Students:

This content is controlled by your instructor, and is not zyBooks content. Direct questions or concerns about this content to your instructor. If you have any technical issues with the zyLab submission system, use the **Trouble with lab** button at the bottom of the lab.

26.3 Project 3, Phases 1 and 2

Purpose

1. To gain experience working with binary trees.
2. To understand how different traversals of trees are used for different purposes.
3. To practice using recursion and other techniques to model and evaluate expressions

Background

In a previous section, we learned that a **fully parenthesized** arithmetic expression is either a number or set of numbers and operations as where each binary operation must be enclosed in parentheses.

Here is an example of an expression converted to fully parenthesized form:

```
Expression: 1 + 2 * 3
Fully Parenthesized: (1 + (2 * 3))
```

This lab will build upon our lab for **`Expression Evaluation`** and must maintain the order of mathematical operations. As a refresher, the priority of operands your program will handle is as follows:

```
Parentheses         ( )

Then from left to right:
```

```
Multiplication      *       Division            /
Addition            +       Subtraction         -
```

Problem Statement

Goal

The goal is to

1. Process mathematical expressions that are provided as strings into binary (expression) trees.
2. Represent the trees in prefix, postfix, and fully parenthesized infix notation.
3. Based on the initial boolean value, you will or will not then evaluate the expression string.

Example

Input:

```
false
1 + 5 * 3
```

The output of this should be:

```
Prefix: + 1 * 5 3
Postfix: 1 5 3 * +
Fully parenthesized: (1+(5*3))
```

Let's say the first parameter were flipped from `false` to `true`. This means that if the input were

```
true
1 + 5 * 3
```

then the output should include the evaluation of the expression, as follows:

```
Prefix: + 1 * 5 3
Postfix: 1 5 3 * +
Fully parenthesized: (1+(5*3))
-----------------
Evaluated: 16.0
```

Overall Program Structure

In this project, you need to code one class that extends the abstract class **`ExpressionTree`**.

Your **`Expression`** class needs to contain:

- one constructor
  - This is the hard part of this project.
  - In terms of planning it, I suggest that you think about which operation is the last to be performed in the expression.
    - Can you write down a rule to identify this operation?
  - Deal with the easier case when there are no parentheses first. Once you accomplish this you can finish quickly using a recursion.
- two output methods
  - The **`fullyParenthesized`** method can also be done using a recursion. This will require an auxiliary recursive method to call on.
  - The **`evaluate`** should be straightforward; it builds upon previous work you have done in this class.
    - Hint: What other labs seem useful here?
- any private helper methods that you find useful

In the main program that tests your class, the Utility **`getInput`** method reads a mathematical expression that is typed as input.

- The expression can contain any combination of numbers, variable identifiers and mathematical operations.
- These mathematical operations are `+`, `-`, `*`, `/`, `(` and `)`.

The constructor should turn an input expression into the content of a binary (expression) tree.

The tree can then be printed in prefix, postfix, and fully parenthesized infix notation. The first two of these methods **have been coded for you** in the abstract class `ExpressionTree`; you need to code the third.

You also need to code the `evaluate()` method which returns the value of the evaluated expression.

Assumptions

- To simplify things in this project the `-` sign can only be used between 2 quantities that are being subtracted.
  - This will prevent you from ever working with a negative number such as `-2` which would involve a different use for the `-` character.
  - However, the simplification means that your code does **not** need to figure out which meaning to attach to any copy of a `-` character.
- You may also assume that only correct mathematical expressions are entered.

Phase 1: No parentheses in the input, and no evaluation

- Write the constructor to deal with expressions that are input without parentheses.
- Write the `fullyParenthesized` output method.

Examples of Phase 1

Example 1A:

Input:

```
false
2 + 4 - 5
```

Output:

```
Prefix: - + 2 4 5
Postfix: 2 4 + 5 -
Fully parenthesized: ((2+4)-5)
-----------------
```

Example 1B:

Input:

```
false
need - more * space
```

Output:

```
Prefix: - need * more space
Postfix: need more space * -
Fully parenthesized: (need-(more*space))
-----------------
```

Phase 2: Handle parentheses in the input, and evaluate the expression

- Adapt the constructor to deal with expressions that are input with parentheses in them.
  - Hint: a Stack might help here, but is not required.
  - Obviously, the **`fullyParenthesized()`** method you wrote in Phase 1 would need to correctly deal with this type of expression
- Fill in the **`evaluate`** method with code you've written in a previous lab.
  - Alternatively, you can choose to evaluate the expression using the prefix or postfix expression.

