Creative Project 2: 12x-Pression

Specification

Learning Objectives

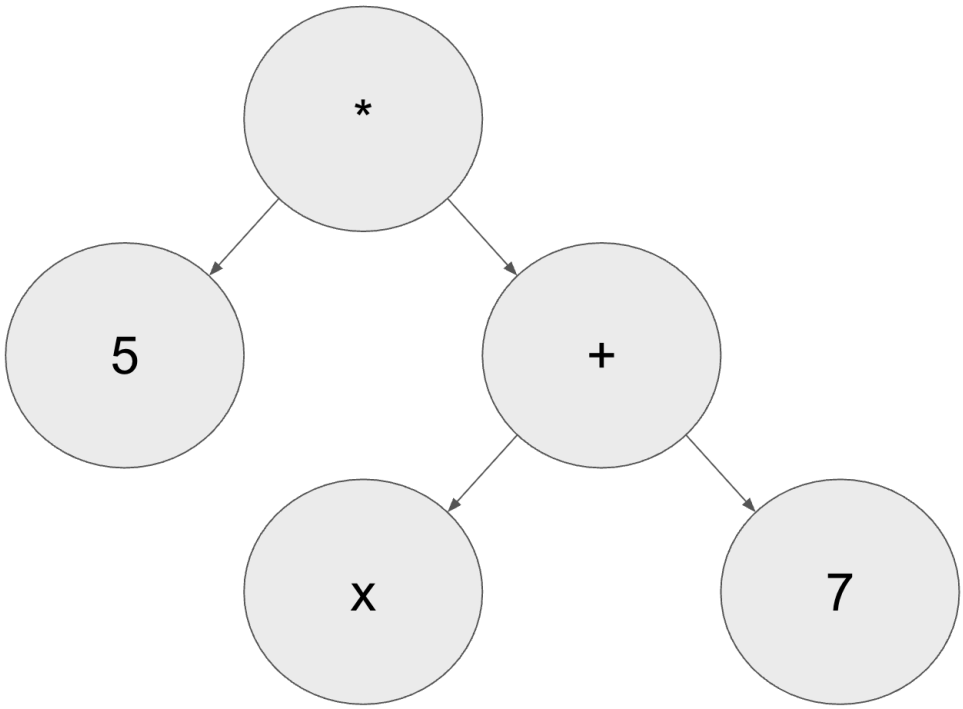
By completing this assignment, students will demonstrate their ability to:

- Define data structures to represent compound and complex data
- Structure data appropriately to efficiently solve a problem
- Write functionally correct Java classes to represent binary trees
- Produce clear and effective documentation to improve comprehension and maintainability of a method
- Write methods that are readable and maintainable, and that conform to the provided guidelines for style and implementation

Background

At this point, you're likely to be super familiar with algebraic expressions. Interestingly, expressions can be defined recursively, and we can represent simple expressions using a binary tree! For our simple definition, we'll say that an expression can be either a variable/constant, or it is a binary operator with subexpressions.

For example, we can represent the following expression `5 * (x + 7)` as:



Notice that the **leaves** of our tree store **constants/variables**, whereas the **branches** store the **operators**. Also notice that the parentheses do not appear in the tree. Nevertheless, the tree representation captured the intent of the parentheses since `+` appears lower in the tree than `*`.

Notations

Infix

Infix notation is the most common notation, where the operator is placed *between* its operands. For example, in `1 + 2`, notice that the operator (`+`) is between its operands (`1` and `2`).

Despite the popularity of this notation, it has one main drawback:

- How do we determine the order of operations?

Though it is entirely possible to create an expression tree using infix notation, it is much more cumbersome. You would need to account for parentheses and/or assign a precedence order for your operations (ex: `*` and `/` before `+` and `-`). As a result, *parsing* (breaking it up into parts) becomes much more difficult to do with code.

Prefix

Prefix notation dictates that the operator *precedes* its operands. For example, we would write the familiar infix expression `1 + 2` as `+ 1 2`! The advantage of prefix is its innate ability to express the intended order of operations without the need for parentheses and other precedence rules.

For example, consider this infix expression: `2 + 4 * 3`. With the normal precedence rules, we would

evaluate `4 * 3` first. If we needed to overwrite any precedence rules, we need parentheses: `(2 + 4) * 3`.

With prefix notation, we don't need parentheses or precedence rules because the order of operations is unambiguous; the notation uniquely indicates which operator to evaluate first.

- `(2 + 4) * 3` can be written as `* + 2 4 3`
- `2 + (4 * 3)` can be written as `+ 2 * 4 3`

Fun fact: Some programming languages actually define their entire language in prefix notation! See Lisp for more.

