Kordel France
Lab 1
Class 605.202 Section 81

Lab 1 Analysis Document

Prefix and postfix expressions are mathematical functions commonly used by calculators to perform calculation. A calculator does not compute the product of the integers 3 and 4 by processing "3 * 4" like a human does. Instead, it reads the expression in order of operations with the operands either preceding the numbers (* 3 4) or succeeding the numbers (3 4 *). *FranceLab1* is a software module written in Python that converts given prefix expressions into their corresponding postfix expressions.

Description of Chosen Data Structures

The data structures used for this program were, primarily, stacks. Arrays were used sparingly but only for operations having nothing to do with the expression evaluation, such as printing expressions to the console or storing command-line arguments. Since the evaluation and conversion of prefix expressions requires characters to be read and compared one at a time, a character stack (or a stack of single-character strings since Python does not support the 'char' type) was used such that each element of the character stack contained only a single character. The stack that stores individual characters is called a *CharStack* and the stack that stores entire operational sequences (3*4/7*9) is called a *StackStack* because it is a stack of *CharStacks*, each *CharStack* containing a group of operations that have been processed from prefix to postfix form.

Justification for Design Decisions and ADT

Other ADTs would have worked as well, but the method of evaluation doesn't so naturally point to their use. For example, a doubly linked list could have been used in replace of the stack ADT, but many more lines of code would have to be written to accomplish the same task. It is because the evaluation of these expressions conveniently work with a LIFO system that the process of prefix-to-postfix conversion naturally points to using a stack ADT. Values (3, 4) as well as entire operations (3*4) are "pushed" and "popped" into an operational stack for a computational sequence.

Complexity

Please refer to *Table 1* for a complete analysis of time and space complexities for every function in the *FranceLab1* module.

FranceLab1 Complexity Analysis				
			Worst Case	
#	File	Function	Time Complexity	Space Complexity
1	__init__.py	*[file]*	N/A	N/A
2	__main__.py	*[file]*	40	159
3	constants.py	*[file]*	2	62
4	conversion.py	*prefixToPostfix*	n^2	n^2
5		*getStringFromStack*	n	n
6	CharStack.py	*init*	n	n
7		*is_empty*	1	1
8		*is_full*	1	1
9		*peek*	1	1
10		*push*	n	1
11		*pop*	1	1
12		*pop_all*	n	0
13	StackStack.py	*is_empty*	1	1
14		*is_full*	1	1
15		*peek*	1	1
16		*push*	n	1
17		*pop*	1	1
18		*pop_all*	n	0
19	file_processing.py	*reverse_stack*	n	n
20		*process_file_data*	n^2	n^2
21	error_checking.py	*check_project_for_input_file_in_correct_directory*	n	1
22		*check_project_for_input_file*	n^3	1
23		*find_input_file*	n^3	1
24		*check_project_for_output_file_in_correct_directory*	n	1
25		*check_project_for_output_file*	n^3	1
26		*find_output_file*	n^3	1

Table 1 - Worst-Case Time and Space Complexities for functions in FranceLab1

The processing of the input file technically proved the most demanding on time complexity. However, this is because the program contains three nested loops to detect the input and output files for the program (iterating through each directory, then iterating through each subdirectory, then iterating through each file). While the maximum number of directories should be 1, the maximum number of subdirectories per directory should be 2, and the maximum number of files per subdirectory should be 8, I decided to use a bit of defensive programming in case the user mistakenly creates subdirectories into the module while trying to run it. Due to these reasons, the functions possessing in *error_handling.py* are highly unlikely to exhibit $\boldsymbol{O(n^3)}$ time complexities. The space complexities for these functions are negligible because the functions are only ever holding a Boolean *file_found* value in memory.

Probabilistically, the highest complexity functions in both space and time will be the *process_file_data* function in *file_processing.py* and the *prefix_to_postfix* function in *conversion.py.*

The function *process_file_data* manages the reading in of all input data character by character. This function uses two nested loops to process the data – one to parse the entire data stream by function, and the other to parse each function by character. Due to this, we reach a converging complexity value of $\boldsymbol{O(n^2)}$ in time (due to the loops) and in space due to the fact that each loop is creating a successively bigger stack at each level.

