Lab 1 Analysis

EN.605.202
Data Structures
Summer 2022

Jose Márquez Jaramillo

Engineering and Applied Science Programs for Professionals
Johns Hopkins University
United States
July 5th, 2022

Abstract

This lab presents a methodology for converting prefix to postfix expressions, it also analyzes the implications of using prefix and postfix expressions along with some of its benefits. A framework for evaluating the validity of prefix expressions is also discussed. Finally, the Python language implementation of these concepts is discussed and alternative methods and improvements are considered.

Contents

1 About prefix and postfix expressions and their conversion

1.1 Important attributes of prefix and postfix notations

Prefix or otherwise known as ”Polish notation” is a mathematical notation in which operators precede their operands. Therefore prefix notation requires that all operators precede the two operands that they work on. Because of this, prefix expression evaluation provides two important benefits over infix evaluation [MR]; namely:

- Prefix expressions do not need the use of parenthesis;
- prefix expressions have precedence built into their organization.

In postfix or ”reverse Polish notation” the operators follow their operands. If there are multiple operators, operators are given immediately after their final operands (often, an operator takes two operands, in which case the operator is written after the second operand). Similar to prefix operations, postfix expressions do not require the use of parentheses. These characteristics provide the following relationships between prefix and postfix expressions:

- Prefix and postfix expressions are polar opposites;
- because they do not require parentheses, they can both provide faster execution times than infix (algebraic) expressions;
- both prefix and postfix expression evaluation can lead to fewer errors since their organization has built-in precedence.

1.2 What is a valid prefix expression?

Up to this point in class and this document, the only way provided to evaluate if a prefix notation is valid is that of computation. However, in this lab, there is a requirement to parse and convert prefix expressions into postfix expressions without having to evaluate and without transforming them to infix expressions. A valid prefix expression by definition always starts with an operator and ends with an operand. Evaluation as indicated in class can begin at the right by pushing operands into a stack, popping twice from the stack whenever an operator is found, pushing back the result of the operation into (b op a) to the stack, and iterating until the end of the string. This leads to a couple of interesting conditions for prefix expressions:

1. When evaluating from right to left, the last found character is going to be an operator. There must remain at least two operands before this operator in order for possible evaluation. Therefore, the total number of operands has to be one more than the number of operators.
2. The rightmost two characters in the expression will be in general operands, excluding those two last symbols, the remaining characters must contain no less operators than operands. This ensures that excluding that first operation, there are enough operands and operators to evaluate the function.

These two conditions are implemented in the final program to identify valid prefix expressions.

1.3 Conversion of prefix expressions to postfix expressions

Given that prefix and postfix expressions are ”reversals” of one another, the simplest of all methods of converting from prefix to postfix is to read the pertinent formula backward [Ham62]. There are however some caveats to this procedure since certain operators such as subtraction (’-’) and exponentiation (’$’) are not commutative (i.e. $a\$b \neq b\$a$ or $a - b \neq b - a$ for some a and bs). We can, however account for this fact during the interpretation of the expressions.

1.3.1 Using stacks for the conversion

Since the operation requires a reversal of order, a stack with its FIFO data abstraction provides an adequate implementation of the procedure. One method to convert the formula can then be to read the prefix expression from right to left (i.e for $S_1 S_{n_2} ... S_{n-1} S_n$ starting from S_n followed by $S_{n-1}, S_{n_2}, ..., S_1$) and parse each symbol S_i as appropriate. From S_n to S_1 follow the following steps:

1. If S_i is an operand, push onto the stack;

2. If S_i is an operator pop twice from the output stack. Create a string with the operators in the right postfix order (in the case of non-commutative operators) followed by the operator, push the resulting string onto the stack;

3. repeat 1 and 2 until S_1 has been parsed.

1.3.2 Complexity of conversion

This approach requires parsing each expression while looping through each symbol (S_i), and it, therefore, has linear worst-case complexity or $O(n)$.

2 Description of program and methods

The program is ”packaged” according to the specifications in the guidelines. The folder SourceCode contains all of the code generated and used for implementation. An effort was made to make the file parsing as accurate as possible, and several different test cases were implemented to alleviate potential incidences. Enhancements beyond those initially specified in the lab were also implemented to both check if a read line contains a valid prefix expression and the level of information generated in output. Below are descriptions of the methods in the package, these are also presented as doc-strings:

- `Stack()` implements a Stack using common list methods from the python language. No input is required, It generates a Stack object. The implementation follows closely the Miller and Ranum implementation in listing 3.1. This stack implementation is based on the use of a built-in list class from Python, making also use of some built-in methods as, for example, `list.pop()`.

