**Kordel France**
**Lab**
**Class 605.202 Section 81**

# Lab 2 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 *). *FranceLab2* 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, binary trees. Binary trees were implemented as a "list of lists". Arrays in other implementations 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. Stack data structures were also leveraged substantially for I/O manipulation. 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*, each *CharStack* containing a group of operations that have been processed from prefix to postfix form. Stacks were *not* used in any way for facilitation of the conversion and/or recursion algorithms.

## Justification for Design Decisions and chosen ADTs

The use of trees as the ADT for evaluation came intuitively. It is easy to see how simple mathematical expressions can be represented with binary trees with the operator as the root node and the two operands occupying the leaf nodes. I originally began building a binary tree data structure as *BinaryTree.py*, but as I began figuring the recursion algorithm, it became more intuitive that a "list-of-lists" implementation would be highly accessible and require less lines of code. Binary trees and descending subtrees were formatted so that, for initial reading-in of the prefix expression, the root node always occupied index 0 and, if applicable, the left and right children always occupied indices 1 and 2 respectively. This provided for a consistent architecture across the entire tree and allowed for easy traversal later on. As one of the final steps for the conversion of the prefix expression, its corresponding binary tree was reformatted through a call to the function *formatRootNodeAsRight* in *conversion.py* by reordering the entire tree so that the root of each subtree occupied the right-most index and children occupied preceding indices. For example, a subtree with both children initially formatted as *[rootNode, leftChild, rightChild]* was reformatted to *[rightChild, leftChild, rootNode];* subtrees with only one child were

reformatted to *[leftChild, rootNode];* and leaf nodes with no children were reformatted as *[rootNode]*. It's important to note here that built-in Python operations such as *list.reverse( )* were not used; the entire subtree was manually reorganized. Other ADTs would have worked as well, but the method of evaluation doesn't so naturally point to their use. In a simple mathematical expression, operations always require exactly two operands and one operator, which instinctively points to a linked-list implementation. Imagining many of these operations stacked on top of each other allows one to easily visualize how a binary tree can smoothly facilitate all of these linked lists together. Furthermore, formatting the expression as prefix, infix, or postfix, requires only a traversal of the tree and simple adjustments of the root and child node indices.

# Complexity and Algorithm Analysis

Please refer to *Table 1* for a complete analysis of time and space complexities for every function in the *FranceLab2* module.

| FranceLab2 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 | *prefix_to_postfix* | n | n |
| 5 | | *prefix_to_postfix_recursive* | n^2 | n^2 |
| 6 | | *format_root_node_as_right* | n | n |
| 7 | | *print_binary_tree_as_string* | n | n |
| 8 | CharStack.py | *init* | n | n |
| 9 | | *is_empty* | 1 | 1 |
| 10 | | *is_full* | 1 | 1 |
| 11 | | *peek* | 1 | 1 |
| 12 | | *push* | n | 1 |
| 13 | | *pop* | 1 | 1 |
| 14 | | *pop_all* | n | 0 |
| 15 | file_processing.py | *reverse_stack* | n | n |
| 16 | | *process_file_data* | n^2 | n^2 |
| 17 | error_checking.py | *check_project_for_input_file_in_correct_directory* | n | 1 |
| 18 | | *check_project_for_input_file* | n^3 | 1 |
| 19 | | *find_input_file* | n^3 | 1 |
| 20 | | *check_project_for_output_file_in_correct_directory* | n | 1 |
| 21 | | *check_project_for_output_file* | n^3 | 1 |
| 22 | | *find_output_file* | n^3 | 1 |

*Table 1 - Worst-Case Time and Space Complexities for functions in FranceLab2*

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.

The file *conversion.py* contains the methods that actually implement the conversion and recursion algorithms. The function *prefix_to_postfix_recursive* is the function recursively called to evaluate the prefix expression. It constructs a binary tree in list format that breaks down each operation inside the prefix expression. We know that a binary tree of depth *d* with *n* leaves contains *2n-1* nodes, so building such a tree requires a time complexity of $\boldsymbol{O(2n) = O(n)}$ and demands space complexity of the same. $\boldsymbol{O(n^2)}$ is mentioned in *Table 1* because, technically, if the tree was an m-ary tree, *m* could be sufficiently large in a worst-case scenario which could account for time and space complexities of $\boldsymbol{O(m*n) \sim O(n^2)}$. The function *format_root_node_as_right* is a traversal algorithm that traverses the entire tree and reformats each subtree such that the left and right children are essentially swapped. This technically requires a time complexity less than $\boldsymbol{O(n)}$, because the algorithm will stop at depth level $d - 1$, but the complexity converges to $\boldsymbol{O(n)}$. Space complexity for this function also converges to $\boldsymbol{O(n)}$ since it is also recursively called until the leaf nodes are reached. The function *print_binary_tree_as_string* is another tree traversal algorithm that traverses the postfix-formatted binary tree and builds a human-readable string in the form of a postfix expression. Since every node (including the leaves) must be reached, the function's time complexity equates to $\boldsymbol{O(n)}$ as well. Time complexity for this function is also $\boldsymbol{O(n)}$ since the returned string requires the same cumulative space as all node objects in the binary tree.

