOR_ADD - representing add operation * OPERATOR_SUB - representing subtract operation * OPERATOR_MULTIPLY - representing multiply operation * OPERATOR_DIV - representing divide operation * OPERATOR_EQUAL - representing equal operation * VARIABLE - representing single variable reference (x) that occurs somewhere in the LHS * CONSTANT - representing RHS of the equation which will always be a fixed number * UNKNOWN - unknown token
Transformer	/** * This class is responsible for Transforming the equation represented by a Pars * we have 'x' on one side , and all the operations on the * other side. For example for the parse tree: * <p> * For eg for equation : 1 + (x * 10) = 21 * <p> * the transformed expression will be: x = (21 - 1) / 10 */

Class Diagram:

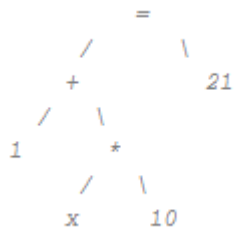
The class diagram and the various interactions are in ClassDiagram.pdf file

Algorithm:

Representing the Equation:

We use ParseTree for representing equation. The ParseTree is a binary tree data structure.

For example for the expression $1 + (x * 10) = 21$, the data structure will look like:



The parse tree building takes O(n) time where n is the number of tokens in the equation.

Algorithm for pretty printing the equation:

This has been done in the class ParseTree. The method is getPrettyPrintEquationFormat()

This can be achieved by in order traversal of the tree(O(n) time). I have printed the braces when a node in the binary tree contains more than 1 node.

Algorithm for transforming the expression so that x is one side:

This has been done in the class Transformer. The method is solveX(ParseTree orig)

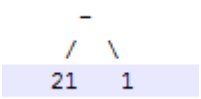
For explaining this algorithm, let me take the example of the parse tree shown above.

Step 1: Locate the operator = . Get the left side of the operator =.

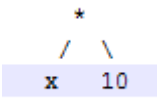
Step 2: The left subtree has operator +. Now locate the constant part of expression in this subtree and the subtree containing variable x.

Step 3: Now move the constant part to RHS taking care of the operator. i.e when we move the constant part of the subtree to RHS, the operator should be made complementary.

Like RHS is now

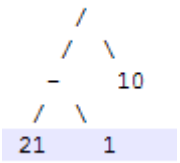


Step 4: Now the LHS is

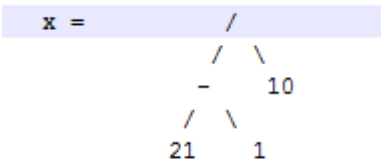


Step 5: Again descend down the LHS part of the subtree following Step 2. So Go to Step 2.

After this the Parse Tree will become like:



Go on descending by bifurcating constant and variable part of sub tree and moving the constant sub tree to RHS. Stop when we locate the variable on LHS. Now we create the new transformed tree with X on LHS and the constant expression on RHS as



The Time complexity depends on the height of the parse tree which is logn

Evaluating the expression:

This has been done in the class ParseTree. The method is evaluateValX()

When we have the Parse tree corresponding to x, i.e LHS contains only x and RHS contains a contant expression, we do an inorder traversal of RHS part of the subtree. When we get a

contant expression in a node, we return that number. When the node represents an operator, we get the binary function represented by that operation

(See TokenType:

```
/**
 *
 * @return the mathematical operation representing the token type.
 * For example for OPERATOR_ADD, this should represent addition
 */
public abstract BiFunction<Double, Double, Double> operation();
```

)

And apply this operation recursively on left and right node

```
/**
 *
 * @return the evaluated value of variable "x"
 */
double evaluateValX() {
    ParseTreeNode right = root.getRight();
    return evaluateValX(right);
}

private double evaluateValX(ParseTreeNode node) {
    if (node == null) {
        return 0;
    }
    Token token = node.getToken();
    if (token.isConstant()) {
        return token.getValue();
    }

    return token.getType().operation().apply(evaluateValX(node.left), evaluateValX(node.right));
}
```

The time complexity is O(n).

Unit Testing:

Wrote integration tests using JUnit4by making various json files. The tests are defined in class EqnSimplificationIntegrationTest.

```
This class does integration testing of the assignment application
EquationSimplifier.
* For testing JUnit 4 is used.
* All the tests parses the json file, builds the equation represented by the json
file,
* tests the pretty print format of the equation
* tests the expression for x
* tests the value of x
* There are positive as well as negative test cases
```

Samples:

```
@Test
public void testEqn1() {
    EquationSimplifier equationSimplifier = new EquationSimplifier( jsonPath: ".\\src\\tests\\testdata\\eqn1.json");

    try {
        Assert.assertEquals( expected: "( 3  + (( x  -  20 ) *  3 )) =  21", equationSimplifier.getEquationInPrettyPrintFormat());
        Assert.assertEquals( expected: "x  = ((( 21  -  3 ) /  3 ) +  20 )", equationSimplifier.getExpressionForX());
        Assert.assertEquals( expected: 26.0, equationSimplifier.solveForX());
    } catch (IOException | ParseException e) {
        e.printStackTrace();
        Assert.fail();
    }
}

@Test
public void testEqn2() {
    EquationSimplifier equationSimplifier = new EquationSimplifier( jsonPath: ".\\src\\tests\\testdata\\eqn2.json");

    try {
        Assert.assertEquals( expected: "( 2  - ( x  *  3 )) =  5", equationSimplifier.getEquationInPrettyPrintFormat());
        Assert.assertEquals( expected: "x  = ((( 5  -  2 ) /  -1 ) /  3 )", equationSimplifier.getExpressionForX());
        Assert.assertEquals( expected: -1.0, equationSimplifier.solveForX());
    } catch (IOException | ParseException e) {
        e.printStackTrace();
        Assert.fail();
    }

}

@Test
public void testEqn3() {
    EquationSimplifier equationSimplifier = new EquationSimplifier( jsonPath: ".\\src\\tests\\testdata\\eqn3.json");

    try {
        Assert.assertEquals( expected: "( -3  *  x ) =  15", equationSimplifier.getEquationInPrettyPrintFormat());
        Assert.assertEquals( expected: "x  = ( 15  /  -3 )", equationSimplifier.getExpressionForX());
        Assert.assertEquals( expected: -5.0, equationSimplifier.solveForX());
    } catch (IOException | ParseException e) {
        e.printStackTrace();
        Assert.fail();
    }

}
```

Goals Achieved

1. Parsed the json file into a binary tree data structure and printed the equation in a pretty print format
2. Transformed the expression so that the variable ‘x’ is one side
3. Evaluated the expression to find the value of ‘’x
4. Wrote Unit Tests to ensure that the algorithm is not broken during refactoring
5. Came up with loosely coupled design so that independent modules can be changed without affecting others