CS255 – Project 03– Infix to Postfix and Postfix Evaluation

Description and Overview

Your program must accept an infix expression, convert it to postfix, evaluate the postfix and print the answer to standard output.

For this project, the infix and postfix operands are restricted to single decimal digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Operators: +, -, *, /, ^, which denote addition, subtraction, multiplication, integer division and exponentiation, respectively.

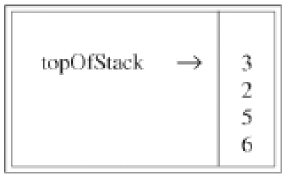
This following examples are taken from pp. 104-110 in Data Structures and Algorithm Analysis in C++, 4th edition, by Weiss. Although your program should first convert infix expressions to postfix expressions and then evaluate the postfix expression, the presentation starts with examining postfix evaluation.

Suppose we have a pocket calculator and would like to compute the cost of a shopping trip. To do so, we add a list of numbers and multiply the result by 1.06; this computes the purchase price of some items with local sales tax added. If the items are 4.99, 5.99, and 6.99, then a natural way to enter this would be the sequence 4.99+5.99+6.99∗1.06=. Depending on the calculator, this produces either the intended answer, 19.05, or the scientific answer, 18.39. Most simple four-function calculators will give the first answer, but many advanced calculators know that multiplication has higher precedence than addition. On the other hand, some items are taxable and some are not, so if only the first and last items were actually taxable, then the sequence 4.99∗1.06+5.99+6.99∗1.06= would give the correct answer (18.69) on a scientific calculator and the wrong answer (19.37) on a simple calculator. A scientific calculator generally comes with parentheses, so we can always get the right answer by parenthesizing, but with a simple calculator we need to remember intermediate results. A typical evaluation sequence for this example might be to multiply 4.99 and 1.06, saving this answer as A1. We then add 5.99 and A1, saving the result in A1. We multiply 6.99 and 1.06, saving the answer in A2, and finish by adding A1 and A2, leaving the final answer in A1. We can write this sequence of operations as follows: 4.99 1.06 ∗ 5.99 + 6.99 1.06 ∗ +.

This notation is known as postfix, or reverse Polish notation, and is evaluated exactly as we have described above. The easiest way to do this is to use a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers (symbols) that are popped from the stack, and the result is pushed onto the stack.

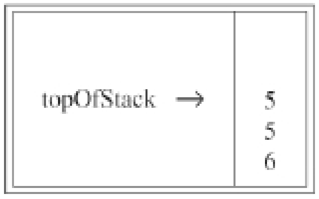
For instance, the postfix expression **6 5 2 3 + 8 * + 3 + *** is evaluated as follows:

6 5 2 3 **+ 8 * + 3 + *** The first four symbols are read and pushed on the stack.

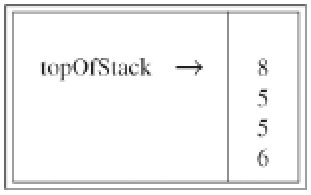


6 5 2 3 + 8 * + 3 + *

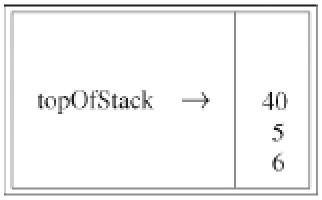
Next, a '+' is read, so 3 and 2 are popped from the stack, and their sum, 5, is pushed.



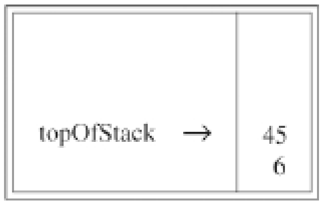
6 5 2 3 + **8** * + 3 + *   Next, 8 is pushed.



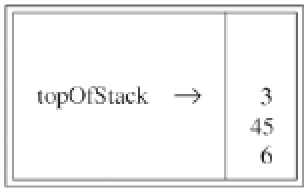
6 5 2 3 + 8 * + 3 + *   Now a '*' is seen, so 8 and 5 are popped, and 5 * 8 = 40 is pushed.



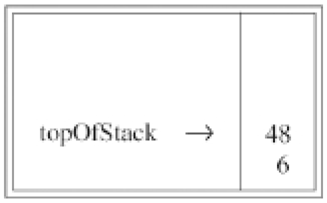
6 5 2 3 + 8 * + 3 + *   Next, a '+' is seen, so 40 and 5 are popped, and 5 + 40 = 45 is pushed.



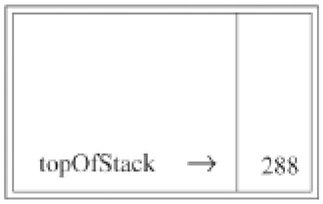
6 5 2 3 + 8 * + 3 + *   Now, 3 is pushed.



