Programming Design, Spring 2014
Homework 11

Instructor: Ling-Chieh Kung
Department of Information Management
National Taiwan University

Submission. To submit your work, please upload the following file to the online grading system at http://lckung.im.ntu.edu.tw/PD/.

1. Your .cpp file for Problems 1.

Each student must submit her/his individual work. No hard copy. No late submission. The due time of this homework is ***8:00am, June 4, 2013***. Note that it is due on ***Wednesday***!

Problem 0

(0 point) Please read Sections 5.20–5.22 and Chapter 19 of the textbook.[1] In any case, I strongly suggest you to read the textbook thoroughly before you start to do this homework.

Problem 1

(100 points) In our daily life, basic mathematical expressions like $1+2\times3$ or $4\times(5-6)/8$ are said to be written as *infix expressions*, i.e., an operator (e.g., $-$) is placed in between the two operands it operates on (e.g., 5 and 6). While infix expressions are intuitive and easy to be processed by humans, it is not natural for computers. In particular, we need to define the precedence rules to make these expressions unambiguous. To see this, note that

$$1+2\times3 \quad \text{and} \quad (1+2)\times3$$

are different. In the former, we know that 2×3 should be done first. The fact that multiplications and divisions should be evaluated before additions and subtractions is a part of the precedence rule. To enforce the order of evaluations we want, we use parentheses. The pair of parentheses in $(1+2)\times3$ ensures that $1+2$ is evaluated first.

For computers, *prefix expressions* are better because there is no ambiguity. In a prefix expression, an operator is placed in front of the two operands. For example, $1+2$ will be written as `+ 1 2` and 2×4 will be written as `* 2 4`.[2] How about $1+2\times3$? It will be

```
+ 1 * 2 3.
```

The first operator `+` needs to operands. While the first one is nothing but `1`, the second one is the outcome of `* 2 3`, i.e., 2×3. A prefix tree can be used to visualize the structure of a prefix expression. Figure 1 is the prefix tree for `+ 1 * 2 3`. In a prefix tree, each internal node is an operator and each leaf is a number. An operation can be evaluated once the two operands of the operator are both ready. In this example, the addition operation can be evaluated only after the multiplication operation is evaluated. Therefore, the final outcome is 1 plus 2×3, which is 7.

As another example, the prefix expression (with it prefix tree in Figure 2) of $4-5\times6+7$ is

```
+ - 4 * 5 6 7.
```

A computer evaluates this expression in the following steps:

[1] The textbook is *C++ How to Program: Late Objects Version* by Deitel and Deitel, seventh edition.
[2] For ease of exposition, all the prefix expressions are typed in the typewriter font. Note that `*` means $\times$.

