COP 3503 – Project, Stage 2

Purpose

This stage revolves around pre-planning your eventual implementation of the overall project, as will be completed to accomplish Stage 3.

Description

Before proceeding to implement all of the functionality listed in the Project Overview, it is wise to first plan out one's approach to coding the program. For this stage, we are not looking for code – we are looking for plans. Your task is two-fold:

- To draw up UML diagrams (we'll talk about these in a near-future lecture) that show the coding structure you plan to implement in your program and the relationships in the classes thereof.
    - What structure do you believe will be necessary within your program to carry out its requirements?

- To provide a few diagrams on exactly how your proposed design would be used to process select incoming expressions.
    - I want to see thought given to exactly how your design will be used to interpret and perform the math requested of the eventual program. Even if you're unsure that your approach will truly be sufficient, I want a good-faith best effort towards this in your submission.

There are no strict requirements on what must and must not be in your program, but as this project is designed to showcase interesting uses of polymorphism, the rest of this document will be devoted to the declaration of a "base class" that can serve as a strong foundation for the data your program will receive from the results of Stage 1's shunting yard algorithm. The methods declared here should completely suffice for implementation of all requirements mentioned in the project overview, when implemented properly by appropriate extensions of the `Expression` class seen on the next page.

You are still permitted to take alternative approaches, though I will warn you that leaving everything in string form as long as possible tends to be overly tedious to manage when it comes time to actually simplify expressions. Those who take this approach usually struggle when presented with complicated (but legal) inputs.

The foundational/base class

```
// Models a component of a mathematical expression.

class Expression
{
     /* Calculates the value of the expression.
      */
     virtual int getValue() = 0;

     /* Returns the requested operand for this expression
      * component.
      */
     virtual Expression* getLeftSide() = 0;
     virtual Expression* getRightSide() = 0;

     /* Returns the individual multiplicative factors of the
      * expression – only those of a positive power. (Thus,
      * belonging in the numerator if fractional.)
      */
     virtual vector<Expression*> getNumeratorFactors() = 0;

     /* Returns the individual multiplicative factors of the
      * expression – only those of a negative power. (Thus,
      * belonging in the denominator if fractional.)
      */
     virtual vector<Expression*> getDenominatorFactors() = 0;

     /* If the expression is the result of addition or
      * subtraction, returns the individual terms.
      */
     virtual vector<Expression*> getAdditiveTerms() = 0;

     /* Signals the expression to produce a simplified version
      * of itself put into lowest terms.
      */
     virtual Expression* simplify() = 0;
}
```

As a reminder, note that as mentioned in the project overview, you do not have to determine multiplicative factors of an addition or subtraction if it can't be reduced to a single arithmetic term.

An extra note: in case this is not otherwise made clear, you are ***not required*** to use this as a base class within your program. If you feel changes might prove beneficial or necessary, make them – this is your project, and you have creative freedom here. You must, however, make clear to me how your proposed changes and/or alternate structure will aid in fulfilling the project's requirements. (This is significantly aided by well-done example expression diagrams, which are covered later in this document.)

This class is designed to provide the core functionality necessary to obtain the information necessary to reduce a combination of expressions into lowest terms for `Expression`s of the following type:

- `class Integer;`
  - Represents a simple integer.
  - The value of an Integer is the Integer itself.
- `class Addition;`
  - Represents the addition of two expressions, which can be an `Integer` or one of the other types implementing `Expression`, such as `Addition`.
- `class Subtraction;`
  - Like addition, but with a flipped sign on the second operand.
- `class Multiplication;`
  - Models multiplying two expressions together.
- `class Division;`
  - Great for representing fractions.
- `class Exponentiation;`
- `class NthRoot;`
- `class Logarithm;`
  - … you probably get the idea from the other examples.

Examples and Clarifications:

Expression: 8* (2 rt 2) - 4

`getNumeratorFactors()` returns {8*(2 rt 2) - 4}. It does not have to determine 4 * (2 * (2 rt 2) – 1), though that would be worthy of some extra credit.

`getAdditiveTerms()` returns {8*(2 rt 2), -4}.

3 * (8 * (2 rt 2) – 4)

`getNumeratorFactors()` returns {3, 8*(2 rt 2)– 4}.
`getAdditiveTerms()` returns {3 * (8 * (2 rt 2)– 4)}, as there is no top-level addition.