More interestingly, the function *prefix_to_postfix* in *conversion.py* uses a single loop to iterate through the entire stack. However, inside the loop are several *stack.push( )* functions. Each of these *stack.push( )* functions has a time complexity of ***O(n)*** and, as the stack grows successively bigger, the space complexity grows with it, resulting in a space complexity of ***O(n).*** In an average case, the space complexity of the entire function is unlikely to reach $\boldsymbol{O(n^2)}$ because the stacks in the nested loops are reset after an entire operation is completed, such as the operation *3 * 4* resulting in an postfix expression *3 4 **. Thus, the nested loop will only reach ***O(n)*** complexity in time and space if the entire prefix expression contains an operation that is a) correct and incredibly large, or b) contains several illegal characters and incredibly large.

Based off of the above analysis and, in consideration of the data presented in *Table 1*, the *FranceLab1* program has a worst-case time complexity of $\boldsymbol{O(n^3)}$ and a worst-case space complexity of $\boldsymbol{O(n^2)}$. However, the *FranceLab1* module probabilistically has worst-case time and space complexities of $\boldsymbol{O(n^2).}$

Lessons Learned

The project honestly taught me a lot about how to use less "sophisticated" ADTs in order to perform operations. Prior to the assignment, without the requirement of only being able to use a stack, I would have elected to use an array and simply iterate over each index. However, there is a lot that can be accomplished by using a main stack with an auxiliary stack or two that I didn't notice prior. The project also taught me to not just reactively select a list for every algorithm needing an array of elements, but to be more conscientious in algorithm design since there may be ADTs better suited for the problem. It really drove home the principles we learned in Modules 1, 2, and 4.

Optimizations for Later Versions

There are some optimizations that could be made to the code in the project. The *process_file_data* method in *file_processing.py* could be optimized for better time and space complexities by altering the way in which the auxiliary stacks are used. I would also add in some unit tests and perhaps optional command arguments such that, if the user opts out of inputting their I/O files when running the *FranceLab1* module, those files are automatically generated.

Addressing Requirements in *Lab 1*

FranceLab1 addresses the requirements as specified in the *Lab 1* handout. Specifically, the program reads prefix expression data from an input text file character by character. The program continues by appropriately parsing the data and allocating each found prefix expression in the input file to a stack. From there, each prefix expression stack is fed into a conversion function that converts the prefix expression into a postfix expression, doing so while still only using a stack. If the conversion is successful, the resultant postfix expression stack is then both logged to the console and written to the output text file. If the conversion fails, the cause of the failure is determined (illegal character, too many operands, etc.) and appropriate information is communicated to the user through both the console and output text file. When the program finishes the conversion of each prefix expression, the program terminates and the user may open the appropriate output text file to view their results. The results of each expression are always communicated alongside its corresponding prefix expression for easy comparison.

Enhancements

The program provides 4 major enhancements (along with many other minor enhancements) on top of the original requirements.

The first major enhancement is centered around the command line arguments. In order to execute the program, the user is required to specify the name of their input and output text files such as `python -m Lab1 input.txt output.txt.` The user is directed to put these files into the *io_files* folder of the module. In the event that the user puts these files into the wrong folder, the program searches the entire program for both files and, if found, relocates them to the designated *io_files* folder. In the event that the user specifies these files and the program cannot find either one of them, the program automatically generates 1) an input file called *input.txt*, and 2) an output file called *output.txt.* Both files are located in the *io_files* folder and the *input.txt* file is auto-populated with demonstration data containing 7 prefix expressions. So, in short, if the user never creates input or output files or moves them to the incorrect file path, the program will still execute successfully.

