Infix, Postfix, and Prefix Notation

Most of the operations we use are binary, such as \*, +, -, /, %. These operators must be used with two operands (the values that are being operated on).

Infix notation places the operators (\*, +, -,…) in between the two operands. Example: 8 + 5

Postfix notation places the operators after (post) the operands. Example: 8 5 +

Prefix notation places the operators before (pre) the operands. Example: + 8 5

Using postfix notation to write expressions makes the evaluation of the expressions much simpler when processing them in a program. There is never any doubt about what operation occurs when. For example, in infix notation we write 4 / 2 + 5 \* 9. The program we use must evaluate the expression AND know what the precedence is for each operation. Without knowing the precedence, the program would evaluate:

4 / 2 + 5 \* 9

4/2 gives us 2

add the 5 to the 2 to get 7

and then multiply the 7 by 9 --- generating an incorrect answer

The equivalent postfix expression is 4 2 / 5 9 \* +

To evaluate that expression: 4 2 / gives us 2

5 9 \* gives us 45

2 + 45 gives 47, the correct answer.

Infix expressions may use parentheses to change the order of operations. Example: (5 + 3) \* 2

The equivalent postfix expression is 5 3 + 2 \* which does not need any parenthesis to force a change in the order of evaluation.

Evaluate these postfix expressions:

1. 2 5 + 3 \* \_\_\_\_\_\_\_\_\_\_\_\_\_

1. 2 5 + 6 3 + \* \_\_\_\_\_\_\_\_\_\_\_\_\_\_
2. 2 6 2 - 5 \* + \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Convert these infix expressions to postfix:

1. 8 + 4 \* 2 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
2. (x + y) \* (z - w) / k\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
3. 3 \* (x + 2) - y / z \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
4. Write the following method using the algorithm specified. Write a driver **application, ExpressionEval,** that asks the user to enter an **infix** expression and use it to test the method by **displaying the contents** of the queue returned from the method:

String convertToPostFix (String infix)

Use the following algorithm to convert an infix expression to a postfix expression. To make the problem less complicated, we will only allow one-digit integer values in the expression and restrict the operators to \*, /, -, and +. The precedence of the operators is \* and / have the same precedence which is higher than + and -, which have the same precedence. Of course, all expressions are evaluated from left to right.

Algorithm to convert an expression from infix to postfix with operator precedence implemented:

The expression to be evaluated is passed to the method as a String. The result returned from the method is a queue containing the expression; numeric digits and operators are all stored as Strings within the queue. For example: the input infix String could be "x + y \* 5", so the resulting queue would be nodes containing these values: "x", "y", "5", "\*", "+".

* + Create a postfix queue (String) to be used as our result.
  + Create a process stack to be used to manipulate the infix values.
  + Examine the infix string from left to right, one character at a time, until all characters have been examined. Use the substring method to pull out each character and treat it as a String, which I will call ch in this algorithm.
    1. if ch is a digit ("0", "1", …,"9") then get the entire integer and add it to the postfix queue.
    2. if ch is a letter (‘a’, ‘b’,…, ‘z’) then add it to the postfix queue
    3. if ch is a space, discard it.
    4. if ch is a left parentheses "(", then push it onto the process stack.
    5. if ch is an operator ("\*"," +", "-", "/") then
       1. pop the operators (if any) from the process stack while they have equal or higher precedence than the current operator in ch and add those to the postfix queue.
       2. push the ch operator onto the process stack.
    6. if ch is a right parentheses ")", then
       1. pop the operators from the stack until the matching left parentheses "(" is the next value on the top of the stack. Add the operators that you pop from the stack onto the postfix queue as you process them.
       2. pop and discard the ")" from the top of the stack.
  + After all of the infix string has been processed, pop the elements still remaining in the process stack and place them in the postfix queue.

Evaluation of the postfix expression cannot be achieved until the program is executed as we do not know what the values for the variables will be until that time. You must write the appropriate SML commands to accomplish the operations using the available memory in the data section to implement your homegrown stack. The stack area must be safeguarded and not used by any other process in order to make loops in the code work properly.

Example:

10 input x symbol table SML lowSML

20 input y 10 line 0 0: 1099 90:

30 let z = x + y x var 99 1: 1098 91:

40 input k 20 line 1 2: 2099 92:

50 let z = z + k \* 2 y var 98 3: 2196 93: 2

60 print z 30 line 2 4: 2098 94: k

70 end z var 97 5: 2195 95: st y

40 line 11 6: 2096 96: st x

k var 94 7: 3095 97: z

50 line 12 8: 2196 98: y

2 literal 93 9: 2096 99: x

60 line 10: 2197

70 line 11: 1094

12:

13:

14:

15:

16: