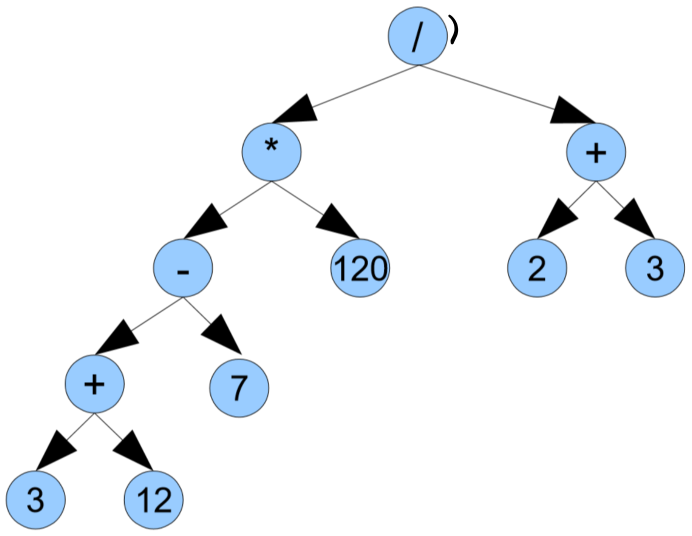
Assignment – Binary Tree Calculator

Goal: The goals of this assignment are to work with binary trees, traversal, recursion, stacks, and parsing string data. The assignment also strives to be somewhat practical, in that you will be processing a mathematical expression and then evaluating it. Your program needs to be able to process statements like ((((3+12)-7)*120)/(2+3)), load it into a binary tree, display what the expression is in postfix notation, and compute an answer.

((((3+12)-7)*120)/(2+3))

The core of this assignment revolves around taking a statement like ((((3+12)-7)*120)/(2+3)), and converting it into a tree like so:



From there, you can read the data using postfix order, which would give you 3 12 + 7 – 120 * 2 3 + /. An expression in postfix notation (Reverse Polish Notation) also allows for a straightforward way to calculate the answer. For example, first read in 3 and 12 and place both on a stack. Then when you attempt to place the + on the stack, you should recognize that + is an operator. So instead of placing + on the stack, you instead retrieve and pop off the prior two elements. Then compute 3 + 12, and place that answer back on the stack. Continue processing. You will next have a 15 and a 7 on the stack, as you attempt to read in a -. Since that is an operator, you will want retrieve, pop off, and compute 15 – 7, and push that result on the stack. After more traversing, you will find 8 and 120 on the stack. Next you will compute 8 * 120 and stick that result on the stack. Continuing, place 2 and 3 on the stack, so your stack at this point should contain 960, 2, 3. When reading the +, you should compute 2+3, and place the result back on the stack. When reading the /, you should compute 960 / 5, and place that result on the stack. Finally, the only number on the stack is your computed solution, which you can read and return.

Implementation:

You will need to start with a Node class/struct. That Node class does not need to be a template, it can just have a string data member. You will not be adding a BinaryTree class to your assignment. Rather, you will merge whatever components of a binary tree needed into the class TreeParser. This means the TreeParser class needs a root pointer, and relevant binary tree methods. This will *not* be a template class.

In my implementation, I used the following class members:

```
private:
        stack<string> mathStack;
        double castStrToDouble(string const &s);
        string castDoubleToStr(const double d);
        void initialize();

        bool isDigit(char c);
        bool isOperator(char c);
        void processExpression (Node* p); //Assuming raw pointers, or you can use shared pointers
        void computeAnswer(Node* p);  //Assuming raw pointers, or you can use shared pointers


protected:
        string expression;
        int position;
public:
        TreeParser();
        void displayParseTree();
        void processExpression(string &expression);
        string computeAnswer();
```

- displayParseTree() will display the binary tree in in-order and post-order notation.
- string expression will hold the mathematical expression you need to work on.
- int position is an index referring to which character of the string you are on.
- mathStack is an STL stack holding strings. You can use it by doing an #include <stack> at the top of your code file.
- castStrToDouble() and castDoubleToStr() have been given to you.
- initialize() will reinitialize your object so it can be used again. This has been given to you.
- isDigit() should accept a char, and see if that char is a digit character. Return true or false appropriately.
- isOperator() should accept a char, and see if that char is a +, -, *, /, or ^ character. Return true or false appropriately.
- The public processExpression() is rather simple. Its job is just to get things started and call the private method. This public method should take a mathematical expression as a string. From there, it should see if there is an expression to compute by checking if the expression is not empty. If so, you want to hold onto that string. So store the expression passed in as an argument into the protected data member called expression.   Set the position data member to zero.   Then create a node, have root point to it, then call the private processExpression(), passing in root.
- The private processExpression() is more work. The private processExpression() is where much of the core of your processing takes place. Specifically, here is how it should work:
  - All of this should be found within a while loop, and you loop so long as position is less than expression's length.
    - You should process each character of your string. Do so by looking at each character, one by one, at the index indicated by the protected data member position.
    - If the character is a '(', create a node, stick it on the left of your current node that your Node * argument is pointing to, then increment position, and then recursively go left.
    - Else if the character is a digit or a decimal (a .), keep reading until you see a non-digit/non-decimal character. Each time, concatenate the new digit/decimal character into a temporary string. Make sure you increment position accordingly as you loop through the string looking for all the consecutive digits. Store that temporary string in your node's data and then call return.
    - Else if the character is an operator, store that operator at the node you are currently in, then create a new node, stick it on the right, increment position, then recursively go right.
    - Else if the character is a ')', increment position then call return.
    - Else if the character is a space ' ', increment position, and then let it go around for the next iteration of the loop. Don't call return.
  - Please remember that you are working with boths strings and chars. The two data types are very different animals. Strings are objects, chars are more like 8 bit integers which represent an ASCII value. If you have string myString = "hello world", and you want the 'w' char, you can obtain it with myString.at(6) or myString[6]. To concatenate chars into strings, you can use +=. For example, string temp = ""; temp += myString.at(6); temp += myString.at(7); will give the temp string the value "wo".
  -
- The public computeAnswer() and private computeAnswer() will take the data that's in your binary tree, process it in a post-

order fashion described in the prior Goal section, and then return the answer. The public computeAnswer() is what is called from main and returns the answer. In other words, the public method can be just two lines of code. The first line gets the recursion started. The second line grabs the last item on mathStack, which is the answer, and returns it. The private computeAnswer() handles the recursion. It uses a postfix traversal, which is a "Go Left-Go Right-Act On Node". This means your code will start with:

```
void TreeParser::computeAnswer(Node* ptr){ // Assuming raw pointers
if (ptr) {
      computeAnswer(ptr->llink);
      computeAnswer(ptr->rlink);
```

After that, it should act on the node.

- If the node's data is a digit, put that node's data on the stack.
- If the node's data is an operator, you need to grab and remove the next two numbers off the stack. From here, use the operator and those two numbers and compute that result. So, if your ptr->data was a +, then you take off numbers A and B off the stack, and then compute A+B, and push the result back on the stack. Remember that to compute an exponent such as A^B, you need to use pow(A,B) included in math.h.
- You will find the methods castStrToDouble() and castDoubleToStr() useful here. Also remember that the STL stack uses top() to retrieve (but not remove) data, and pop() to remove (but not retrieve) data. You can still push data onto the stack normally with push().