Andrew Taylor
atayl136

Data Structures Lab 2 Analysis – Prefix to Postfix Conversion via Recursion

Introduction

The goal of this lab project builds on the last lab, to build a prefix to postfix converter, but with a recursive solution instead of a stack. Because of the overlap, I was able to re-use portions of the code from Lab 1 since I had done those parts correctly. I felt there was no need to reinvent the wheel, and set out to swap the core of the program, the stack solution, for a recursive one.

Once again I am programming in Python, and for the input there is the same "Required Input.txt" file and required test cases to process. I decided to go with the same test cases as a way to contrast the output of the two projects. The test cases I chose originally spanned the gamut of possibilities as far as input goes, so there was really no need to come up with redundant cases. I incorporated the feedback I received from Lab 1, and made sure there was complete error checking in this version.

The implementation is a lot simpler in this version, but the concepts were more difficult to grasp and I had to start over several times. In the end I came away with a better understanding of recursion in general and specific lessons about designing and building recursive code.

The Recursive Data Structure

The combination of string data and a recursive function makes a recursive data type. In the code we have two functions. An outer function and an inner function so we have two functions to the ADT. The data is simply taken as input and stored in temporary variables. The recursive function has a stopping case, a base case, and two recursive calls.

Recursion Versus Iterative Solution

Recursion was used to process and convert the strings to postfix as required, by there are iterative elements of the program where that makes more sense: file I/O and process files. The recursion itself simplifies the code by at least 70% roughly, and doesn't require any constructed datatype it just uses strings. Doing this project changed my mind about recursion: it can be the simpler, more elegant solution. While it may be easier to think in terms of for loops. It can pay to think and program recursively especially when the recurrence relation is the focus of the process. In this case the recurrence relation is rather weak, so the efficiency trade off for using recursion to solve this problem may not be worth it. Nonetheless, recursion here makes sense because of the size and number of the inputs and improvement in the style of code.

Recursion can come with a cost of a memory burden because each step builds upon the previous step and is stored in memory separately. For a problem such as this, where the desired output is only kept at the stopping case, I think recursion is inefficient. I find recursion to be a much better solution when every iteration provides an operation or piece of data desired on its own. In the case where 1000's of

records are being altered, or the problem exhibits a natural mathematical recurrence relation like the Fibonacci sequence, a recursive solution would be a stronger choice.

Design

I tried a couple list based designs before coming to the solution I chose. In all designs, the common element is an index I had added to mark the recursions and provide a track on the progress of the converter through the input, as well as define the stopping case for the recursion. In my early attempts, I thought about a list of lists, reordered by the recursive case, but this was unwieldy, and involved associating an index of the list with an index of the character and corresponding relationships among other characters processed. It was not the way to go. Then I tried an approach involving two lists, building out the first list with concatenations of the second list plus an operator, which gave me the basis for the recursive case I developed with strings. The code changed a lot as I tried out ideas.

I took a similar approach to the stopping cases, I worked backwards from what I needed the function to output. One stopping case was clear, when the recursion on the string was finished, but it became evident later that a second stopping case was needed to assign temporary variables to concatenate with. It took me a second, but I saw that there was no need for two rules of concatenation, so instead I made a second stopping case for the operands and a second recursive call was created, one for each list. Then I realized I could build out the string directly without need for joining the list at the end.

What needed to happen when, naturally ordered the structure of the recursion, and I looked at the algorithm from Lab 1 to keep a check that I was building out the algorithm correctly. In the end I added the error checking, which was easy because the types of input were already separated by the stopping case and the recursive cases. I tried out several inputs, compared the result to my outputs from Lab 1, and satisfied with the results, I had built a recursive solution by going from the output, to algorithm, to code.

The remainder of the project could be preserved: the code to take in the data, process the data, and output the data, then provide statistics. I decided to segment my program in the same way so I ended up with 4 files: a main driver (which was also responsible for file I/O), a converter file, a process file, and runtime statistics. The other files required very little modification because they had been implemented correctly the first time around.