Required use-case Diagrams:

Provide diagrams that show how your design would be used to process the following expressions:

1. 2 * (5 + 2 ^ 3)
2. 3 + 2 * 3 rt 81
3. 1 / 10 + 1 / 10
4. 2 rt 2 * 2 rt 8

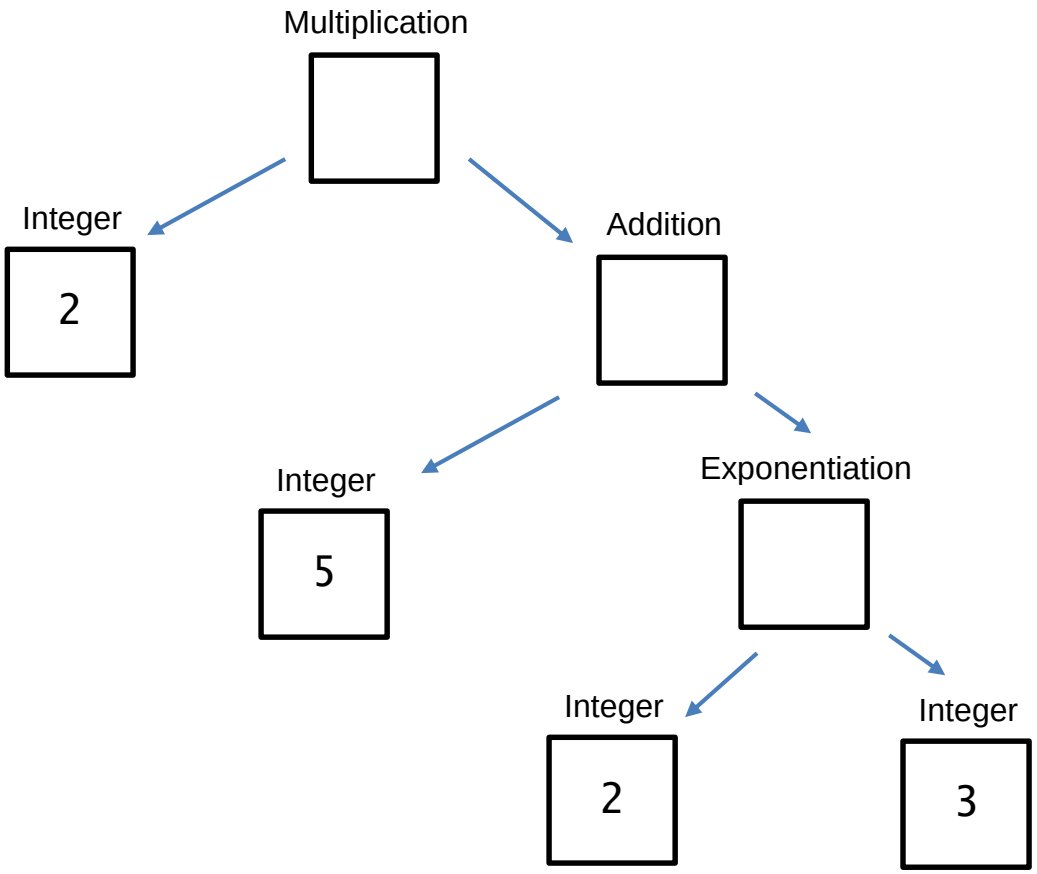
You should use a visual, step-by-step breakdown that would correspond to "showing all your work". Indicate which objects and/or functions are used to transform the expression at each step.

A personal example diagram for #1 is provided at the end of this document.

