You should be able to convince yourself that this algorithm works. It is clearly linear and actually makes only one pass through the input. It is thus on-line and quite fast. Extra work can be done to attempt to decide what to do when an error is reported—such as identifying the likely cause.

Postfix Expressions

Suppose we have a pocket calculator and would like to compute the cost of a shopping trip. To do so, we add a list of numbers and multiply the result by 1.06; this computes the purchase price of some items with local sales tax added. If the items are 4.99, 5.99, and 6.99, then a natural way to enter this would be the sequence

$$4.99 + 5.99 + 6.99 * 1.06 =$$

Depending on the calculator, this produces either the intended answer, 19.05, or the scientific answer, 18.39. Most simple four-function calculators will give the first answer, but many advanced calculators know that multiplication has higher precedence than addition.

On the other hand, some items are taxable and some are not, so if only the first and last items were actually taxable, then the sequence

$$4.99 * 1.06 + 5.99 + 6.99 * 1.06 =$$

would give the correct answer (18.69) on a scientific calculator and the wrong answer (19.37) on a simple calculator. A scientific calculator generally comes with parentheses, so we can always get the right answer by parenthesizing, but with a simple calculator we need to remember intermediate results.

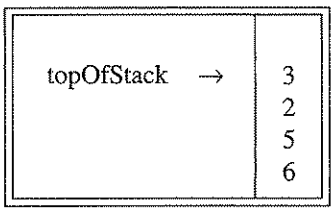
A typical evaluation sequence for this example might be to multiply 4.99 and 1.06, saving this answer as A_1. We then add 5.99 and A_1, saving the result in A_1. We multiply 6.99 and 1.06, saving the answer in A_2, and finish by adding A_1 and A_2, leaving the final answer in A_1. We can write this sequence of operations as follows:

$$4.99\ 1.06 * 5.99 + 6.99\ 1.06\ * +$$

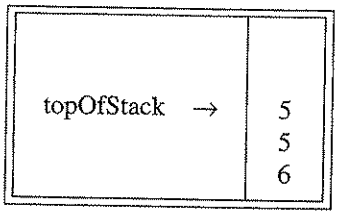
This notation is known as **postfix** or **reverse Polish notation** and is evaluated exactly as we have described above. The easiest way to do this is to use a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers (symbols) that are popped from the stack, and the result is pushed onto the stack. For instance, the postfix expression

$$6\ 5\ 2\ 3 + 8 * + 3 + *$$

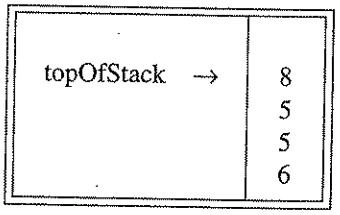
is evaluated as follows: The first four symbols are placed on the stack. The resulting stack is



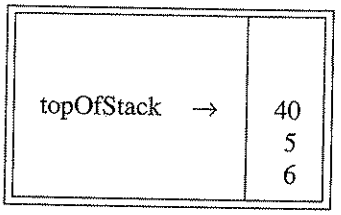
Next a '+' is read, so 3 and 2 are popped from the stack and their sum, 5, is pushed.



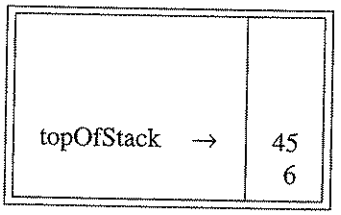
Next 8 is pushed.



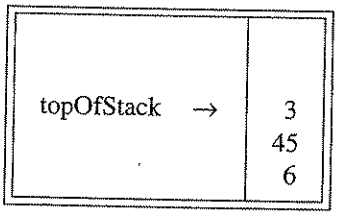
Now a '*' is seen, so 8 and 5 are popped and $5 * 8 = 40$ is pushed.



Next a '+' is seen, so 40 and 5 are popped and $5 + 40 = 45$ is pushed.

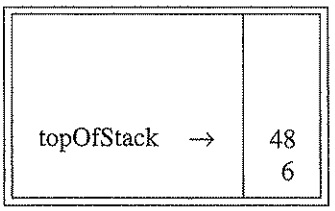


Now, 3 is pushed.

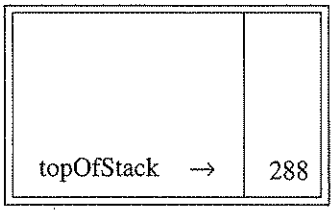


Next '+' pops 3 and 45 and pushes $45 + 3 = 48$.

shed.



Finally, a '*' is seen and 48 and 6 are popped; the result, $6 * 48 = 288$, is pushed.



The time to evaluate a postfix expression is $O(N)$, because processing each element in the input consists of stack operations and thus takes constant time. The algorithm to do so is very simple. Notice that when an expression is given in postfix notation, there is no need to know any precedence rules; this is an obvious advantage.

Infix to Postfix Conversion

Not only can a stack be used to evaluate a postfix expression, but we can also use a stack to convert an expression in standard form (otherwise known as **infix**) into postfix. We will concentrate on a small version of the general problem by allowing only the operators +, *, (,), and insisting on the usual precedence rules. We will further assume that the expression is legal. Suppose we want to convert the infix expression

```
a + b * c + ( d * e + f ) * g
```

into postfix. A correct answer is `a b c * + d e * f + g * +`.

When an operand is read, it is immediately placed onto the output. Operators are not immediately output, so they must be saved somewhere. The correct thing to do is to place operators that have been seen, but not placed on the output, onto the stack. We will also stack left parentheses when they are encountered. We start with an initially empty stack.

If we see a right parenthesis, then we pop the stack, writing symbols until we encounter a (corresponding) left parenthesis, which is popped but not output.

If we see any other symbol (+, *, (), then we pop entries from the stack until we find an entry of lower priority. One exception is that we never remove a (from the stack except when processing a). For the purposes of this operation, + has lowest priority and (highest. When the popping is done, we push the operator onto the stack.

Finally, if we read the end of input, we pop the stack until it is empty, writing symbols onto the output.

The idea of this algorithm is that when an operator is seen, it is placed on the stack. The stack represents pending operators. However, some of the operators on the stack that have high precedence are now known to be completed, and should be popped, as they will no longer be pending. Thus prior to placing the operator on the stack, operators that are on