Programming Project 3

Due: Thursday, 10/1/15 at 11:59 pm

Infix Expression Evaluator

For this project, write a C program that will evaluate an infix expression. The algorithm REQUIRED for this program will use two stacks, an operator stack and a value stack. Both stacks MUST be implemented using a linked list.

For this program, you are to write functions for the linked list stacks with the following names:

```
int isEmpty (stack);          // return TRUE if the stack has no members
void push (stack, data);      // add the data to the top of the stack
data top (stack);             // return the data value on the top of the stack
void pop (stack);             // remove the data value from the top of the stack
```

The exact type of each parameter (including the type of parameter passing being used) is left up to you as the programmer. You may also wish to write a "topPop( )" function that does both the top and pop operations in a single function. Also, an init( ) function and a reset( ) function may prove useful.

These functions must take the head of the linked list (or a structure containing the head of the linked list) as its FIRST parameter. **The actual head of each linked list MAY NOT BE A GLOBAL VARIABLE.** It MUST be somehow declared as a local variable to some function (it may be as a local variable to a function other than main). Each operation performed on the linked list MUST be done in its own function.

Please note that you will need a stack of integer values and a stack of character operators for this project. However, since the operators can all be represented as characters (which is a sub-class of integers), you are more than welcome to write a single linked list structure to be used for both stacks.

The commands used by this system are listed below and are to come from standard input. Your program is to prompt the user for input and display error messages for unknown commands. The code in proj3base.c already does this for you. It is expected that you will use this code as the starting point for your project.

Command	Description
q	Quit the program.
?	List the commands used by this program and a brief description of how to use each one.
<infixExpression>	Evaluate the given infix expression and display its result.

The **<infixExpression>** will only use the operators of addition, subtraction, multiplication, division and the parentheses. It will also only contain unsigned (i.e. positive) integer values. We obviously could allow for more operators and more types of values, but the purpose of this assignment is to focus on linked list structures and not on complicated arithmetic expressions.

Infix Expression are when the operators of addition, subtraction, multiplication and division are listed IN between the values. Infix Expressions require the use of parentheses to express non-standard precedence. Examples of Infix Expressions are:

42 + 64
60 + 43 * 18 + 57
(60 + 43) * (18 + 57)
18 – 12 – 3
18 – (12 – 3)

Infix Expressions are normally converted to a Postfix Expression in order to be evaluated. Postfix Expressions are when the operators of addition, subtraction, multiplication and division are list AFTER (i.e. "post") the values. Postfix Expression never require the use of parentheses to express non-standard precedence. The fact that postfix expressions do not need parentheses makes them much easier to evaluate than infix expressions. The conversion from an Infix Expression to a Postfix Expression can be done using a stack.

A postfix expression is evaluated also using a stack. When a value is encountered, it is pushed onto the stack. When an operator is encountered, two values are popped from the stack. Those values are evaluated using the operation implies by the operator. Then the result is pushed back onto the stack. When the end of a postfix expression is encountered, the value on the top of the stack is the result of the expression. There should only be one value on the stack when the end of the postfix expression is encountered. Examples of Postfix Expressions are:

42 64 +
60 43 18 * + 57 +
60 43 + 18 57 + *
18 12 – 3 –
18 12 3 – –

Both the algorithm to convert an infix expression to a postfix expression and the algorithm to evaluate a postfix expression require the use of stacks. Note that the conversion algorithm requires a stack of operators, while the evaluation algorithm requires a stack of values. Also note that the following algorithm merges the conversion and evaluation algorithms into one, so it never actually creates the Postfix Expression.

Algorithm for Infix Evaluation

First we will define the function popAndEval (OperatorStack, ValueStack) as follows:

```
op = top (OperatorStack)
pop (OperatorStack)
v2 = top (ValueStack)
pop (ValueStack)
v1 = top (ValueStack)
pop (ValueStack)
v3 = eval ( v1, op, v2 )
push (ValueStack, v3)
```

If you are trying to perform a top operation on an empty ValueStack, print out an error message and return the value of -999. If the operator op for eval () is not one of *, /, + or -, print out and error message and return the value of -999.

When getting the input for an infix expression, the program **proj3base.c** breaks everything down into "tokens". There are 3 types of tokens we are concerned with: an OPERATOR token, a VALUE token and the EndOfLine token. The code in the function processExpression() already checks for these token but you will need to add the code how to interact with the stacks.

First step, have both the OperatorStack and the ValueStack empty

```
While (the current token is not the EndOfLine Token)
        if  ( the current token is a VALUE )
                push the value onto the ValueStack
        if  ( the current token is an OPERATOR )
                if  ( the current operator is an Open Parenthesis )
                        push the Open Parenthesis onto the OperatorStack
                if ( the current operator is + or - )
                        while ( the OperatorStack is not Empty   &&
                                the top of the OperatorStack is +, -, * or / )
                                        popAndEval ( OperatorStack, ValueStack )
                        push the current operator on the OperatorStack
                if ( the current operator is * or / )
                        while ( the OperatorStack is not Empty  &&
                                the top of the OperatorStack is * or / )
                                        popAndEval ( OperatorStack, ValueStack )
                        push the current operator on the OperatorStack
                if ( the current operator is a Closing Parenthesis )
                        while ( the Operator Stack is not Empty &&
                                the top of the OperatorStack is not an Open Parenthesis )
                                        popAndEval ( OperatorStack, ValueStack )
                        if (the OperatorStack is Empty )
                                        print an error message
                        else
                                        pop the Open Parenthesis from the OperatorStack
        get the next token from the input

Once the EndOfLine token is encountered
        while (the OperatorStack is not Empty )
                popAndEval ( OperatorStack, ValueStack )
        Print out the top of the ValueStack at the result of the expression
        Popping the ValueStack should make it empty, print error if not empty
```

The error statements in the above algorithm should only occur if an invalid infix expression is given as input. You are not required to check for any other errors in the input.

Command Line Argument: Debug Mode

Your program is to be able to take one optional command line argument, the -d flag. When this flag is given, your program is to run in "debug" mode. When in this mode, your program should display the operators and values as they are encountered in the input. The printf statements for these are already included in the program proj3base.c, so you just have to "turn off" the printf statements when the –d flag is given.

When the flag is not given, this debugging information should not be displayed. One simple way to set up a "debugging" mode is to use a boolean variable which is set to true when debugging mode is turned on but false otherwise. This variable may be a global variable. Then using a simple if statement controls whether information should be output or not.

```
if ( debugMode == TRUE )
        printf (" Debugging Information \n");
```

Provided Code for the User Inteface

The code given in proj3base.c should properly provide for the user interface for this program including all command error checking. This program has no code for the linked list. It is your job to write the functions for the specified operations and make the appropriate calls. Most of the changes to the existing proj6.c program need to be made in the **processExpression ( )** function. Look for the comments of:

```
// add code to perform this operation here
```

Note: the heads of the linked list are required to be a local variable in a function and you are required to pass the head of the linked to the operation functions. The function processExpression() is strongly suggested as the functions in which to declared the linked lists.

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