Postfix to Machine Instruction Using a Stack

Jeffrey Wan

John Hopkins University

<u>Author Note</u>

It was actually pretty great to learn how to code a Stack in Java even though I work as a full time software engineer. This has made reading Java and even Scala so much easier. Thanks to my classmates and my professor at Hopkins for helping me through this course as well.

Abstract

This program is a postfix converter that uses a stack to do the conversion from postfix to machine instruction. It is intended to mimic a cpu converting postfix instructions to machine instructions. It is written in Java and uses OOP principles to maintain a stack and keep track of necessary state needed to do the conversion.

Design and Analysis

The converter was made using 4 main classes: The main PostfixConverter class which is the application runner is composed of the Operand, LinkedListStack, CPU, and Operator classes.

The Operand class checks to see if the character of the postfix command is a character. This is needed to check if the characters are valid and for the edge case of the postfix command being one character. The Cpu class is responsible for managing the stack and converting the postfix command into machine instructions using the stack. There are two stacks, one to hold the converted machine instructions, and one to hold the postfix commands as they are being converted to machine characters. At the end, the stack holding the converted machine instructions are reversed because the bottom of the stack is actually the first command to be executed. The Operator class is used to check if the character of the postfix command is an operand. The operator class is also responsible in determining which operator the incoming character is which is used by the Cpu class to build the proper instruction. The main PostfixConverter is responsible to input and output management and sending the character to the Cpu class to be converted.

Errors arise when the characters are neither operators nor operands and if the stack that keeps track of postfix commands has too many commands leftover which would imply too few operators. They are eventually caught by the PostfixConverter class and the appropriate action is taken. If the character is invalid, the postfix command is skipped and it is noted in the output file and standard output that the postfix command was invalid and skipped. If the postfix command is invalid (too few operators or operators that occur before a sufficient number of operands), it is also noted in the output file and stdout.

A stack made sense to keep track of both the postfix conversion to machine instruction and the final machine instructions. The stack is implemented using a Linked List which offers two advantages: unlimited size, and O(1) append time, both of which were higher priority than the main advantage offered by an array-based implementation: random access. We're doing fewer reads than writes so I think the linked-list implementation is better.

A stack is also useful when converting postfix to machine instruction. The conversion algorithm necessitates storing combined intermediate postfix expressions which a stack was perfect for. For example, in AB+C-, the AB+ is combined into A+B which is stored on the stack. A 'C' is then added to the stack with the 'C' on top and so the stack now has two items. When the final '-' is reached, both items are popped and combined into A+B-C and placed back on the stack. A+B-C is the final infix. Even though we were not using the infix, the stack is still needed to make the proper conversion into machine instruction because it can keep track of the intermediate expressions. The final machine instructions are stored on another stack and reversed since the bottom of the stack is actually the first instruction. Perhaps a queue is better suited to store the machine instructions in a future implementation of this converter.

My solution to the problem was an interactive one. I looped through the postfix expressions and then looped through each character in each expression so it is an O(N^2) solution to this problem. This is the only solution I see at the moment unless there's some way to convert each postfix expression into machine code without looping through each character. If I were to recursively solve this problem, instead of looping through each postfix expression, I would implement a method that looked like:

```
cpu = new Cpu() // creates new stacks.
convert(instruction, index) {
    if instruction[index] == null
        writeToFile(cpu.convertedLinkedListStack.machineInstructions())
    else {
        machineInstruction = cpu.process((instruction[index]))
        if machineInstruction != null {
                cpu.convertedLinkedListStack.push(machineInstruction)
        }
        convert(instruction, index++)
    }
}
```

A recursive solution would have to convert each character to machine instruction and the Cpu class would keep track of the instructions processed so far. It would call itself again until the postfix command was fully processed (the base case) and when that happens, it writes the converted machine instructions to file.

Lessons Learned

I think the biggest lesson learned here was how to use stacks to manage ordered items. During the development stage of this project, I made several mistakes using stacks. I was writing to output commands that were in reverse order because commands that were supposed to be listed on top and first were actually at the bottom of the stack and so when I was popping the stack to write to output, the last item to be written to output was actually first in the stack. It wasn't until I wrote the `reverse` method that this problem was fixed.

Other lessons learned were actually related to Java. I learned how to package classes so that they had access to other classes in the same package and I also learned how to write assertions for testing.

Testing:

These were the test cases that I used:

```
AB+
AB-
AB*
AB/
AB+C-
AB+C*
ABC+-
ABC*+DE-/
AB-C+DEF-+$
ABCDE-+$*EF*-
ABC+*CBA-+*
ABC+/CBA*+
AB-*CBA+-*
ABC-/BA-+/
ABC+$CBA-+*
AB0+/CBA+-/
-AB
+AB
*AB
/AB
C-AB
C+AB
C*AB
C/AB
A

AB
AB@
AB#
!
```

- It includes an empty string, valid expressions, invalid expressions with only operands and too many operators, invalid expressions with invalid characters like #, and simple expressions that were easy to develop against like AB-. I think these test cases resulted in some pretty robust code.

A few things to note:

1. This was an odd example:

```
Handling Postfix line: AB0+/CBA+-/
LD B
AD 0
ST TEMP1
LD A
DV TEMP1
ST TEMP2
LD B
AD A
ST TEMP3
LD C
SB TEMP3
ST TEMP4
LD TEMP2
DV TEMP4
ST TEMP5
```

I chose to handle this by treating 0 as a constant and so I can add 0 to the loaded register B and store the result in Temp1. I don't think this is a syntax error and so chose to treat it as so.

2. If the expression has an operator that occurs when the linkedList has not enough items to combine into another expression, an error is thrown. Some examples of this are +AB (the first operator is processed with an empty stack) and AB-*CBA+-* which processes a * when the stack only has one items in it. In these cases, an error should thrown and recorded and the next postfix expression should be processed.

3. If the expression has too many operands and not enough operators, the stack that keeps track of intermediate postfix expressions will have more than 1 item in it at the end. In this event, an error should be thrown that states that there were not enough operators in the original expression which should be recorded. The Converter class should continue to process the next expression.

4. If the expression has an invalid operand like # or @, it will ignore the expression and continue processing the next expression.

<u>Some additional features:</u>

- can now handle postfix expressions with exponents like AB-C+DEF-+$ and ABCDE-+$*EF*-
- handles empty strings, only operand expressions, invalid expressions with invalid characters.
- included Javadocs in the doc folder.