Extension (that means extra-credit: not required)

Part I.

A postfix expression lists operators after operands (post = after). The infix expression 5 + 7 becomes 5 7 + in postfix. The advantage of postfix is that it can be evaluated more easily than infix. Your job is to evaluate a postfix expression stored as a string.

You may assume that each character in the string is either a digit or an operator (+, -, \*, /). You may assume that each operand is only a single digit. You do not have to input the expression to be evaluated; for instance, in main you might have a command like:

String s = "57+";

int x = evaluate(s);

System.out.println(s + " = " + x);

where the output should show "57+ = 12".

The algorithm for evaluating a postfix expression involves a stack. Loop over the expression (remember, it's stored as a string). For each symbol, or character in this case, do one of two things:

If the symbol is an operand, just push it on the stack.

If the symbol is an operator, pop the top two operands from the stack,

apply the appropriate operation, and push the result back on the stack.

At the end of this process the result of the entire expression will be the only item left on the stack. You may assume that the string contains a valid postfix expression (our algorithm does not do error checking). Use an ap.ArrayStack object.

Trace the algorithm for "354\*+7\*", which equals 161.

Part II is on the next page….

Part II.

No one wants to write out an expression in postscript, so we want to be able to translate an infix expression to postfix. Given the input string "(3+5\*4)\*7" we want to produce "354\*+7\*", which of course we now know how to evaluate. Assume you'll be given a valid infix expression. To convert to postscript, loop over the string. For each character:

Scan through an expression, getting one token at a time.

1) Fix a priority level for each operator. For example, from high to low:

3. - (unary negation)

2. \* /

1. + - (subtraction)

Thus, high priority corresponds to high number in the table.

2) If the token is an operand, do not stack it. Pass it to the output.

3) If token is an operator or parenthesis, do the following:

-- Pop the stack until you find a symbol of lower priority number than the current one. An incoming left parenthesis will be considered to have higher priority than any other symbol. A left parenthesis on the stack will not be removed unless an incoming right parenthesis is found.

The popped stack elements will be written to output.

--Stack the current symbol.

-- If a right parenthesis is the current symbol, pop the stack down to (and including) the first left parenthesis. Write all the symbols except the left parenthesis to the output (i.e. write the operators to the output).

-- After the last token is read, pop the remainder of the stack and write any symbol (except left parenthesis) to output.

Trace the algorithm for "(3+5\*4)\*7", which equals "354\*+7\*" in postfix.

Example on next page…..

Convert A \* (B + C) \* D to postfix notation.

|  |  |  |  |
| --- | --- | --- | --- |
| **Move** | **Curren Ttoken** | **Stack** | **Output** |
| 1 | A | empty | A |
| 2 | \* | \* | A |
| 3 | ( | (\* | A |
| 4 | B | (\* | A B |
| 5 | + | +(\* | A B |
| 6 | C | +(\* | A B C |
| 7 | ) | \* | A B C + |
| 8 | \* | \* | A B C + \* |
| 9 | D | \* | A B C + \* D |
| 10 |  | empty |  |