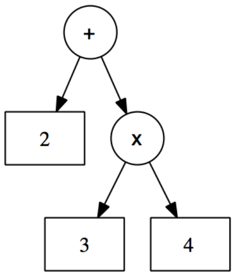
CSCI-605 Advanced Object-Oriented Programming Concepts

Homework 3



1 Introduction

An interpreter is a program that executes instructions written in a programming language. For example, the Java Virtual Machine (JVM) is an interpreter that executes compiled Java bytecode. This assignment involves writing an interpreter for evaluating simple arithmetic expressions.

The interpreter will receive a mathematical expression as a string from the standard input. This expression will be in prefix form; the mathematical operator is written at the beginning of the expression rather than at the middle (e.g., the expression 2 + 3 * 4 is written + 2 * 3 4). Only the operators +, -, *, /, and % are allowed. The only supported operand is integer literals (e.g., 8). Once it receives the expression, the interpreter will:

- convert the expression into a parse binary tree
- evaluate the expression
- display the infix form of the expression (referred as emitting)

1.1 Goals

This homework helps students to gain experience working with:

- Interfaces and Classes
- Method overriding
- Polymorphism
- Unit testing

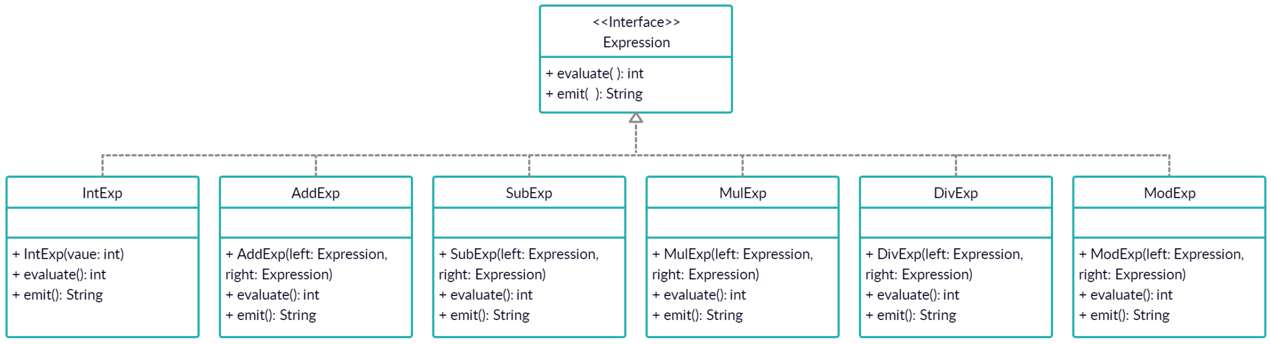
1.2 Provided Files

1. The `test` folder includes examples for unit testing. You must follow these examples to create your own unit testing classes.

2 Implementation

2.1 Design

The following design only shows the expression classes, the class in charge of reading the input expression and parsing it into a tree is not included. Notice also that the `public` methods from the expression classes are shown, but not their private state. You will need to design that on your own.



2.2 `Interp` Implementation

The main program is the class `Interp`. It is responsible for reading a prefix expression from the standard input, generating the corresponding parse tree, and then emitting and evaluating the tree.

You are expected to implement this class from scratch. **To receive full credit you should follow all these suggestions**:

- Your main program should be fully object oriented (no static methods except for the main). This means the main method should only contain the required code to get the interpreter running.
- This class must contain a method for reading in prefix expressions from the user until they wish to quit. It is responsible for building a list of tokens for the prefix expression. It should then pass control to a helper function that builds the parse tree. After that, it uses the root of the tree to emit and evaluate the expression.
- The helper function takes the current list of tokens as an argument and returns the appropriate `Expression` node for the token at the front of the list. This is a recursive function that gets the token at the front of the list, and based on the string, it builds and returns the appropriate concrete Expression node by recursively calling to complete the work. This recursive process continues until it reaches an operand (the base case).

For purposes of this lab you may assume all input is correct and represents a valid parse tree.

