CPS 112: Computational Thinking with
Algorithms and Data Structures
Homework 8 (40 points)

Handed out: April 11, 2018 (Wednesday)
Midway Checkpoint Due: 11:59pm April 18, 2018 (Wednesday)
Late submissions accepted with penalty until 11:59 pm April 20, 2018 (Friday)
Final Due Date: 11:59 pm April 25, 2018 (Wednesday)
NO LATE SUBMISSIONS

If you run into trouble or have questions, arrange to meet with me or a tutor!

- **If you run into trouble or have questions, arrange to meet with me or the tutor!**
-
- **Before you submit:**
- **Please review the homework guidelines available on Canvas**

- Read the assignment very carefully to ensure that you have followed all instructions and satisfied all requirements.

- Create a zip file named like this: *yourusernameHWX.zip* (with your username and replacing the *X* with the assignment number) that contains a directory named *yourusernameHWX*, which contains all of your .java files, your debug log, and your cover sheet.

- Make sure to compile and thoroughly test all of your programs before you submit your code. **Any file that does not compile will receive a 0!**

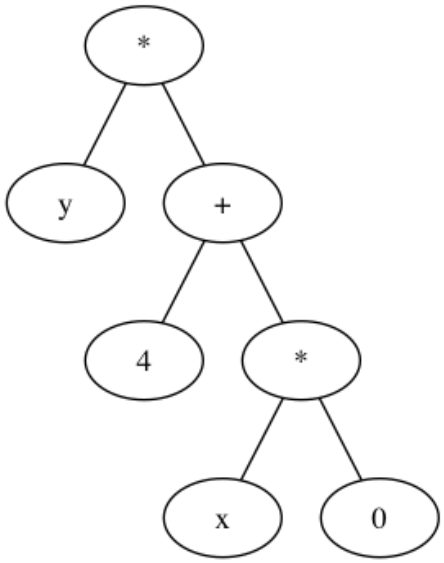
- Ensure that your code conforms to the style expectations set out in the homework guidelines.

    - Make sure your variable names are descriptive and follow standard capitalization conventions.

    - Put comments wherever necessary. Comments at the head of each file should include a description of the contents of the file **(no identifying information please!)**. Comments at the beginning of methods describe what the method does, what the parameters are, and what the return value is. Use comments elsewhere to help your reader follow the logical flow of your code.

    - *Program readability and elegance are as important as correctness.* After you've written your function, read and re-read it to eliminate any redundant/unused lines of code, and to make sure variable and function names are intuitive and relevant.

We discussed in class that binary trees are a natural representation of arithmetic expressions. For instance, the expression *y* * (4 + *x* * 0) can be represented by the following tree.

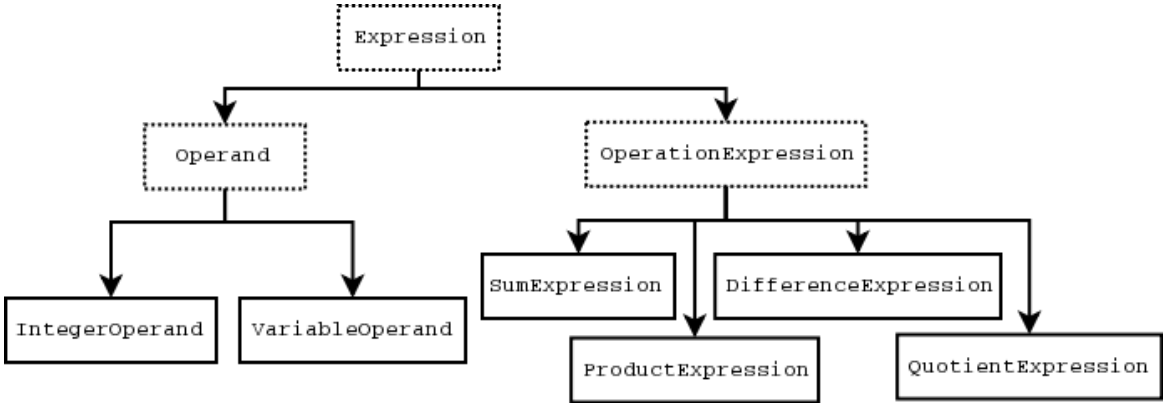


In this assignment you will use tree structures to represent, simplify, and evaluate arithmetic expressions. This project has a lot of moving/interacting parts. **You should read the entire assignment and develop a plan before you start writing code!**

