Course: EEE2020 Data Structure

Homework 3 **Due date: 2014.5.8 before class starts**

Things you should do for this homework.

(1) **Upload *Report File* on YSCEC** (Report file should contain 'Flowchart', 'source code' and 'results')
(2) **Upload all *source codes* and report on YSCEC**.
(3) ***Print your Report File***, and ***hand it in*** before class

Late submissions will NOT be accepted.
This homework is not a group assignment. Group submissions will NOT be accepted. You need to submit all text files which are used as inputs or outputs of your program.

1. Parenthesis Matching
We need to design a program to check if an equation has balanced parentheses, that is, (,). The equation includes many arithmetic operations, and parentheses. This program checks whether the parentheses are matched or not.

- Parentheses are opened by a ')' and closed by a ')'.

A formula is said well-matched when the following conditions are met:

1. Every parenthesis has a corresponding pair. ')' corresponds to '('.
2. Every parenthesis pair is opened first, and closed later.

Input
Input is an equation including parentheses. This equation includes numbers, arithmetic operations, and parentheses. This equation may have some faults on arithmetic operation, but you do not have to care about the faults for this matching test.

Output
For each case, print a single line "**Matched**" when the formula is well-matched; print "**Not Mathced**" otherwise.
ex)

Input	Output
(3+2)*(7-1/2)	Matched
)2^1-13(7*1)	Not Matched
(32+-1/11)/*(1-13)	Matched
((1+13)-(1/13)	Not Mathced

2. Infix to Postfix Conversion

Infix, Postfix (Reverse Polish notation) notations are different but equivalent ways of writing expressions. The following examples demonstrate the difference.

Infix	Postfix	Notes
A * B + C / D	A B * C D / +	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	divide C by D, add B, multiply by A

Infix to postfix conversion algorithm is as follows.

Infix to Postfix Conversion

In normal algebra we use the infix notation like a+b*c. The corresponding postfix notation is abc*+. The algorithm for the conversion is as follows :

- Scan the Infix string from left to right.
- Initialise an empty stack.
- If the scannned character is an operand, add it to the Postfix string. If the scanned character is an operator and if the stack is empty Push the character to stack.
- If the scanned character is an Operand and the stack is not empty, compare the precedence of the character with the element on top of the stack (topStack). If topStack has higher precedence over the scanned character Pop the stack else Push the scanned character to stack. Repeat this step as long as stack is not empty and topStack has precedence over the character.
- Repeat this step till all the characters are scanned.
- (After all characters are scanned, we have to add any character that the stack may have to the Postfix string.) If stack is not empty add topStack to Postfix string and Pop the stack. Repeat this step as long as stack is not empty.
- Return the Postfix string.

Example :

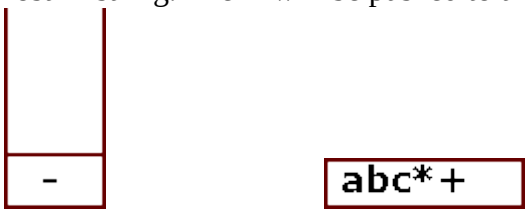
Let us see how the above algorithm will be implemented using an example.

Infix String : a+b*c-d → Postfix String : a b c * + d -

Initially the Stack is empty and our Postfix string has no characters. Now, the first character scanned is 'a'. 'a' is added to the Postfix string. The next character scanned is '+'. It being an operator, it is pushed to the stack.

Next character scanned is 'b' which will be placed in the Postfix string. Next character is '*' which is an operator. Now, the top element of the stack is '+' which has lower precedence than '*', so '*' will be pushed to the stack.