Required Class

ExpressionTree.java

In this assignment, you will implement an expression tree capable of representing simple expressions consisting of constants, variables, and the four basic operators (`+`, `-`, `*`, and `/`). **You may assume that variables do not have any coefficients**.

```
public ExpressionTree(Scanner exprScan)
```

- Construct a new expression tree given a scanner connected to the expression. The provided expression will be in **prefix notation** as described above.
  - You may assume that each variable/constant/operator in the prefix expression is properly separated by whitespace.

HINT: What traversal order should we use for **pre**fix notation?

```
public String toString()
```

- Returns the string representation of the expression tree in **infix** notation as described above.
  - Each operator should have a space between it and its operands. For example, instead of `1+2`, it should be `1 + 2`.
  - The returned string should contain parentheses around each operator in order to be equivalent to the expression in prefix notation. For example, if provided the prefix expression `* + 2 4 3`, the returned string should be `((2 + 4) * 3)`.

HINT: What traversal order should we use for **in**fix notation?

```
public boolean containsVariable()
```

- Returns true if the expression tree contains any variables. False otherwise.

```
public int evaluate()
```

- Evaluate the expression and return its value.
  - When dealing with subtraction and division, we will read the operands from left to right. For example, `- 1 2`, indicates that we should calculate `1 - 2`.
- Throws an `IllegalStateException` if the expression contains any variables.

```
public void substitute(Map<String, Integer> bindings)
```

- A **binding** is an association of a name to a value. For example, `x = 5` is a binding.
- Replace any variable in the expression tree with its bound value.
  - **NOTE:** This does not imply that a variable inside of your tree is guaranteed to be bound.

`ExpressionNode`

We have provided a **private static inner class** called `ExpressionNode` to represent the nodes of the tree. The class contains the following fields:

- `String symbol`
  - This field is analogous to the `data` field in the `IntTreeNode` class we've been working with in lecture, and can store an operator, variable, or constant.
  - This field is declared as `final`, which means it cannot be reassigned after initialization.
- `ExpressionNode left`
- `ExpressionNode right`

Additionally, we have provided a method inside of the class, `isConstant()`, that returns true whether this node is a constant, and false otherwise. You might find `Integer.parseInt(String s)` helpful for converting a string representing a number into an integer.

