Computer Science Courses

Project 2- Part 1: Evaluating Expressions

- Project starts on: October 9, 2015
- Project early submission: **October 18, 2015**
- Projected Score date: **October 19, 2015**
- Project due date: **October 23, 2015**

> Please read the description very carefully. You will stand to lose significant amount of points if you do not follow instructions perfectly in your projects.

Purpose: Learn how to evaluate an expression tree.

Input: Expressions using addition, subtraction, multiplication and division, each terminated by a semicolon.

Output: Evaluation of those expressions for numeric constants.

Project skeleton

Your project skeleton can be found on piazza. Download the files and start working

The following files are related to Part 1:

- `Project2.java`: contains the empty methods that you need to implement
- `Tree.java`: binary tree
- `Parser.java`: a recursive descent parser to build a tree using an expression
- `Token.java`: the Token class from Project 0.
- `Project2Part1.java`: contains the main function of Part 1 (look for comments that say Part 1)

> Do not change the original function names and input parameters in project skeleton. You are not allowed to refactor the project skeleton. In order to get full credit, you must use the same function names, same input and output parameters.

Overview

In this project, you will leverage the tokenizer of Project 0 to read the textual input and convert it to tokens. The token stream gets parsed and expression trees will be constructed. When an expression tree has been built, you then evaluate it and print the value along with the type of the value, which is integer or float.

Each Tree nodes stores an information record and two pointers referencing the left and right subtrees, respectively. The information in leaf nodes is a number with a type attribute (integer and float). The pointers are null in this case. The information in interior nodes are operator tokens.

The routine that allows you to build an expression tree is a parser. We will provide you with a recursive descent parser. You will expand it adding the code that builds the tree. Here is the structure of the parser:

```
expression := factor '+' factor '+' ... '+' factor
factor     := primary '*' primary '*' ... '*' primary
primary    := '(' expression ')'
            | number
```

The tree T is evaluated recursively:

- If T is a leaf, then its value is the number stored in the leaf.
- If T is not a leaf, let T_L and T_R be the subtrees of the root R of T. Then the value of T is the result of applying the operation of R to the values of T_L and T_R.

The type of result is calculated by the following rules which mirror evaluation of the expression tree:

- int <op> int –> int when <op> is +,-, or *
- int <op> int –> float when <op> is /
- float <op> float –> float for all operations
- float <op> int –> float for all operations
- int <op> float –> float for all operations

It is possible to evaluate expressions without first building a tree. You should not do that, We will extend the tree operations in Part 2, so building the tree explicitly is a good idea.

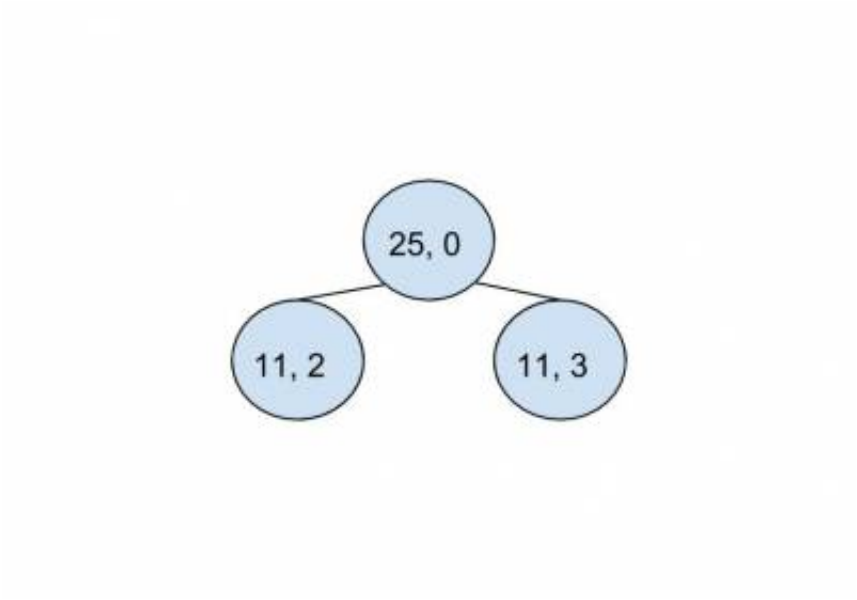
An expression tree will be printed in the following way:

```
([root_token],[left_sub-tree_token],[right_sub-tree_token])
```

Example 1: Single expression

Input: 2*3;?

