CS29003 ALGORITHMS LABORATORY
ASSIGNMENT 1
Date: 12 – August – 2021

Your friend is an investment banker. His job requires him to do a lot of calculations. For instance, for a particular investment plan: if he deposits 1000 Rs. for 3 years, in first year he gets 2% interest, while in the next two years, he gets 5% and 3% interest, respectively. The mathematical utterance corresponding to this calculation would look like this:

$$1000 * (1 + 0.02) * (1 + 0.05) * (1 + 0.03)$$

He has to actually write many such utterances to compare against various plans available in the market. He is looking for some way to reduce his work, even if marginally!

He comes to you for help. You think about it a lot and come up with a **new notation** of expressing these mathematical utterances which would save him some keystrokes.

Normal Notation: Mathematical utterances are made of operators (+, *, etc.) and operands. The operands may be numbers, variables or (recursively) smaller mathematical utterances. The utterances can be written in a variety of notations. The normal notation is the one where the operator is written between its operands. Such a notation requires notions of ***precedence*** (indicating, for example, that division has higher precedence than addition, so that $3 + 15/3$ is 8, not 6) and ***associativity*** (indicating, for example, that subtractions are to be done from left to right, so that $10 - 3 - 2$ is 5, not 9). It also normally requires a way to override the precedence rules by using ***parentheses***. For example, expression $7 + 3 * 6$ has value 25, but $(7 + 3) * 6$ has value 60.

Token	Operator	Precedence	Associativity
( )	parentheses	3	left-to-right
* / %	multiplication, division, modulo	2	left-to-right
+ -	addition, subtraction	1	left-to-right

Table 1: Operator Precedence and Associativity

New Notation: In this new notation, the operator is written after both of its operands. For example, you write 756+ to indicate the result of adding 7 and 56. This notation does not require any notions of precedence or associativity, and does not require parentheses. For example, an arithmetic calculation 7 + (32 * 6) is equivalent to 7 32 6 * + in the new notation, but the expression (7 + 32) * 6 is equivalent to 7 32 + 6 *.

Now, your friend is quite happy with this notation and wishes to completely move to this new notation. However, he realizes that he already has quite a few utterances stored in the normal notation. So, he needs to convert those to the new notation. You need to help him by writing a program.

Further, you also decide to help him by writing another program that will take any utterance in new notation, and compute the final value.

Exercise 1: Convert a given mathematical utterance into new notation

In this exercise, you are required to write a program that reads a file titled "input.txt" containing an mathematical utterance on each line written in normal notation, converts it into its corresponding new notation using an ***Operator Stack*** and writes the converted notation on each line in a file titled "part1-output.txt". The utterances that your program needs to handle involve (decimal) integer constants, such as 4 and 57 and operators as detailed in Table 1. **Make sure to include space between tokens (constants/operators) while writing the converted output into the file** in order to ensure that utterances like 4+57 and 45+7 do not have the same notation 457+.

Algorithm and Templates

1. Create an Operator Stack and its associated functions strictly following the given templates.

   ```
   struct operatorNode {
   char data;
   struct operatorNode* next;
   };

   struct operatorStack {
   int size;
   struct operatorNode* top;
   };

   struct operatorStack* createStack();
   bool isEmpty(struct operatorStack* stack);
   char peek(struct operatorStack* stack);
   char pop(struct operatorStack* stack);
   void push(struct operatorStack* stack, char op);
   ```

2. Scan the given arithmetic expression from left to right. For each token t (multi-digit operands like *72, 45* etc. are to be considered) in the input string:

   (a) if (t is an operand): append t onto the end of the output string.
   int isOperand(char* ch);

   (b) else if (t is an operator):

      i. Pop tokens (if any) from the top of the stack while they have equal or higher precedence than t and append them onto the end of the output string. You can also stop if the top of the stack is a left parenthesis, or if the stack is empty.
      int precedence(char op1, char op2);

      ii. Push t into the stack.

   (c) else if (t is a left parenthesis): Push t into the stack.

   (d) else if (t is a right parenthesis):

      i. Pop tokens from the top of the stack and append them onto the end of the output string until a left parentheses is encountered.

      ii. Pop and discard the left parentheses from the stack.

3. Pop tokens (if any) and append them onto the end of the output string until the stack is empty.

For each line in the input file titled “input.txt”, scan the contents (a mathematical utterance) into a character string “exp”, call a function with the following template to convert it into the new notation and finally write it to “part1-output.txt”. **char* convert(const char* exp);**

Exercise 2: Evaluate the mathematical utterances in new notation

In this exercise, you are required to write a program that simulates a calculator. More specifically, the program should take, as input, an utterance in the new notation, and return, as output, the computed value of the given utterance using an ***Operand Stack***.

Algorithm and Templates

1. Create an Operand Stack and its associated functions similar to the previous exercise; only difference being that here you would push operands (integer constants) into the stack instead of operators (characters).
2. Scan the arithmetic expression in the new notation from left to right. For each token t (separated by “space”) in the input string:
   (a) if (t is an operand): Push t into the stack.
   (b) else if (t is an operator):
      i. Pop the top two operands from the stack.
      ii. Apply the operator t to the two popped operands.
      iii. Push the result of the operation into the stack.
3. When there are no tokens left, the stack only contains one item: the final value of the utterance. Pop the stack one final time and write the result into the output file.

For each line in “part1-output.txt”, scan the mathematical utterance in the new notation into a character string “exp”, call a function with the following template to evaluate it and finally write the obtained value to “part2-output.txt”. **int evaluate(const char* exp);**

Sample Input - “input.txt”

```
3 * 7 + 5
45 / 3 * 5
56 * ( 3 + 72 )
11 % 3 - 2
```

Sample Output - “part1-output.txt”

```
3 7 * 5 +
45 3 / 5 *
56 3 72 + *
11 3 % 2 -
```

Sample Output - “part2-output.txt”

26
75
4200
0

File Naming Convention

Please note that the output file names used in this document till now are generic and for explanation purpose only. **Your submissions will not be evaluated unless** you follow the below specified file naming convention for naming your files:

1. Program/Code file Naming Convention :
   <ROLLNO(IN CAPS)>_A<Assign_No>_P<Exercise_No>.c/cpp
   Eg: Exercise 1: 20CS50004_A1_P1.cpp
   Eg: Exercise 2: 20CS50004_A1_P2.cpp
2. Output file Naming Convention :
   <ROLLNO(IN CAPS)>_A<Assign_No>_P<Exercise_No>_output.txt
   Eg: Exercise 1: 20CS50004_A1_P1_output.txt
   Eg: Exercise 2: 20CS50004_A1_P2_output.txt
3. Submit them in a zipped folder named 20CS50004.zip.

Build

1. Compile : g++ 20CS50004_A1_P1.cpp
2. Run: ./a.out or .\a.exe