The main file is where the program first begins so it is where I decided to read the input. I parsed and processed the input as dictated by the Programming Assignment Guidelines. After that it is built into strings character by character and then into a list of input strings. The process file is where I connected this input to the conversion, and the conversion to the output. The conversion file converts recursively. The main driver imports the runtime tools. Finally, the main driver organizes the above into a main() function and runs it, once, generating a report in the console and a log written to file. Generally I tried to keep it simple and yet efficient. Looking at the assignment instructions, the focus was on the recursion.

Efficiency

Time complexity in recursion depends on the number of recursive calls before reaching the base case. Because in the case of my solution each function calls itself twice unless it has been recursed an

amount of times equal to the length of the string (index), if we call this number n this gives us O(2^n). So, a great drawback of this approach is we are looking at an exponential time complexity whereas the stack was constant time.

Error Checking

During Lab 1 it occurred to me that the unavoidable errors anticipated in the application were of a data input in nature, so for that and the following reasons I decided to purely implement print statements warning the user rather than any try…except statements and exceptions:

1) Python does not allow overriding built-in exceptions at the module scope level. They are only triggered when loaded. That is, functions that call non-existing stacks will have Python's error not mine.

2) Exceptions are meant to halt a program, and I wanted to process as much of the file as possible for the user. It is possible to continue execution but I don't think that is in fact what was in mind with their creation, thus the traceback. With my selected approach the user is informed of the error but receives maximum output.

3) Print statements give just as much information when done correctly, because the point is the program functions save some problem that must be brought to the users attention.

4) Exceptions to me are more fitting to problems with the program features, requiring the attention of the programmer, than formatting details about the input for the end user. When was the last time you encountered an exception as an end user? This is a really bad sign when this happens, it means the program doesn't even work.

The errors of interest were common input errors: invalid characters, transposed characters, missing characters, things of that nature. This basically boiled down three scenarios, and a helpful print statement to flag when a space in the input was ignored. I tried to build in a print statement (and thus one in the log) that would catch each of these scenarios, and account for any key or key combination the user could press on the keyboard. All things being equal it seemed like a simple question, either the character fit the required format or it didn't. Either there was the right number of operands and operators in the expected format or there wasn't. I couldn't think of many scenarios beyond that to test. Extreme cases were included in my test cases as required. There could be infinite ways for a user to "fix" an invalid input, so the best I could do is give them a general idea of why it didn't qualify. The errors I warned to the user included:

1) Invalid character (not a letter or operator)
2) Error from not enough operands
3) Error from not enough operators
4) Extra Space in input string ignored

Enhancements

In the first lab I took some inspiration from the sample project and built a module for runtime statistics. This simple module helped track the time the program takes, and the size of the files. I also

started keeping track of the errors. The resulting report when printed to console and logged to file includes:

1) The date and time the converter was started
2) How many nanoseconds the program took to run
3) The total lines of input
4) The number of bad characters there were
5) The size of the input after it was read
6) The total lines of output, to compare
7) The lines converted to postfix without error
8) The size of the output before it was written to file

I kept this work active for Lab 2. Thinking of a small enhancement to add this time around, I decided to display a graphic which explains recursion. It basically sums up my experience with this module. You'll have to run the program to see it!

Lessons Learned

Working through this lab I got a better understanding of recursion and how to program it. Some of my lessons learned were:

1) Multiple Base Cases can be Used
   When I came to the realization that the alpha characters needed to be provided via a base case, a lightbulb went on. It was then when I realized that a base case and a stopping case are not the same thing. This has opened up new doors for me in thinking about recursion.

2) Recursive Code is More Elegant
   The same conversion that took me a page of code last time only took up 1/3 of a page when programmed recursively. This can be a major advantage if recursion can be applied multiple times in a project, and clear documentation would make that advantage even stronger.

3) Multiple Recursive Calls is a Great Tool
   When handling a problem in sub-problems, I will now think in terms of multiple recursion because different sub-problems may lend themselves to different calls.

Starting over, I would try to see if I could approach things from the other angle. Do recursive calls follow a structure that can be memorized? Is there a top-down strategy? 1 or 2 base cases and 1 or more recursions gives a lot to choose from, I wonder if the choices there line up with different types of problems. I think my chosen solution was very clear and Pythonic, so I wouldn't do anything different stylistically. In the end I'm very satisfied with my solution. All in all this was a very fun and educational project. Thank you!