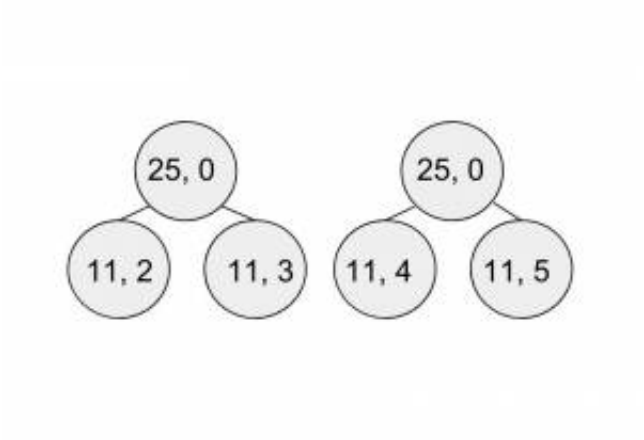
The tree generated by this string of Tokens should look like:



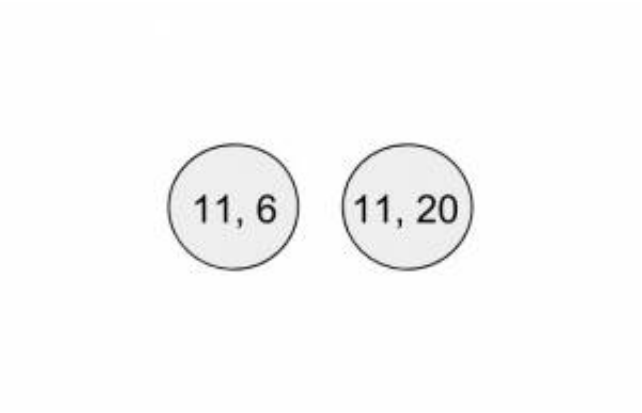
Next, after evaluation, there should be a single node as follows:

Example 2: Two expressions

Input: 2*3; 4*5;? Two separate trees will be generated:



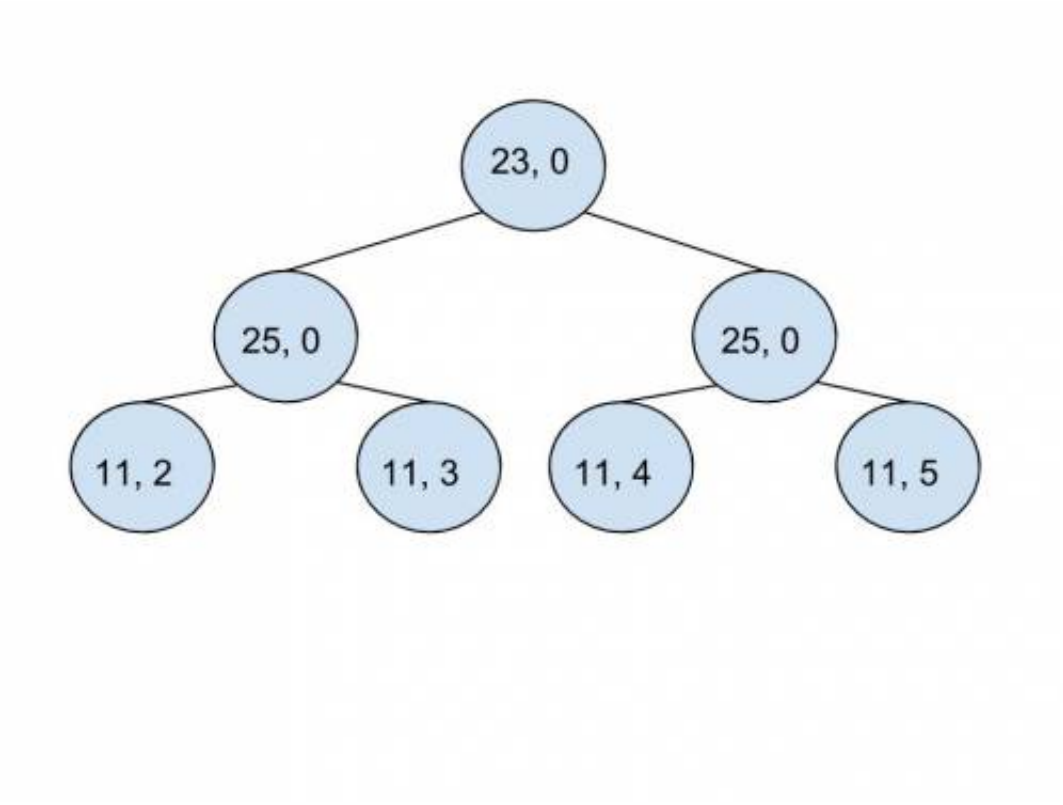
Next, after evaluation of both trees, you will have two separate Tree nodes:



