Infix to Postfix Converter

This problem requires you to write a program to convert an infix expression to postfix format. 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

A + B * C	*Given infix form*
A + B C *	*Convert the multiplication*
A B C * +	*Convert the addition*

(The part of the expression that has been converted is underlined.)

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 form*
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.

As discussed earlier, the postfix form requires no parentheses. The order of the operators in the postfix expressions determines the actual order of operations in evaluating the expression, making the use of parentheses unnecessary.

Input

The 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 symbols in an expression. A symbol may be an operand (a single upper-case letter), an operator (+, -, *, or /), a left parenthesis, or a right parenthesis.

Sample Input:

```
A + B - C ;

A + B * C ;

( A + B ) / ( C - D ) + E ) / ( F + G) .
```

Output

Your output should consist of each input expression, followed by its corresponding postfix expression. All output (including the original infix expressions) must be clearly formatted (or reformatted) and also clearly labeled.

Sample Output:

```
Infix:     A + B - C ;

Postfix: A B + C -

Infix:     A + B * C ;

Postfix: A B C * +

Infix:     ( A + B ) / ( C - D ) ;

Postfix: A B + C D - /

Infix:     ( ( A + B ) * ( C - D ) + E ) / ( F + G ) .

Postfix: A B + C D - * E + F G + /
```

Discussion

When converting infix expressions to postfix notation, the following fact should be taken into consideration: with 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, subexpressions 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* subexpression within that group to be transformed. This last-in, first-out behavior should immediately suggest the use of a stack.

In addition, you must devise a Boolean 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 symbol 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.

Algorithm

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 symbol scanned is an operand, it may be immediately appended to the postfix string.

Rule 3: If the next symbol 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 precedence higher than or equal to 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.) All these remaining operators should be popped and appended to the postfix string.

StackType Data Structure

You are required to use a linked list implementation of the Stack ADT described in Chapter 5. Outside of the given Stack ADT operations, your program may not assume knowledge of the stack implementation; in other words, you may not add any additional methods to the Stack ADT. If you need additional stack operations, you should specify and implement them using the operations from the Stack ADT.

It is highly recommended you start with the stack implementation first and verify it can work with different types.

Examples

Here are two examples to help you understand how the algorithm works. Each line on the following table 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 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+B*C/D-E	A		2
	A	+	3
	A B	+	2
	A B	+ *	3
	A B C	+ *	2
	A B C *	+ /	3
	A B C * D	+ /	2
	A B C * D / +	-	3
	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*
(A+B*(C-D))/E		(	4
	A	(	2
	A	(+	3
	A B	(+	2
	A B	(+ *	3
	A B	(+ * (	4
	A B C	(+ * (	2
	A B C	(+ * (-	3
	A B C D	(+ * (-	2
	A B C D -	(+ *	5
	A B C D - * +		5
	A B C D - * +	/	3
	A B C D - * + E	/	2
	A B C D - * + E /		6

Deliverables

- A listing of your program including any classes used
- A listing of your test plan as input to the program
- A listing of the output file from the test runs (can be screenshots)