Jon Powers Lab 1—Analysis Document

USAGE

```
Lab1PostfixConverter inputFile.txt outputFile.txt [optimize]
```

`inputFile.txt:`	The input file containing postfix expressions, one per line, to be converted into machine instructions by the program.
`outputFile.txt:`	The file that output will be written to. Any existing file with the same name will be overwritten.
`optimize:`	Optional. If "optimize" (without quotes) is provided as a third argument to the program, the Compiler will invoke optimization routines that attempt to reduce the number of machine instructions generated to evaluate the postfix expression. This will be overridden by any configurations specified in the input file.

GENERAL OVERVIEW

This program has three main components: an InputFileHandler, a Compiler, and the main Lab1PostfixConverter. Each makes extensive use of the StringStack class, which implements a stack ADT for Strings; the InputFileHandler also uses the CharStack class, which is a similar implementation but for char data. The program leans on InputFileHandler.getLinesFromFileAsStacks() to process the specified input file, which returns an array of StringStacksDataPacks, each of which holds a StringStack containing the input string along with any metadata parsed from the input file. Each StringStack contains a postfix expression from the input file in a stack that will pop off items from left to right. If the file contains no data or there is an issue reading the file, InputFileHandler will return an empty array to the main program, which signals to it that some error has occurred.

The workhorse in Lab1PostfixConverter is its convertPostfix() method, which takes in a StringStackDataPack containing the postfix expression and its metadata. It reads the expression elements one at a time by popping them off the input stack. Operands are placed onto an operand stack, and when operators are encountered, they are applied to the two operands at the top of the stack. The result is pushed back onto the operand stack. The operand stack serves as an error checking mechanism: if we run out of operands before we finish processing the input, then we know that there are not enough operands for a valid postfix expression; conversely if there is more than one final value on the operand stack after processing the input expression, then we know there were too many operands for a valid postfix expression. In either case, the error is reported up the call stack.

The actual compilation to machine instructions is performed by instances of the Compiler. Optimizations are available but are disabled by default. The basic compilation process takes in two operands and an operator in the form OperandA Operator OperandB—e.g., A + B or C / D—loading OperandA into the register, and applying the Operator-specific instruction to the register using OperandB; the result of the operation is then stored in a temporary variable. this results in three machine instructions for each operation. I implemented a number of optimizations that reduce the number of overall instructions in cases of commutative operations (addition and multiplication), chained operations like A + B + C where there is no need to store intermediate results, and special cases where the operand is 0 or 1. These optimizations are detailed below in the ENHANCEMENTS section.

Input File Format, Accepted Operations, and Accepted Operands

This program requires that the input file contain one postfix expression per line. Lines beginning with the character '#' are treated as comments and ignored, and lines beginning with '>' provide configuration information for the expression, such as whether or not to enable optimizations in the Compiler. All whitespace characters (Unicode value < 20 and Unicode 32) are ignored, and both CRLF and LF line endings are accepted. The program treats '+' as addition, '-' as subtraction, '*' as multiplication, '/' as division, and '$' as exponentiation. Exponentiation is supported but only if OperandB is a digit between 0 and 9. Except in the case of exponentiation and the special cases of operands equaling 0 or 1 (detailed below), all characters are treated as input operand values; I assume that these would be associated with numerical values if this program were actually run.

Data Structures Used

This program makes extensive use of stack structures throughout. This is primarily because nearly all of the processes required for this program require us to maintain an ordered list of actions to take or to maintain an ordered history of items that we calculate. For example, by convention we apply mathematical operations from left to right, so we must track the order in which operands appear in the postfix string. By pushing all operand values onto a stack when processing the postfix expression, we maintain their order and know that items on top of the stack appear to the right of items lower on the stack.

Stacks also provide a convenient way to process our input one item at a time. The process for evaluating a postfix expression requires us to look at one character at a time, which is exactly what the stack provides to us. In addition, the stack takes care of all the work of tracking how much of the input is left to process and telling us when we've exhausted our input source—a feature that serves a critical validation role when evaluating postfix expressions.

The Compiler also makes use of stacks internally to store the machine instructions it generates. Although I could have had the Compiler simply append instructions to the end of a string, some of the Compiler's optimizations require the ability to backtrack and remove previously generated instructions. For example, using the non-optimized behavior, a postfix expression like ABC++ would result in storing the result of BC+ into a temporary variable, loading A into the register, and then adding that temporary variable to the contents of the register. The optimizer recognizes that this is unnecessary because addition is commutative, so it removes the previously generated store instruction and simply adds A to the register after BC+ is computed. Without a stack to backtrack, it would be much more difficult to implement these types of optimizations.

