# CS2351 Data Structures

Homework 4

Due: 11:59 pm, May 30, 2011

In this assignment, you are asked to write a program which makes use of the stack to convert an expression from infix notation to postfix notation, using only linear time. You should submit the source code of your program to the iLMS system (`http://lms.nthu.edu.tw/`) before the deadline. To grade your assignment, we will arrange a 15-minute session with you for a demonstration of your program, at General Building II Room 734 on either June 02 or June 03. Please make sure that the source code can be compiled without any error. Late submission will get at most 60% for the grade.

**Problem Description**

In a mathematical expression, there are well-defined rules that tell us which operator should be applied before the other. Intuitively, each operator is associated with a *precedence order*, and the higher the order, the earlier that we should apply the operator. For instance, $\wedge$ (the power) has higher order than $*$ and /, while $*$ and / have higher order than $+$ and $-$. When the expression may contain a contiguous sequence of operators with the same precedence, we need to specify whether the earlier one should be applied first or the latter one should be applied first.

For instance, in the expression $5 + 3 - 4 - 2$, all operators are of the same precedence, and we know that the evaluation in this case should be from left to right. We thus call $+$ or $-$ a *left associative* operator. Similarly, $*$ and / are also left associative. On the other hand, in the expression $2 \wedge 3 \wedge 2$, we know that the evaluation should be from right to left (that is, the value is $2 \wedge (3 \wedge 2)$ instead of $(2 \wedge 3) \wedge 2$). We thus call $\wedge$ a *right associative* operator.

In other words, even though operators may start with the same precedence, if an operator is left associative and if it appears before another one with the same precedence (like $5 + 3 - 4$), then the earlier one ($+$ in the example) should be applied first, and thus has a higher precedence. In contrast, if an operator is right associative and if it appears before another one with the same precedence (like $2 \wedge 3 \wedge 2$), then the latter on should be applied first.

Using the above concept of precedence, the pseudocode in Algorithm 1 can convert an expression from infix notation to postfix notation in linear time. The correctness shall be explained during the tutorial.

**Your Task**

Given an expression in the infix form, write a program to compute the corresponding postfix form. Your program should read the input from the file named `input.txt` and output the results to the file named `output.txt`. The following is an example of the input file. The first line contains only one number which indicates the total number of expressions to be converted. Each of the following line is a string (or character array) represents an infix expression, with terms separated by a space. Operands are always an unsigned 32-bit integer, and operators include $+$, $-$, $*$, /, $\wedge$ with their usual precedence orders as described above. The output file contains the expressions in postfix form, each being printed on a separate line, and each term separates from the other with a space. See the following for an example.

**Example Input:**

```
3
1 + 2 + 3
```

32 + 6 * 3 - 45
5 * 2 ^ 3 ^ 2

**Example Output:**
1 2 + 3 +
32 6 3 * + 45 -
5 2 3 2 ^ ^ *

**Bonus**

A 10% bonus will be given if your program can handle the case with parentheses. For instance, given the expression 4 * ( 3 - ( 2 + 5 ) ) - 1, the output should be 4 3 2 5 + - * 1 -. This requires only minor modification to the code. To know more on how to do that, read the reference book by Horowitz and others, or consult with the tutors during the tutorial.

**Algorithm 1** Infix-to-Postfix Conversion

**Input:** An expression in infix form
**Output:** The expression in postfix form

```
1: Initialize an empty stack S
2: while there are unprocessed terms in the expression do
3:    Get the next term T from the expression
4:    if T is an operand then
5:       Output T immediately
6:    else
7:       {/* T is an operator */}
8:       if T has higher precedence than Top(S) then
9:          Push T to S
10:      else
11:         while Top(S) has a higher precedence than T do
12:            Output Top(S) and Pop(S)
13:         end while
14:         Push T to S
15:      end if
16:   end if
17: end while
18: {/* Output remaining operators in S when all terms are processed*/}
19: while S is not empty do
20:   Output Top(S) and Pop(S)
21: end while
```