Lab 01

**Adam Rich**

Section EN.605.202.87

Spring 2018

March 6, 2018

**Approach**

The desired pseudo-ML output always comes in chunks of 3 lines every time an operator is encountered.

1. Put the first variable in the register
2. Operate on 1st var with 2nd, leaving result in register
3. Copy value in register to temp variable (even if that is never used)

The requirements of the lab say to "leave the final result in the register" and the example output has a final line that copies the final result to a temp variable. My code does the same, even though that final variable is never used.

I am processing each character, one character at a time. This does make the code a little harder to follow since there are several conditional expressions within a while loop. However, to make the output file more readable, the ML instructions for a single statement are cached in a String variable until the entire postfix expression is processed. This allows me to output the converted postfix expression at the top of the block. However, the buffer string is never re-processed, in line with the spirit of the lab.

An integer stack is used. The stack is my own code complete with its own test cases. Please see the README file for a sample command line to run these.

I am using just one integer stack for the entire file, re-initialized for each new line of input. All variables are pushed to this stack, even the temporary ones. To differentiate between temp IDs and actual character variables A-Z, temp variable IDs are stored as negative values. I could have stored these as positive and still processed strings with up to 64 operations. However, I thought that this was an unnecessary limit (in fact, I have one sample input that includes an expression with several hundred operations).

Another approach I considered was to have two stacks, one for names variables and another for temp variable IDs (or even a String stack pushing the whole "TEMPn"). This would require a marker in the character stack, say zero. Every time a zero is encountered on a pop from the character stack, pop from the TEMP stack. However, since this requires special "encoding" of a value in the character stack, I opted to just use the one stack. Both approaches require a compromise, so just use one was my philosophy.

**Command Line Args**

The lab instructions dictate that all input should be passed by using command line arguments, and not rely on user input. However, in the file "Error Handling and Exceptions.pdf" available on the course website, it gives an example of dealing with command line args that are invalid. The example recommends alerting the user that the command line args are invalid and then gracefully giving them an opportunity to fix the input without erroring out. This code is carved out of main() and put in a separate private method of the application class.

Valid command line args can be passed and they do not then require any user input. User input is validated, within reason, but using output files that are really directories will throw an IOException, which is caught and gives an output message to contact the developer for help, if needed.

**Extra Operator and White-Space**

Sample input includes "$". This is not in the list of operators that we are required to support. But, instead of treating this as an invalid character, this is instead handled using the new ML expression: PW. There might be other ways to do this, say with a GOTO ML instruction and a "comparison operator" and that would be another interesting thing to try.

Extra spaces, tab characters, blank lines and a last expression with no EOL character are all supported. EOL is marked by either CR or LF characters, to support Windows and/or Linux files. Since I am relying on the default behavior of FileReader and FileWriter to handle files, not all file encodings are supported. My example are UTF-8 or ANSI. Both encodings work just fine.

**Using integers instead of Strings**

Following on Dr. Erhan's example I/O code, I am treating the one-at-a-time read characters as integers. This allows me to push them to an integer stack instead of creating a String stack or a char stack.

**"StackInteger" class and tests**

The StackInteger class is my own creation. It is an array implementation with all the standard stack methods: push, pop, isEmpty, and peek. It also includes a toString method that prints the contents of the stack, space delimited, with the top most element on the right.

The class also includes a private method called autoGrow. This helps to prevent stack overflow. When you try to push to a full stack it instead copies the array to a bigger array to create room. This of course takes time for big operations, but you can also initialize the original stack with a specified size, if you can anticipate needing it to remove the need to autoGrow in some cases.

There are two constructors: a default one and a custom one. The inputs to the custom are the initial size of the private array and whether to allow automatic autoGrow. Passing false for the autoGrow makes for a stack that is like the array implementations we studied in class.

**Use of "String" class**

I used the string class in my code just for creating strings to append to in preparation for writing to the output file. Processing is still one character at a time and strings are never re-processed. Output, the original input expression, and error messages are cached in separate strings to allow for more logical arrangement of the output file.

**Output Format**

Output uses "//" as a comment marker. Each expression has its own block of white-space delimited output. The top line gives the original expression followed by the correct ML. If an expression is invalid the output gives the expression and tries to tell the user where the expression is invalid. This is not perfect as sometimes the error is just "not enough operators".

**Style**

In an effort to keep lines shorter in the presence of many nested conditionals and loops, I have opted to use 2 characters as the default indent. Some expressions needed to be broken at "+" or other continuation characters, but these always appear in the preceding line to be less confusing.

Comments are in the application code to help future maintainers quickly understand what is going on. They do not repeat what is readily understandable from the code. They do precede each method, main and private, to help explain inputs and purpose. In particular, main's comment block makes it clear the usage of the args variable.

The flowerbox in each file attempts to give future developers some context. Full context, of course, can be had my reading this file, the associated README file and the original lab assignment, all of which are included in the lab folder.

**Test Cases**

Several test case files were created:

* default (the assigned inputs)
* default-CR (tests different EOL characters)
* really-bad (stuff that is not even post-fix expressions)
* test-symbols (what happens when non-support symbols are passed)
* empty (what does it do with an empty file)
* no-newline (last line has no EOL character)
* default-spaces (test whether certain white space is allowed: space, tab, extra newlines)
* really-big (really long expressions)

The default input includes test cases with too many variables and some with too many operators.

The output of all these sample input files is in the output directory. See the README file for more information on convenience batch scripts to run these.

**Things Learned**

I learned a lot of Java, or re-learned it. I also learned that sometimes the hardest part of writing an app is dealing with potential errors. It was great to write and use a stack in a real example. I really liked learning about postfix expressions. And, it was a nice challenge to do the dev without an IDE.

**What would I do differently?**

I would like to spend some more time making the code more readable and making sure that I have exhausted all possible errors with input (that is, invalid postfix expressions). The output file could include more information about what the input was, how many statements there are, and how many successfully compiled, but given that I spent my time on helping the user find errors in their expressions, I think this is a fair compromise.

**Efficiency**

I am processing one character at a time. so the app has Big-O of n (being the number of characters in the input file). However, if the file is made up of expressions that just get bigger and bigger, and the initial array size is not large enough, the autoGrow function in the underlying array will require an additional number of operations, but I am not sure how many that would be. A default stack array implementation would just error out in this case as n gets larger.

Big Omega notation is also Big Omega (n).

**IDE**

My IDE was notepad++ and the command line. Instructions on how to build and run the code, from the command line at the project base folder, are in the README file.

**github**

I used github, a private repo, to backup and track important chunks of development.