Infix to Postfix Conversion

This problem requires you to write a program to convert an infix expression to a postfix expression. The evaluation of an infix expression such as A + B * C requires knowledge of which of the two operations, + and *, 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 first convert the portion of the expression that is evaluated first,* namely the multiplication. 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 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 AB + * C to AB + C *, AB+ 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 five binary operations: addition, subtraction, multiplication, division, and exponentiation. These operations are denoted by the usual operators, +, –, *, /, and ^, respectively. There are three levels of operator precedence. Both * and / have higher precedence than + and –. ^ has higher precedence than * and /. Furthermore, when operators of the same precedence are scanned, +, –, * and / are left associative, but ^ is right associative. Parentheses may be used in infix expressions to override the default precedence.

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

A collection of *error-free* simple arithmetic expressions. Expressions are presented one per line. The input has an arbitrary number of blanks between any two symbols. A symbol may be a letter (A – Z), an operator (+, – , *, or /), a left parenthesis, or a right parenthesis. Each operand is composed of a single letter. The input expressions are in infix notation.

Example

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

Output

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

Example
(Only the four postfix expressions corresponding to the above sample input are shown here.)
A B + C –
A B C * +
A B + C D - /
A B + C D - * E + F G + /

Discussion

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, 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 the group to be transformed. This last-in, first-out behavior should immediately suggest the use of a stack.

Your program should utilize the basic stack methods. You will need to PUSH certain symbols on the stack, POP symbols, test to see if the stack is EMPTY, look at the TOP element of the stack, etc.

In addition, you must devise a boolean method that takes two operators and tells you which has higher precedence. This will be helpful, because in Rule 3 below, you need to compare the next symbol to the one on the top of the stack. [Question: what precedence do you assign to '('? You need to answer this question since '(' may be on top of the stack.]

You should formulate the conversion algorithm using the following six rules:

1. Scan the input string (infix notation) from left to right. One pass is sufficient.
2. If the next symbol scanned is an operand, it may be immediately appended to the postfix string.
3. If the next symbol is an operator,
   i. Pop and append to the postfix string every operator on the stack that
      a. is above the most recently scanned left parenthesis, and
      b. has precedence higher than or is a right-associative operator of equal precedence to that of the new operator symbol.
   ii. Push the new operator onto the stack.
4. When a left parenthesis is seen, it must be pushed onto the stack.
5. When a 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.
6. When the infix string is completely scanned, the stack may still contain some operators. [Why are there no parentheses on the stack at this point?] All the remaining operators should be popped and appended to the postfix string.

Examples

Here are two examples to help you understand how the algorithm works. Each line below 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: 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: (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