Postfix Expressions

Arithmetic expressions written in what is called infix notation:

Operand1 op Operand2

Examples

x = 2 * 3   x = 5 + 4 * 3   x = 2 + 4 * 3

Rules indicate which operations take precedence over others.
Use parentheses to override those rules.

x = (2 + 4) * 3   x = 2 + (4 * 3)

Write a program to evaluate the following expression (remember, a program reads tokens left to right 1 at a time)

Example   Example

2 * 3 + 4 =   2 + 3 * 4 =

Reading 1 token from left to right results in the wrong answer.

Another way to write arithmetic expressions is using postfix notation, the 2 operands come before the operation.

Operand1 Operand2 op

Examples (all equations will use 1 digit numbers)
2 3 * 4 + =   2 3 4 * + =   3 4 + 5 1 + * =   5 4 3 + + 6 3 / + =   8 2 / 4 3 2 * + - =

Advantages
- When you read an operation, you can immediately perform it
- No need for parentisis

With postfix notation, it is possible to use a stack to find the overall value of an infix expression by first converting it to postfix notation.

Infix	Postfix
5 * (6 + 2) - 12 / 4:	5 6 2 + * 12 4 / -

There are two algorithms involved. One converts an infix expression to postfix form, and the other evaluates a postfix expression. Each uses a stack.

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)
{
      Read in next token
      If an operand is found
      {
        push it onto the stack
      }

      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
      }
}

Pop the stack (this is the final value)
```

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

Examples

Use a Stack to evaluate the following postfix expressions

3 4 5 + *	8 3 – 2 5 + *	9 2 + 7 3 8 - +	
5	5	5	5
4	4	4	4
3	3	3	3
2	2	2	2
1	1	1	1
0	0	0	0
Top=______ ___________	Top=____ _________ _____	Top=___ ________ _______	Top=__ ________ _____

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
      If (a left parenthesis is found)
          Push it onto the stack
      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
      Pop the left parenthesis from the stack and discard it
  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
           }
           Push the latest operator onto the stack
  }
}
```

```
    End-While
    While (the stack is not empty)
        Pop the stack and add the popped value to P
```

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.

Resources

http://www.youtube.com/watch?v=uh7fD8WiT28

http://www.cs.nthu.edu.tw/~wkhon/ds/ds10/tutorial/tutorial2.pdf