EN.605.202.81 Section 84

Alexander Shah

Lab 2: Prefix to Postfix with Recursion Analysis

October 22, 2023

Lab 2 Analysis

This program takes prefix expressions from an input file, converts them to postfix format using a recursive algorithm, and writes the results to an output file. The program effectively manages error handling, reading files, processing incorrect lines, and managing invalid expressions. The program features more comprehensive error handling compared to Lab 1 including cases where there are too few operands or too few operators and reports the errors both to the console and the output file.

Justification for Design Decisions

A majority of Lab 1 was able to be used for Lab 2, the major difference was of course that we did not make use of the MyStack class and the stack based conversion was done with recursion instead, as necessitated by the Lab guidelines. File processing had to be modified to handle the recursive algorithm returning a tuple, keeping track of the expression and end index, in order to correct a deficiency in Lab 1 that did not detect some invalid expressions. Specifically, if the recursive algorithm returns an expression that is shorter than the input expression, it raises a ValueError. This resolves the case where a valid sub-expression could be formed but the algorithm stops considering further characters, it did not check that all characters were used.

Because expressions can be broken down into sub-expressions, using recursion is a natural fit to converting expression types. Recursion works by dividing and conquering problems into sub problems and we can divide up any expression into an operator and two operands, and combine 2 expressions together with another operator, the same general type of sub units. There is a hierarchy established by operators and operands you can visualize as a tree structure which helped me to change from an iterative approach to recursive.

Efficiency

As seen in Lab 1, the efficiency of both stack and recursive implementations are similar. Time and space complexity should be O(n) for each, however the recursive implementation can grow in recursive depth depending on the input. Recursive depth may come at an overhead cost. Comparatively the stack will grow in size based on the input which does not change how it operates or in terms of overhead. Each access and retrieval (pop, push) on the stack is O(1). Recursion also uses the computer's stack compared the stack implemented in Lab 1 would use the heap, the stack and heap have different rules and uses. The stack may be faster than the heap, but a thread can reserve space on the stack in LIFO order and is freed when the thread returns and the heap is more dynamically allocated as it can be allocated or freed any time. This might make the iterative Lab 1 implementation faster than the repeated function calls due to recursion in Lab 2.

What I Learned/What I Would Do Differently

Initially I did not keep track of the position within the expression the recursive call considered and used additional storage to hold sub expressions, it was much harder to figure out what wasn't working. It also seemed likely that holding sub-expressions in another data structure would be impossible to implement without accidentally recreating a small stack structure so I abandoned it. But by tracking one more parameter, the index, I was able to implement the recursive case with minimal overhead outside of what is introduced by recursion. It was also much harder to figure out problems with the recursive implementation compared to the stack based implementation due to the non-intuitive nature of parsing the expression into a tree and operating that way. Debugging the call stack was challenging, such as trying to find enough information about where problems were occurring and what had happened up to that point. Overall the implementation turned out to be easier than I expected but

getting there felt harder. Maybe a hybrid implementation could end up addressing the overhead and complexity of working with recursion while maintaining the potential speed benefits of the stack structure.