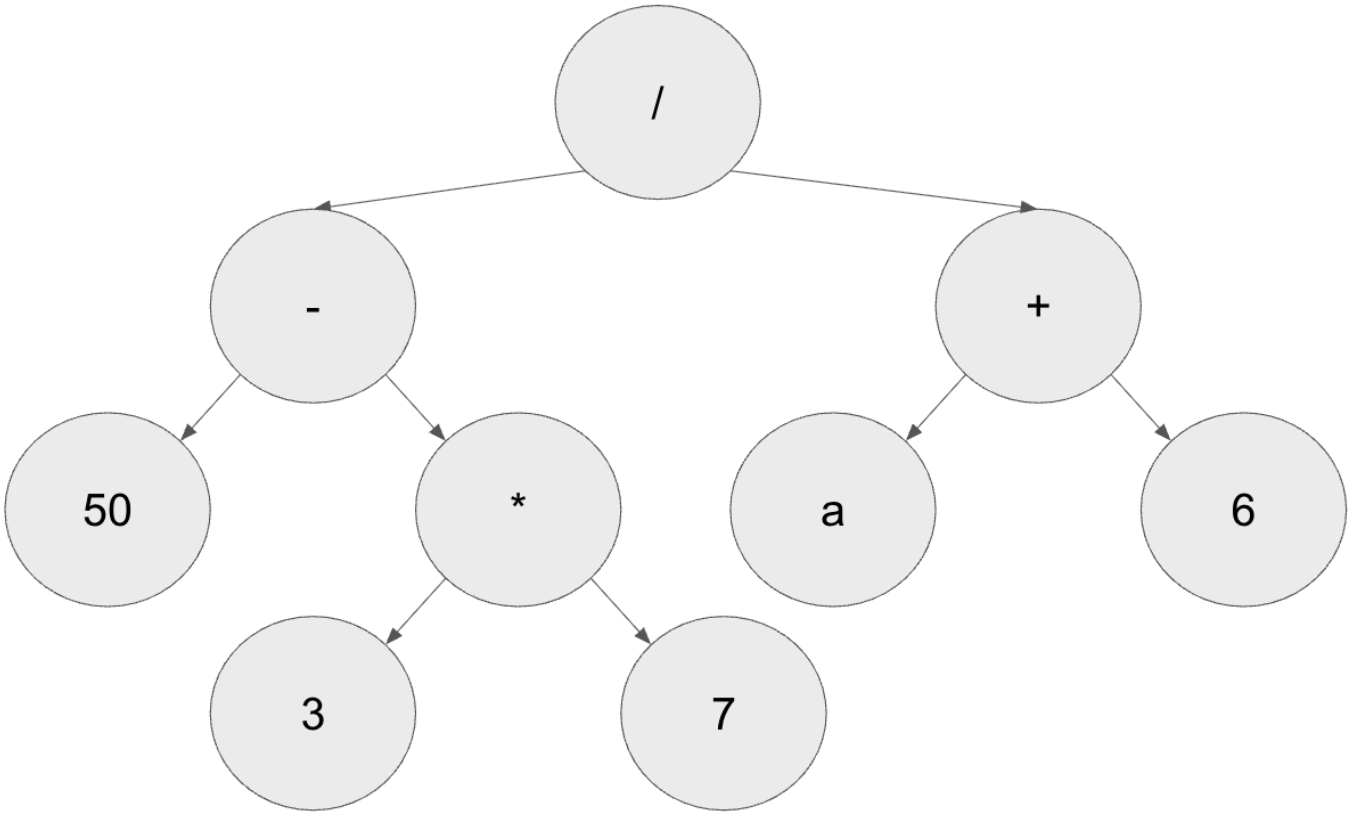
Creative Extension

To earn an E for Project Code, you must implement one of the four provided methods as your creative extension:

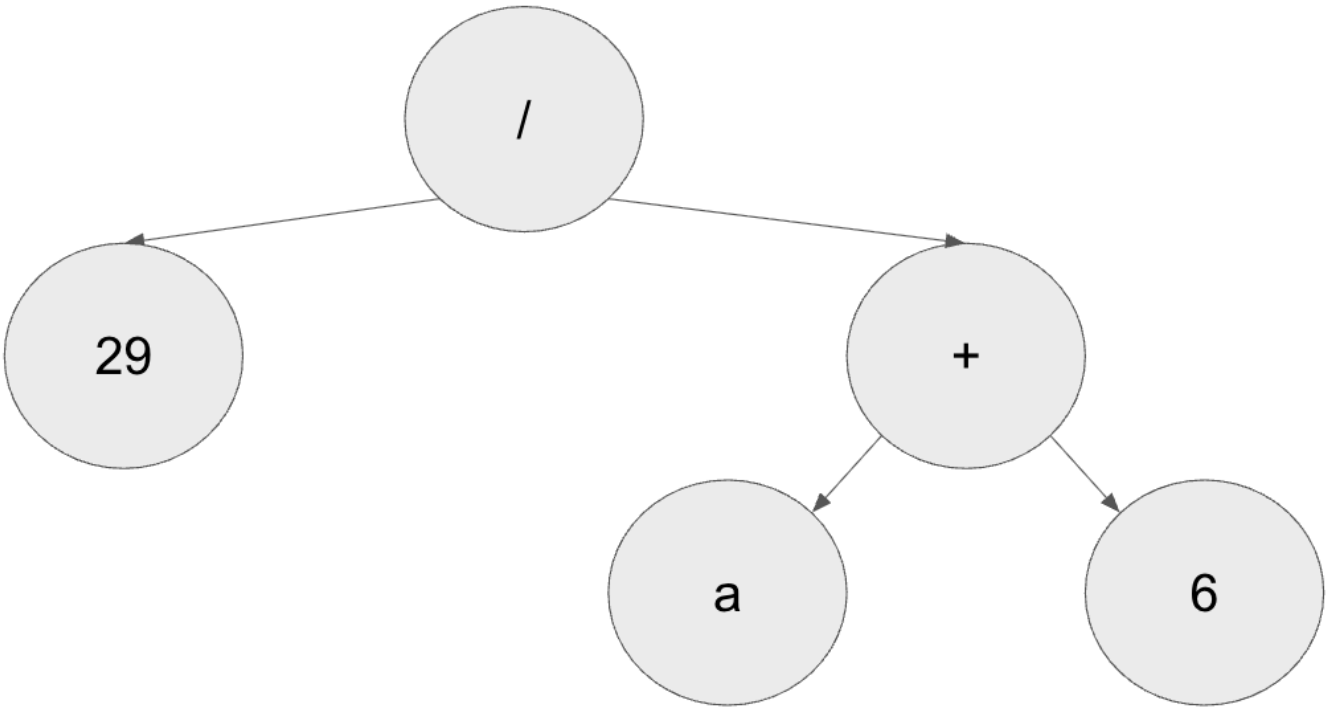
```
public void simplify()
```

- Simplify the expression tree by combining constants where possible.
  - **NOTE:** You should not combine any like variables. For example, `a + a` should not be simplified to `2a`, since we won't deal with coefficients.