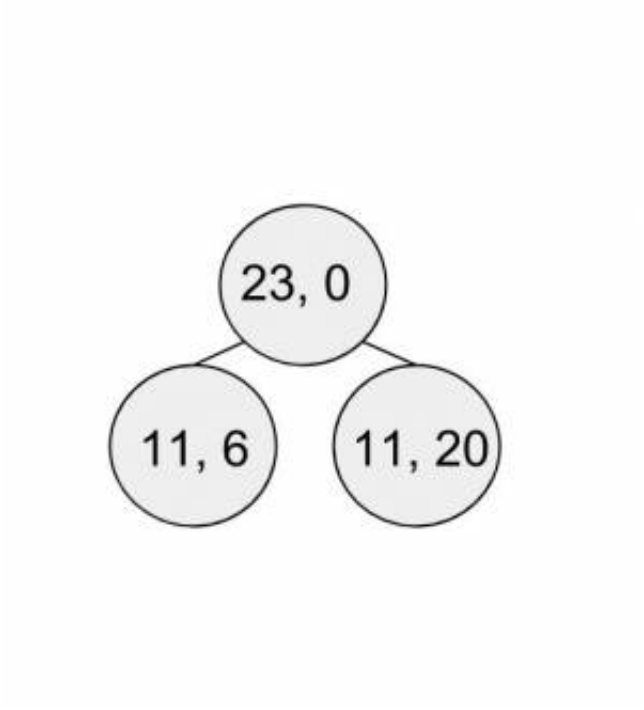
Example 3: Single expression with multiple operators

Input: 2*3+4*5;?

The tree generated by this string of Tokens should look like:



Then, the evaluation of the tree should start at the subtrees of the root:



Finally, the evaluated expression is simplified down into a single node:

Expected Output: the first line of output represents the tree in pre-order and the second line represents the outcome of the evaluation.

```
(23,((25,((11,(),()),(11,(),())),(25,((11,(),()),(11,(),()))))
```

```
(11,26,i)
```

In the second line: 11 means the token is a value, 26 is the outcome and i means the data type is integer.

Note : There are no lingering blanks in the output.

Tasks

You need to implement the three empty methods inside `Parser.java`, `Project2Part1.java` and `Project2.java`. They will be labelled with a comment saying "TODO: PART1.

1. `build_expression_tree`: receives a list of tokens and builds an expression tree. This method utilizes definitions/ methods implemented inside `Parser.java`. **Output** the built expression tree using the `print` method predefined for a `tree` class.
2. `Evaluate`: Receives an expression tree as a parameter and evaluates the expression tree into a single node. **Output** the evaluated expression tree's token using the `print` function predefined for the `token` class.
3. `main`: Reads input from user, construct expression tree, evaluate expression tree and print the formatted output.