Examples of Phase 2

Example 2A:

Input:

```
true
6 * ( 7 + 5 ) / 4 - 2
```

Output:

```
Prefix: - / * 6 + 7 5 4 2
Postfix: 6 7 5 + * 4 / 2 -
Fully parenthesized: (((6*(7+5))/4)-2)
-----------------
Evaluated: 16.0
```

Example 2B:

Input:

```
false
6  - 5 + 6 / 2 + ( 4 + 5 / 3 ) / 3 - (2 + 3 * 4)
```

Output:

```
Prefix: - + + - 6 5 / 6 2 / + 4 / 5 3 3 + 2 * 3 4
Postfix: 6 5 - 6 2 / + 4 5 3 / + 3 / + 2 3 4 * + -
Fully parenthesized: ((((6-5)+(6/2))+((4+(5/3))/3))-(2+(3*4)))
-----------------
```

Example 2C:

Input:

```
true
8 * 16 / 5 + (2 - (3 - 6))
2 * (3 + 6 - 2 / 5 + (2 * 3 * 4 - (6 / 2)) - 7 + 2 * 3) + (16 / 4 + 8)
```

Output:

```
Prefix: + / * 8 16 5 - 2 - 3 6
Postfix: 8 16 * 5 / 2 3 6 - - +
Fully parenthesized: (((8*16)/5)+(2-(3-6)))
```

```
-----------------
Evaluated: 30.6

Prefix: + * 2 + - + - + 3 6 / 2 5 - * * 2 3 4 / 6 2 7 * 2 3 + / 16 4 8
Postfix: 2 3 6 + 2 5 / - 2 3 * 4 * 6 2 / - + 7 - 2 3 * + * 16 4 / 8 + +
Fully parenthesized: ((2*(((((3+6)-(2/5))+(((2*3)*4)-(6/2)))-7)+(2*3)))+((16/4)+8))
-----------------
Evaluated: 69.2
```

Requirements (applies to both phases)

- The **`fullyParenthesized()`** method is expected to return the properly fully parenthesized expression as a String.
    - This expression must be in infix notation and fully enclosed in parentheses.
    - The expression **must** also retain its order of operations and **should not** be evaluated yet.
- The evaluation tests will only be run on fully parenthesized numerical expressions
    - This means that you will not have to handle evaluation of anything other than numbers.
- You are limited to **15** submission attempts.
- The code will be expected to read in **`String`** values.
- The **`evaluate()`** method is expected to return the properly evaluated expression as a **`double`**.
- You may write additional private helper methods if need be; however the **`evaluate()`** and **`fullyParenthesized()`** methods **must** be implemented by the end of the project.
- All code you write should exist solely in the **`Expression`** class.
    - NO CODE SHOULD BE ADDED TO THE MAIN FUNCTION
- Base your code on the given template.
    - Any code not adhering to the given template may work, but may not pass all tests upon submission and will not receive a full grade.

LAB ACTIVITY | 26.3.1: Project 3, Phases 1 and 2 | 0 / 60

Downloadable files

`Node.java` , `Utility.java` , `Expression.java` ,
`BinaryTree.java` , `ExpressionTree.java` , `BNode.java` **Download**
`Tree.java`

File is marked as read only Current file: **Node.java**

```
1  import java.util.Iterator;
2
3  abstract class Node<T> {
4     protected Node<T> parent;
5     protected T data;
6
7     public abstract Iterator<? extends Node<T>> children();
8
9     public void setParent(Node<T> n) {
10       parent = n;
11    }
12
13    public void setData(T t) {
14       data = t;
15    }
16
17    public Node<T> getParent() {
18       return parent;
```

Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → Node.java (Your program) → Output (shown below)

Program output displayed here

Signature of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

Trouble with lab?