- For example, the provided tree

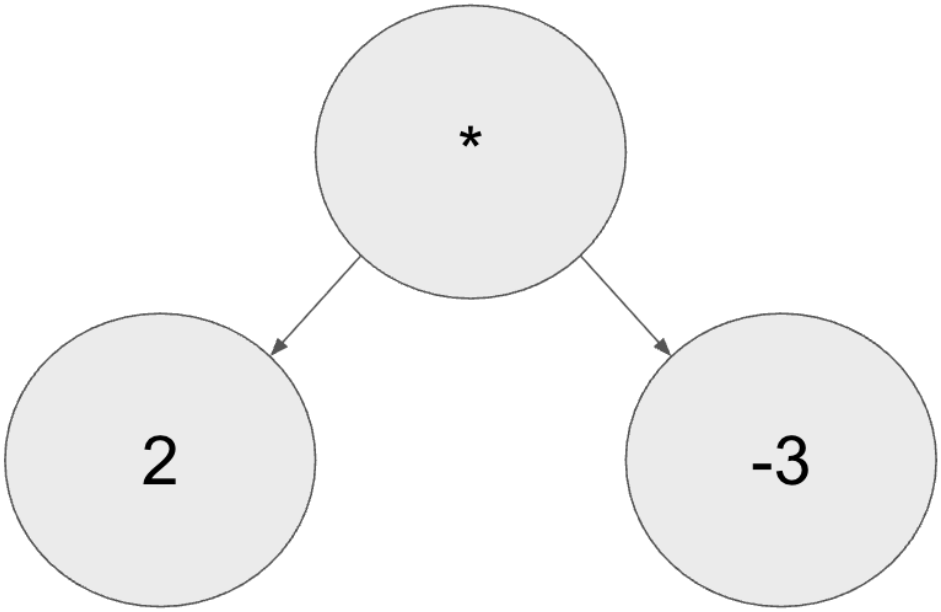


- would be simplified into

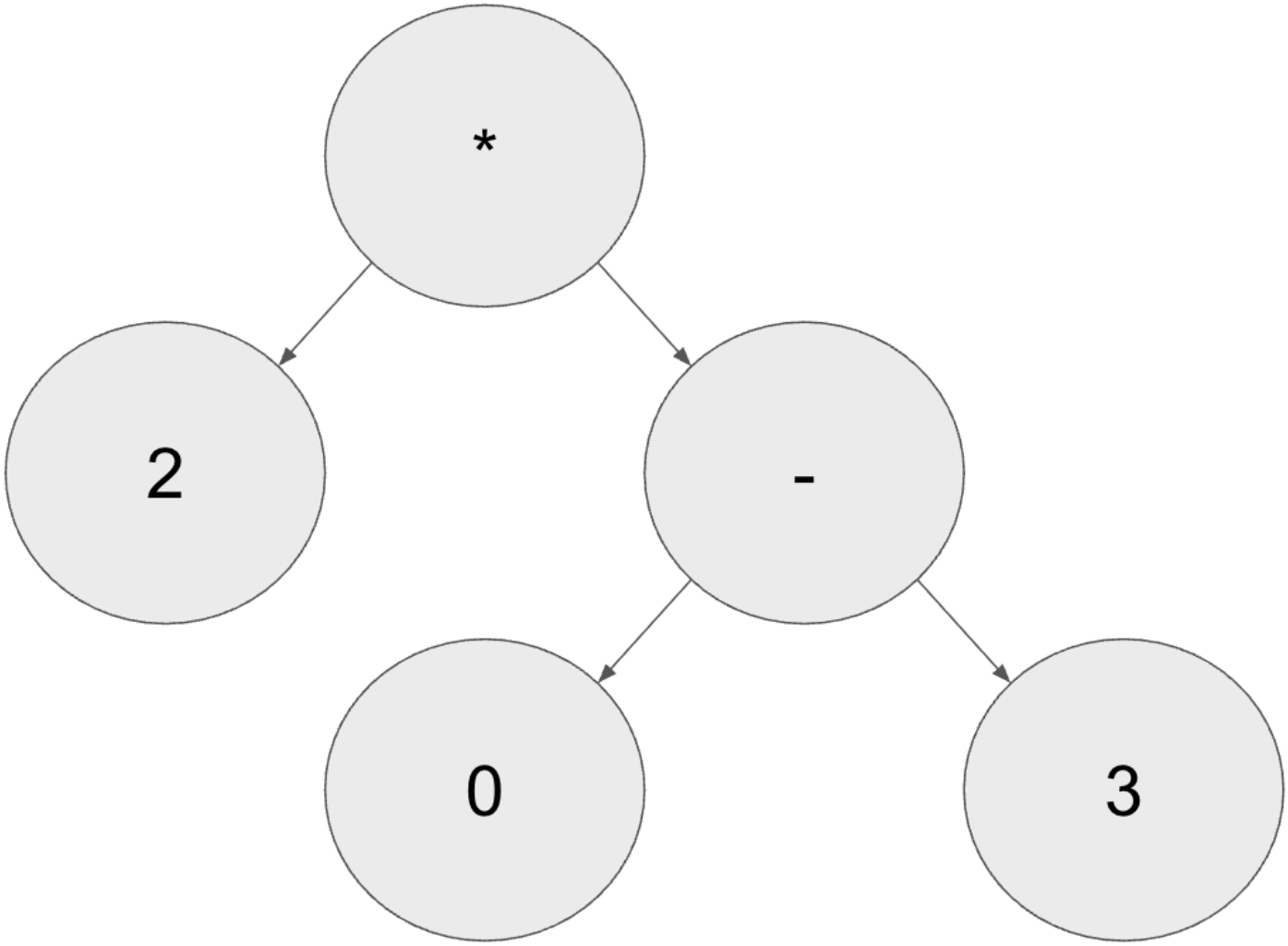



```
public void removeNegatives()
```