NOTE: This is a two week project! To help you pace your work, there is a midway checkpoint. Your final submission should contain the *entire project*. The whole thing will be graded, including the problems that appear before the midway checkpoint. **The final score for Parts 1-4 will be the average of your score at the checkpoint and your score for the final submission.** This is to allow you the chance to encounter difficulties and recover or simply to get credit for improving on your initial solutions to these problems.

Part 0: Expression Trees

You will implement several classes representing different types of expression (i.e. different types of nodes). The class hierarchy is shown below. There are three abstract classes: `Expression` (superclass of them all), `Operand` (represents a number or a variable, a leaf of the tree), and `OperatorExpression` (represents an expression using an arithmetic operator, an internal node of the tree). Each `OperatorExpression` object has two `Expression` objects that represent its operands (its children). The rest of the classes are specific, concrete types of expression.


Expression
Operand
OperationExpression
IntegerOperand
VariableOperand
SumExpression
DifferenceExpression
ProductExpression
QuotientExpression

`Expression` has two static methods:

- `expressionFromPostfix` – converts a prefix expression to an `Expression`
- `expressionFromInfix` – converts an infix expression to an `Expression`

The following methods are abstract and must be implemented by subclasses (more details in the subsequent instructions):

- `toPrefix` – returns the expression in prefix notation
- `toInfix` – returns the expression in fully parenthesized infix notation
- `toPostfix` – returns the expression in postfix notation
- `evaluate` – takes a `HashMap` that maps variables to values and evaluates the expression
- `simplify` – returns a new, (possibly) simpler `Expression`
- `getVariables` – returns a `Set` of the variables contained in the expression
- `equals` – compares `Expressions`, taking commutativity of + and * into account

You will implement all of these classes in *Expression.java*. You can have more than one class in a file, but only one public class; in this case `Expression` is the public class. As you can see, the class files have been provided with stubs for the methods so you should be able to compile and test even if you haven't written everything. **You may not add any new classes or change the instance variables or public methods of the classes**. You are *encouraged* to add protected methods and move public methods in the class hierarchy as you see fit in order to reduce code duplication.

Wait! Do not start coding yet!

- There are a lot of classes and methods to write. You need to take them step by step and test as you go. Writing a bunch of code without testing is not going to end well! You might want to add a `main` function to `Expression` to contain your testing code and add to it as you progress.
- You will need to take advantage of the class hierarchy to prevent code duplication. Think carefully about what functionality is common between the concrete classes and use the superclasses to hold that common functionality. You may add protected helper methods to the classes to help you out with this.

Got it? Are you ready to proceed in a careful, deliberate manner?

For Midway Check: To simplify coding and testing a bit, you only need to handle integers, variables and addition (+) for all coding required for the Midway Check.

After the Midway Check you will implement subtraction(-), multiplication(*), and division(/) for **all** the code as well as implement some addition functionality.

Part 1 (3 pts): `expressionFromPostfix`

Implement the static method `expressionFromPostfix`. It will take a postfix expression represented as an array of "tokens", where each token is a `String` representing an integer, a variable, or an operator. The method should return an `Expression` object that is the root of a tree representing the given expression. So the list `["y", "4", "x", "0", "*", "+", "*"]` should produce the tree shown above, where the root is a `ProductExpression` with a `VariableOperand` and a `SumExpression` as its operands.

