Expression Trees - A3
Due: Nov 7, 2022 at 11:59PM

1 Expression Trees - A3 Specification

1.1 Assignment Instructions

- This is an *individual* assignment.
- Make sure to read through all of the specifications so your submission is complete.
- Follow all the submission steps in the Setup document by the assignment deadline.

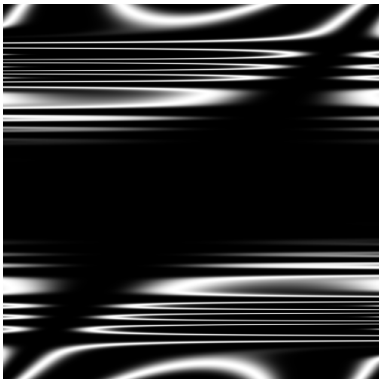
1.2 Assignment Link

The skeleton code for the assignment is available at `https://classroom.github.com/a/-RBEn4rz`.

1.3 Grammars

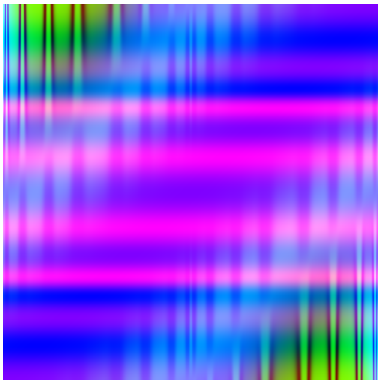
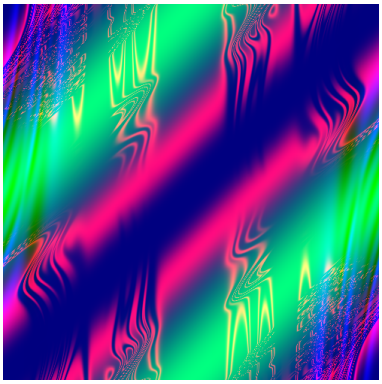
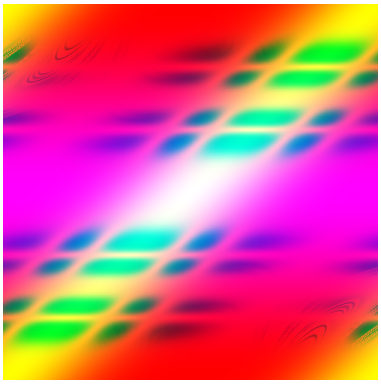
Formal grammars give a list of rules to define all possible phrases in a context-free language. In this assignment, you will be implementing a specific grammar in order to generate random phrases in a language. The language that we will define is part of mathematics: specifically, random mathematical expressions. Grammars can also be used to precisely define other languages, like programming languages. (A more complicated class of grammars, context-sensitive grammars, define natural human languages.)

While this is moderately interesting in its own right, we can carefully define our mathematical language to visualize the results of the expressions. For example, if we have a function $f(x, y)$ that we are sure will output values between specific bounds (say, between -1 and 1), we can apply the expression to the coordinates of every pixel in an image. We can scale the result to a grayscale value, to effectively plot this function on the image grid (with $f(x, y)$ visualized by the brightness of the color). With a sufficiently complicated function, we can produce interesting images such as the one below:



Generated with: cos(pi * cos(pi * sin(pi * y * sin(pi * cos(pi * sin(pi * cos(pi * cos(pi * y))) * cos(pi * cos(pi * cos(pi * y)))) * (x + cos(pi * cos(pi * cos(pi * x)))) / 2) * y)))

By applying different random expressions to each color channel (red, green, and blue), we can produce more interesting effects:

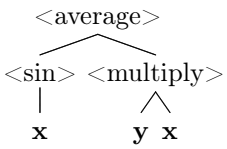
1.3.1 Expression Grammar

Your program will generate expressions with the following grammar, which is in roughly Backus-Naur Form:

```
<expression> ::= <multiply> | <average> | <sine> | <cosine> | <terminal>
<multiply> ::= <expression> * <expression>
<average> ::= ( <expression> + <expression> ) / 2
<sine> ::= sin(pi * <expression> )
<cosine> ::= cos(pi * <expression> )
<terminal> ::= x | y
```

In a grammar, all terms in < >'s are *non-terminals*. Each line has one non-terminal rule on it, defining what children a tree node of that type might have. The | character is an OR separator. For example, a `<terminal>` node could either have an `x` node as a child, or a `y` node.

We can construct a binary tree that is valid in the grammar by creating a root that is one of the specific `<expression>` types. We continue to construct the tree by applying a rule to any non-terminal that is currently a leaf node. For each non-terminal node, the grammar rule gives a clue as to how many branches that node should have. For example, `multiply` has two branches, and `x` has none.



This tree represents the expression $(sin(pi * x) + y * x)/2$. With a computer's aid, we can easily solve this equation for a given x, y input. Note that every node is a subtype of an expression node, and the leaf nodes are only terminal symbols. Consider other possible trees that could be constructed from the grammar. Do you see how to build a variety of longer expressions from the grammar?