- Remove any negative constants from the tree by subtracting them from 0.
- For example, the provided tree

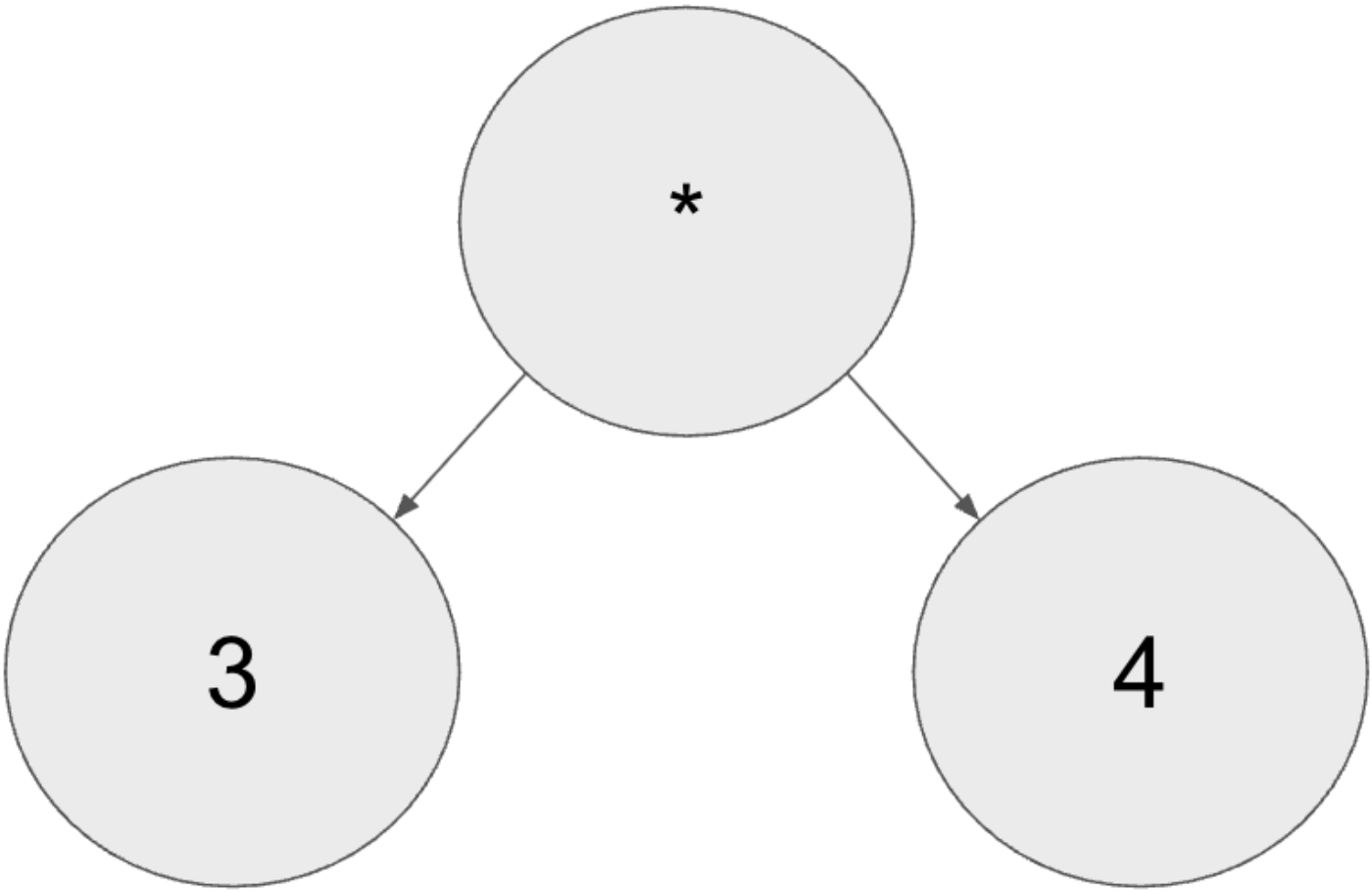


- would turn into

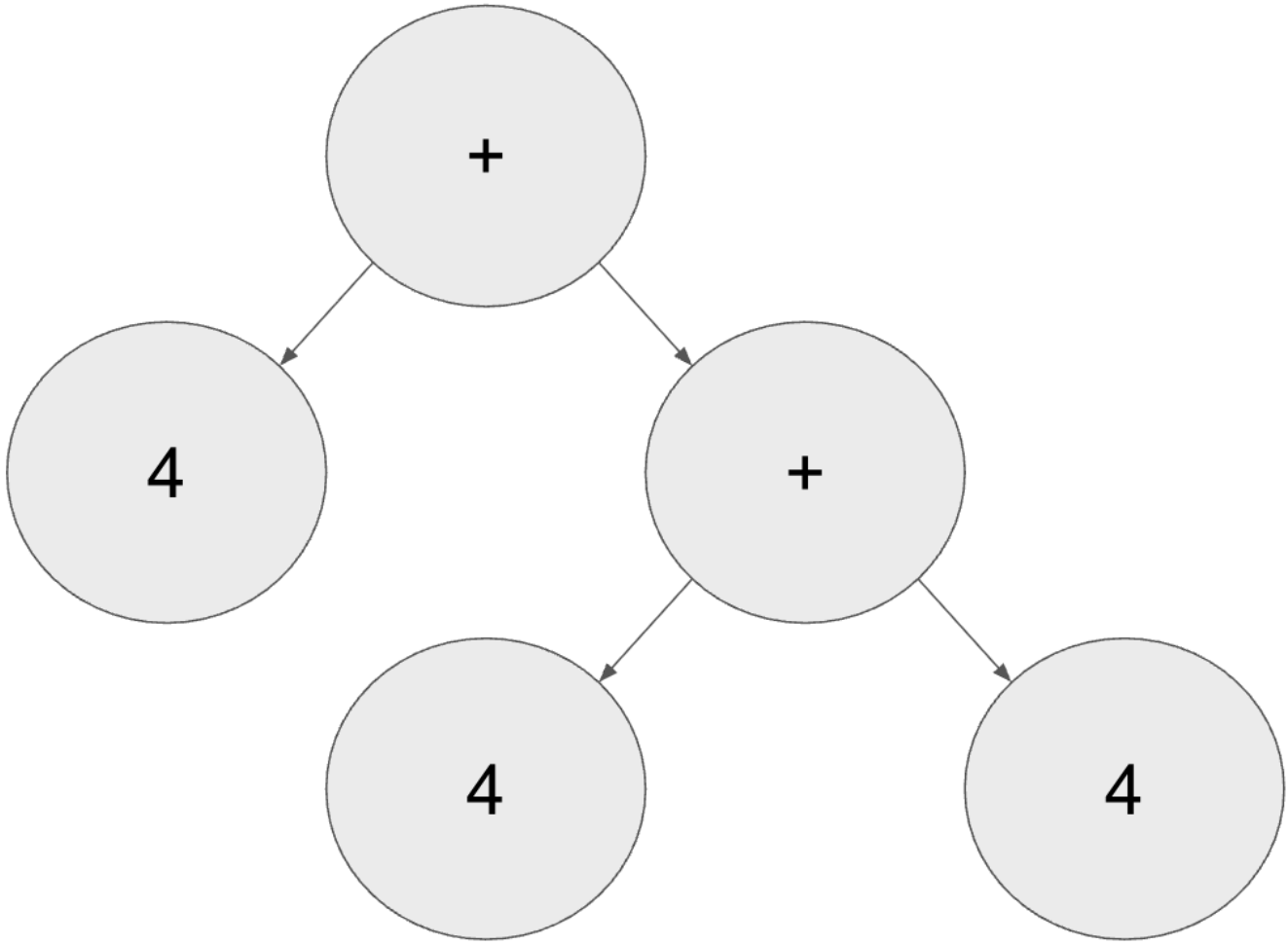


```
public void expandMultiplication()
```