Input:

Input is a list of expressions. Each followed by a ";" indicating the end of each expression. "?" denotes the end of the input.

Output:

Each **nonempty** expression should be printed based on their input order.
For each of them, print a line for the initial parse tree and followed by a line with evaluation result.

Test Cases

"Input:" and "Expected Output" should not be printed. They are simply used for illustration.

```
Input: 1+1; ; 2*3.5?
Expected Output: (23,((11,(),()),(11,(),()))
                 (11,2,i)
                 (25,((11,(),()),(12,(),()))
                 (12,7.0,f)

Input: 1+1; ; ?
Expected Output: (23,((11,(),()),(11,(),()))
                 (11,2,i)
```

Project Specific Instructions

Testing: You will begin by using the sanity test script provided to you. It can be found on Piazza resource page . Place `sanity_test` and `data` in the same directory as `Project2Part1.java`. **Always follow the format of sanity test!** Assuming you are in the directory mentioned previously, you must use it as shown below:

```
$sh sanity_test.sh testCaseNumber
The output for this should be:
$Sanity Test Passed!
```

Furthermore, you should also create your own test cases and test your program against them. The program takes input from standard input, so you should have to feed in your test cases accordingly. It is highly recommended that you use redirection to test your programs. Talk to your TA if you want to learn how to do it. You could also just look at what `sanity_test.sh` is doing to understand how to use redirection. Move ahead only once you are convinced of the correctness of your work. PLEASE

REMEMBER THAT `sanity_test.sh` IS THERE ONLY TO TEST YOUR PROGRAM AGAINST ELEMENTARY TEST CASES. It is up to you to construct all kinds of test cases and validate your work.

Grading

Overview

Tests	Points
Program Compiled and Run	10
Coding Standard	10
Passing All Test Cases	80
Total	100

Details

Program Compiled and Run: 10 pts

We can compile your program and run it successfully, according to our requirements.(If your code cannot compiled by our script you lose 10 points).
We **DO** care about warnings. You will lose points if warnings are raised even though your code compiles and runs.
You should **NOT** change the signatures of the given methods.This can even lead to worse situation: failing most test cases.

Coding Standard: 10pts

Your code should be well structured.
Rule of the thumb: TA can understand any method in less than 10 seconds.
Suggestions:

1. Add Comments. Usually you will have the same amount of comments as code.
2. Friendly Variable Names.
3. Lots of small methods rather than several large methods.
4. **Indentation. You will lose all 10 points if your code is not well indented.**

Passing All Test Cases: 80pts

1. build_expression_tree: 45 pts
2. Evaluate: 25 its
3. main: 10 its

You are responsible for the robustness of the program. Passing only the provided vanilla test cases may result in very low scores.

Early Submission

If you submit your project before "early submission" deadline, You will receive a projected score before the real deadline.

Your program will be tested with exact same test cases for the final judgment. But you are allowed to resubmit if you want to improve the score.

After the projected score shows up in Blackboard, check Piazza. We will post students common mistakes to help you improve your project.

We will **NOT** release test cases before the real deadline.

Submission

You **must** create a directory "project2p1". You **must** Put all your .java files under "project2p1" directly. If you are using Eclipse or other IDE, make sure you put .java files under project2p1 directly. Failing to do so will result in 10 points off.

Your file structure should looks like the following:

```
project2p1:
  |- all java files(No extra folder like "src")
```

Note:

1. you should **not** literally copy the following commands.
2. Replace yourLogin with your login ID like "zhan1015".

To resubmit, you just need to retype the following commands and type "yes" after the third command.

```
ssh yourLogin@data.cs.purdue.edu
cd directoryContainsProject2
turnin -v -c cs251 -p project2p1 project2p1
```

After the third command is executed, the system will give you some feedback about which files and folders have been submitted. If you resubmit a project, the previously submitted files will be overwritten.

---

cs25100i/fall15/projects/project2p1.txt · Last modified: 2015/10/09 03:34 by zhan1015