This program relies primarily on String-based stacks rather than character based ones. This was originally driven by the need to store temporary variable names (made up of multiple characters) in the operand stack. While there is a limited workaround of using digits to represent those variable—such as "1" for "TEMP1"--that approach would max out at 10 variables (one each for 0-9) and would eliminate the ability to treat digit characters as integers for the special cases of 0 and 1 and for exponentiation. Given this extra flexibility, I chose to rely mainly on String-based stacks rather than character based ones. The one place this created some headache is in the InputFileHandler because we were asked to read in the source file by character rather than by String, which requires a somewhat awkward and clunky approach to convert into StringStacks.

Time and Space Efficiency

Both the StringStack and CharStack implementations used in this program are implemented using an array to hold the stack contents. I did this both for simplicity and to facilitate some of the"bonus" stack features they contain. All the major stack operations run in $O(1)$ time, such as push(), pop(), isEmpty(), and similar methods. Most of the bells and whistles, such as copy() and viewStack(), run in $O(n)$ because they must perform some actions on each element of the stack. But the implementation of reverse(), which was needed in several places in this program, takes advantage of the array to swap elements on each end of the stack until reaching the midpoint, which ends up halving the time required to be on the order of $O(\frac{n}{2})$.

Array stack implementations are generally space inefficient because they are statically allocated and we have to overestimate how large to make them in order to accommodate items that will be pushed onto the stack; we risk overflow if the stack is too small. To address both the space and overflow concerns, both stack implementations used in this program are self-growing in that they will automatically increase their size using the private method stretchStack() if an attempt is made to push onto the stack when the underlying array is full. This allows us to provide a small-sized array initially (minimizing memory use) without risking overflow. It also ensures that the underlying array is reasonably close in size to the amount of room being used. Although not implemented, it would be reasonable to also include a shrinkStack() method to shorten the underlying array when the stack grows smaller in order to conserve memory. These methods do add some computational overhead to the stack due to periodically requiring us to resize and copy over the stack contents, but the benefit of avoiding overflow outweighs the minor time costs.

Recursive Comparison

The handling of the input stack, operand stack, and calls to the compiler could be handled recursively, with each iteration passing on a smaller input sub-stack as well as the most recent operand stack until the input stack is empty. While this is a perfectly valid approach, it feels like a bad fit because the string processing required to evaluate a postfix expression is inherently sequential, which lends itself to an iterative approach. And because we're also tracking our progress all along the way with operand stacks and machine instruction lists, it seems unwieldy to be passing these things back and forth, which we would have to in order to use recursion.

An alternative approach to recursion would be to develop a way to identify subproblems in the postfix expression—equivalent to parenthesized infix expressions or portions of the infix expression that have order-of-operations precedence—that could be evaluated and then passed back back up the chain to be used in computations that involve the solutions to those smaller subproblems. I wanted to develop a routine along these lines so that expressions that reduce to zero like AB+0* could be pulled out and simplified to a single machine instruction: LD 0. Others like AB+B- could be similarly reduced. Breaking the expression down into subproblems that could be reduced to simple instructions could allow for a wider range of optimizations than the simple iterative approach that I chose.

ENHANCEMENTS

This program contains a number of enhancements that go beyond the basic Lab 1 requirements and specifications. It implements some limited functionality for '$' as an exponentiation operator, and it contains extensive attempts to optimize and the reduce number of output machine instructions. Please note that these optimizations are only used if the "optimize" argument is passed to the program from the command line or if optimizations are specified for an expression in the input test file.

Exponentiation. The '$' character is not defined in the program specification but is commonly used to denote exponentiation. The Compiler contains a private exponentOperation() method to provide some basic exponent functionality. The special case of x^0 is defined to result in an output of 1 in all cases other than 0^0, which is undefined and raises an exception and results in rejecting the expression as invalid. The exponentOperation() method will provide naive exponentiation where the exponent component is a digit character between 1-9 by multiplying the base by itself that many times. If the exponent is a variable, the expression is rejected because we do not have the flow-control, comparison, or bit-shifting machine instructions that would allow us to properly evaluate these at run time.