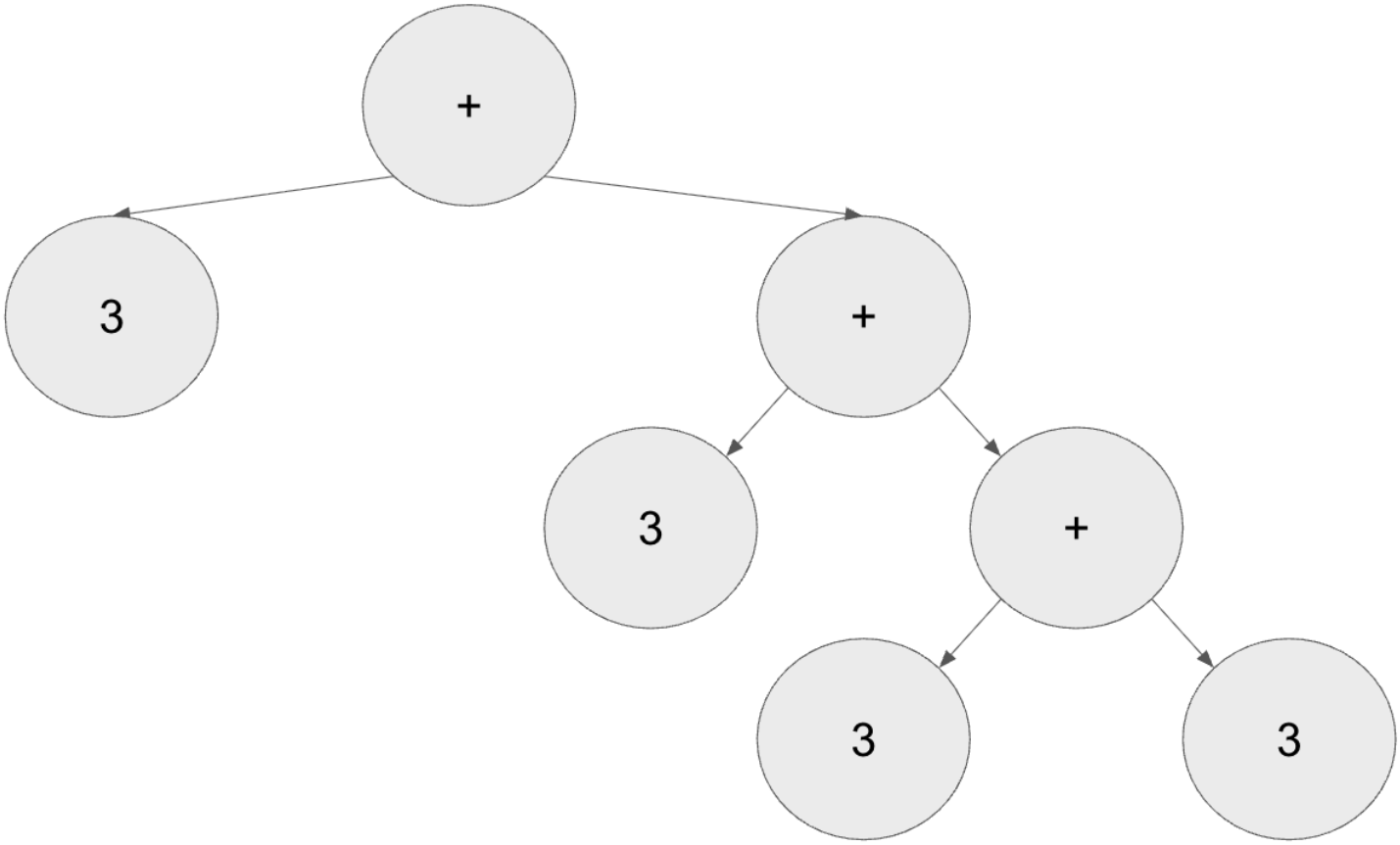
- Expand out any multiplication operators that have two constant operands into a series of addition/subtraction expressions. You can expand the multiplication by using either operand.
- For example, the provided tree



- can turn into



- or



- For negative numbers, you can choose to subtract from 0 or add negative numbers. Either approach is accepted.

> **NOTE:** There are more ways the trees can look. The key point is that the multiplication node is removed and replaced with multiple addition/subtraction operators.

```
public int solve(int rhs)
```

- Given a value for the right-hand side, compute and return the value for the variable that solves the equation.
  - For this method, you may assume that the expression tree has exactly **one** instance of a variable, and the equation is solvable with integer math to exactly one solution
- For example, if you had an expression such as `x + 5`, and you are provided with the right-hand side to form an equation such as `x + 5 = 12`, your goal is to solve for `x`.

