← Project 5: Trees 🌴🌳🌲🎄

Due by midnight, Sunday 12/9 (or late on Monday)

The expressions return!!!

In this project, you will be dealing with a tree data structure that can represent **mathematical expressions.** Don't worry, there's no parsing this time, and the evaluation is far easier!

Starting point

Download and extract these materials. Contained are:

- `Expression.java`, **the file you will modify.**
- `ExpressionError.java`, just an exception type.
- `Driver.java`, which is used to test `Expression`.
  - I've given you a starting point, but **you should expand upon this and add more tests!**

To compile and run, do:

```
javac *.java
java Driver
```

You'll see that it prints out a bunch of 0s and `<not implemented>`s. You need to fix that :)

Expression trees

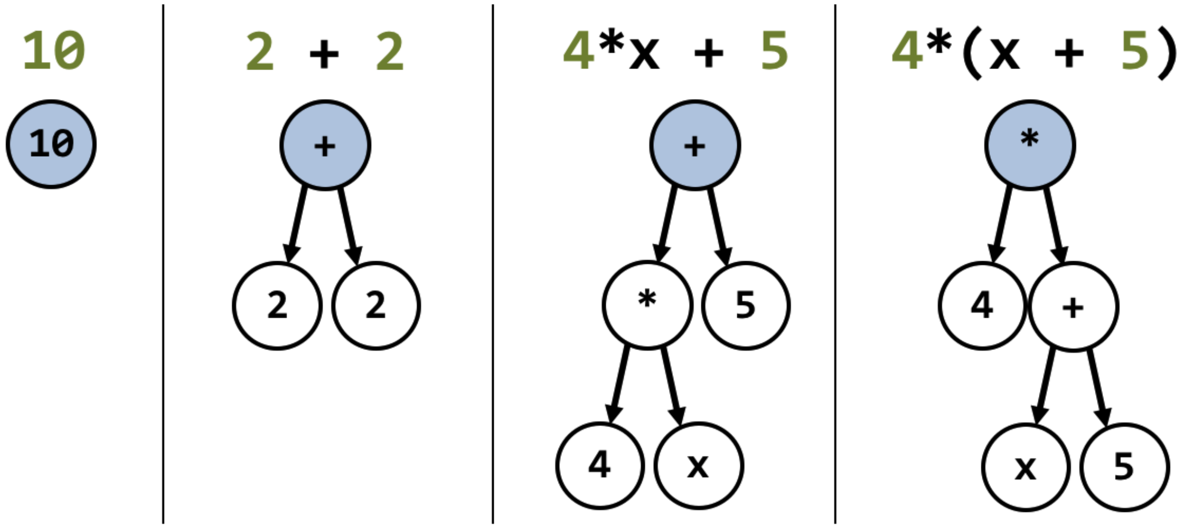
Like I showed in class, trees come up a lot in representing languages, such as math and programming languages.

The `Expression` class represents a **node** in an expression tree. Each instance of `Expression` can be one of three things:

- a **Number**
  - in which case its `_value` is a string representation of the value.
  - you can use `getNumberValue()` to easily convert that string to a `double`.
- a **Variable** name
  - in which case its `_value` is the variable's name.
- an **Operator**
  - in which case its `_value` is one of these operators: `"+", "-", "*", "/",` or `"^"`
  - also, `_left` **and** `_right` **are its children** - the operands to that operator.

There are no "bracket" nodes; order of operations is entirely determined by the tree's structure.

Here are some examples of mathematical expressions and the trees which represent them:



Compare the last two trees. You can think of parentheses as saying, “force this part of the expression to be a sub-tree.”

Your task

Open `Expression.java` and look through it. There are already some methods written, but there are several stubbed-out ones with `TODO` inside them.

Each method gives you practice writing very common kinds of tree algorithms: **visiting** all the nodes in a tree; **searching** through a tree; creating a **new tree** which is a modification of an old one; and **building a tree from scratch.**

> The next section documents some methods I wrote for you that will be helpful in writing these.

`String toString()`

- returns a human-readable **infix** string representation of this expression tree.
- **for numbers:** return the string representation of its value.
- **for variables:** return the variable name (the `_value` member).
- **for operators:** return a string of the form `"(left op right)"`, where:
    - `left` is the string representation of the `_left` member
    - `op` is this operator (the `_value` member)
    - `right` is the string representation of the `_right` member
- the result will have lots of parentheses. that’s correct :)

`String toPostfix()`

- returns a human-readable **postfix** string representation of this expression tree.
- this should be a very slight modification of `toString()`.
- *don’t forget to call `toPostfix()` recursively.*
- there should be no parentheses in the output.

`double evaluate(Map<String, Double> variables)`

- given a set of variables, evaluate the expression tree and return the result.
- **for numbers:** return the numerical value of the node (`getNumberValue()`).
- **for variables:**

    - check if the variable's name (`_value`) exists in the set of variables, using `variables.containsKey()`
    - if not, throw an `ExpressionError` with a descriptive error message.
    - if so, return the value from `variables.get()` .
    - Here is the documentation for `Map` . You can find the docs for `containsKey()` and `get()` there.
- **for operators:**
    - recursively evaluate the `_left` and `_right` children, using the same `variables` argument to them.
    - based on this operator (`_value`), perform the right calculation on those two values and return the result.