The second major enhancement incorporated time delays. When the program is conventionally executed, the command prompt flashes a plethora of data and finishes executing in under a second. The user is then forced to go back and read through the steps executed. *FranceLab1* incorporates time delays to allow the user to read information in the command prompt as processing steps are completed. For example, when the program starts up and accepts the command line arguments, a time delay of 3 seconds allows the user to know whether or not their *args* were accepted. Additionally, when the program successfully imports the I/O files, this information is logged to the command prompt and a time delay of 3 seconds halts processing so that the user can take in this information accordingly. Finally, after each prefix expression finishes conversion, a time delay of 1 second briefly holds processing as well to allow the user to see whether or not that step failed.

The next major enhancement revolves around error handling. If a prefix expression contains illegal characters, the program notifies the user through both the console and the output file. However, instead of simply letting the user know that the expression contained invalid characters, they are notified of exactly which characters that were illegal in their expression. Additionally, all white space is concatenated so that any gaps between characters in an expression (i.e. **3 4*) are stripped and any new lines within the input file are not counted as prefix expressions.

Finally, the fourth major enhancement builds a summary metric for the user. At the end of both the console output and output file, a summary of exactly how many prefix expressions were processed is displayed along with how many successively converted to postfix expressions and how many failed.

Recursion

While the current algorithm used to convert prefix expressions to postfix expressions uses iteration, recursion could also be used to convert the expressions. While it isn't exactly obvious, a recursive solution for the *prefix_to_postfix* might resemble something like the following:

```
function prefix_to_postfix(prefix_stack, length) {
        postfix_stack = Stack(length: length)
        operators = ['*', '/', '+', '-', '$']
        if (not prefix_stack.peek( ) in operators) {
                a = prefix_to_postfix(prefix_stack.pop( ) )
                b = prefix_to_postfix(prefix_stack.pop( ) )
                postfix_stack.push(a + b)
        } else {
                posfix_stack.push(prefix_stack.pop( ) )
        } endif
        return postfix_stack
} endfunction
```

I experimented a bit with it in code to try and codify the results. All recursive functions can be performed iteratively, but the base case must be concretely established in order to use recursive methods. In the above, my base case is when *prefix_stack.peek( ) == operator*. As a personal

bias, the iterative solution was obvious to me while the recursive solution took quite a bit of debugging and plenty of print statements to convince myself that a solution somewhat resembling the above was feasible.

Test Results

Running the command `python -m Lab1 input.txt output.txt.` produced appropriate results for an input file called *input.txt* containing the following prefix expressions:

```
-+ABC
-A+BC
$+-ABC+D-EF
-*A$B+C-DE*EF
**A+BC+C-BA
/A+BC +C*BA
*-*-ABC+BA
/+/A-BC-BA
*$A+BC+C-BA

837


//A+B0-C+BA
yt*h/t
*$A^BC+C-BA
*$A^BC+C-BA&&&&%
```