- `readfile(input_file)` reads the file in the location specified character by character. As part of the function, characters that are non-alphanumeric nor valid sign operators are not considered to be parsed into the prefix expression. Both the original string and the parsed ”clean” expressions from each line are stored in a dictionary for further analysis. The function returns a dictionary with line numbers as its keys. Since this function loops through the file, it has linear time complexity or $O(n)$.

- `is_prefix_valid(expression: str)` determines if the prefix expression is a valid one or not. This method loops over each expression twice (non-nested) and it therefore has linear time complexity or $O(n)$
- `prefix_to_postfix(entry: dict)` converts a prefix expression to a postfix expression. The function takes in a dictionary entry which is a dictionary. The function uses `is_prefix_valid()` to check if it is a valid prefix expression. If valid, a stack of class `Stack()` is used to convert it into a postfix expression. The methodology of conversion is as previously explained. The function does not return anything but adds entries into the dictionary corresponding to the expression. As previously indicated this conversion function has linear complexity or $O(n)$.
- `process_files(input_file: Path, output_file: Path)` reads in the file using `readfile()`, then loops through the resulting dictionaries with `prefix_to_postfix()` to check if the expressions are valid, convert them, and add results to their corresponding dictionaries. An output file is generated as specified with a summary table and then specification for each line to the corresponding string read, the prefix expression processed, and corresponding conversion to postfix. In the cases where the lines do not contain valid prefix expressions, the output indicates it as well. In order to generate the output as specified, this function loops outputs the file with linear complexity or $O(n)$.

3 Recursion

In the current implementation a stack has been used to convert prefix to postfix expressions. Another possible way to solve this problem is to use recurssion. A possible implementation of recursion would be:

1. Start at the beginning of the expression, that is evaluate from left to right.
2. If the first symbol is an operator, call the function again to the second symbol,
3. If the first symbol is not an operator, compose the postfix expression.
4. Repeat 2 and 3 until the last symbol in the expression.

Since the function will effectively iterate over every symbol on each line, the recursive conversion will have linear complexity or $O(n)$. Recall that the stack implementation of the conversion was also $O(n)$; however, the run time of the stack implementation depends solely in the number of symbols meanwhile the recursive implementation depends on the number of symbols as well as the number of operators.

4 Enhancements

4.1 Pre-processing as an enhancement

Pre-processing is an important step in the program because valid symbols need to be recognized in each line to parse them and try to discern a valid prefix expression. It is also important to be able to identify valid prefix expressions to avoid producing invalid postfix expressions as well as improving the efficiency of the program. The dictionary as a storage data structure also provided flexibility when later providing summary information and output.

4.2 Output as an enhancement

As previously indicated emphasis was placed on the output generated, in particular on summarizing the information parsed. The output summary therefore indicates:

- the name of the file processed,
- the total number of lines evaluated,
- the number of lines containing valid prefix expressions,
- and finally the line numbers of those which did not contain valid prefix expressions.

Next the output file goes line by line indicating:

- the line number,
- the input as read,
- whether a valid prefix expression was found in the line, if no valid prefix expression is contained in the line,
- and finally the postfix conversion (if available).

5 What I learned and what I would do differently

5.1 What I learned

I have summarized above several concepts which have contributed to the development of this lab, below I will explicitly list some of the concepts that I learned as a consequence of going through this exercise:

- How are prefix and postfix notations similar or different?
- Why is it possible to convert directly from prefix to postfix without converting to infix?
- What makes a prefix expression valid?
- How can prefix expressions be converted to postfix? What is the cost of that conversion?
- Can the conversion from prefix to postfix expressions be done recursively?
- Developing python packages in Pycharm.
- Python File I/O basics
- Running python programs through the command line.

5.2 What I would do differently

- The program could identify if the expression contains numbers instead of letters and produce a mathematical solution.
- Along with this mathematical solution there can be checks for mathematical errors such as division by zero or taking a root of a negative number.
- The code could be further optimized and other implementations can be used.

References

[Ham62] C. L. Hamblin. Translation to and from polish notation. *The Computer Journal*, 5(3):210–213, nov 1962.

[MR] Bradley N. Miller and David L. Ranum. *Problem Solving with Algorithms and Data Structures Using Python*. Franklin Beedle Associates.