Data Structures

EN.605.202.81.SP24

Lab 2: Prefix to Postfix Using Recursion

Logan Griffin

3/26/2024

Analysis

Implementation and Design

The purpose of this project was to utilize recursion in order to convert a prefix expression into a postfix expression. The objective was the same as Lab 1, except for the implementation requirement to use recursion.

The program works by taking the file paths for an input and output file. It runs by evaluating each character in the file, one by one, building a tree structure from the input data. The method used to build the tree is recursive. The characters are taken and evaluated, calling the method recursively until either an invalid character or a letter (operand) is found. From there, the program returns and restarts the search for one of the base cases. Creating a tree structure this way allows the expression to be rearranged very easily. Each parent node will contain an operator and have two child nodes, one left, one right, that contain the operands for the operator. Once gathered, a single subtree consisting of one operator and two operands will automatically become a single operand for the operator that is serving as the parent to the subtree operator. The operands and operators are combined until there is a single operator in the root node, with two children made up of the rest of the characters.

Given that making the postfix expression is completed by combining the items of subtrees in a specific order, it was easy to take the tree and output the infix expression as well. Instead of gathering the operands and then the operator for a term like AB+, the operator can simply be gathered in between the operands to achieve a term like A+B.

There are numerous checks throughout the program that will flag an invalid expression. Any invalid expression that is found will result in the appropriate message being written to the output file. If an expression makes it all the way through, the prefix version will be printed to mimic the input file, followed by the infix and postfix variations.

Iterative vs. Recursive

As previously mentioned, both Lab 1 and 2 achieved the same results. Lab 1 followed an iterative approach, while Lab 2 followed a recursive approach. Both strategies were viable to convert an expression, and there were advantages and disadvantages to each. As someone with very little exposure to recursion, the logic was slightly difficult to follow. Further, debugging provided more challenges. I wanted to set and use variables as I have done in the past with iterative methods, but that does not work with recursion. These factors lead me to believe that my iterative solution is marginally better than my recursive solution. That being said, I also believe that more practice with recursion would yield an overall better solution.

Recursion works well for converting between statements because trees, a naturally recursive data structure, can be implemented and used to output any other form of expression. Converting between expressions presents an inherent base case represented by encountering a letter serving as an operand. Once a tree is made, it can be traversed in numerous ways to achieve specific results. This allowed for simple enhancement. On the other hand, for an iterative approach to convert between all the types of expressions, long, complete methods would have to be written for each, requiring substantially more work.

Enhancements

In terms of enhancements, the program is able to convert the prefix expression to both an infix expression, and a postfix expression. Traversing the tree data structure differently, but still recursively, allowed each form to be generated with very few lines of code. The infix statement was output by traversing through the tree going left each time until a leaf was found. The left leaf was printed, then the parent node was printed before the traversing of the right side began. Similarly, the postfix expression was generated by going left, printing the left leaf node, then going right and printing the right leaf node, before finally printing the parent node, which was the operator.

An additional enhancement could be to set up the program so that it could take in any type of expression, identify what the type is, build a tree from the expression, and output the other two expression types. This would require checking for an identifier, such as an operator as the first character representing a prefix statement, an operator as the last character representing a postfix statement, or a letter as both the first and last characters, representing an infix statement. The recursive method to build out the expression tree would need two variations, with the ordering of the base cases and recursive calls varied.

Error Handling

There are numerous exceptions that can be encountered while running the program. First, the program checks for exactly two command line arguments, assuming the first is the input file and the second is the output file. The next check is to make sure both files can be opened. After that, the program will try to read a character.

Once it's been determined that the input file actually contains data, the program can move on to validating the expression on the first line. Any invalid character that is encountered will result in a node in the tree that has the string "Invalid" as its data. Even when an invalid character is found, the program will still build out the rest of the expression in the tree structure. This slightly decreases the program's efficiency, but, on average, there shouldn't be many cases where the expression is many characters long with an invalid character near the beginning. Immediately exiting the method building the tree would be the most efficient way to handle an invalid character. After the tree is built, the program recursively checks for an "Invalid" node, and, if one or more is found, the program will determine that the overall expression on the line is invalid. Additionally, there is a recursive call that counts the number of nodes and leaves in the tree. A regular node will contain a reference to either a left child or right child, while a leaf will not have any references to any other nodes. For an expression to be valid, there must be one more leaf than regular node in the entire tree. Lastly, the program will check for any additional characters on the line after a valid tree has been built. Characters other than the carriage return or new line following a successful tree will result in an invalid statement.

What I Learned

By working through this lab, I gained valuable experience working with recursion. Still being relatively new to programming, I find it difficult to have the discipline to work out the code on paper before jumping right into writing it. The recursive method calls forced me to go back to working with paper, as it was much easier to see what was happening by tracking the method calls and returns.

At first, I tried to implement a solution that would write the postfix equivalent of a subtree immediately after the second operand was identified. I found it difficult to achieve this recursively, and, after many failures, decided it would be more efficient to build the entire tree out first.

I learned that sequence has a major impact on how a recursive function returns. When a base case is reached, the method returns to where it was called, which happens to be somewhere within itself. Each call returning to a specific location seemed to complicate the sequence logic. Therefore, it was crucial to carefully examine where each check/operation was placed in the recursive methods.

Description of Data Structures

Trees are an extremely useful data structure, often chosen for applications involving locating specific data. They are hierarchical, which helps maintain an organized structure. Traversing through a tree to visit all the nodes can be done with simple recursion calls.

Terms used to describe trees include parent, child, node, root, leaf, level, subtree, etc. A parent node shares a link to a child node. Depending on the tree, a parent node can have any number of child nodes. The root node is the node at the lowest level. When looking at a diagram, it is the node at the top of the tree. All other nodes in the tree can be accessed through the root node. A subtree is any combination of parent/child nodes that are part of a larger tree. A leaf is a specific type of node that has no references to any child nodes. It can be thought of as the end of a branch.

Efficiency

The efficiency of the program can vary depending on the structure of the input file. A valid statement will result in a time complexity of O(N), since each character will be evaluated twice during the initial tree build, once for valid checks, and twice for each of the three methods used to output the statement. An invalid statement would result in a smaller time complexity since it would not be printed, but the overall value would remain O(N).

If the total number of characters in the file was N, a file of N lines of 1 character each would be less complex, in both space and time, than a single line of N characters. Each recursive call has to be revisited, resulting in a higher number of overall operations. A massive tree also takes much more space to store compared to single nodes made and removed N times.