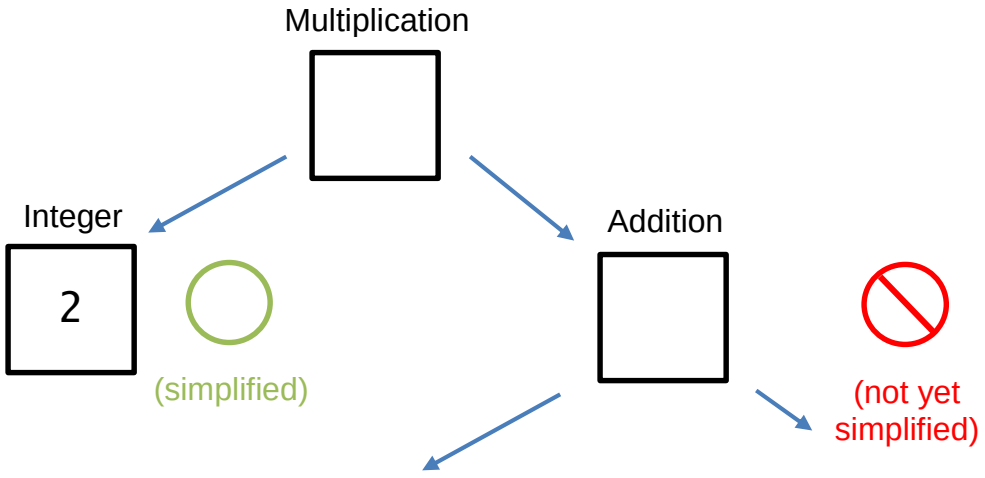
Submissions:

Please submit all work in *.doc or *.pdf form. If including stand-alone images, please use *.gif, *.jpg, or *.png formats.

2 * (5 + 2 ^ 3)

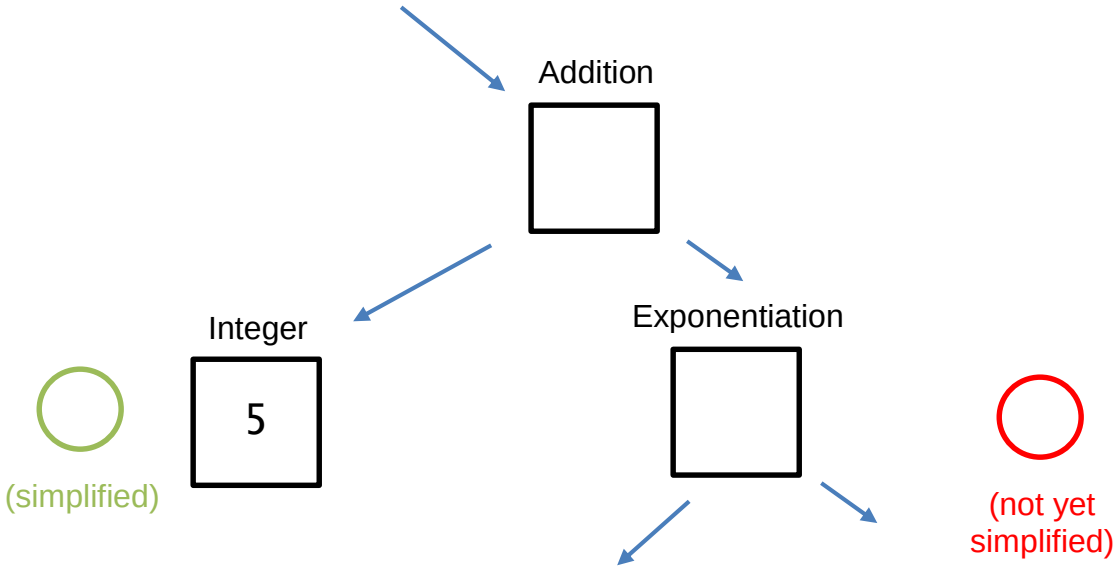


Multiplication.simplify()



