CS201 – Data Structures and Discrete Math Fall 2014

Programming Project 1

Due: Wednesday, 9/24/13 at 11:59 pm

Infix Expression Evaluator

For this programing project, you will read in a line of input that contains an infix expression and evaluate it. This assignment will require the use of two stacks. One stack for values and one stack for operators.

Both stacks are to be implemented using dynamic arrays. When the program starts, both of these two stack should only have space for 2 items on the stack. When we grow either stack, we will only add space for 2 additional items on the stack. Thus the size of the arrays are to change from 2 to 4 to 6 to 8 to 10 to ….

When writing your code, you MUST place all of the data items needed for each stack in a C struct. These data items must include the following (and may include others if needed).

- the pointer to the dynamic array that actually holds the stack
- the integer variable specifying the current size of the dynamic array
- the integer variable specifying the top of the stack

Since the program requires the use of two stacks (one for values and one for operators), you may need to create two different C structures. (While it possible to use only one type of struct for this project, the most "straight forward" implementation would be to create one struct for the value stack and a second struct for the operator stack.) The instances/variables of the struct(s) MUST be declared as local variables in your C code. They may NOT be global variable. Thus you must pass the variables to the various functions via parameter passing.

You MUST write functions for

- initializing the stack,
- checking if the stack is empty,
- pushing an element onto the stack,
- popping an element off of the stack,
- accessing the top element on the stack, and
- resetting the stack so that it is empty and ready to be used again. Note: a stack that has grown to contain more than two items DOES NOT need to be resized back to contain only 2 items.

The stack may need to grow when pushing an element onto the stack. You may write the grow code as its own function or as part of the push function. When the stack grows, you are to print out a message stating which array is growing and the old and new size of the dynamic array.

Input

The input for this program will come from standard input. Each line of input will be a single infix expression expression. You may assume that each line of input is less than 300 characters long. This allows us to use the fgets() functions from <stdio.h> to read the input into a character array. To get fgets() to read from standard input, the third parameter of fgets() should have the value of **stdin**.

The only exception is if the input on the line just contains the letter q. In this case, quit the program.

To keep things simpler, the values given in the input will only be given as positive integers. This keeps us from worrying about floating point values (division will be integer division) and from confusing the subtraction operator with the negative sign.

Our infix expressions will use the following operators:

Operator	Associativity	Precedence	Operation
( )	Left to Right	Special	Parenthesized Expression
^	Right to Left	Highest	Exponentiation
*, /	Left to Right		Multiplication and Division
+, -	Left to Right	Lowest	Addition and Subtraction

Stack Use Algorithm for Evaluation of Infix Expressions

To evaluate an Infix Expression we need to use two stacks. One stack to hold the values of the expression and one stack to hold the operators of the expression.

When a new expression is read in, both stacks should be cleared. We inspect the expression from left to right. The expression might contain the following information:

- Values (or operands): unsigned integer values
- Operators: ( ) ^ * / + -
- The end of the expression: fgets() will include the newline character \n which we can use to indicate the end of the expression.
- Other stuff: spaces, tabs, other character input

When an error occurs in any of the following steps, your program should print out some description error message and then have your program read in the next line of input.

When "Other stuff" in encountered in the expression, it should be ignored. This is not considered to be an error.

When a digit is encountered in the expression, all adjacent digits are to be used to create a single value. Once the entire value is determined, the value is to be pushed onto the value stack.

When an operator is encountered in the expression, we may need to "process the operator". To "**process an operator**", two values are popped from the value stack, the action of the operator is performed on those two values and push the resulting value onto the value stack. If the value stack empties before two values have been popped, the expression had too many operators and an error has occurred.

The exact steps performed when an operator is encountered differs based on which operator is encountered. Thus you will perform one of the following:

- When a open parenthesis "(" is encountered, it is pushed on the operator stack.
- When a close parenthesis ")" is encountered, we pop the operator stack until we pop an open parenthesis. For each operator popped from the operator stack (except for the open parenthesis), we "process the operator" (see below). If an open parenthesis is never popped before the operator stack is emptied, then we have an imbalance of parentheses and an error has occurred. Also note that we never remove the open parenthesis from the operator stack expect when processing a closing parenthesis.

- When an exponentiation operator "^" is encountered, it is pushed on the operator stack.
- When a multiplication or division operator is encountered, while the operator on the top of the operator stack has an exponentiation, multiplication or division operator (^ * or /); pop and "process that operator" from the operator stack. Then push the encountered operator onto the stack.
- When an addition or subtraction operator is encountered, while the operator on the top of the operator stack has an exponentiation, multiplication, division, addition or subtraction operator (^ * / + or -); pop and "process that operator" from the operator stack. Then push the encountered operator onto the stack.

When the end of the infix expression is encountered, all remaining operators are popped from the operator stack and are "processed" in order that they are popped off the stack. If we pop an open parenthesis at this time, then we have an imbalance of parentheses and an error has occurred. After all of the operators have been processed, if the value stack contains more than one value, the expression had too many values and an error occurred. If no error have occurred and the end of the infix expression is encountered, pop the one value from the value stack and print it as the result of the infix expression.

Programming Style

Your program must be written using good programming style which includes:

- Meaning variable names
- Header comments for the file
- Header comments for each function
- Inline comments
- Proper indentation of code
- Blank lines between code sections
- Use of functions

Program Submission

You are to submit the programs for this lab via the Assignments Page in Blackboard.

To help the TA, name your file with your net-id and the assignment name, like:

- ptroy1LabX.c