2.3 Standalone Output

When executed as a standalone program, your solution should produce output as follows:

```
Welcome to your Arithmetic Interpreter v1.0 :)
> + 4 1
Emit: (4 + 1)
Evaluate: 5
> * / 8 4 + 3 2
Emit: ((8 / 4) * (3 + 2))
Evaluate: 10
> * + - 15 3 / 20 4 % 36 11
Emit: (((15 - 3) + (20 / 4)) * (36 % 11))
Evaluate: 51
> quit
Goodbye!
```

First, a welcome message is displayed. The user is then prompted with ">" and the program waits for the user to enter the expression. Once entered, the program builds the parse tree. It then emits the inorder string of the expression and displays the evaluation of it. This process is repeated until the user enters "quit" - this causes the program to display a closing message and then the program gracefully terminates.

Please notice that all the components in an expression are separated by white spaces (e.g., + 4 1).

3 Testing

Up-front software design is one way to avoid making costly mistakes and coding by trial-and-error. Another technique for avoiding costly errors is *test driven tevelopment*, or *TDD*. There is not enough time or space here to go into detail regarding TDD, but you can think of it as writing your classes one-at-a-time, and making sure that each class is fully tested before moving to the next class.

So how does a developer make sure that a class is fully tested? By writing a unit test. A unit test is a separate class that contains a suite of tests to make sure that one specific class is working the way that it is intended to work. Each test is encapsulated as a separate method and follows this pattern:

1. Setup - The class and any of its dependencies are instantiated.
2. Invoke - The method(s) of the class being tested are called.
3. Analyze - Verify that the method had the expected result.

Each test method should only test one small thing, e.g. one of the constructors of the class to ensure that it sets the values of all of the class's fields properly.

To help you to understand how unit testing works, and to write tests of your own, a few test unit files have been provided to you in the `test` folder.

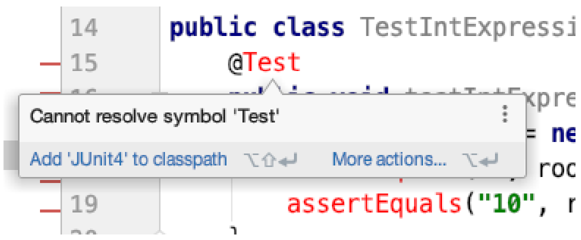
- `tests.TestIntExpression` - A unit test for the `hw3.IntExp` class.
- `tests.TestAddExpression` - A unit test for the `hw3.AddExp` class.

You should use these as a guide for writing your own unit tests. Note that each test method follows the setup/invoke/analyze pattern mentioned previously.

In order to receive the full credit for testing, you will need to write a unit test in the `hw3.test` folder for each of the concrete `Expression` classes that you create in the `hw3` folder. You do not need to create new unit test classes for `IntExp` and `AddExp`, you can use the ones we have provided. Remember, a *good* test method only tests one small thing and follows the setup/invoke/analyze pattern that you see in the provided example.

3.1 JUnit

For writing the unit tests, you will use a library called JUnit. This library is contained within IntelliJ, but you may need to configure it first. Import `TestIntExpression` and `TestAddExpression` classes into your project in a `test` package. Open the `TestIntExpression` class, if the source code have compiled errors due to JUnit, you should hover over the Test annotation that precedes the `testIntExpression` method. A popup should come up with blue text to **Add Junit 4 to classpath**.

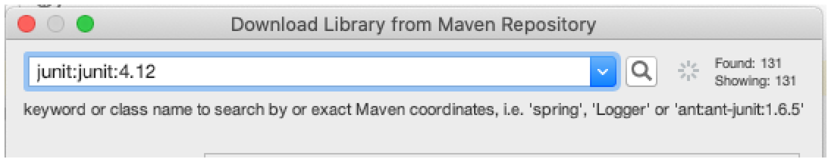


```
public class TestIntExpressi
    @Test
        assertEquals("10", r
```

Cannot resolve symbol 'Test'

Add 'JUnit4' to classpath ⌥⇧↩   More actions... ⌥↩

Select that and a dialog should come up to download it - click the **OK** button.



