CS 220 – Data Structures and Systems Programming

PEX 3 – RPN Calculator

Part I Submission – Due at 2300 on Lesson 24

Part II Submission – Due at 2300 Lesson 27

Help Policy:

AUTHORIZED RESOURCES: Any, except another cadet's program.

NOTE:

- Never copy another person's work and submit it as your own.
- Do not jointly create a program.
- You must document all help received from sources other than your instructor or instructor-provided course materials (including your textbook).
- **DFCS will recommend a course grade of F for any cadet who egregiously violates this Help Policy or contributes to a violation by others.**

Documentation Policy:

- You must document all help received from any source other than your instructor.
- The documentation statement must explicitly describe WHAT assistance was provided, WHERE on the assignment the assistance was provided, and WHO provided the assistance.
- If no help was received on this assignment, the documentation statement must state "NONE."
- If you checked answers with anyone, you must document with whom on which problems. You must document whether or not you made any changes, and if you did make changes you must document the problems you changed and the reasons why.
- **Vague documentation statements must be corrected before the assignment will be graded and will result in a 5% deduction on the assignment.**

Turn-in Policies:

- On-time turn-in is at the specific time listed above.
- Late penalty is a 50% cap on the points you can earn on the PEX. For example, the most you could earn on a 100 point PEX is 50 points. Late PEXs can be turned in up to 1 week past the due date.
- There is no early turn-in bonus, but there is extra credit for this assignment.

0. OBJECTIVES

- Demonstrate understanding of stack and queue ADTs
- Demonstrate understanding of pointers
- Use dynamically allocated memory in a linked data structure
- Demonstrate the ability to create and use external libraries

1. BACKGROUND

In this programming exercise you will create a calculator that accepts an arbitrary length algebraic expression, and displays the Reverse Polish Notation translation of the expression followed by an equal sign and the result of the expression. The algebraic expression may contain standard mathematical operations (+,-,/,*, and ^), and parentheses. The calculator must be able to accept rational numbers and integers. Rational numbers are all real numbers, both positive and negative, which can be written as a fraction.

The program must begin execution with a message describing the functionality of the calculator to the user followed by a prompt for the user. i.e. INPUT: or EXPRESSION: After each successful calculation, this prompt will re-appear asking the user to enter the next algebraic expression. If the user enters just the return key without an algebraic expression, then the program will terminate.

2. WHAT IS REVERSE POLISH NOTATION?

Reverse Polish Notation (RPN) is a way of preserving precedence in an arithmetic expression without the need for parentheses. An algebraic expression in ordinary notation is shown below:

(3 + 5) * (7 - 2)

The parentheses tell us to first add 3 to 5, and then subtract 2 from 7, and finally to multiply the two results. In RPN, the numbers and operators are listed one after another, and an operator always acts on the most recent numbers in the list. The numbers can be thought of as forming a stack, like a pile of plates. The most recent number goes on the top of the stack. An operator takes the appropriate number of arguments from the top of the stack and replaces them by the result of the operation.

In RPN notation the above expression would be represented as shown below:

3 5 + 7 2 - *

Reading from left to right, this is interpreted as follows:

- Push 3 onto the stack.
- Push 5 onto the stack. The stack now contains (3, 5).
- Apply the + operation: take the top two numbers off the stack, add them together, and put the result back on the stack. The stack now contains just the number 8.
- Push 7 onto the stack.
- Push 2 onto the stack. It now contains (8, 7, and 2).
- Apply the - operation: take the top two numbers off the stack, subtract the top one from the one below, and put the result back on the stack. The stack now contains (8, 5).
- Apply the * operation: take the top two numbers off the stack, multiply them together, and put the result back on the stack. The stack now contains just the number 40

Polish Notation was devised by the Polish philosopher and mathematician Jan Lucasiewicz (1878-1956) for use in symbolic logic. In his notation, the operators preceded their arguments, so that the expression above would be written as

* + 3 5 - 7 2

The 'reversed' form is more convenient from a computational point of view. Hewlett-Packard championed Reverse Polish Notation for years.

3. CONVERTING AN ALGEBRAIC EXPRESSION TO RPN

A pioneer in Computer Science, Edsger Dijkstra, invented the Shunting-yard algorithm to convert infix expressions to postfix (RPN). The algorithm received its name because its operation resembles that of a railroad shunting yard. The link above is crucial to solving this problem.