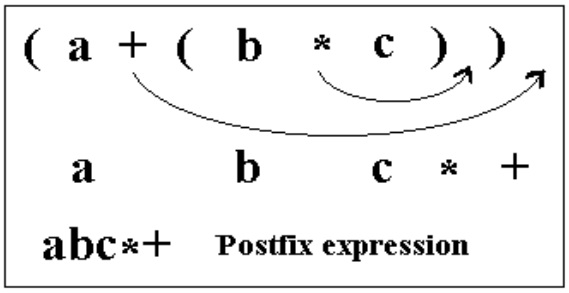
The next character is 'c' which is placed in the Postfix string. Next character scanned is '-'. The topmost character in the stack is '*' which has a higher precedence than '-'. Thus '*' will be popped out from the stack and added to the Postfix string. Even now the stack is not empty. Now the topmost element of the stack is '+' which has equal priority to '-'. So pop the '+' from the stack and add it to the Postfix string. The '-' will be pushed to the stack.



Next character is 'd' which is added to Postfix string. Now all characters have been scanned so we must pop the remaining elements from the stack and add it to the Postfix string. At this stage we have only a '-' in the stack. It is popped out and added to the Postfix string. So, after all characters are scanned, this is how the stack and Postfix string will be :

End result :
Infix String : a+b*c-d

Postfix String : abc*+d-
Using the conversion algorithm, write down a program to read a line of Infix expression and convert it to Postfix expression and print out converted Postfix expression and the result of input.



The input contains a collection of error-free infix arithmetic expressions.
The Postfix notation doesn't need to write down parentheses such as '(' or ')'.

Sign
We need to consider the sign of integer. To express negative integer, we use the exclamation mark as negative sign symbol. For example, !4 is equal to -4. The sign – is only used as a binary operator over two operands. The sign ! indicates a unary operator. We would like to distinguish the unary and binary operators to make the problem simple.

Operators
We consider the four fundamental arithmetic operators. Additionally, we consider the power operator. '^' is a symbol of the power operator. For example, 2^3 is equal to 8. If this operators are cascaded, the usual rule is to work from the top down, thus a^b^c = a^(b^c). For example, 2^1^2 is equal to 2 (not 4). We allow only natural numbers for power calculation.

Input	Output
2+3*4	Postfix : 2 3 4 * + Ans : 14
(1+2)+7	Postfix : 1 2 + 7 + Ans : 10
3 + 2 * 3 – 4	Postfix : 3 2 3 *+ 4- Ans : 5
8 / 4 – 10 + 20 * 3 – 15 * 2	Postfix : 8 4 / 10 – 20 3 * + 15 2 * - Ans : 22
5 * 2 ^ 4 + 3 * 2 – 4 * 8	Postfix : 5 2 4 ^ * 3 2 * + 4 8 * – Ans : 54
3^2^2 – 4*10 – 4/10 +4	Postfix : 3 2 2 ^ ^ 4 10 * - 4 10 / - 4 + Ans : 44.6
2^2^1^2 - 2*!7	Postfix : 2 2 1 2 ^ ^ ^ 2 7 ! * - Ans : 30
3 + !2^(3 + !1) +3	Postfix : 3 2! 3 1! + ^ + 3 + Ans : 10
1^2 + 3 - !1	Postfix : 1 2! ^ 3 + 1! – Ans : 5
3^!2^2 - 81	Postfix : 3 2! 2 ^ ^ 81 – Ans : 0

You need to use a stack to evaluate a postfix notation for the arithmetic result. The procedure is given in the textbook or lecture note. Refer to the algorithm.

Tasks to be completed:

You need to execute the parenthesis matching program first, and then run a program to convert an infix notation into a postfix notation. Then you need to run a program to evaluate the postfix notation. Three programs as module functions should be provided together and be run to complete the above goal. Each algorithm is given in details in the textbook. All the basic codes are given in the textbook and the lecture note. You can use the codes given in the textbook. The codes are not sufficient to complete the above process and you need to modify the codes appropriately.

You need to provide the results after the parenthesis matching program, after conversion into a postfix notation and then after the evaluation over the postfix notation. You need three different stacks for the above whole program. If the parenthesis matching is not successful, the program cannot follow the next steps. You need to demonstrate the program run for many expression examples including the above examples.

For this homework, you need to write a report in English or in Hangeul for the above results and your program code (12pt font is recommended, 3-4 pages would be appropriate for the report not including the program code. Please follow the instructions given in the course syllabus) and add the source code and result at the end. Please give us your algorithm analysis with asymptotic notations.