Notice that for all of the above functions, if their domain is $[-1, 1]$, their range is also $[-1, 1]$. This is important. You will be asked to add one more function to this grammar – this function should also produce a value between -1 and 1 if the inputs are between -1 and 1.

2 Implementation

The goal of this assignment is to practice writing Java code that uses overloaded classes, and to practice constructing and traversing binary trees in a recursive manner.

You will be constructing and then evaluating trees similar to the one above.

2.1 New Classes

For this assignment, you will define a number of new Java classes: one for each non-terminal in the grammar. The **Expression** superclass has already been defined for you. Since an **Expression** in the grammar is always substituted by exactly one other grammar symbol, it makes sense to have this be an abstract class that other classes extend. The other classes should be concrete.

Each new class you make should extend the provided **Expression** class. Each class will need three methods: a constructor, **evaluate()** (required by the parent class), and a **toString()** method.

Also define a node for the `x` and `y` terminals. This could be a single class where each object is `x` XOR `y`, or can be two separate classes.

Consider what parameters your constructors will need to take: what variables are required to produce the non-terminal represented by your class? For example: what variables might we need to define a new **Multiply** node in a tree? How can it keep track of the local tree structure? Remember to store these parameters as fields in the class!

Your **toString()** method should return a string representation of the mathematical function. Note that you can use recursion to also include the string representation of any class variables – this involves recursion across multiple functions, rather than treating a single function as a recursive loop. What is the base case for this sequence of calls? What node types will *not* have a recursive call?

Your **evaluate()** method should work similarly, but instead of returning a `String`, you will be calculating a `double`. All of the functions are easily calculated directly, or are in the **Math** library. `Math.pi` is also conveniently in the library. Note that you can call evaluate methods in other classes to get doubles from them as well.

These methods should be short. There is not a lot of work to do in any given node. All of the behavior of tree construction and evaluation should come together with a small amount of code due to the recursive design of the project.

2.2 Recursion

If you want to calculate the results of your full expression (at the root of your tree), you will need to calculate the results of all the subexpressions first. This means that, from a given node, you should evaluate its branches first, and then its contribution to the expression.

If you want to build a string of an expression with the root at a node, the approach should be similar: convert any subtrees to strings, and then build a string from those, including your current node.

2.3 Testing

Incrementally test your classes in **main** methods by manually constructing small expression trees (without a lot of nesting), and then using **toString()** to test that you can print out the expression. You should see sensical mathematical functions as a result.

2.4 RandomExpression

You also need to complete the **RandomExpression** class to represent a randomly constructed expression. This expression will be made by randomly expanding a rule the grammar: for example, you may start with `<sine>`, then fill the expression inside that with `<cosine>`, then fill the expression of that with `<average>`, and then fill the two expressions of that, and so on. Eventually you will reach a `<terminal>` variable and your overall expression will be complete.

In addition to **evaluate()** and **toString()** methods, you'll need to complete **buildRandomExpression()** to randomly construct a new expression from the grammar using the process described above. This method will need to create a new random expression, and then recursively call itself in order to create random expressions to fill in the non-terminals when appropriate.

Each time you need a new node, randomly pick from among any of the grammar symbols. Although not

reflected in the grammar listing above, you can pick different expressions at a different frequency. For example, you might pick `<sine>` 50% of the time and `<cosine>` 30% of the time. Use **Math.random()** to generate numbers between 0-1, and assign different ranges to different function choices. Try to write your code such that the ranges are easy to modify!

I saw interesting results with a $< 20\%$ chance of a terminal.

The algorithm described above has the potential for infinite recursion. This is a desired behavior of a good grammar, but isn't useful when we want to generate finite trees. In order to avoid `StackOverflowError`s, keep track of your current depth in the tree, and add a terminal automatically if you hit the maximum. Note that the higher your maximum, the longer your program will take to produce a random expression.

Be sure to test your random expression generation by printing it out! You'll need to run the program multiple times to test the random outcomes.

2.5 New function

In addition to the classes listed in the grammar, you should create one more class representing another function that fits the grammar's requirements: the function has at least one parameter, and it maps values in $[-1, 1]$ back to $[-1, 1]$. Try some functions or composites of functions that make your art results more interesting. Update your **RandomExpression** frequencies to include this new function.

2.6 Results

Run the **RandomArtFrame** class to kick-start your random expression generator, and show you results. Not every random expression will give interesting results, but you can reroll the dice with the buttons supplied. Save three results that you think look interesting as `out1.png`, `out2.png`, and `out3.png`. Move them to the root of your **a3** directory as part of your submission.

3 Double Check:

- Have you implemented all the `TODO` methods?
- Have you created classes for all the math functions?
- Have you written a new math function to include in the expression grammar?
- Have you added/committed/pushed your code from **all** the files you created and modified?

4 Grading

- All classes are defined: +15%
- All classes have working toString methods: +15%
- All classes have working evaluate methods: +15%
- RandomExpression generates expression tree: +15%
- RandomExpression generates expression string: +10%
- RandomExpression evaluates expression: +15%
- New function: +5%
- Interesting images: +10%