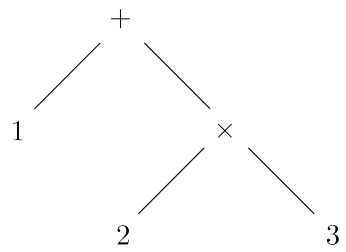

Figure 1: `+ 1 * 2 3`

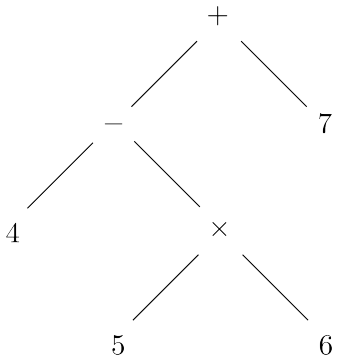

Figure 2: `+ - 4 * 5 6 7`

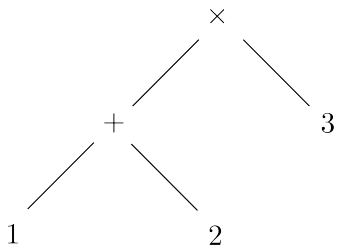

Figure 3: `* + 1 2 3`

1. As it first sees an operator `+`, it looks for the two operands for `+`. As it sees that the second term is `-`, it knows that the first operand of `+` is the outcome of a subtraction operation, so it should put `+` asides and evaluate the subtraction first.
2. To evaluate the subtraction operation, it again looks for two operand. The first one is `4`, and the second one is the outcome of a multiplication operation! So it puts `-` and `4` asides and switch to evaluate the multiplication operation.
3. The multiplication operation needs two operands, which are `5` and `6`. The outcome is 30.
4. Now the computer goes back to the subtraction operation with `4` and `30` as the two operands. The outcome is `-26`.
5. Finally, the computer goes back to the addition operation with `-26` as the first operand. Note that at this moment, the first six terms (from `+` to `6` have all been used. Therefore, the second operand is the last term, `7`. The outcome of the addition operation, or the whole expression, is `-19`.

In general, when a computer looks for operands for an operator, it first checks the term right after the operator. If it is an operand, then it is the first operand. Otherwise, it is an operator and the computer faces a *subexpression* whose solution can be used for evaluating the original expression! This is exactly a recursive process. For the expression `+ - 4 * 5 6 7`, there are two subexpressions: `- 4 * 5 6` and `* 5 6`. The evaluation of each subexpression is the same as the evaluation of the original expression, except that the subexpression is shorter. Then if a function returns the outcome of evaluating the original expression, it can also be used for returning the outcome of evaluating each subexpression. In short, the evaluation of each operation can be done with two recursive calls for calculating the two operands. The base case of this recursion is, of course, a single number.

How about infix expressions with parentheses? For example, the prefix expression (with it prefix tree in Figure 3) of $(1 + 2) \times 3$ is

`* + 1 2 3`.

Why? Let's evaluate this prefix expression. First, we see `*`, so we look for two operands. The first operand is the outcome of `+ 1 2`, and the second operand is `3`. Therefore, `+ 1 2` will be evaluated before the multiplication operation can be evaluated. This is exactly what happens with the pair of parentheses. In summary, with prefix expressions, there is no need to define precedence rules for operators.

In this problem, you will be given infix expressions for you to evaluate their outcomes. You may still choose your way for implementation. However, if recursion is a natural way, why not try it?

Input/output formats

The input consists of 35 lines of characters `+`, `-`, `*` and integers 0, 1, ..., 9. In each line, these characters form a prefix expression. We say a term is either an operator or a number. To make the problem

easier, all numbers are integers within 0 and 9 (so no number is negative or having two or more digits). Two terms are separated with a white space. You may assume that the expression is indeed a correct prefix expression. Your program should output the outcomes of all the operations in a given expression according to the order that they are evaluated. When an operation's operands both require evaluating subexpressions, the first subexpression (for the operator at the left) should be evaluated first. Two outcomes (which will both be integers) should be separated with a white space.

Below are some examples:

- Input: `+ + - + 1 2 3 4 5`. Output: `3 0 4 9`.
- Input: `+ + - * 1 2 3 + 4 5 6`. Output: `2 -1 9 8 14`. Note that when we evaluate the subexpression `+ - * 1 2 3 + 4 5`, the two operands are `- * 1 2 3` and `+ 4 5`. Your program should evaluate `+ 4 5` only after `- * 1 2 3` has been evaluated. That is why it should output `2 -1` before `9`.
- Input: `- 1 + 2 * 3 - 4 5`. Output: `-1 -3 -1 2`.

Grading criteria

- 70 points are given based on the correctness of your output. Each correct line of output gives you two points. The input will be organized in the following way:
  - For the first five lines, there are one operator and two operands.
  - For the sixth to tenth lines, there are two operators and three operands. Moreover, the first two terms will be operators and the last three terms will be operands.
  - For the eleventh to fifteenth lines, there are n operators and $n+1$ operators (of course, n will not be give to you). The first n terms are operators and the last $n+1$ terms are operands.
  - For the last twenty lines, everything is possible.
- 30 points will be based on how you write your program, including the logic and format. Please try to write a robust, efficient, and easy-to-read program.