

# COMP251 - Assignment2

## Deadline: Nov 12, 2021

### Goal:

In this assignment you will write a calculator. You should use stack ADT and binary trees to write a simple parsing algorithm that takes an expression and builds an equivalent expression tree. Then the expression tree will be used to evaluate the input expression.

### Problem:

Your calculator takes a file that contains an expression (written in infix format) and executes. The input file has a specific format.

### Input File Format:

Each file contains exactly one infix expression like:

$$(x - 2) * (y + 3)$$

So the expression might have some variables ( $x, y, \dots$ ) and literal values ( $2, 3, \dots$ ). The input file will start with lines that contain the variables and their values and the last line will be the expression. An example:

$$y = 4$$
$$x = 3$$
$$(x - 2) * (y + 3)$$

### Syntax:

**Variables:** the variables are one letter characters: “a” to “z” and “A” to “Z”. Variables are case sensitive so “a” and “A” refer to two different variables.

**Infix-expression:** an infix-expression is an algebraic expression in infix format. The operators in an infix-expression include: “+”, “-”, “\*”, “/” and “%”. The operators are integer **binary operators** (there is no unary operator such as  $-x$ ). The operands are either variables or positive integer constants. **You can assume that there is exactly one space between operands and operators (including parenthesis) in an infix expression. This means that your program**

**will be tested for the programs that include statements with exactly one space between their operands and operators.** Below are some examples of valid infix-expression:

```
Y
X + 2 * y
( 0 - 45 )
( x - y ) * ( x + y )
G * ( ( m * n ) / ( r * r ) )
```

and here some invalid infix-expression:

```
2y           //2y is not a variable.
- 45 * u + ( - Z ) //no unary operator is allowed.
( ( y + 7 ) * z //unbalanced parenthesis.
Sum * J + I
//sum is not a valid variable (variables consist of one letter
only).
[ x + y ]     //invalid token ]
x ^ 2         //invalid token ^
```

Here are some sample input files and the expected output for them:

Input file

```
x = 1
y = 3
x * x + y * y
```

Output

```
10
```

Input file

```
z = 10
( z + 2 ) / 2
```

Output

```
6
```

Input file

```
T = 10  
T
```

Output

```
10
```

Input file

```
( 2 + 4 * 5 ) / 3
```

Output

```
7
```

Input file

```
S = 4  
( 2 + 4 * 5 ) / 3
```

Output

```
7
```

Input file

```
9
```

Output

```
9
```

**Note:** the number of lines that initialize the variables depends on the input file. But all those lines will come first (the last line will be the expression). And the format of initialization lines is fixed:

```
variable = value
```

Value is an integer number. And all the tokens are separated by white-space.

## Syntax Errors

Your calculator should take the input file as a command line argument. Note that it is possible to have errors in the infix expression, in that case, the calculator should print an appropriate message about the error, and then terminate the execution (your program should not crash).

There are two possible types of errors:

1. **Syntax error:** if the infix expression is invalid, it is a syntax error. Look at examples of errors mentioned above (unbalanced parentheses, invalid variable names such as x2, invalid operator or character such as !, ^, &, ...). For these types of errors it's enough that your program prints "syntax error".
2. **Undefined variable error:** If the infix expression tries to access the value of a variable that was not initialized before, then your program must print "undefined variable error" message. For example:

**The following shows some examples:**

Input file

```
read x = 1
x ^ 2
```

Expected output:

```
Syntax error.
```

Input file

```
x = 24
y = 7
x + ( - y )
```

Expected output:

```
Syntax error.
```

Input file

```
P = -4  
Q = 2  
d = 5  
c * d + P
```

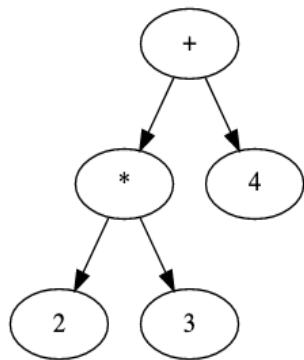
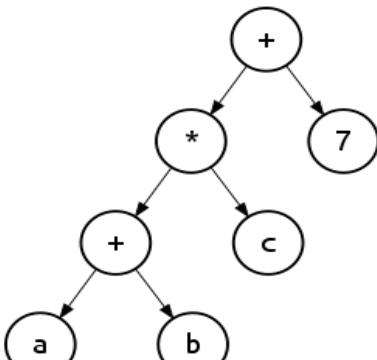
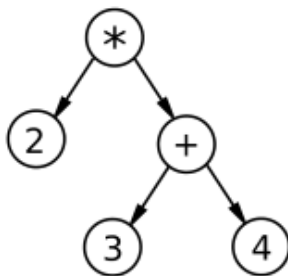
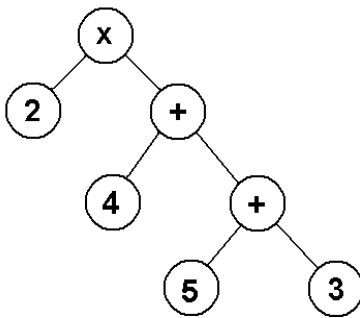
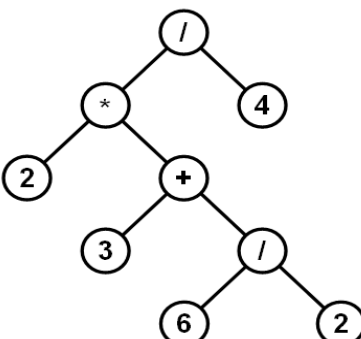
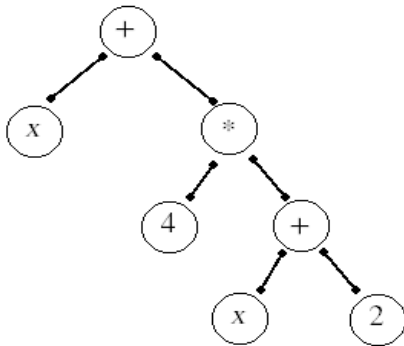
Expected output:

```
error: variable c is not defined.
```