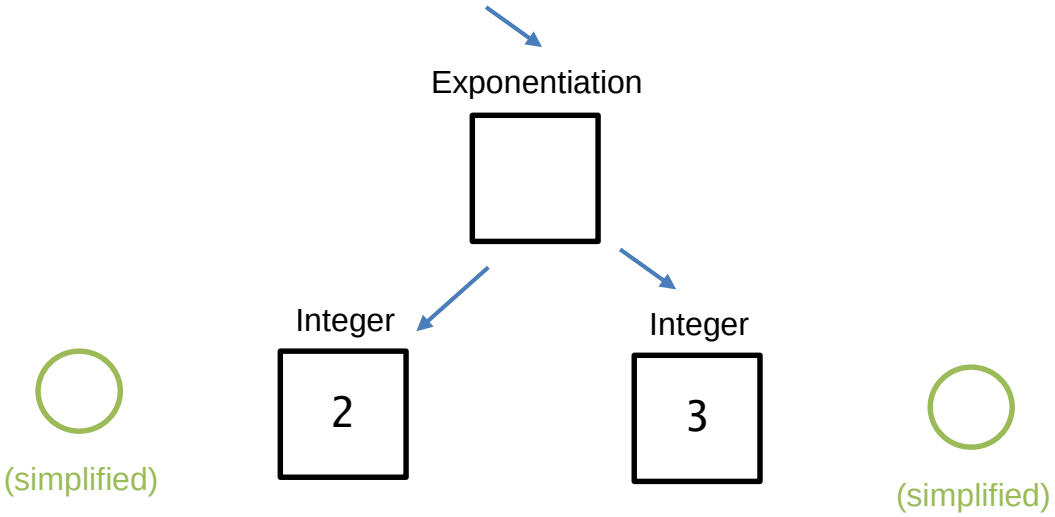
This triggers Addition.simplify() on Multiplication.getRightSide().

Addition.simplify()



This triggers Exponentiation.simplify() on Addition.getRightSide().

Exponentiation.simplify()

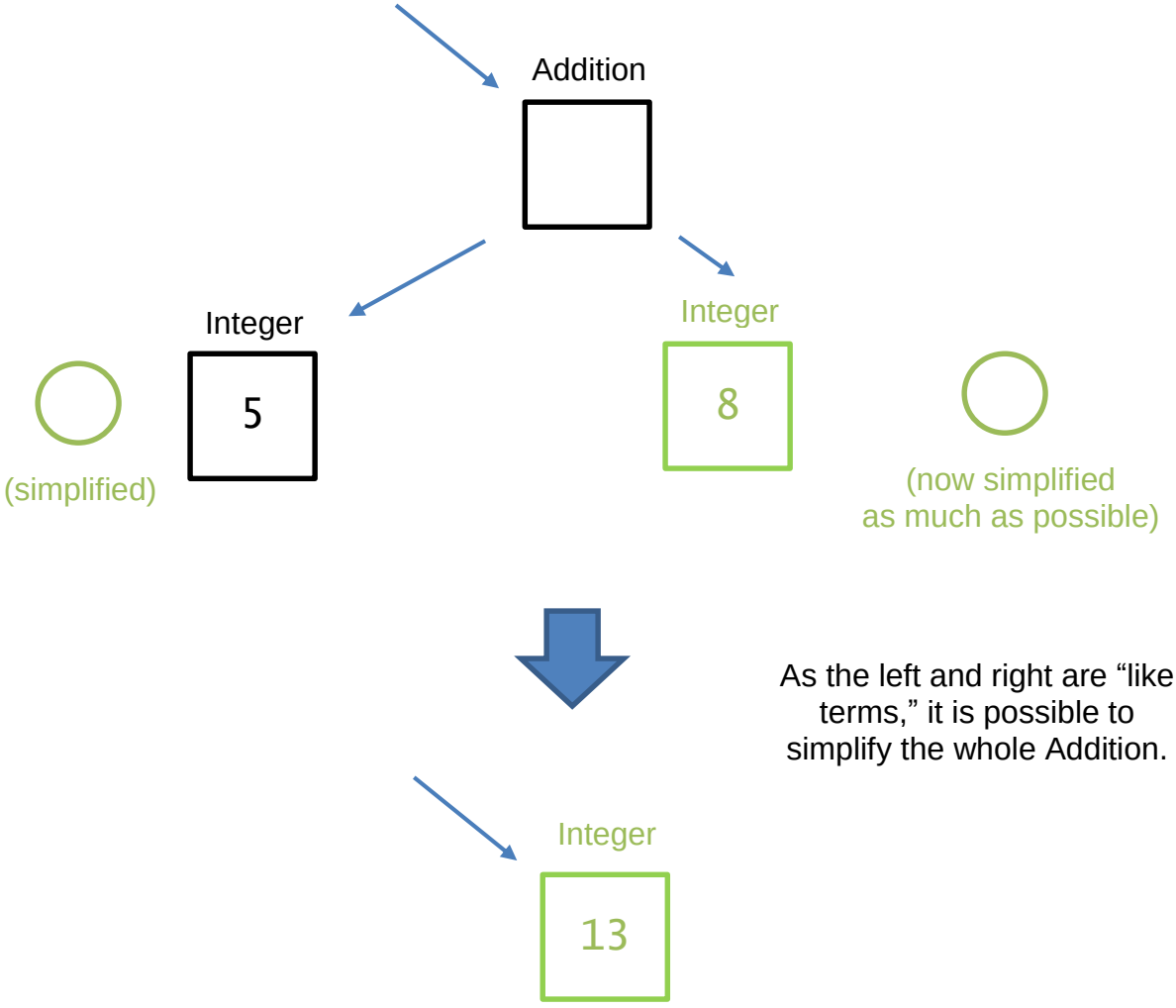


Both left and right are simplified, *and* we can simplify the combination further! Now to do the actual calculation…

Still Exponentiation.simplify()…

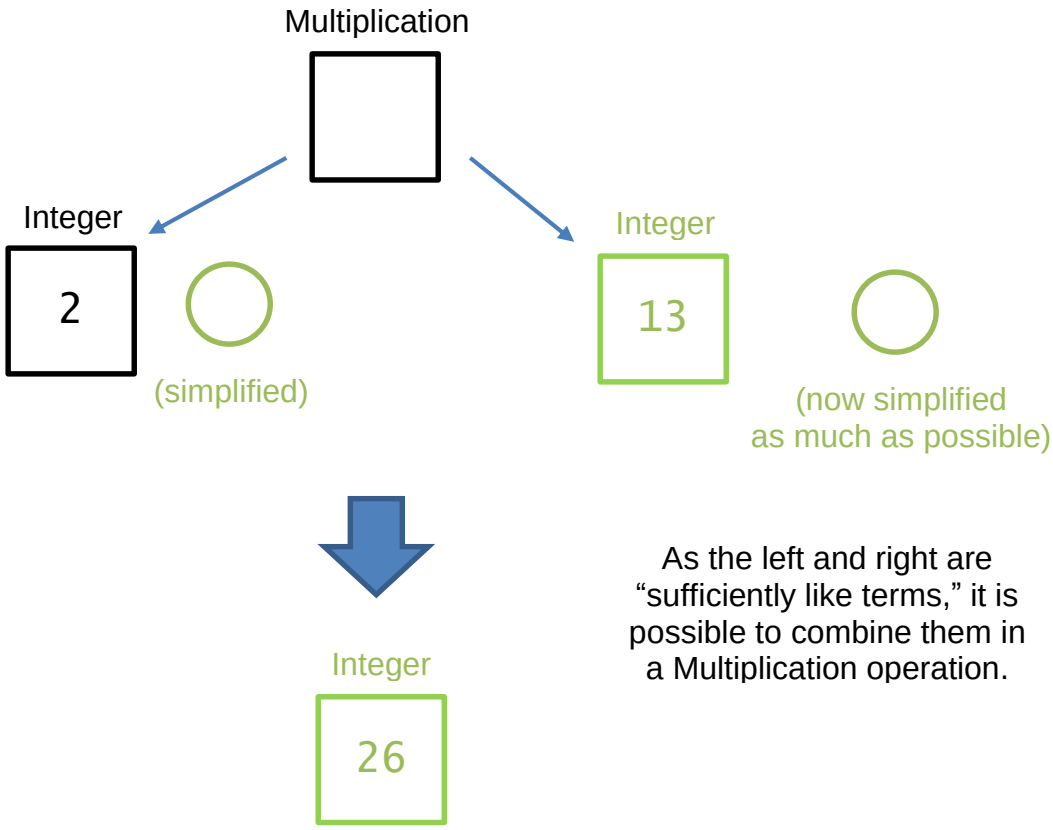
Given the simplified left and right sides, we note that we can actually evaluate this one and simplify the overall expression, **return**ing an Integer with value 8.

Back in Addition.simplify()…



As the left and right are “like terms,” it is possible to simplify the whole Addition.

As in, this Addition.simplify() **return**s an Integer with value 13.
Back in Multiplication.simplify()…



As the Multiplication was the top-level operation in the expression, this means that the overall expression simplifies to a single, simple Integer: 26.