`Expression reciprocal()`

- returns a **completely new** expression tree that is the reciprocal of this one.
- you will not be making recursive calls to `reciprocal()` , but **you should use** `clone()` **where appropriate.**
- there are 3 cases:
    - **numbers:** return a **new** number node whose value is the reciprocal.
    - **division:** return a **new** division node whose children are cloned and swapped.
    - **everything else:** return a **new** division node whose children are 1 and a clone of this.

`Set<String> getVariables()`

- returns a set containing all the unique variable names which appear in the expression tree.
- the code I gave already creates the `Set<String>` for you.
    - it has an `.add()` method that you can use to add Strings to it.
- you will have to write a **private recursive method** to actually find the variables, and have this call that one.
    - you will pass that `variables` set as an argument to it.
    - think about how each kind of node will change the set (if at all).

`static Expression geometricMean(double[] numbers)`

> You may not use `quickParse()` to implement this method. Sorry ;)

- creates an `Expression` that represents the **geometric mean** of the array of numbers given as an argument.
- the resulting `Expression` should be of the form:
    - `(numbers[0] * numbers[1] * ... * numbers[n-1]) ^ (1 / n)`
    - where `n` is the length of the array.
    - (it's OK to assume that the array is always at least 1 item long.)
- use the `Number()` , `Operator()` , and `reciprocal()` methods to create the expression.
- making the chain of multiplications can be done iteratively or recursively.
    - it's a fun little puzzle :)

The methods I wrote for you

- `Number(double)`
    - makes a new `Expression` node containing a number.
    - e.g. `Expression e = Number(3.1415);`
- `Variable(String)`

    - makes a new `Expression` node containing a variable name.
    - e.g. `Expression e = Variable("num_people");`
- `Operator(Expression, String, Expression)`
    - makes a new `Expression` node containing an operator, and which points to two children.
    - e.g. `Expression e = Operator(Number(4), "/", Number(5));` for the expression `4 / 5`.
- `quickParse(String)`
    - parses a string into a tree of `Expression` nodes. supports `+-*/^` and regular parentheses `()`.
    - e.g. `Expression complex = Expression.quickParse("1 / (5*x^2 + 3*x - 9)");`

> `quickParse` has very little error checking and will likely crash or give weird results with erroneous input. But it's really there for testing purposes, so just give it valid expressions please :)

- `isOperator()`, `isNumber()`, `isVariable()`
    - return `boolean`s saying what type of node this is.
    - e.g. `if(expr.isOperator()) ...`
- `getNumberValue()`
    - for number nodes, parses the `_value` member into a `double`.
    - for operator and variable nodes, will probably crash. (that's why it's private.)
- `clone()`
    - makes a complete copy of an expression, recursively.
    - **have a look at how this method is implemented!**

Testing

`Driver.java` has a small amount of code in it to test your `Expression` methods. However it does pretty minimal testing. Like it tells you, **TEST MORE THOROUGHLY!!!** Use `Expression.quickParse()` to easily create test cases.

Here are the outputs I got from my implementation:

```
toString:         (((4.0 * x) + (y / 9.0)) + 12.0)
toPostfix:        4.0 x * y 9.0 / + 12.0 +
evaluate:         55.0
reciprocal:       (1.0 / (((4.0 * x) + (y / 9.0)) + 12.0))
reciprocal(num):  0.14285714285714285
reciprocal(div):  (10.0 / x)
getVariables:     [x, y]
geometricMean:    (((((4.0 * 9.0) * 3.0) * 7.0) * 6.0) ^ 0.2)
it evalutes to:   5.3868466094227525
```

Extra Credit [+10]

`String toNiceString()`

- Turns the expression into a nice string. ;)
- This is like `toString()` but it will **only put parentheses where needed.**
- Hints:
    - Don't forget to call `toNiceString()` recursively.
    - Decide whether to put parentheses around *each* of an operator's *children*.
    - Think about when you, as a human, need to put parentheses in an expression. What is the rule there? What does it have to do with?

Done correctly, if you just have `toString()` call this method, the relevant lines of the above output would now look like:

```
toString:        4.0 * x + y / 9.0 + 12.0
reciprocal:      1.0 / (4.0 * x + y / 9.0 + 12.0)
reciprocal(div): 10.0 / x
geometricMean:   (4.0 * 9.0 * 3.0 * 7.0 * 6.0) ^ 0.2
```

Grading Rubric

- **[5]:** Submission
  - Incorrectly submitted projects will lose all 5 points.
  - Please follow the submission directions carefully. There's no reason not to.
  - *It's 5 free points, people.*
- **[15]:** `toString()`
- **[10]:** `toPostfix()`
- **[25]:** `evaluate()`
- **[15]:** `reciprocal()`
- **[15]:** `getVariables()`
- **[15]:** `geometricMean()`
- **[+10]:** `toNiceString()`

Submission

You will submit a ZIP file named `username_proj5.zip` where `username` is your Pitt username.

Do not put a folder in the zip file, just the following file(s):

- All the `.java` files
  - Including any changes you made to `Driver.java`
- **If you did the extra credit, please also add a file named EC.txt**
  - It can be an empty file
  - It's just there to let the grader know you did it

> Do **not** submit any IDE project files.

Submit to the Box folder at this link. Drag your zip file into your browser to upload. **If you can see your file, you uploaded it correctly!**

You can also re-upload if you made a mistake and need to fix it.