Lastly, the function *prefix_to_postfix* in *conversion.py* uses a single loop to iterate through the entire expression for error checking. This iteration is dependent on the size of the prefix expression passed so the time complexity results in ***O(n)*** while the space complexity is represented by the same since the entire expression must be stored through this function for evaluation.

Based off of the above analysis and, in consideration of the data presented in *Table 1*, the *FranceLab2* 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 *FranceLab2* module probabilistically has worst-case time and space complexities of $\boldsymbol{O(n^2).}$

## Lessons Learned

The lectures and textbook opened up my approach to this Lab quite a bit. While assessing how to use recursion to convert the prefix expression during the *Lab 1* analysis, I did not anticipate using a tree at all. However, after a few conceptual examples were provided in the lecture notes, it became obvious that a tree could easily model this problem. I've also been able to acknowledge how models of several other problems would be easily represented through trees whereas 3 weeks prior I would have reactively elected to use a simple, non-linked array. For *Lab 2*, it was easy to get away with using the "list-of-lists" implementation of the binary tree, but for more complex problems such as m-ary trees, I would not be so keen to use the list format. It's easy to see how this format can become complicated, messy, and difficult to manipulate as the depth and breadth of the tree grows. Taking this into consideration, I would construct a custom tree object with methods that allow for easy accessibility and readability.

Additionally, *Lab 2* further drove home the power of recursion. When facilitated correctly, constructing one perfect function gives the programmer a lot of advantage and simplicity over an iterative version that may be more complicated. It is also easy to see how harmoniously recursion and trees work together. These two simple concepts give a lot of power to the programmer with very few lines of code. Using recursion allowed me to remove an entire file from the program that was otherwise needed in *Lab 1*.

## Optimizations for Later Versions

As specified previously, I would choose to implement the binary tree through nodes and references in a custom object in a later version. The "list-of-lists" implementation worked really well for this project but would be difficult to scale. There are also some optimizations I would incorporate that I mentioned in *Lab 1* as well. Particularly, 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 *FranceLab2* module, those files are automatically generated.

## Addressing Requirements in *Lab 2*

*FranceLab2* addresses the requirements as specified in the *Lab 2* 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 Lab2 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. *FranceLab2* 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.

## Recursive vs. Iterative Evaluation

The constraints posed in *Lab 1* and *Lab 2* illuminate the pros and cons of using both recursive and iterative procedures in evaluating mathematical expressions. While I know that iterative methods can occasionally out-perform recursive methods in time and space complexities, both *Lab 1* and *Lab 2* showed consistent algorithmic behavior across my code in this respect. Under both scenarios, the project produced worst-case time complexities of $\boldsymbol{O(n^3)}$ and worst-case space complexities of $\boldsymbol{O(n^2)}$. However, the recursive solution wins my preference due to the facts that it is easily represented the mathematical evaluation rules, resulted in less lines of code, and worked seamlessly with the binary tree ADT. The recursive solution was much more intuitive to program and, on a much larger data input, would scale more swiftly.

## Test Results

Running the command `python -m Lab2 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
********************************************************************************
Processing input...
   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 could not be
found.
There are too many operands or operators in given prefix string.
--------------------------------------------------------------------------------
7) For the prefix string: *-*-ABC+BA, the equivalent postfix string could not be
found.
There are too many operands or operators in given prefix string.
--------------------------------------------------------------------------------
8) For the prefix string: /+/A-BC-BA, the equivalent postfix string could not be
found.
There are too many operands or operators in given prefix string.
--------------------------------------------------------------------------------
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.
There are also too many operands or operators in given prefix string.
Only alphabetical characters and operands of type ['*', '/', '+', '-', '$'] are
acceptable.
--------------------------------------------------------------------------------
12) For the prefix string: yt*h/t, the equivalent postfix string could not be found.
There are too many operands or operators in given prefix string.
--------------------------------------------------------------------------------
13) For the prefix string: *$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.
--------------------------------------------------------------------------------
********************************************************************************
********************************************************************************
6 out of 14 total read prefix expressions were successfully converted to postfix
expressions.
8 out of 14 total read prefix expressions failed to convert to postfix expressions.
********************************************************************************
```

## Conclusion

The *FranceLab2* Python module performs a conversion of mathematical prefix expressions specified by the user and returns (if available) its corresponding postfix expression. The module does this explicitly through recursion and the construction of binary trees. 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 2 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 2 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.

16) Chlan, Eleanor; et al. (2021). Graphs Notes, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 10 March– 15 March 2021.

17) Chlan, Eleanor; et al. (2021). List Notes, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 10 March– 15 March 2021.

18) Almes, Scott. (2021). Homework 7 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 11 March 2021.

19) Chlan, Eleanor. (2021). Homework 7 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 14 March 2021.

20) Almes, Scott. (2021). Lab 2 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 19 March 2021.

21) Chlan, Eleanor. (2021). Lab 2 Office Hours, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 21 March 2021.