Transform an infix expression to postfix notation

Suppose Q is an arithmetic expression in infix notation. We will create an equivalent postfix expression P by adding items to on the right of P. The new expression P will not contain any parentheses.

We will use a stack in which each item may be a left parenthesis or the symbol for an operation.

```
Start with an empty stack.  We scan Q from left to right.

While (we have not reached the end of Q)
   If (an operand is found)
      Add it to P
   End-If
   If (a left parenthesis is found)
      Push it onto the stack
   End-If
   If (a right parenthesis is found)
      While (the stack is not empty AND the top item is
             not a left parenthesis)
         Pop the stack and add the popped value to P
      End-While
      Pop the left parenthesis from the stack and discard it
   End-If
   If (an operator is found)
      If (the stack is empty or if the top element is a left
          parenthesis)
         Push the operator onto the stack
      Else
         While (the stack is not empty AND the top of the stack
                is not a left parenthesis AND precedence of the
                operator <= precedence of the top of the stack)
            Pop the stack and add the top value to P
         End-While
         Push the latest operator onto the stack
      End-If
   End-If
End-While
While (the stack is not empty)
   Pop the stack and add the popped value to P
End-While
```

Notes:

- At the end, if there is still a left parenthesis at the top of the stack, or if we find a right parenthesis when the stack is empty, then Q contained unbalanced parentheses and is in error.

EVALUATE A POSTFIX EXPRESSION

Suppose P is an arithmetic expression in postfix notation. We will evaluate it using a stack to hold the operands.

```
 Start with an empty stack.  We scan P from left to right.

 While (we have not reached the end of P)
    If an operand is found
       push it onto the stack
    End-If
    If an operator is found
       Pop the stack and call the value A
       Pop the stack and call the value B
       Evaluate B op A using the operator just found.
       Push the resulting value onto the stack
    End-If
End-While
Pop the stack (this is the final value)
```

Notes:

- At the end, there should be only one element left on the stack.
- This assumes the postfix expression is valid.

How can this be implemented?

Work like this is usually done by an assembler, compiler or interpreter. A programmer uses an expression in her or her code, and evaluating it is someone else's problem.

Suppose it is our problem (maybe we are writing an interpreter). The interpreter is reading a line at a time from a file as a string, such as

```
A = ((B + C) / 3 - 47 % E) * (F + 8)
```

The string needs to be parsed--that is, we need to break it up into substrings, each of which is one meaningful part. These substrings are often called tokens. The tokens are separated by spaces, in many cases, but also a token ends if we find a left or right parenthesis or the symbol for an operator. Thus for instance, in the above example, we have "E)", and this consists of two tokens "E" and ")". Bear in mind that the symbol for an operator can be more than one character.

We then have a list of tokens, perhaps in an array or a linked list. Somewhere we will have an Evaluate function which takes such a list as an argument and returns a numeric value.