> **NOTE:** This extension is harder than the previous ones, so beware!

Implementation Requirements

To earn a grade higher than N on the Project Code dimensions of this assignment, **your *core* algorithms** (aka, traversing the tree) **for each method must be implemented *recursively*. You will want to utilize the *public-private pair* technique discussed in class.** You are free to create any helper methods you like, but the core of your implementations must be recursive.

Implementation Guidelines

For this assignment, you should follow the Code Quality guide when writing your code to ensure it is readable and maintainable. In particular, you should focus on the following requirements:

- An important concept introduced in lecture was called `x = change(x)`. This idea is related to the proper design of recursive methods that manipulate the structure of a binary tree. **You should follow this pattern when necessary when modifying your trees**.
- All methods present in `Classifier` that are not listed in the specification must be `private`.
- Make sure that all parameters within a method are used and necessary.
- When designing helper methods, avoid unnecessary returns.
- Clients of the class should never have to manually set fields of an object immediately after construction (when possible) — there should be a constructor included for this situation.
  - For example, if you were the implementor of the `Point` class:

```
Point coord = new Point(); // Poor usage of constructor
coord.setX(5); // Have to manually set field
coord.setY(7); // Have to manually set field

Point coord = new Point(5, 7); // Correct usage of constructor
```

- You should comment your code following the Commenting Guide. You should write comments with basic info (a header comment at the top of your file), a class comment for your `ExpressionTree` class, and a comment for every method.
    - Make sure to avoid including *implementation details* in your comments. In particular, for your object class, a *client* should be able to understand how to use your object effectively by only reading your class and method comments, but your comments should maintain *abstraction* by avoiding implementation details.
    - Continuing with the previous point, keep in mind that the client should **not** be aware of what implementation strategy your class/methods utilize.
- All methods present in your class that are not listed in the specification must be private.

⭐ [GRADED] 12x-pression

Download starter code:

C2_12xpression.zip

Reflection

For this week's reflection, we would like everyone to read this short passage and then watch and engage with the following video.

Trees aren't just a data structure we introduced just to make your lives harder. There are practical, real-world applications of expression trees that you just implemented!

For example, screen readers use specialized tree structures to parse complex code, web pages, and math formulas so visually impaired developers can navigate and explore them step-by-step. Below is a demonstration of a screen reader stepping through a math *expression.*

Watch **A demonstration of math accessibility, by Richard Orme** (3m 40s)

Question 1

Think about how traversing a tree differs from reading a plain text string left-to-right. Why is a hierarchical tree structure necessary for someone using a screen reader to explore a nested mathematical formula or code block without getting overwhelmed?

Question 2

Recall the section in the specification that talks about the difficulty in parsing infix expressions. Now think about the different ways expressions are displayed on webpages: maybe rendered as a flat

image or an unformatted string. How can this increase the difficulty of making a webpage accessible?

Question 3

The following questions will ask that you practice **metacognition** to reflect on the topics covered on this assignment and your experience completing it. For each question, focus on your plan and/or process for working through the assignment along with the CS concepts. Think about things like how you organized your working time, what sorts of things tended to go wrong, and how you dealt with those errors or mistakes.

What is another way we could have represented expressions instead of using binary trees? What advantages and disadvantages do you think that representation would have had compared to the binary tree representation?

Question 4

Describe how you went about testing your implementation. What specific situations and/or test cases did you consider? Why were those cases important?

Question 5

What skills did you learn and/or practice with working on this assignment?

Question 6

What did you struggle with most on this assignment?

Question 7

What questions do you still have about the concepts and skills you used in this assignment?

Question 8

About how long (in hours) did you spend on this assignment? (Feel free to estimate, but try to be close.)

Question 9

Was any part of the specification or requirements unclear? If so, which part(s), how was it unclear, and how could it have been made more clear?

Question 10

[OPTIONAL] Do you have any other feedback, questions, or comments about this assignment?

(Note that we may not be able to respond to questions here, so please post on the message board if you would like a response!)

Final Submission

Final Submission

Fill out the boxes below and click "Submit" in the upper-right corner of the window to submit your work.

Question

I attest that the work I am about to submit is my own and was completed according to the course Academic Honesty and Collaboration policy. If I collaborated with any other students or utilized any outside resources, they are allowed and have been properly cited. If I have any concerns about this policy, I will reach out to the course staff to discuss *before* submitting.

(Type your preferred name as your response.)