Kick Start 2021 - Round C

Analysis: Binary Operator

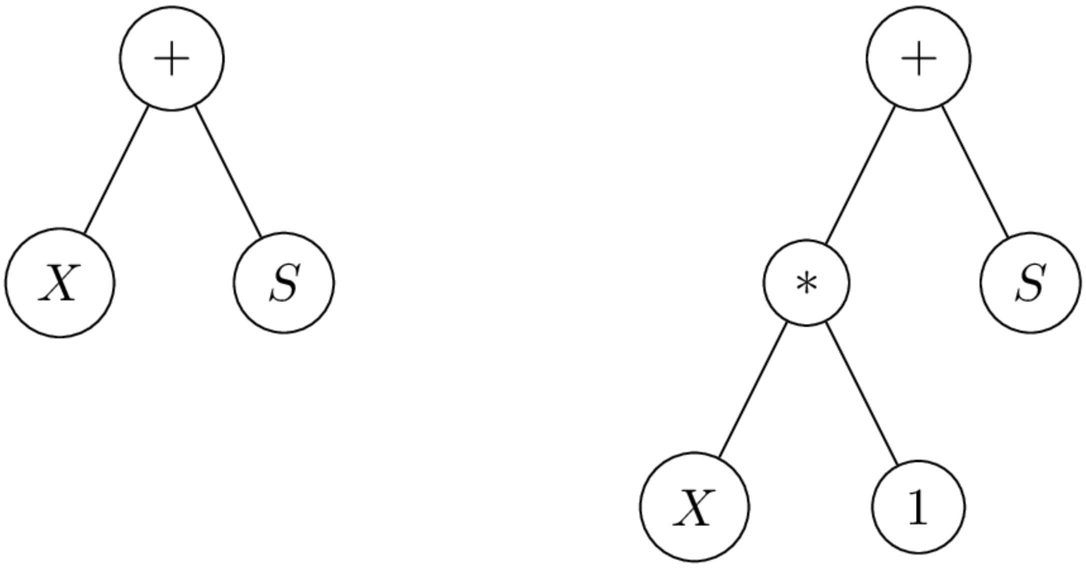
For each expression, we could construct an expression tree for it and consider it as an arthimetric tree, with, `+, *, #` as its operators. We can then start trying to reduce the whole tree so that we could compare different expressions more easily.

Test Set 1

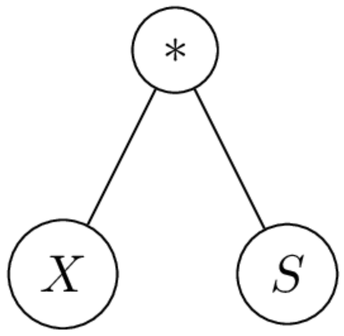
In the first test set, for each expression we could first examine if there is a `#` in it. If not, a simple traversal through the arthimetric tree would give us the value represented by the equation. Otherwise, we could start from the node where the operator is `#`. We can reduce this node's left and right subtrees into a single node each by traversing each subtree and calculating its value accordingly. This will result in a subtree with leaf values L and R, and root `#` (i.e. $L\#R$ in expression form). Now we have a tree where there are only leaf nodes under the single `#`, the next step will be trying to process node `#`'s ancestors, and ultimately we want to reduce the whole expression into a general form of $(A\#B)*C+D$ in order to do further categorization. For simplicity we would denote $L\#R$ as X from now on.

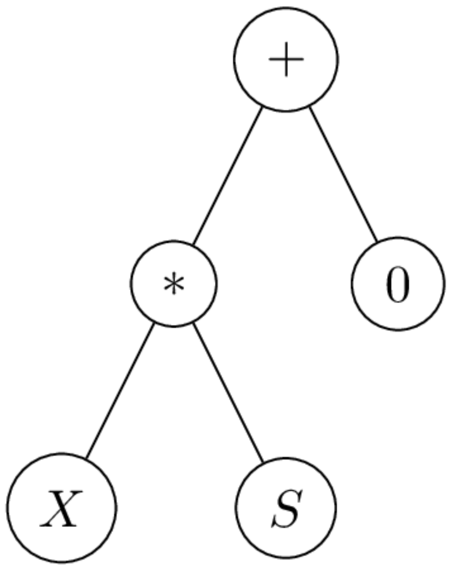
We would start try to transform the expression into our desired form from the subtree where X is the left leaf. Specifically we want to Consider X's sibling tree, since there is no `#` in that tree, we could use the same traversal method to get its value S. Now consider the parent node of X:

1. X's parent is a `+`, or X is the root node: we would reconstruct X into a subtree, where its root node is a $*$, and its left node being X, and right node being 1.



