Intro
My phase 1 solution is a top-down enumerative synthesizer. It uses a weighted breadth first search via a priority queue and custom cost function, and does minor pruning on infeasible partial programs.

General Design Choices
In general, every part of the program, be it the expand function, ASTNode deep-copy function, or otherwise, is designed with memory efficiency in mind. At a certain point, the priority queue becomes too large, with too many ASTNodes stored in memory. As such, the algorithm focuses on memory efficiency to postpone the event in which it runs out of memory. For example, ASTnode children lists are forced to be unmodifiable, since they never change, and ASTnode variables like its cost (see below), and whether it is complete, are accumulated rather than calculated.

Cost Function Design Choices
The weighted nature of the BFS algorithm works by selecting to test the "cheapest" synthesized programs first. To achieve this, the ASTNode class (ASTNode.java) was made to implement the "Comparable" interface. The ASTNode class has a private member variable called "cost". This value is a float, and represents an estimated cost of the program it represents. For efficiency's sake, ASTNode costs are calculated only once, upon creation of the object. Costs are a sum of the ASTNode's symbol cost, along with the cost of each of its children. Thus, costs are accumulated, and will never change, as ASTNodes do not change. The symbol class stores a static hashmap relating symbol names (strings) to their respective costs.

Variables, constants, and non-terminals have a cost of 0. All "functions" start with a cost of 1.0, and 0.1 is added to that value for every argument it takes greater than 1. So, for example, the "Not" function has a cost of 1.0, the "Add" and "Or" (among others) functions have a cost of 1.1, and the "Ite" function has a cost of 1.2.

The reason why it is set up this way, and more specifically, why non-terminals have a cost of 0, is because I wanted to prioritize simpler programs. Instead of giving a cost to non-terminals, I gave costs to the only things that could introduce non-terminals: the functions. By doing so, I prioritize the add and multiply operations over introducing boolean logic to the program, and when operating with boolean logic, I prioritize logical not. If instead I gave costs to non-terminals, then I would still prioritize these, just to a greater extent than necessary. In such a case, Ite(B, E, E) would have the same cost as Multiply(E, Multiply(E, E)). This essentially "devalues" conditional logic in the search, and makes it so that any solutions needing conditional logic are much harder to find than need be.

This is of course a tradeoff. It is entirely possible that devaluing conditional logic could be a good thing, depending on the types of

input-output examples the program is using. I made the executive decision that, without any further knowledge on the input-outputs received, that solutions involving conditional logic should be valued similarly to ones without.

A possible advanced solution to this dilemma could be if there is any way to calculate something like a “degree of branching” of the input-output examples. In other words, if a computer program could produce a statistically significant confidence value on whether or not the theoretical program defined by the input-output examples involved conditionals, then the synthesis algorithm could dynamically shift its cost function to better suit the program it is trying to reach.

Pruning

The synthesizer does minor pruning on infeasible partial programs. At the start of the synthesis algorithm, it calculated the maximum output value from the given examples, as well as the maximum x, y, and z variable values. It shares these values with ASTNode, which uses them to accumulate a maximum value for each ASTNode. If the maximum value for an ASTNode while being expanded is less than the maximum output value, then the branch is pruned.

The ASTNode maximum values are calculated by assuming the maximum possible value from all permutations. So for example, an ASTNode with the symbol “x” has a maximum value equal to the maximum value of x in the input examples. An ASTNode with the symbol “Multiply” and children “y” and “3” has a maximum value of the maximum y value, times 3. In the case that an ASTNode or any of its children are non-terminal symbols, the maximum value is set to the maximum output value – theoretically, an expression symbol could eventually be evaluated to an indefinite string of Multiply calls, and so no pruning is done until all non-terminal symbols are gone.

This pruning does help some, but ultimately, does not push the maximum program length that can be possibly solved by this synthesizer.

Issues

The main issue with the synthesizer is that it runs out of memory before it can generate sufficiently complex programs. Although it solves simple programs with 5 or 6 terminal symbols almost instantly, it can generally only create programs of up to 8 terminal symbols. In testing on my machine, it can, on rare occasions, create programs of up to 9 terminal symbols, but no further. A better cost function, and better pruning is needed to push the synthesizer further – or perhaps, a completely different method from the one used.

One small issue is that, because it searches for a program until it runs out of memory, it doesn’t print “null” after not finding a solution – it will simply print some kind of memory error message.