6 5 2 3 + 8 * + 3 + *   Next, '+' pops 3 and 45 and pushes 45 + 3 = 48.



6 5 2 3 + 8 * + 3 + *
Finally, a '*' is seen and 48 and 6 are popped; the result, 6 * 48 = 288, is pushed.



CS255 – Project 03– Infix to Postfix and Postfix Evaluation

Infix to Postfix Conversion:

A stack be used to evaluate a postfix expression, but we can also use a stack to convert an expression in standard form, also known as infix form, into postfix form. We will concentrate on a small version of the general problem by allowing only the operators +, *, (,), and insisting on the usual precedence rules.   Reminder: * has precedence over + and both * and + are assumed to use left association. For example, for the infix expression, x*y*z, the conversion should assume to evaluate the leftmost * first: (x*y)*z.

For the purpose of this discussion, we will further assume that the expression is legal. Suppose we want to convert the infix expression a + b * c + (d * e + f) * g into postfix.

The correct answer is a b c * + d e * f + g * +.

We start with an initially empty stack.

- When an operand is read, it is immediately placed onto the output.
- Operators are not immediately output, so they must be saved somewhere.
  - The correct thing to do is to place operators that have been seen, but not placed on the output, onto the stack. We will also stack left parentheses when they are encountered.
  - If we see a right parenthesis, then we pop the stack, writing symbols until we encounter a (corresponding) left parenthesis, which is popped but not output.
  - If we see any other symbol (+, *, ( ), then we pop entries from the stack until we find an entry of lower priority. One exception is that we never remove a (from the stack except when processing a). For the purposes of this operation, + has lowest priority and (highest.
  - When the popping is done, we push the operator onto the stack.
- Finally, if we read the end of input, we pop the stack until it is empty, writing symbols onto the output.

The idea of this algorithm is that when an operator is seen, it is placed on the stack. The stack represents pending operators. However, some of the operators on the stack that have high precedence are now known to be completed and should be popped, as they will no longer be pending. Thus prior to placing the operator on the stack, operators that are on the stack, and which are to be completed prior to the current operator, are popped.

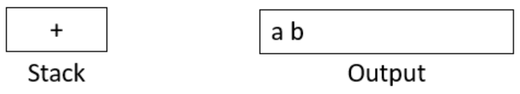
Expression	**Stack** When Third Operator Is Processed	**Action**
a*b-c+d	-	- is completed;+ is pushed
a/b+c*d	+	Nothing is completed; * is pushed
a-b*c/d	-*	* is completed; / is pushed
a-b*c+d	-*	* and - are completed; + is pushed

Parentheses simply add an additional complication. We can view a left parenthesis as a high-precedence operator when it is an input symbol (so that pending operators remain pending) and a low-precedence operator when it is on the stack (so that it is not accidentally removed by an operator). Right parentheses are treated as the special case.

To see how this algorithm performs, we will convert the long infix expression above into its postfix form.

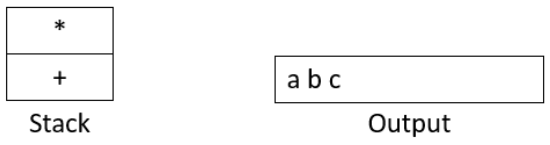
a+b*c+(d*e+f)*g

First, the symbol a is read, so it is passed through to the output. Then + is read and pushed onto the stack. Next b is read and passed through to the output. The state of affairs at this juncture is as follows:



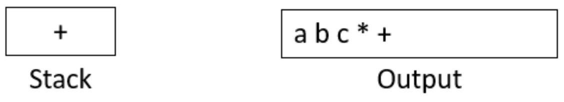
a+b***c**+(d*e+f)*g

Next, a * is read. The top entry on the operator stack has lower precedence than *, so nothing is output and * is put on the stack. Next, c is read and output. Thus far, we have



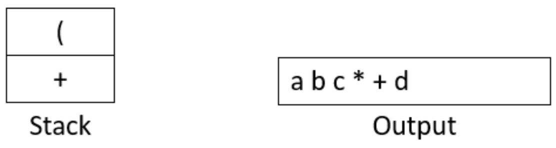
a+b*c**+**(d*e+f)*g

The next symbol is a +. Checking the stack, we find that we will pop a * and place it on the output; pop the other +, which is not of lower but equal priority, on the stack; and then push the +.



a+b*c+**(d***e+f)*g

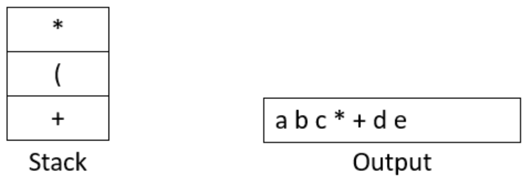
The next symbol read is a (. Being of highest precedence, this is placed on the stack. Then d is read and output.