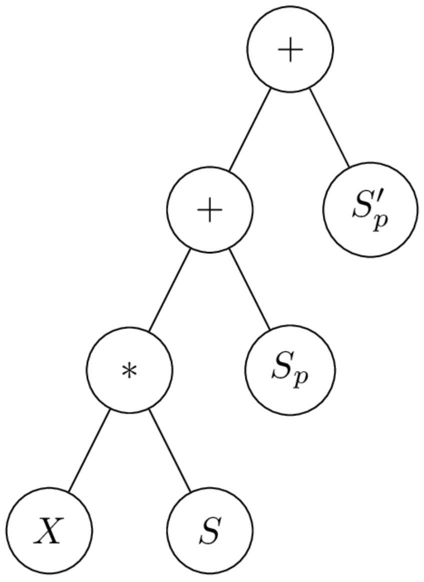
2. X's parent is a `*`: in this case, we added a $+$ node as a new parent, and the value 0 as the new right-sibling S_p.

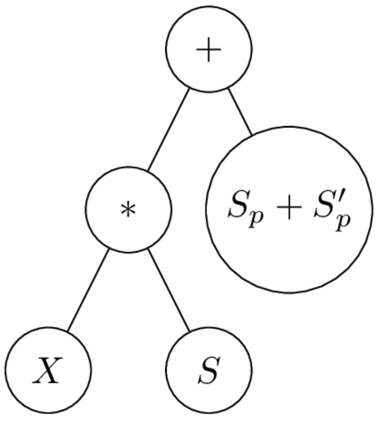




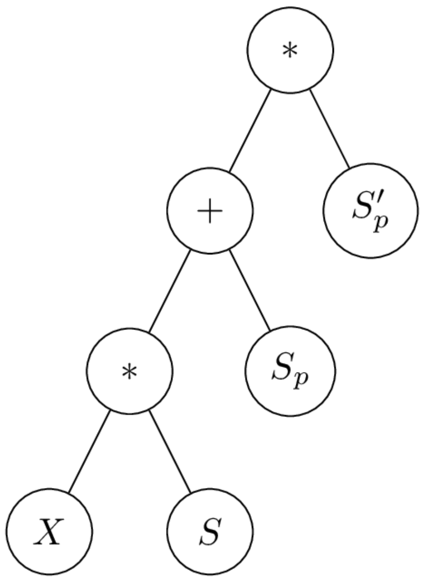
We then consider X's great-grandparent, and the corresponded reduced sibling S'_p:

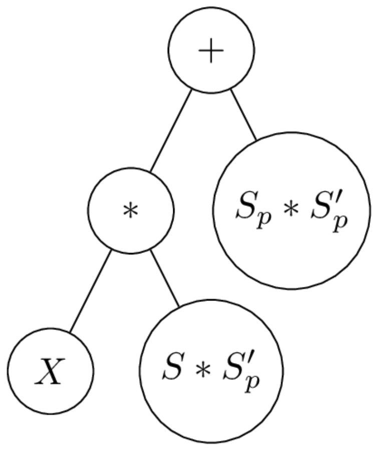
1. the node is a $+$, or there is no such node: we could reduce the height of the tree by replacing the tree with its left subtree, and substitute the value of its right subtree from S_p to $S_p + S'_p$





2. the node is a $*$: samely, we would reduce the subtree with its left subtree, but this time we substitute the value of its right subtree with $S_p * S'_p$, and also replace X's sibling with $S * S'_p$





Notice that, with this process, we could reduce the height of the tree by 1 each time, we could then perform this reduction process recursively until the expression is irreducible.

After the reduction process, the expression would be simplfied into the form of `(`A`#`B`)*`C`+`D, where A, B, C, D are all integers. Therefore the expression can be represented by a 4-tuple (A, B, C, D). We can then use a hashtable or any similar data structure to categorize them into different equilavence classes. Note that when $C = 0$, the only comparator matters here will be D only, since any value of A`#`B does not affect the value at all.

Test Set 2

For the second test set, we could follow the similar solution as mentioned above: reduce the expressions, then categorized into different classes. However, this method would become much more complicated since there might be more than one `#` in each expression. As a result, we cannot represent each expression as a 4-tuple, and the reduction rules are much more complicated. In fact, consider the following expression (not fully parenthesized for the sake of simplicity):

```
((0#1)+1)*((0#2)+1)*((0#3)+1)*...
```

If we utilize the same denotation in test set 1, naming each expression as $x_1, x_2, \ldots, x_n$, this will result in a polynomial with n variables, and therefore 2^n coeffcients to consider in terms of categorization. Luckily, since `((0#1)+1)` already takes 9 characters per block, and there is also need to spend additional characters on enclosing brackets and operators, the total number of coefficients for any given input string will be small enough to allow such deterministic solution to pass.

Alternative solution

There is also a simpler nondeterministic approach to this problem. Since the operator `#` represents a total function, for each calculation of some X`#`Y, we could randomly assign a value as the result, and memorize it for further calculation of the same X`#`Y. We could utilize it to calculate the exact value of an expression, then state that two expressions belong to the same equivalence class if their corresponding values are the same. This process will never put two expressions from the same class into different classes, but may result in a collision that mistakenly puts two expressions from different classes into the same class. Unfortunately, we do not have a simple proof that such solution has sufficiently small probability of failure. To further decrease the probability of false matches, the process can be repeated several times, putting two expressions into the same class only if their evaluations matched every time.