Infix to Postfix Notation Converter and Postfix Expression Evaluator

Introduction

We are accustomed to writing expressions using *infix* notation, such as: Z = X + Y. Computers prefer that we use *postfix* notation: Z = XY +. One important advantage of postfix notation is that postfix form requires no parentheses. The order of the operators in the postfix expression determines the actual order of operations in evaluating the expression, making the use of parentheses unnecessary.

The evaluation of an infix expression such as A + B * C requires knowledge of which of the two operations, or , should be performed first. In general, A + B * C is to be interpreted as A + B * C unless otherwise specified. We say that multiplication takes *precedence* over addition. Suppose that we would now like to convert A + B * C to postfix. Applying the rules of precedence, we begin by converting the first portion of the expression that is evaluated, namely the multiplication operation. Doing this conversion in stages, we obtain the following: (The part of the expression that has been converted is underlined.)

`A + B * C`	*Given infix format*
`A + B C *`	*Convert the multiplication*
`A B C * +`	*Convert the addition*

The major rules to remember during the conversion process are that the operations with highest precedence are converted first and that after a portion of an expression has been converted to postfix it is to be treated as a single operand. Let us now consider the same example with the precedence of operators reversed by the deliberate insertion of parentheses:

`( A + B ) * C`	*Given infix format*
`A B + * C`	*Convert the addition*
`A B + C *`	*Convert the multiplication*

Note that in the conversion from "A B + * C" to "A B + C *", "A B +" was treated as a single operand. The rules for converting from infix to postfix are simple, provided that you know the order of precedence.

We consider four binary operations: addition, subtraction, multiplication, and division. These operations are denoted by the usual operators, + , -, *, and /, respectively. There are two levels of operator precedence. Both * and / have higher precedence than + and -. Furthermore, when unparenthesized operators of the same precedence are scanned, the order is assumed to be left to right. Parentheses may be used in infix expressions to override the default precedence.

Problem Statement

Using your favorite stack implementation developed in class, write a program to convert an infix expression to postfix format and then evaluate the resulting postfix expression.

Input

An input file contains a collection of error-free infix arithmetic expressions, one expression per line. Expressions are terminated by semicolons, and the final expression is followed by a period. An arbitrary number of blanks and end-of-lines may occur between any two tokens in an expression. A token may be an operand (you may assume it is a positive single-digit integer), an operator , a left parenthesis, or a right parenthesis. Here is a sample input:

```
3 + 4 – 5;
3 + 5 * 2;
( 6 + 4 ) / ( 6 - 4 );
( ( 2 + 1 ) * ( 4 - 4 ) + 1 ) / ( 2 + 3 ).
```

Output

Your output should consist of each input expression, followed by its corresponding postfix expression, followed by the result of the postfix expression, followed by a blank line. All output (including the original infix expressions) must be clearly formatted (or reformatted) and also clearly labeled. Here is a sample output for the sample input given above.

```
Infix:     3 + 4 - 5
Postfix:   3 4 + 5 -
Result:    2

Infix:     3 + 5 * 2
Postfix:   3 5 2 * +
Result:    13

Infix:     ( 6 + 4 ) / ( 6 - 4 )
Postfix:   6 4 + 6 4 - /
Result:    5

Infix:     ( ( 2 + 1 ) * ( 4 - 4 ) + 1 ) / ( 2 + 3 )
Postfix:   3 1 + 4 4 - * 1 + 2 3 + /
Result:    0
```

Processing Notes (Infix to Postfix Conversion)

In converting infix expressions to postfix notation, the following fact should be taken into consideration: in infix form the order of applying operators is governed by the possible appearance of parentheses and the operator precedence relations; however, in postfix form the order is simply the "natural" order--i.e., the order of appearance from left to right.

Accordingly, sub-expressions within innermost parentheses must first be converted to postfix, so that they can then be treated as single operands. In this fashion, parentheses can be successively eliminated until the entire expression has been converted. The last pair of parentheses to be opened within a group of nested parentheses encloses the first sub-expression within that group to be transformed. This last-in, first-out behavior should immediately suggest the use of a stack. Your program may utilize any of the operations in any of the Stack class implementations developed in the class.

In addition, you must devise a function that takes two operators and tells you which has higher precedence. This is helpful because in Rule 3 below you need to compare the next input token to the top stack element. Question: What precedence do you assign to '('? You need to answer this question because '(' may be the value of the top element in the stack. You should formulate the conversion algorithm using the following six rules:

Rule 1. Scan the input string (infix notation) from left to right. One pass is sufficient.
Rule 2. If the next token scanned is an operand, it may be immediately appended to the postfix string.
Rule 3. If the next token is an operator,
- a) Pop and append to the postfix string every operator on the stack that
  - i. is above the most recently scanned left parenthesis, and
  - ii. has equal or higher precedence than that of the new operator symbol.
- b) Then push the new operator symbol onto the stack.

Rule 4. When an opening (left) parenthesis is seen, it must be pushed onto the stack.
Rule 5. When a closing (right) parenthesis is seen, all operators down to the most recently scanned left parenthesis must be popped and appended to the postfix string. Furthermore, this pair of parentheses must be discarded.

Rule 6. When the infix string is completely scanned, the stack may still contain some operators. (No parentheses at this point. Why?) All these remaining operators should be popped and appended to the postfix string.

Below are two more examples to help you understand how the algorithm works. Each line in the following tables demonstrates the state of the postfix string and the stack when the corresponding next infix symbol is scanned. The rightmost symbol of the stack is the top-of-the-stack symbol. The rule number corresponding to each line demonstrates which of the six rules was used to reach the current state from that of the previous line.

Example 1: Input expression is
A + B * C / D - E

Next Symbol	Postfix String	Stack	Rule
A	A		2
+	A	+	3
B	A B	+	2
*	A B	+ *	3
C	A B C	+ *	2
/	A B C *	+ /	3
D	A B C * D	+ /	2
-	A B C * D / +	-	3
E	A B C * D / + E	-	2
	A B C * D / + E -		6

Example 2: Input expression is
(A + B * (C - D)) / E

Next Symbol	Postfix String	Stack	Rule
(		(	4
A	A	(	2
+	A	(+	3
B	A B	(+	2
*	A B	(+ *	3
(	A B	(+ * (	4
C	A B C	(+ * (	2
-	A B C	(+ * (-	3
D	A B C D	(+ * (-	2
)	A B C D -	(+ *	5
)	A B C D - * +		5
/	A B C D - * +	/	3
E	A B C D - * + E	/	2
	A B C D - * + E /		6

Processing Notes (Postfix Evaluation)

Now that you have the expression converted to postfix notation, it needs to be evaluated. The evaluation algorithm is as follows:

1. While the end of the postfix expression has not been encountered, read the expression from left to right:
   a. If the current token is an operand, push it onto the stack
   b. Otherwise, if it is an operator:
      i. Pop the two top elements of the stack into variables x and y
      ii. Calculate y operator x
      iii. Push the result of the calculation onto the stack
2. When the end of the postfix expression is encountered, pop the top value of the stack. This is the result of the postfix expression

Deliverables

- The source code of your program renamed using the following naming convention:

  LastName-FristName-AssignmentNumber.cpp

- The input file you used to test your program.

Submission Instructions

- A listing of the source code of your program named as described above.
- Submit your deliverables as indicated above in the drop box dedicated for this assignment.

Dr. Iyad A. Ajwa **Programming Assignments**
CS–230 Data Structures **Assignment 5**