## Guideline:

### Expression Trees:

As we discussed in class, it is not a very efficient way to design an algorithm to work on infix notations. Instead, infix notation is first converted into either postfix notation or an expression tree and then the expression will be computed. We discussed how to parse the infix expressions and convert them to postfix expressions in class. **In this assignment you will parse the infix expressions and build the equivalent expression trees.** The following shows some example of infix expressions and equivalent expression trees:

$2 * 3 + 4$	$(a + b) * c + 7$	$2 * (3 + 4)$
		
$2 * (4 + (5 + 3))$	$2 * (3 + 6 / 2) / 4$	$x + 4 * (x + 2)$
		

In expression trees, internal nodes correspond to operators and leaves are operands either literal values like 3, 5, or variables like a, b.

## Parsing Infix Expressions:

Parsing algorithm to convert infix expression to expression tree is very similar to the algorithm for converting infix expression to postfix expression that we discussed in class. But here we build the tree instead of printing the postfix expression. The following is the algorithm:

- Create an empty stack for operators opStack
- Create an empty stack for subtrees treeStack
- Read the infix expression from left to right (let e be the next token in the infix expression)
  - if e is an operand, build a tree with e as root and push the tree to treeStack
  - if e is a left parentheses, "(", push it to the opStack
  - if e is an operator:
    - if opStack is empty, push e to the opStack
    - otherwise, pop operators of greater or equal precedence from the opStack
    - For each operator, op, that you popped, pop two trees from treeStack, (t1, t2)
    - Build a new tree with op as the root, t2 as left subtree and t1 as right subtree
    - Push the new tree to treeStack
  - You should stop when you see an operator of lower precedence or "(" and then push e to the opStack
  - if e is ")" , pop operators from the opStack until you see matched "(" (NOTE: you should also pop "(" )
    - For each operator, op, that you popped, build a tree: (similar to previous case)
    - pop two trees from treeStack, (t1, t2)
    - Build a new tree with op as the root, t2 as left subtree and t1 as right subtree
    - Push the new tree to treeStack
- When you reach to the end of infix expression, pop all operators in opStack:
  - For each operator, op, that you popped, build a tree: (similar to previous case)
  - pop two trees from treeStack, (t1, t2)
  - Build a new tree with op as the root, t2 as left subtree and t1 as right subtree
  - Push the new tree to treeStack
- at the end there should be just one tree in treeStack, and it will be the full expression tree

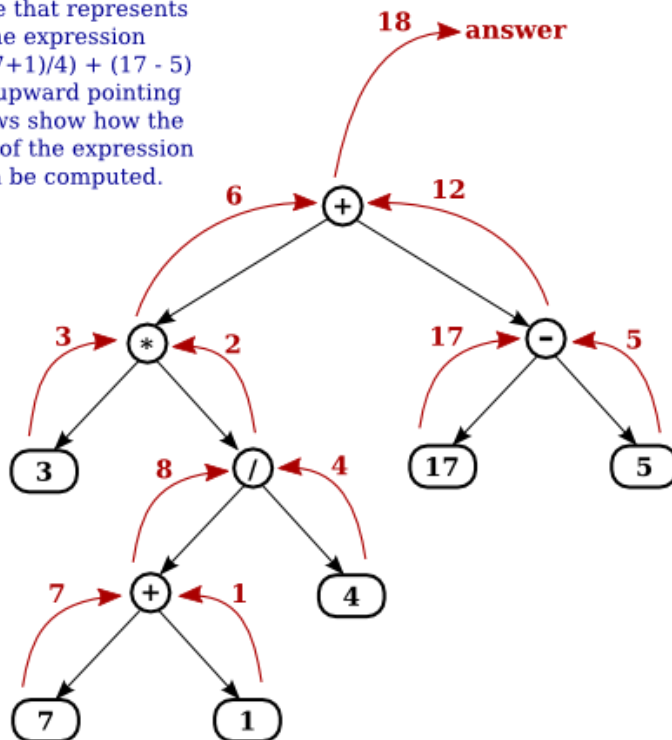
## Evaluating the Expression:

Now that we have the expression tree, it's easy to find the value of the input expression. For each node in the tree define a value (a field in the class node).

