Converting Infix Expressions to Postfix

First consider the problem of parsing an expression from infix to postfix form. Our three-step procedure is not easily adaptable to machine code. Instead, we will use an algorithm that has as its essential data structures the following:

1. A queue, Infix, containing the infix expression. This expression consists of individual *tokens*, which may be subdivided into the following categories:
   a. Operator tokens, such as +, −, *, and /. We will assume the existence of a boolean-valued ValidOperator function, which receives a token and returns true if that token is a valid operator for the expressions we are considering.
   b. Operand tokens. These may be thought of as variables and constants defined according to the rules of a particular language. We will assume the existence of a boolean-valued function ValidOperand, which receives a token and returns true if that token is a valid operand.
   c. Left and right parentheses tokens, denoted by LeftParen and RightParen, respectively.
   d. A special EndToken, which delimits the end of the infix expression.
2. A stack, OperatorStack, which may contain operator tokens, LeftParen, RightParen, and EndToken.
3. A queue, Postfix, which contains the final postfix expression.

For the moment, we will consider only expressions that involve the operands ÷, −, /, and *. In the exercises and programming problems, you will consider some variations introduced by allowing additional operators. The description of the algorithm is as follows:

1. Define a function InfixPriority, which takes an operator, parenthesis, or EndToken as its argument and returns an integer as specified below:

Token	*	/	÷	−	LeftParen	RightParen	EndToken
Returned value	2	2	1	1	3	0	0

   This function reflects the relative position of an operator in the arithmetic hierarchy and is used with the function StackPriority (defined in step 2) to determine how long an operator waits in the stack before being enqueued on the postfix expression.
2. Define another function StackPriority, which takes the same possibilities for an argument and returns an integer as specified below:

Token	*	/	+	−	LeftParen	RightParen	EndToken
Returned value	2	2	1	1	3	undefined	0

   This function is applied to operators in the operator stack and returns their priority in the arithmetic hierarchy—a value that is to be compared to the infix priorities of incoming operators from the infix queue. The result of this comparison determines whether or

not an operator waits in the stack or is enqueued on the postfix expression.

3. Initialize OperatorStack by pushing EndToken.
4. Dequeue the next Token from the infix expression.
5. Test Token and
   - 5.1 If Token is an operand, enqueue it on the Postfix expression.
   - 5.2 If Token is a RightParen, then pop entries from OperatorStack and enqueue them on Postfix until a matching LeftParen is popped. Doing this ensures that operators within a parenthesized portion of an infix expression will be applied first, regardless of their priority in the usual arithmetic hierarchy. Discard both left and right parentheses.
   - 5.3 If Token is EndToken, pop all entries that remain on the stack and enqueue them on the Postfix queue.
   - 5.4 Otherwise, pop from the stack and enqueue on the Postfix queue operators whose StackPriority is greater than or equal to the InfixPriority of Token. This comparison, keying on the priority of Token from the Infix queue and operators that have previously been pushed onto the OperatorStack, ensures that operators are applied in the right order in the resulting Postfix queue. After popping these operators, push Token.
6. Repeat 4 and 5 until Token is the delimiting EndToken.

The key to the algorithm is the use of the stack to hold operators from the infix expression that appear to the left of another given operator even though that latter operator must be applied first. The defined functions InfixPriority and StackPriority are used to specify this priority of operators and the associated pushing and popping operations. This entire process is best understood by carefully tracing through an example.

Example 5.1 Parse the infix expression

$$A * B + (C - D/E) \#$$

into its equivalent postfix form. Trace the contents of the operator stack and the postfix string as each token is processed. Here the symbol # is used for EndToken. The solution to this problem is presented in Table 5.1. In this table, the parenthesized numbers in the Commentary column refer to subcases of step 5 in the preceding algorithm. We only show the content of a data structure when it changes.

The following PSEUDO procedure fully implements our algorithm. Note how the use of functional parameters ValidOperator, ValidOperand, InfixPriority, and StackPriority make the algorithm completely general. That is, by passing in appropriately defined functions for these parameters, we can control the parse in a variety of different ways—allowing different definitions for operators, operands, and the hierarchy of operations. You will explore ways of doing this in the exercises and problems.