Download Library from Maven Repository

junit:junit:4.12

Found: 131
Showing: 131

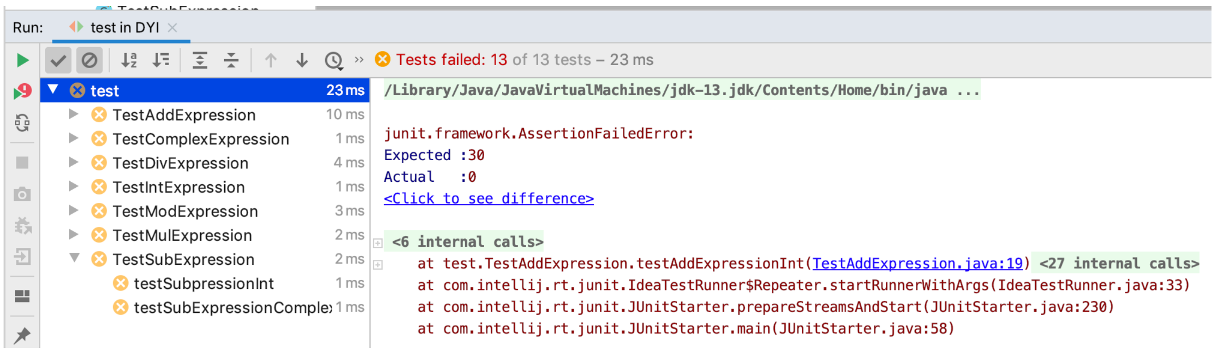
keyword or class name to search by or exact Maven coordinates, i.e. 'spring', 'Logger' or 'ant:ant-junit:1.6.5'

At this point there should be no errors related to JUnit. You may have still have errors if you haven't implemented the `IntExp` class yet.

3.1.1 Run Tests

You can run all the tests at once by right clicking on the `test` package, or you can run each test case individually by right clicking on it.

When a test fails it shows you what it Expected, and the Actual which is what your program produced. These two things must match *exactly* in order to pass the test. If strings are being compared they must match character by character exactly.



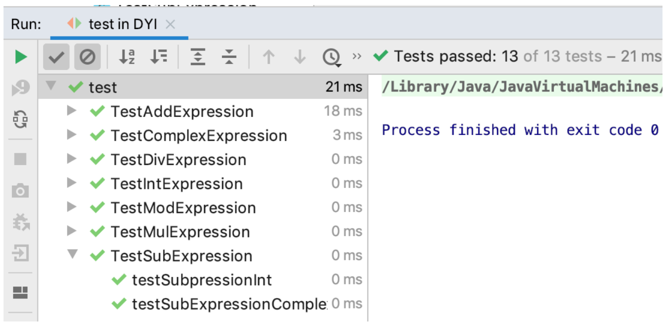
```
test                          23 ms
  TestAddExpression           10 ms
  TestComplexExpression        1 ms
  TestDivExpression            4 ms
  TestIntExpression            1 ms
  TestModExpression            3 ms
  TestMulExpression            2 ms
  TestSubExpression            2 ms
    testSubpressionInt         1 ms
    testSubExpressionComple    1 ms
```

```
/Library/Java/JavaVirtualMachines/jdk-13.jdk/Contents/Home/bin/java ...

junit.framework.AssertionFailedError:
Expected :30
Actual   :0
<Click to see difference>

<6 internal calls>
    at test.TestAddExpression.testAddExpressionInt(TestAddExpression.java:19) <27 internal calls>
    at com.intellij.rt.junit.IdeaTestRunner$Repeater.startRunnerWithArgs(IdeaTestRunner.java:33)
    at com.intellij.rt.junit.JUnitStarter.prepareStreamsAndStart(JUnitStarter.java:230)
    at com.intellij.rt.junit.JUnitStarter.main(JUnitStarter.java:58)
```

Your goal is to implement a test unit for every concrete `Expression` class and make sure that all tests pass.



```
test                          21 ms
  TestAddExpression           18 ms
  TestComplexExpression        3 ms
  TestDivExpression            0 ms
  TestIntExpression            0 ms
  TestModExpression            0 ms
  TestMulExpression            0 ms
  TestSubExpression            0 ms
    testSubpressionInt         0 ms
    testSubExpressionComple    0 ms
```

```
/Library/Java/JavaVirtualMachines/

Process finished with exit code 0
```

4 Submission

You will need to submit all of your code, including your tests, to the MyCourses assignment before the due date. You must submit your hw3 and `hw3.test` folders as a ZIP archive named “`hw3.zip`” (if you submit another format, such as 7-Zip, WinRAR, or Tar, you will not receive credit for this homework).

5 Grading

The grade breakdown for this homework is as follows: The grade breakdown for this homework is as follows:

- Design: 15%
- Functionality: 60%
- Testing: 15%
- Style: 10%