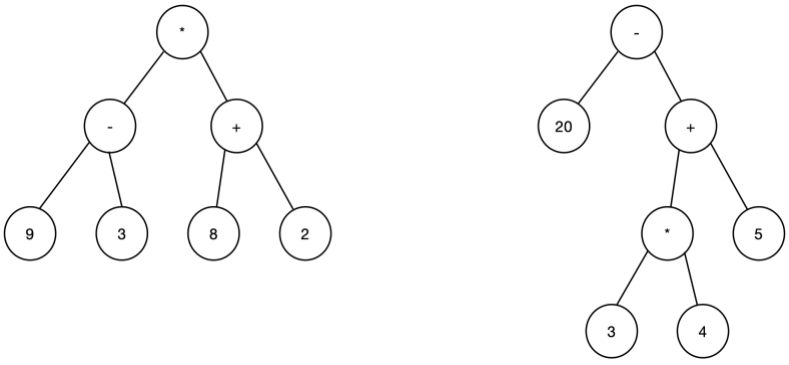
Programming Project 4

Due: Thursday, 3/18/21 at 11:59 pm

Simple Calculator with Binary Expression Trees

For this project, write a C program that will implement a simple calculator that can evaluate expressions of arbitrary length. The program MUST use **a binary expression tree** to implement the calculator. The calculator will only use the operators of addition, subtraction, multiplication, division. 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 writing C structs and recursion, not performing complicated arithmetic expressions.

A binary expression tree is a binary tree that can store and evaluate expressions. A binary expression tree has two types of node: *leaf nodes* that have no children and contain *operands* (integer values) and *internal nodes* that have one or two children and contain the *operators*. Below you can see two examples of binary expression trees.



The expression tree on the left represents the expression *(9 – 3) * (8 + 2)* and evaluates to *60*. The expression tree on the right represents the expression *20 – ((3*4) + 5)* and evaluates to *3*.

Expression trees are evaluated recursively, first the left subtree is evaluated, then the right subtree is evaluated and finally both values are combined with the operator stored in the root to give the final solution.
The conventional way of writing expression such as *20 – ((3*4) + 5)* is called infix notation. 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 the order of operations.

The algorithm that builds an expression tree from an infix expression is relatively complex, so for this project, we want to use another syntax known as **postfix** notation (also known as reverse Polish notation). 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.

Due to unambiguity of postfix expressions, most computers first convert infix expression to a postfix expression and then evaluate that postfix expression.

The postfix notation of expression given above are:

9 3 - 8 2 + *
20 3 4 * 5 + -

You must write a calculator program that reads a postfix expression, converts it to a binary expression tree, displays the notation in standard (infix) notation, and finally displays the result of evaluating the expression.

The commands used by this calculator 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 **proj4Base.c** already does this for you. You are expected to use that code as the base 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.
<postfixExpression>	Store the given expression in an expression tree, display its infix, prefix, and postfix notations, evaluate and display its result.

For completing this project, you need to use two data structures: a binary expression tree and a stack. You must write C structs for a tree node for the binary expression tree and a **stack that holds tree nodes.**

Parsing the expression string

The first step is to write a function called **parseExpression( )** which takes the string containing your postfix expression and returns the root node of an expression tree build based on that string. Implement the following non-recursive algorithm in **parseExpression( )** to build a binary expression tree from the given string. You are required to use a stack that can hold tree nodes.

***Function**: parseExpression*
***Input**: string e*
***Output**: root of an expression tree for e*

- Start with an empty stack
- Tokenize the string into an array of operators and operands
- For each token (moving left to right)
    - if the element is an operand
        - Create a tree node containing that operand
        - Push the tree node into the stack
    - if the element is an operator
        - Create a tree node containing that operator

    - Pop the first element off the stack
    - Set it as the right child of the newly created tree node
    - Pop the next element off the stack
    - Set it as the left child of the newly created tree node
    - Push the newly created tree node into the stack
- Pop the stack and return the node

For a valid expression, when you reach the end of the expression, there will be only one tree node in the stack which will be the root of a new expression tree.

If you detect an invalid postfix expression you should immediately return NULL. You can add statements to the algorithm above to check for invalid expressions. An expression is invalid if

- When processing an operator token, the stack does not contain at least two tree nodes, or
- Upon reaching the end of the expression, there are more than one tree node in the stack

Tokenizing a string in C

One method for tokenizing a given string is to use strtok( ) function. strtok( ) allows us to break a string based on a delimiter and tokenize it. The following code will break an expression into tokens using space as a delimiter:

```
char input[] = "60 43 18 * + 57 +";
char *ptr = strtok(input , " ");

while (ptr != NULL) {

     printf("%s   ", ptr);
     ptr = strtok ( NULL, " ");

}
```

The code prints the following (pay attention that tokens are ***string***, not character or integer)

```
60   43   18   *   +   57   +
```

Display different notations

For the next step, you should display different notations of the expression stored in the tree. First display the infix notation, then the prefix notation, and last the postfix notation. Each notation should be printed on a separate line. For displaying different notations, you have to write functions that perform the different traversal on the binary expression tree. The postfix notation is the result of a postorder traversal, the prefix notation is the result of a preorder traversal and the infix notation is the result of an inorder traversal.

Infix notation requires a proper placement of parentheses. Recall the process for inorder traversal first visits the left subtree, then the root node and finally the right subtree. In order to get the

proper notation, you should check the current root node and if it is an operator, print open parenthesis "(" before the recursive call to left subtree and place close parenthesis ")" after returning from recursive call from right subtree.

Use the following algorithm to display proper infix notation:

- If root node contains an operator
  - print "("
- Make a recursive call to the left subtree
- Print the value stored in the root
- Make a recursive call to the right subtree
- If root node contains an operator
  - Print ")"

Evaluate the expression

Finally, you have to evaluate your expression tree and display the result of the expression. Write a recursive function **evalExpression( )** which takes the root of your current expression tree and returns the value of the expression tree. This is the actual calculator part of the project that performs the simple evaluations.
The algorithm for evaluation is very simple and is based on the post order traversal. You should implement the following algorithm:

Function*: evalExpression*
Input*: root of an expression tree*
Output*: the value of the expression tree*

- If root node contains an operand
  - return the root's int value
- if root node is not an operand (it contains an operator)
- evaluate the left subtree
- evaluate the right subtree
- apply the root's operator to the values calculated in the previous steps
  - `leftval op rightval`
- return the result

printf statements for Gradescope

Processing an invalid expression:

```
printf("Invalid expression. Enter a valid postfix expression \n");
```

Displaying the outcome of the expression:

```
printf(" Expression result: %d \n", …);
```

Use of C struct and C functions

When writing your code, you MUST create two C structs: one struct for nodes in the expression tree and one struct for the stack. You can decide whether you want to implement the stack using a dynamic array or a linked list. Remember that the values this stack stores are Tree nodes.

The root for the expression tree MUST be declared as a local variable in main() or some other function. **It may NOT be global.**

MULTIPLE SOURCE CODE FILES

Your program is to be written using at least three source code files. It must also have a makefile and a header file to help with the compilation of the program. Place your functions for binary expression tree in a `btree.c` file and the functions for stack in a `stack.c`. The rest of the code which is the main function should be placed in `main.c`.

The parse and evaluate function should be considered as binary tree function and you should place them in `btree.c` file.

If you add additional functions to your program, you can add those to whichever source code file seems appropriate for that function (or create a fourth source code file or even a fifth).
You must also create a header file. The job of the header file is to contain the information so the source code files can talk to each other. The header file (.h file) should contain the function prototypes and any struct and/or typedef statements. Please review the make file discussions from lecture or lab 7.

The makefile MUST separately compile each source code file into a ".o" file and separately link the ".o" files together into an executable file. Review the makefile discussion to see how this is done. The command to create the .o file is:

```
gcc –c program1.c
```

The command to link the files program1.o, program2.o and program3.o into an executable file is:

```
gcc program1.o program2.o program3.o
```

The above command will just name the executable file using the default name of a.out, most often the –o option is given to provide a specific name for the executable file.

```
gcc program1.o program2.o program3.o –o program.exe
```

Team Submission

You have the option of working and submitting your project 4 in a team of 2. If you decide to choose a team submission, you must make sure that you are adding the other member to the submission on Gradescope. You can also work on the project individually similar to the previous projects.

Coding Style

Don't forget to use good coding style when writing your program. Good coding style makes your program easier to be read by other people as the compiler ignores these parts/differences in your code. Elements of good code style include (but may not be limited to):

- Meaningful variable names
- Use of functions/methods
- Proper indentation
- Use of blank lines between code sections
- In-line comments
- Function/method header comments
- File header comments

The Code Review Checklist also hints at other elements of good coding style.

Program Submission

You must zip all of your files needed for the program together and then submit the zip file for this program via the proper submission links on **Gradescope.**

In your makefile, please rename the final executable program to "**Project4**", e.g. ,
gcc –o Project4 program1.o program2.o program3.o

Create a directory called "Project4", then put source code files, your header file and your makefile into this directory. Zip this directory and submit the zip file.

Note that the folder name and file name are case sensitive.