Commutative Operations. Probably the biggest enhancement implemented in this program is for commutative operations. As an example, consider the postfix expression provided in the Lab 1 assignment, ABC*+DE-/. After BC* is computed, the example stores the result in Temp1, and then computes ATemp1+. Of course because addition is commutative, it makes no difference whether we calculate $A + Temp1$ or $Temp1 + A$. So the intermediate storing of Temp1 serves no computational purpose. As a result, when optimizations are enabled, the program will see that the result of BC* is already in the register and will simply add A to the register. This eliminates the ST and LD instructions, which made up 40% of the non-optimized instructions. Over a large number of calculations, this sort of optimization could make a huge difference.

Zero-Based Operations. This program contains a number of optimizations based on the properties of operations involving zero. For example, an addition operation with zero as one of the operands will always equal the other operand. So rather than generating instructions to do that calculation, we simply go directly to the result—the nonzero operand. (In the case of 0+0, both operands are zero and I arbitrarily chose to keep the second operand as the "nonzero" number to keep.) Similarly for subtraction, subtracting zero from a number leaves the number unchanged, so we skip the subtraction step. Multiplying a number by zero always equals zero, so we skip the operation and go directly to zero. And finally zero divided by a number will always be zero (except for $\frac{0}{0}$), and a number divided by zero is undefined and will result in rejecting the expression as invalid.

Since the majority of input cases here use letters to represent values, we can't know what values would actually be sent if the program were run. As a result, the division by zero protection and computational shortcuts will not work as intended when the compiled instructions are run and values are substituted for the variables. This virtual machine lacks the instruction set necessary to provide the additional checks needed to catch those situations.

One-Based Operations. This program also takes advantage of the known results of identity operations involving 1 as an operand. Namely, multiplication of a number by one will result in that number, and dividing a number by 1 will also result in that number. These routines could apply when the other number

is zero, but as a matter of program-flow, the zero-based operations take precedence over the one-based operations.

Input File Enhancements. This program takes its input from a text file, which is required to contain one postfix expression per line. However, to facilitate reading and writing of test input I implemented some parsing routines that allow the inclusion of file comments, expression comments, as well as Compiler configuration. Lines beginning with '#' are ignored by the InputFileHandler when parsing it for input data. This allows us to provide comments within the file about its contents. Lines beginning with '>' are treated as metadata for the expression, which will be passed along to the program. The metadata lines must appear before the postfix expression. The strings OptimizeOn and OptimizeOff are captured as configuration commands, which will dictate whether optimizations are enabled when the Compiler evaluates the postfix expression. This configuration will override the optimization setting selected from the command line. So, for example, if the user runs the program with optimizations off, an expression will still be evaluated with optimizations enabled if OptimizeOn is specified in the input file. If both OptimizeOn and OptimizeOff are specified in the input file, then both are ignored (since we can't do both), and the program will use the user-selected setting.

Lessons Learned and Additional Opportunities

Although not the point of this project, one of the biggest lessons I learned is that optimization is hard. What started out as some simple ideas took a lot more work than I anticipated to generalize and make work over multiple and varied test cases. And there is still more low-hanging-fruit optimization that could be done. As an example, the expression AB+0* reduces to zero but I did not have time to develop a routine to look back to find the beginning of that expression (or sub-expression) and eliminate the machine instructions that it generated (though certainly some sort of stack would be involved in backtracking to work that out).

I also ran into some of the limitations of stacks in the project. While they were very convenient to use in many cases, they are limited to a one-time use. So if I needed to, for example, pop down the stack to generate a string of its contents, I would lose all of the information that was in that stack. For example, one requirement was that we echo the input expression in the output. Since my InputHandler passes the expression to the program as a stack (and I used that stack up in the process of processing it), I was left with an empty stack that I couldn't get the original expression out of. Pulling out the input expression before processing it would leave me with the same problem. So I ended up adding a copy() function to the stack classes and passed copies of the stack in situations where I needed to exhaust the stack without losing its information forever. In this regard, a list might have been a more appropriate data structure to use for certain items.

I also ran into some issues with the stack being backwards from what I needed, since it only pops from the top. For example, the main program loops through the postfix expressions from the file and pushes the output onto a stack until all the expressions have been processed. This stack is then used to feed information to the output file. But if we just pop off the output stack's contents, everything is going to be in reverse order from the input file (because the first input expression's output is at the bottom of the stack). While this is a somewhat minor point, it could lead to confusion for the user reading the output. And, in fact, the very same issue came up in the Compiler because the machine instructions are stored in a stack as they are processed, and that order matters. So I implemented a reverse() method that reverses the order of the stack by swapping elements in the underlying array. The same effect could be achieve by

dumping the stack out onto another stack (which is, in fact, what's done with the machine instructions in the Compiler) or by using a FIFO queue.