Lab 1

Justin Ely

615.202.81.FA15 Data Structures

06 October, 2015

Comments

A stack-based implementation make sense for translating postfix expressions largely because stacks are a well-understood and robust way to evaluate a postfix expression. This ties into the current assignment, as to translate an expression into machine language you first need to determine each individual expression.

From each individual expression, the machine code is straight-forward: load argument 2, perform operation on arg2 and arg1, store result.

Providing adequate tests for the translator was integral to the final version of the code. With the initial version, there had been no accounting for blank strings, spaces, numbers, or unused variables. Putting in many different test-cases with strange edge-cases caused new errors (or lack there of) which lead to new handling of the expressions.

Recursion

While a recursive solution would be possible, it's not the standard way to address postfix expressions. The iterative technique is straightforward and naturally works with a stack.

Design

For my stack, I chose to implement an array based stack of fixed size (50 elements) and type (String). The String type seems like a natural choice given the design requirements. It allows each element to be of different lengths, which would not be possible with a char array, and is thus able to store the "TEMPX" strings needed in the problem. The fixed type was motivated more from simplicity than anything else. Dynamic typing was not needed for this problem, and so implementing it would cause additional tests to be run and additional failures to be possible. A set length of 50 elements was chosen because memory is cheap, and a 50 variable stack is unlikely to happen in a typical postfix expression.

I chose to implement a custom exception for BadPostfixExpression. Both for clarity to the user and specificity when error handling. Providing clearer output to the user is a clear advantage; constraining the error as much as possible and pointing right to what went wrong reduces the amount of time spent debugging. The added specificity when handling errors is particularly good practice so as to not accidentally handle an unexpected error simply because the names were the same.

Enhancements

While the requirements were to read expressions from named files, I felt that it is cumbersome to need to write a text file just to feed a single command into the translator. Because of this, I allow both named files and explicit postfix expressions to be supplied as input. This does have some trade-offs: namely if an input argument is the name of a file on the disk, then it will be read line-by-line, otherwise it will be treated as a single postfix expression. This means that an incorrectly spelled file will be read as postfix and will generate the same error as would an invalid expression.

Another enhancement is the support of exponentiation. The design requirements stat that only +,-,*,/ need to be supported, but it was simple to implement exponentiation ($) as well. The machine command corresponding to $ is PW, short for power.

The last enhancement made is an additional convenience function that returns the current size of the stack. While not needed for this problem, for an array-based stack of fixed size, such a method could be useful to avoid overflow and so was included.

Efficiency

The efficiency of the module is tied to the stack implementation. Since insertions and deletions can happen only at one end, the cost of either operation is O(1). With constant-time operations to pop and push from the stack, the complexity of the entire translation process is O(n) - since to evaluate a postfix expression, a single pass over each variable is required.

There will be differences in the constant factor for the complexity in each run however. As an example, translating the simple expression AB+ requires two pushes and two pops. For a longer expression with nested operations, the results of previous computations will be pushed back onto the stack for evaluation later.

What I learned

This project was my first serious experience with Java, and as such I learned quite a lot about the language and how the language semantic enforce a particular design. The very structure of needing to split each class into a separate file enforced a high amount of modularity. While I appreciate this to an extent, such aggressive splitting is a distinct departure from the logical groupings I'm used to. As a particular example, I would have liked to include my custom exception, BadPostfixExpression, in with the code for parsing and translating the expressions. This particular exception is custom-designed for this application, and it would make sense to me to be able to find it with the other postfix specific code.

In addition, throwing exceptions was a major learning experience. Beyond simply needing to declare the thrown exception in the function definition, the two distinct exception types was a stumbling block for a time. The fact that some errors must be caught by their calling functions is a good example of the language enforcing particular behavior. By throwing an error that requires catching upon compilation, I'm forced to write code that is less-prone to unintended failure.

Complaints

This wouldn't be a learning experience without some complaints, so to round out the experience I might as well detail them. Declaring types of functions and variables is a pain. I do understand why they are there, but they simply make programming less fun!

Next time

The main thing I would change would be generic typing of the Stack data structure. Although restricting the type to Strings is fine for the current designs of this project, creating stacks of different types would be beneficial both as a learning experience and to allow more features to the Postfix translator.

I would also either use a dynamically allocated stack, or have the maximum size of the stack be set upon initialization. Although my imposed limitation of 50 elements seems generous for the current application, allowing the stack size to be set based upon the length of the input expression would be much safer and generalizable.

Exploring the available unit-testing frameworks is another aspect that I would work towards. Although my test-cases work fine, the amount of work required to test them up without the benefit of a unit-test environment was burdensome and unnecessary.