Those results were logged to an output file called *output.txt* and resembles the following:

```
********************************************************************************
********************************************************************************
********************************************************************************
                              Welcome
********************************************************************************
                  Starting Prefix-To-Postfix Program
********************************************************************************
   14 total expressions were read in.
   Beginning conversion of prefix expressions to postfix expressions...


1) For the prefix string: -+ABC, the equivalent postfix string is: AB+C-
--------------------------------------------------------------------------------
2) For the prefix string: -A+BC, the equivalent postfix string is: ABC+-
--------------------------------------------------------------------------------
3) For the prefix string: $+-ABC+D-EF, the equivalent postfix string is: AB-C+DEF-+$
--------------------------------------------------------------------------------
4) For the prefix string: -*A$B+C-DE*EF, the equivalent postfix string is: ABCDE-
+$*EF*-
```

```
--------------------------------------------------------------------------------
5) For the prefix string: **A+BC+C-BA, the equivalent postfix string is: ABC+*CBA-+*
--------------------------------------------------------------------------------
6) For the prefix string: /A+BC+C*BA, the equivalent postfix string is: ABC+/
--------------------------------------------------------------------------------
7) For the prefix string: *-*-ABC+BA, the equivalent postfix string could not be
found.
Too many operators for operands in prefix string: *-*-ABC+BA.
--------------------------------------------------------------------------------
8) For the prefix string: /+/A-BC-BA, the equivalent postfix string could not be
found.
Too many operators for operands in prefix string: /+/A-BC-BA.
--------------------------------------------------------------------------------
9) For the prefix string: *$A+BC+C-BA, the equivalent postfix string is: ABC+$CBA-+*
--------------------------------------------------------------------------------
10) For the prefix string: 837, the equivalent postfix string could not be found.
The characters ['7', '3', '8'] are illegal in the given prefix string: 837.
Only alphabetical characters and operands of type {'/', '*', '+', '$', '-'} are
acceptable.
--------------------------------------------------------------------------------
11) For the prefix string: //A+B0-C+BA, the equivalent postfix string could not be
found.
The characters ['0'] are illegal in the given prefix string: //A+B0-C+BA.
Only alphabetical characters and operands of type {'/', '*', '+', '$', '-'} are
acceptable.
--------------------------------------------------------------------------------
12) For the prefix string: yt*h/t, the equivalent postfix string could not be found.
Too many operators for operands in prefix string: yt*h/t.
--------------------------------------------------------------------------------
13) For the prefix string: *$A^BC+C-BA, the equivalent postfix string could not be
found.
The characters ['^'] are illegal in the given prefix string: *$A^BC+C-BA.
Only alphabetical characters and operands of type {'/', '*', '+', '$', '-'} are
acceptable.
--------------------------------------------------------------------------------
14) For the prefix string: *$A^BC+C-BA&&&&, the equivalent postfix string could not
be found.
The characters ['&', '&', '&', '&', '^'] are illegal in the given prefix string:
*$A^BC+C-BA&&&&.
Only alphabetical characters and operands of type {'/', '*', '+', '$', '-'} are
acceptable.
--------------------------------------------------------------------------------
********************************************************************************
********************************************************************************
7 out of 14 total non-blank lines were successfully converted to postfix expressions.
7 out of 14 total non-blank lines failed to convert to postfix expressions.
********************************************************************************
```

Conclusion

The *FranceLab1* Python module performs a conversion of mathematical prefix expressions specified by the user and returns (if available) its corresponding postfix expression. In hindsight, there are some optimizations that could be made to enhance the time and space complexities and fine-tune the conversion algorithm. The module contains several enhancements that aid in usability, user interface, and error handling. All tested prefix expressions correctly output the corresponding postfix expression and appropriately communicates any errors to the user.

References

1) Almes, Scott; et al. (2021). Stacks Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 4 February and 8 February 2021.

2) Almes, Scott; et al. (2021). Stacks Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 12 February 2021.

3) Almes, Scott; et al. (2021). Queues and Lists Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 19 February 2021.

4) Almes, Scott; et al. (2021). Lab 1 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 25 February 2021.

5) Almes, Scott; et al. (2021). Lab 0 Sample Project, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 26 February 2021. Web. URL: https://blackboard.jhu.edu/bbcswebdav/pid-9469422-dt-content-rid-100642966_2/xid-100642966_2

6) Almes, Scott; et al. (2021). Lab 1 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 1 March 2021.

7) Chlan, Eleanor; et al. (2021). ADT and Complexity Notes, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 28 January – 2 February 2021.

8) Chlan, Eleanor; et al. (2021). ADT and Complexity Notes, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 2 February– 9 February 2021.

9) Chlan, Eleanor; et al. (2021). Stacks Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 14 February 2021.

10) Chlan, Eleanor; et al. (2021). ADT and Complexity Notes, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 10 February– 16 February 2021.

11) Chlan, Eleanor; et al. (2021). Queues Notes, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 17 February – 18 February 2021.

12) Chlan, Eleanor; et al. (2021). Lists Notes, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 17 February – 18 February 2021.

13) Chlan, Eleanor; et al. (2021). Queues and Lists Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 21 February 2021.

14) Chlan, Eleanor; et al. (2021). Queues and Lists Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 28 February 2021.

15) Miller, B. N., & Ranum, D. L. (2014). Problem solving with algorithms and data structures using Python (2nd ed.). Decorah, IA: Brad Miller, David Ranum.