If the node is a leaf node, then its value is simply the number that the node contains (if it is a variable it should have been initialized and you have the value for that in the input file).

If the node is an internal node (corresponding to an operator), then the value for the node will be computed by finding the values of its left and right subtrees (children) and then applying the operator to those values. The process is shown by the upward-directed arrows in the illustration. The value computed for the root node is the value of the expression as a whole.

A tree that represents  
the expression  
 $3 * ((7+1)/4) + (17 - 5)$   
The upward pointing  
arrows show how the  
value of the expression  
can be computed.



[source](#)

You need to define a **recursive** method to evaluate the tree (Hint: it is a kind of post traversal algorithm over the tree).

## Handling errors:

There are two types of errors as we discussed but they may occur and be captured and handled in different steps.

- Syntax errors can be captured while you are converting the infix expression to expression tree (invalid variable name, unbalanced parentheses, invalid operator, invalid token...)
- Undefined variable errors can be captured when you are evaluating the expression tree.



## Input program file

The calculator must get its input filename as a command line argument.

If you are using the eclipse compiler go to the Run menu and click on “Run Configuration” (if you are using Windows: then click on “Java Application” on the left bar). Select the “Arguments” tab and type the name of the input file in the Program arguments box.

If you are using the command line java compiler, type the name of the input file after the main class. For instance if the input program is **test.esp** then type **java Calculator test.esp** in the linux prompt. This way the `arg[0]` value in the main method below will be the string **“test.esp”**.

```
public static void main(String arg[]){
    //arg[0] is the first argument which should be the name of the input program.
    ...
}
```

## Requirements:

Make sure your final submission passes all the following requirements to get the full mark:

- Your code should run without error.
- Your code should handle all the syntax errors (print appropriate message then terminate the execution). Your code should not crash.
- A set of test cases will be posted to test your code. But you have to write more test cases and test your code and make sure you capture all possible errors.
- You should implement all the incomplete methods given in the source codes posted along with this assignment. (You may need to add more methods, it is your choice).

## Submission

- Please record a short video in which you explain your work. Run through your code and show how it works. The design of your classes and all the methods (just a brief description, algorithms and data structure you use). If you could not finish all parts of the assignment, mention them in your video.
- **Important:** post your video on youtube and set the **publish time to be three days after the deadline** of the corresponding assignment.
- Submit: the **source code** (.java files), and a **readme** file which you should put the link to your video on youtube.

- **Important:** I'll randomly choose some students for **oral presentation**, and they will be notified by email. If you get selected, I'll schedule a time slot and **you should present your assignment to me and answer some questions.**

## Important Remarks<sup>1</sup>

It's very important that you make sure your program compiles without any problem (you may get 0 if your program didn't compile)

For this project, you must work alone!

**By alone rule:** All code that you submit should be written by you alone, except for small snippets that solve tiny subproblems (of course you are allowed to use the source codes we discussed in class or I post on blackboard).

**Do Not Possess or Share Code:** Before you've submitted your final work for a project, you should never be in possession of solution code that you did not write. You will be equally culpable if you distribute such code to other students or future students of COMP251 (within reason). **DO NOT GIVE ANYONE YOUR CODE – EVEN IF THEY ARE DESPERATELY ASKING. DO NOT POST SOLUTIONS TO PROJECTS ONLINE** (on GitHub or anywhere else)! If you're not sure what you're doing is OK, please ask.

## Permitted:

- Discussion of approaches for solving a problem.
- Giving away or receiving significant conceptual ideas towards a problem solution. Such help should be cited as comments in your code. For the sake of other's learning experience, we ask that you try not to give away anything juicy, and instead try to lead people to such solutions.
- Discussion of specific syntax issues and bugs in your code.
- Using small snippets of code that you find online for solving tiny problems (e.g. googling "uppercase string java" may lead you to some sample code that you copy and paste into your solution). Such usages should be cited as comments in your hw, lab, and especially project code!

---

<sup>1</sup> The description of rules about code submission are from [UBerkeley](#).

## **Absolutely Forbidden:**

- Possessing another student's project code in any form before a final deadline, be it electronic or on paper. This includes the situation where you're trying to help someone debug. Distributing such code is equally forbidden.
- Possessing project solution code that you did not write yourself (from online (e.g. GitHub), staff solution code found somewhere on a server it should not have been, etc.) before a final deadline. Distributing such code is equally forbidden.