Big Hint: Go back to the algorithm for evaluating *a postfix expression. Building an expression tree is very similar. Instead of performing operations you should create tree nodes. Try it out by hand first! Remember too that you may not (indeed you don't need to) add any classes....So no you don't need a Node class*

A few notes:

- You should assume that variable names start with a letter.
- You need only support the operators +, –, *, and /.
- Integer operands might be negative...

You have been provided with a method, `drawExpression`. Just like in lab, this method takes a filename and creates a text file. That text file can be used to produce an image of the tree. For instance, if you have an `Expression e` and you call `e.drawExpression("expr.dot")` then it will create the file *expr.dot*. You can then, in the command line, run the command `dot -Tpdf -o expr.pdf expr.dot` to create the file *expr.pdf*, which should contain a picture of your tree.

You should make use of this capability to thoroughly test your method. Try it out on lots of different kinds of expressions and ensure that the trees it builds are correct.

Part 2 (3 pts): Traversals

Implement the `toPrefix`, `toInfix`, and `toPostfix` methods. These should be recursive as we have seen in class and lab, but note that you don't need an if-statement! You can implement the method differently in the different subclasses of `Expression`. Which classes represent the base case of the recursion?

Note that `toInfix` should return a *fully parenthesized* expression. So, for the tree above, it should return the string `"(y*(4+(x*0)))"`.

Test out a variety of expression trees. Make sure you try out positive and negative integers, variables, all the operations, big trees, small trees, etc.

Once you have your traversals working you can try out the program *Calculate.java*, which is provided. *Calculate* expects a postfix expression with spaces separating the tokens. For instance, to build the example tree above, you should enter the string `"y 4 x 0 * + *"` when prompted. Right now all it will do is print out the expression you give it in the various formats. As you implement more methods you will complete its functionality.

Part 3 (3 pts): `getVariables`

Implement the method `getVariables`. It should return a `Set` of `Strings`, the variables contained in the expression. A `Set` is a collection that contains only one of any given item. You can think of it like an associative map where the values are always the same as the keys. So for the tree above, calling `getVariables` on the root should return a set containing the strings `"x"` and `"y"`. Calling it on the root's left child should return a set containing only `"y"`.

Java provides multiple concrete implementations of the `Set` interface. Your method should create and return a `TreeSet`. Guess what data structure underlies a `TreeSet`? It's a binary search tree! The

methods of the `Set` interface you probably need the most are `add` and `addAll`. Look them up!

Now that you've implemented this, *Calculate* should prompt you for the values of the variables in the expression you give it. If it doesn't prompt you for all the variables in the expression, or if it prompts you for extra variables...something is wrong. It will then say that it is evaluating the expression, but it will just print out 0. That's because you haven't implemented `evaluate` yet!

Part 4 (3 pts): `evaluate`

Write the `evaluate` method. This method takes a `HashMap<String, Integer>` that associates variable names with integer values. It should return the evaluation of the expression when the variables are replaced with their values. Note that division here is integer division, so the result should always be an integer.

For the above tree, if the variables are assigned $x = 3$, $y = 4$, then it should evaluate to 16.

Once this method is implemented *Calculate* should print out the correct value of the expression, given the values you have assigned to the variables. Again, you should test this on a variety of trees.

Part 5 (4 pts): `equals`

Implement the `equals` method. It takes an `Object` and should return true if that `Object` is an `Expression` object that represents the same expression as `this`.

You should pattern `equals` on examples we've seen before. For instance, you could go back to the *Date.java* example from the beginning of the semester. That said, unlike that example this method will have to be recursive, since two expressions can only be equal if their operands are equal!

There's one more wrinkle. Your `equals` method should take basic commutativity into account. Specifically, it should say that $x + y$ is equal to $y + x$ and that $x * y$ is equal to $y * x$ (where x and y might represent any arithmetic expressions). You do not need to handle more complicated cases like noticing that $x + y + z$ is the same as $z + x + y$.

