EN.605.202.81 Section 84

Alexander Shah

Lab 1: Prefix to Postfix with Stacks Analysis

September 26, 2023

Project 1 Analysis

This program receives prefix expressions and uses an implementation of a stack data structure to parse and convert to a postfix expression. The program effectively manages error handling, addressing issues such as reading files, processing incorrect lines, and managing invalid expressions that cannot be successfully converted. The stack, implemented as MyStack, is crucial to the process where we store and retrieve elements of the expression.

Justification for Design Decisions

The stack is implemented as MyStack which has the methods pop and push to add or remove elements from the top of the stack. A stack is useful in this application as it allows us to parse the expression and build an output expression by contextually combining elements of the expression and putting them on or off the stack. A LIFO data structure is ideal for this application. This allows an iterative solution to process the input string from right to left, popping the elements off and adding back on expressions that fit the postfix scheme.

Recursion

Compared with a recursive approach, the iterative solution is potentially more efficient in space complexity since the recursive method has memory overhead for repeated calls, and is easier to control for debugging and ensuring that edge cases are covered. In addition, the iterative solution will work the same way regardless of the size of the input compared to a recursive solution which may need larger and larger depth to its recursive calls. This project showed how the stack data structure is an appropriate choice for the implementation of this iterative solution. A stack is well suited to the operations that the problem calls for, and performs each operation during the run time with good efficiency.

Efficiency

When assessing time and space complexity, the iterative approach exhibits O(n) time complexity. This efficiency is achieved through the use of O(1) stack operations, resulting in O(n) space complexity to store the stack's elements. A recursive implementation may depend on how deep the recursion needs to accommodate the operands and operators in the stack, but in the worst case the recursive implementation runs averagely in O(n) time and O(n) space.

What I Learned/What I Would Do Differently

I might have chosen to handle errors more effectively, such as showing the steps taken when parsing an expression, such as showing the elements moving on and off the stack and where the program runs into an invalid expression or when the program discards an input. In addition, using more output during debugging could be beneficial to keep track of the program logic to find more instances where edge cases could affect the output of the program. For example, the program does not handle parenthesis, comma separated values, or combining integers, and works by discarding white space which could be used significantly.

Specifically on debugging, I ran into an issue where the arrangement of elements in the stack were correct up until a point, then the ordering of the final arrangement of elements on the stack failed to achieve the correct output, even after making sure the program logic was correct. For example an expression like "-+ABC" resulted in "-AB+C", when it should have been "AB+C-". Checking the last state of the stack, the stack contained [-, AB+, C]. It occurred to me that the last step of recombining the individual subexpressions was happening in the wrong order. By using debugging statements I got a better view of how the stack changed and I was able to figure out the solution. By taking the ending stack, and unpacking the elements in reverse order while joining them to an output string, I was able to recombine them in the correct output order. Debugging in this way helped me to more deeply understand the steps that were occurring and where they were going wrong.