CS255 – Project 03– Infix to Postfix and Postfix Evaluation

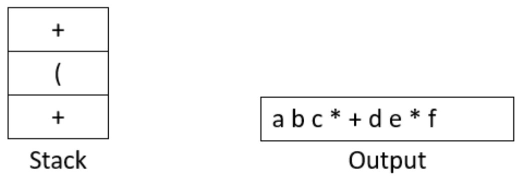
a+b*c+(d***e**+f)*g

We continue by reading a *. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, e is read and output.



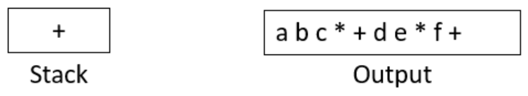
a+b*c+(d*e**+f**)*g

The next symbol read is a +. We pop and output * and then push +. Then we read and output f.



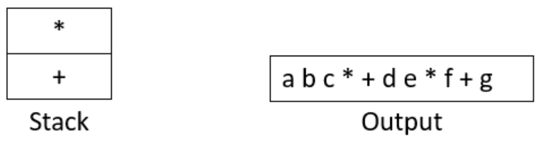
a+b*c+(d*e+f**)***g

Now we read a), so the stack is emptied back to the (. We output a +.



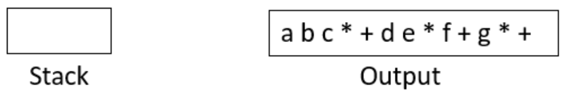
a+b*c+(d*e+f)***g**

We read a * next; it is pushed onto the stack. Then g is read and output.



a+b*c+(d*e+f)*g **<end>**

We’re at the end of input, so we pop and output symbols from the stack until it is empty.



We can add subtraction and division to this repertoire by assigning subtraction and addition equal priority and multiplication and division equal priority. A subtle point is that the expression a - b - c will be converted to a b - c - and not a b c - -. Our algorithm does the right thing, because these operators associate from left to right. This is not necessarily the case in general, since exponentiation associates right to left: 2^2^3 = 2^(2^3) = 2^8 = 256, not (2^2)^3 = 4^3 = 64.

CS255 – Project 03– Infix to Postfix and Postfix Evaluation

Input /Output Requirements:

- Your program must read an infix expression represented as a string.
  - Operands: Single decimal digits
  - Operators: +, -, *, / , ^
- Your program must output the corresponding expression and then the result of the postfix evaluation. Use sample runs shown below
- Use integer division. Check for division by zero.
- Your program should keep reading infix expressions and processing each one until an empty string is read.
- Use the given code as an outline for implementing this project and complete the code to accomplish the above requirements.

Sample Run:

```
Enter an infix expression: 3 ^ 2 ^ (1+2)
The postfix form is 3 2 1 2 + ^ ^
Value of the expression = 6561

Enter an infix expression: 3 * (4 - 2 ^ 5) + 6
The postfix form is 3 4 2 5 ^ - * 6 +
Value of the expression = -78

Enter an infix expression: (7 + 8*7
infix2Postfix: Missing ')'

Enter an infix expression:
```

CS255 – Project 03– Infix to Postfix and Postfix Evaluation

Design and Implementation Requirements:

InfixToPostfixDriver.cpp: Main driver class. Use the given code. Study and understand it.

MyStack.h: Template stack class. You must implement the template Stack class using singly-linked list without a header node or a tail pointer. It has a single member variable that is a reference to a **StackNode**, nodes put on the stack. (StackNode is defined as a struct within the MyStack.h file.)

It will be used by both the InfixToPostfix and the PostfixEval class. Member functions top(), topAndPop(), and pop() should throw an UnderflowException for accessing an empty stack.
See the "CS255Exceptions.h" below.

- Complete isEmpty()
- Complete makeEmpty()
- Complete pop()
- Complete push()

InfixToPostfix.h: Converts an infix string to a postfix expression. Use the starter code and complete the member function so that the given infix expression is converted to the correct postfix expression.

- Complete the postfix() member function

PostfixEval.h: Use the starter code and complete the member functions so the postfix expression is evaluated and produces the correct result.

- Complete the compute() member function: watch for integer division by zero and 0^0 error.
- Complete the evaluate() member function

ExpressionSymbol.h: Used to set the stack and input precedence of operators.

CS255Exceptions.h: This is used for exceptions and error processing and is limited for this program. Your program must utilize the exception classes given in this file. Your MyStack class should check for empty stack. The InfixToPostfix class and member function, postfix(), should throw an ExpressionException when an infix expression has a missing right parenthesis. See the above sample runs. Also watch for integer division by zero and 0^0 indeterminate form error.

Due Date: See Programming Projects on Course Blackboard Site.