Part 6 (4 pts): `simplify`

The `simplify` method should return a *new* `Expression` that represents a (possibly) simplified version of `this`. Specifically, you should apply the following simplification rules:

- $x + 0 = x$, $0 + x = x$
- $x * 1 = x$, $1 * x = x$
- $x * 0 = 0$, $0 * x = 0$
- $0/x = 0$
- $x/1 = x$
- $x/x = 1$
- $x – 0 = x$
- $x – x = 0$
- Any `Expression` whose (simplified) operands do not have any variables should simplify to a single `IntegerOperand`.

You do not need to handle more complicated cases (like noticing that 3 + (*x* + 4) can simplify to *x* + 7).

Once this is working, you should see that *Calculate.java* prints the correct simplified expression. For instance, the example above should simplify down to `(y * 4)`.

---Midway Checkpoint (due 11:59pm April 18th)---

Turn in your work so far. You score on parts 1-6 will be averaged between your checkpoint submission and your final submission. If you are ahead of schedule, keep going! There's plenty more to do. If this first part of your project still needs work, don't panic! You still have time to recover. That said, you should come talk to me immediately so we can work together to get you back on track!

Part 7 (2 pts): Fix problems with parts 1-6
I understand that some of you will have more to fix than others. Points will be awarded here based on the comments given to you at the Midway Check and what you did (or didn't do) to address them. If you have very little to fix or only minor issues to address, that will be taken into account.

Part 8 (10 pts): Implement the methods in Parts 2-6 for the other operators (-, *, /).
Test, test, test (...Did I say test?) Make sure to test and code in small chunks.

Also, make sure `expressionFromPostfix` works with all the operators.

Part 9 (6 pts): `expressionFromInfix`

Implement the static method `expressionFromInfix`. It will take an infix expression represented as array of "tokens", where each token is a `String` representing an integer, a variable, an open or closed parenthesis, or an operator. The method should return an `Expression` object that is the root of a tree representing the given expression. You may **not** assume that the expression is fully parenthesized.

Important rule: You may not convert the expression to postfix notation as an intermediate step. You must convert directly from infix notation to an expression tree.

This is a bit tricky, but you can do it. Here's a Big Hint. You need to interleave aspects of the shunting yard algorithm for converting from infix to postfix with the algorithm you already have for building a tree. Try doing a conversion from infix to postfix by hand and try to identify at what point in that process you can safely create a node in the tree, and what node it should be. Here's an Even Bigger Hint: you'll need *two* stacks...

Once this works, you should be able to use the infix option in *Calculate.java*. It expects spaces between the tokens (even parentheses). So to build the tree above, enter `"y * ( 4 + x * 0 )"`.

Part 9 (2 pts): Code duplication

Hopefully you have been doing this all along, but look over your code for opportunities to make it more elegant. Take advantage of the class hierarchy to prevent code redundancy. Some hints:

- Look for common functionality that is repeated in multiple methods and/or classes.
- Put some implementation in the superclasses to be inherited by the subclasses.

- Create protected helper methods that capture common functionality to avoid repeating it.
- Make sure you keep testing as you make changes so you don't break anything!

Your maximum grade for this part will be determined by the number of *meaningful* lines in your code. Specifically, I will count the number of semi-colons. You can count them too using the following command in the shell: `grep -o ";" Expression.java | wc -l`. Your maximum score will be determined as follows:

- *> 299 semi-colons*: 0 pts
- *275-299 semi-colons*: 0.5 pts
- *250-274 semi-colons*: 1 pts
- *225-249 semi-colons*: 1.5 pts
- *< 225 semi-colons*: 2pts

Please keep the main goals of elegance and error prevention in mind. I reserve the right to give less than the maximum score if your attempts to make your code more compact make it difficult to read or if, despite having few semi-colons, you still have significant code redundancy.

---Final Submission (due 11:59pm April 25th)---

This is the end of the project! Your final submission of the *entire project* will be graded. Your score for parts 1-6 will be the average of your checkpoint submission and your final submission. Make sure you've tested everything thoroughly and fixed as many problems with your midway checkpoint submission as you can (feel free to come see me if you are stuck on something). Go back through your code to look for opportunities to improve readability or elegance.