The pseudocode below is from the Shunting-yard algorithm link above. Implementation of this algorithm will require an understanding of the below pseudocode and the use of stacks and queues.

```
/*
This implementation does not implement composite functions, functions with variable
number of arguments, and unary operators.

See http://www.reedbeta.com/blog/the-shunting-yard-algorithm/ to convert this
implementation to include unary minus (making a number negative)
*/

while there are tokens to be read do:
    read a token.
    if the token is a number, then:
        push it to the output queue.
    if the token is a function then:
        push it onto the operator stack
    if the token is an operator, then:
        while ((there is a function at the top of the operator stack)
               or (there is an operator at the top of the operator stack with
greater precedence)
               or (the operator at the top of the operator stack has equal
precedence and the token is left associative))
              and (the operator at the top of the operator stack is not a left
parenthesis):
            pop operators from the operator stack onto the output queue.
        push it onto the operator stack.
    if the token is a left paren (i.e. "("), then:
        push it onto the operator stack.
    if the token is a right paren (i.e. ")"), then:
        while the operator at the top of the operator stack is not a left paren:
            pop the operator from the operator stack onto the output queue.
        /* if the stack runs out without finding a left paren, then there are
mismatched parentheses. */
        if there is a left paren at the top of the operator stack, then:
            pop the operator from the operator stack and discard it
/* After while loop, if operator stack not null, pop everything to output queue */
if there are no more tokens to read then:
    while there are still operator tokens on the stack:
        /* if the operator token on the top of the stack is a paren, then there are
mismatched parentheses. */
        pop the operator from the operator stack onto the output queue.
```

```
exit.
```

4. PART I – SUBMISSION REQUIREMENTS

The first part of Programming Exercise 3 is the creation and submission of libraries for a stack ADT and a queue ADT implemented as a linked list, and a library to handle the mathematical calculations. These libraries require implementation of *at least* the following functions:

- Stack and Queue
    - Initialize
    - Push / enqueue
    - Pop / dequeue
    - Display
- Mathematical Functions
    - Addition
    - Subtraction
    - Multiplication
    - Division

Be sure to include your ***documentation statement*** in the file header of your .c files. Additionally, ensure all of your source code meets the C programming standards for the course.

Please place all of your files in a folder named PEX3_LastName, zip the folder, and submit your zipped file to the correct location on the course website by the due date and time.

5. PART II – SUBMISSION REQUIREMENTS

The second part of Programming Exercise 3 is the creation of the functionality for the calculator, which will use the libraries submitted for Part I. Specifically, your calculator must provide the following:

- The calculator must read a complete algebraic expression that can consist of integers, fractions, and mixed numbers (a whole number and a fraction) from the console.
- The calculator must present the result in proper form (fractions reduced, integer results must have just the integer and no 0/1, purely fractional results must not have a leading integer 0, negative signs must be in the proper places within mixed numbers, and so forth)
- The calculator must provide an error message if the user input is not algebraically correct (missing a closing parentheses, a negative sign on a denominator, etc.). To ease the parsing, the following rules must be followed:
    - Mixed numbers can have only one space between the integer and the fraction
    - Unary minus signs must appear directly beside the value that they are making negative

Extra credit may be earned on this assignment by implementing a graphical user interface for the calculator instead of utilizing the console for input and output. The use of curses is suggested and additional documentation is provided on the website for support; however, you may choose to implement this functionality in other ways.

Be sure to include your ***documentation statement*** in the file header of your .c file. Additionally, ensure all of your source code meets the C programming standards for the course.

Please place all of your files (to include those from Part I) in a folder named PEX3_LastName, zip the folder, and submit your zipped file to the correct location on the course website by the due date and time.

6. PROGRAMMING EXERCISE – HELPFUL HINTS

You should consider the following during the design and implementation of your calculator.

- Parse the user input into the following:
  - Operand stack
  - Operator stack
  - Expression queue to hold the algebraic expression
    - This queue must hold different data types (operators, operands, parens)
    - Investigate pointers of type void
- Consider defining a structure of type MIXED_NUMBER

CS 220 – PEX 3 – Part I – Grade Sheet Name:_____

		Points
Coding standards / Execution	**15%**	
Your code follows coding standards (comments, naming & indention etc)	**2**	
Your submission compiles and executes	**2**	
Functionality	**85%**	
Stack Library	**7**	
Queue Library	**7**	
Mathematical Function Library	**7**	
Subtotal	**75**	
Vague/missing documentation statement (-5%)	**-1.25**	
Submission requirements not followed (-5%)	**-1.25**	
Late penalties – 50% Cap (may turn in 1 week later than due date / time)	**max score=12.5/ 25**	
Total	**25**	

Comments from Instructor:

CS 220 – PEX 3 – Part II – Grade Sheet Name:_____

		Points
Coding standards / Good Design	**10%**	
Your code follows coding standards (comments, naming & indention etc)	**2.5**	
Your PEX is well designed (functionally decomposed & efficient)	**2.5**	
Functionality	**90%**	
Describe program functionality and provide user prompt	**5**	
Parse user input	**20**	
Display algebraic expression in RPN	**10**	
Display result of user-input algebraic expression	**10**	
Subtotal	**50**	
Graphical User Interface for Calculator	**10**	
Vague/missing documentation statement (-5%)	**-2.5**	
Submission requirements not followed (-5%)	**-2.5**	
Late penalties – 50% Cap (may turn in 1 week later than due date / time)	**max score=25/50**	
Total	**50**	

Comments from Instructor: