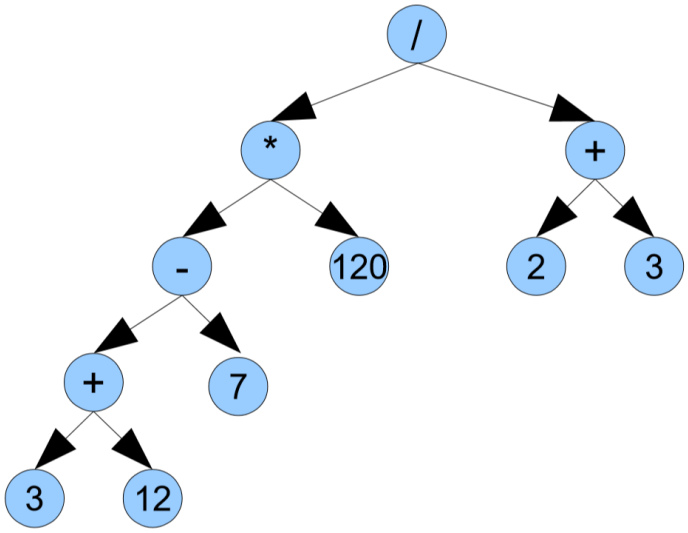
Assignment – Binary Tree Calculator

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 should process a mathematical expression, load it into a binary tree, display what expression in postfix notation, and compute an answer.

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, read the data with post-order traversal, which would give 3 12 + 7 – 120 * 2 3 + /. This format is also called postfix notation or Reverse Polish Notation.

An expression in postfix 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 + is read, because it is an operator, retrieve and pop off the prior two elements. Then compute 3 + 12 and place that answer back on the stack. Continue processing. The 7 is processed and also put onto the stack. Then when - is processed, use the stack to retrieve and pop off 15 and 7, then compute 15 – 7, and push that result on the stack. After more traversing the stack will have 8 and 120. Compute 8 * 120 and place that result on the stack. Continuing along, place 2 and 3 on the stack. The stack at this point should contain 960, 2, 3. When reading the +, compute 2+3, and place the result back on the stack. When reading the /, compute 960 / 5, and place that result on the stack. Finally, the only number on the stack is the computed solution.

Implementation:

The .h file is shown below. Your task is to implement all needed functions.

```
struct Node
{
 string data;
 Node* left{ nullptr };
 Node* right{ nullptr };
};

class TreeParser
{
public:
 ~TreeParser();
 void clear();
 void inOrderTraversal() const;
 void postOrderTraversal() const;
 void processExpression(string& expression);
 double computeAnswer();
private:
 void clear(Node* p);
 void inOrderTraversal(Node* p) const;
 void postOrderTraversal(Node* p) const;
 bool isDigit(char c) const;
 bool isOperator(char c) const;
 void processExpression(Node* p);
 void computeAnswer(Node* ptr);
 Node* root{ nullptr };
 string expression;
 unsigned int position;
 stack<double> mathStack;
};
```

The details and purposes of the methods and data members are given below:

- ~TreeParser() calls clear().
- The public clear() calls the private clear() method, passing in root. After that call, set the root to nullptr. Set expression to an empty string. Set position to 0. Pop all items from the mathStack.
- The private clear() method uses post-order traversal to delete all nodes.
- The public inOrderTraversal() calls the private inOrderTraversal(), passing in root.
- The private inOrderTraversal performs an in-order traversal, using cout to display the nodes contents, with a space character after to help separate the display for the next displayed item.
- The public postOrderTraversal() calls the private postOrderTraversal(), passing in root.
- The private postOrderTraversal performs a post-order traversal, using cout to display the nodes contents, with a space character after to help separate the display for the next displayed item.
- isDigit() accepts a char and checks if that char is a digit character. Return true or false appropriately.
- isOperator() accepts a char, and checks 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 prepared for the private recursive method. First, it should call clear() to clear out any tree in the object that many have existed from a prior processExpression() call. The method should check if there is an expression to compute by checking if the expression is not empty. If so, have the object copy the mathematical expression into the 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 public computeAnswer() and private computeAnswer() will take the data in the binary tree, process it in a postorder fashion described in this document's Goal section, and then return the answer. This public method can be just two lines of code. The first line starts the recursion. The second line obtains the last item on mathStack, which is the answer, and returns it.
- root points to the first node in the tree.
- string expression will hold the mathematical expression.
- unsigned int position is an index referring to the current character in expression being analyzed.
- mathStack is an STL stack holding double values.
- The private processExpression() is where much of the core of the processing takes place. It will process each character of the expression string. Do so by looking at each character, one by one, at the index indicated by the data member position:

- All of this logic should be found within a while loop, and loop so long as position is less than expression's length.
  - If the character is a '(', create a node, place it on the left of the current node, then increment position, and then recursively go left.
  - Else if the character is a digit (characters '0' through '9') or a decimal (the character '. '), keep reading until a char is a non-digit/non-decimal character. Each time, concatenate the new digit/decimal character into a temporary string. (Use += to concatenate a char to a string.) Make sure the position data member is incremented accordingly as it iterates through the string looking for all the consecutive digits. Store that temporary string in the node's data, then call return.
  - Else if the character is an operator, store that operator at the current node, then create a new node, place 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 (the character ' '), increment position, and then let it go around for the next iteration of the loop. Don't call return.
- Note that this logic works with both strings and chars. The two data types are very different things. Strings are objects. Chars are 8 bit integers which represent an ASCII value. Suppose a string called myString contains "hello world", and the char 'w' is needed. It can be obtained with myString.at(6) or myString[6]. To concatenate chars into strings, use +=. For example, string temp = ""; temp += myString.at(6); temp += myString.at(7); will give the temp string the value "wo".

- The private computeAnswer() handles the recursion. It uses a postfix traversal, which is a “Go Left-Go Right-Act On Node”. This means the code will start with:

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

  - The action part of postorder traversal is as follows:
    - If the node's data is a digit, put that node's data on the stack. Note that the numbers in the tree are strings, but the stack holds doubles. At this part of the logic, convert the string to a double with std::stod. For example: double some_double_output = std::stod(some_string_input)
    - If the node's data is an operator, then obtain the top two numbers from the stack. From here, use the math operator found in the node and those two numbers and compute that result. For example, if the node’s data was a +, then obtain numbers A and B and remove them from the stack, then compute A+B, and push the resulting sum back on the stack. Remember that to compute an exponent such as A^B, use pow(A,B) given from #include <cmath>.

Tips for success:

Don’t try to solve this all at once. For debugging, do not lazily just run step over debugging. Rather, carefully use step into and step over as this assignment uses recursion. While debugging, be acutely aware of exactly how far along the input string has processed and how much of the tree has been created. This assignment is simply not the kind of assignment where you can solve without detailed debugging.

Once you get the tree built for the first test problem, then move into the computeAnswer() method for the first test. Again debug carefully as this involves post-order traversal.