The first part of Project 4 is based on the “Basic Calculator Program” of Main and Savitch’s Section 7.2, with the changes detailed below. Only fully parenthesized infix expressions will be involved. Do Lab Exercise 5 on string reversal and parentheses matching with a stack and with recursion, and study all of DSO’s sections 7.1 and 7.2 before you begin.

Make sure you can do all the data manipulations ON PAPER FIRST, before you attempt to program the computer to do them!!

The second part of Project 4 is to develop a simple binary tree linked node structure, C/C++ functions to help build expression trees out of these nodes and to traverse them, and to demonstrate the use of these functions.

The project name for turnin is Proj4. The executable file names to be built by your build script are part1 and part2.

Unlike previous projects, you MAY use STL classes string and/or stack instead of C-strings and/or stacks you implement yourself, but you don’t have to.

1 Part 1

1.1 Differences From Section 7.2

1. We will use expressions (1) whose numbers are only single digits, with no decimal points, and (2) with no blanks.

2. The expression’s length can be limited to 200 characters. The expression will be supplied in a single line. In view of requirements below, I think it would be wise to read lines of input into a char array buffer using getline as in previous projects (or STL string) instead of using the peek and >> operations as illustrated in calc.cxx of DSO (7.2).

3. The expressions will use 3 kinds of parentheses pairs: ( and ), { and }, and [ and ]. Unlike Main’s program, yours must verify the expression is properly parenthesized with a left parenthesis of one kind being allowed to match ONLY with a right parenthesis of the SAME kind. (Different kinds of parentheses make it begin to look like HTML!)

To implement this, your program should push each left parenthesis into the “Operations” stack (in addition to each operation symbol). When a right parenthesis is encountered, your program should pop and perform operations from the Operations stack until a left parenthesis is popped, and then test whether the kinds of this left parentheses and the right parenthesis last encountered are the same. (Tip: Encapsulate parenthesis kind testing into a function.)

As soon as a mismatch is detected, your program should print:

EXPRESSION SYNTAX ERROR (MISMATCHED PARENTHESIS)

and exit.
4. As soon as the program detects the input expression has illegal or unsupported syntax, your program should print:

**EXPRESSION SYNTAX ERROR (SOME OTHER ERROR)**

and exit.

5. In addition to printing the prompt and the message with the evaluated result, your program must print a report about each non-atomic subexpression and its value immediately after each subexpression is evaluated. (An atomic subexpression means a single input number.)

This report has 3 parts: (1) a copy of the subexpression, (2) the two numbers being operated on and operation symbol, which is the top-level operation of the subexpression and (3) the arithmetic result. Some partial credit will be given for parts (2) and (3) only; but very little for work little more than copying the code from the book!

Here is the sample input and output corresponding to the example explained within section 7.2 of DSO:

Type a fully parenthesized arithmetic expression:

```
((6+9)/3)*(6-4)
```

```
(6+9)=
6+9=
15

[(6+9)/3]=
15/3=
5

(6-4)=
6-4=
2

[((6+9)/3)*(6-4))=
5*2=
10
```

That evaluates to 10

To make part (1) of the reports happen, you must put store, in a stack, information what will enable the program to print the proper substring from the input string. (Hint: Use two index numbers!) This information may be stored in a structure (of your design) type (instead of **double**) for the items in the “**numbers**” stack; or in a separate stack.

Here’s simple code to print a substring of a C-string or STL **string**:
// A is a C-string or STL string.
// 0<=start_index<=end_index<=length of the string.
for( int i = start_index; i < end_index; i++ )
    { cout << A[i]; }
// Chars from A[start_index] up to A[end_index-1] have been printed.

Reminder: All requirements from previous projects are in force for this one (except of course
the limitation on STL and the string class).

2 Part 2

The first objective of part2 is to develop a simple binary tree linked node structure, C/C++
functions to help build expression trees out of these nodes and to traverse them, and to demonstrate
the use of these functions.

Here is an alternative, object oriented description: Develop an TNode class for node-of-an-
expression-tree that uses the linked node implementation. This class has member functions for
building a node tree from its top level operator or constant and possibly addresses of the roots of
its subtrees. The TNode class should have member functions to traverse the expression tree whose
root node is the object for which each member function is called. The project must demonstrate
the use of these member functions.

It is your choice as to whether or not to use a class with constructors or other member functions,
or just a plain struct. You must make use a struct or class that contains two pointer-to-tree-node
fields or members, and some data.

The second objective is to gain deeper understanding between expression parsing using operand
and operator stacks, and (1) building the expression tree, (2) traversing the expression tree, (3)
evaluating the expression tree, (4) printing the expression tree by traversing it several ways. This
objective is supposed to reveal the relationship between post-order traversal, postfix notation and
evaluation.

2.1 Getting Started, etc.

Read the entire assignment and plan how you will do the project! The relevant chapters in Main
and Savitch for this project are: 5 on dynamically allocated linked data structures, 7 on stacks
and expressions, 9 on recursion and, for trees, parts of 10 (sections 10.1, “representing a tree with
nodes” in 10.2 and the beginning of 10.3, 10.4 except for templates and function parameters, and
the self-tests of 10.4).
You may use the following declarations to get started on Part 1:

```c++
struct TNode
{
    //enum {CONST, OPERATION} NType; //optional, look up enum in DSO index.
    char symbol; //optional: maybe modify to support multi-char constants.
    //double value; //optional: value==the constant's value if NType==CONST
    //    //optional: otherwise, symbol=='+', '-', '*', or '/'
    //    //optional: and NType==CONST.
    TNode *pleft;
    TNode *pright;
};
TNode * newTNode( char ch ); //Dynamically allocates one TNode and returns
    //its address.
void InOrderTraverse( TNode *pnode, int depth )
{
    if(pnode->pleft) { InOrderTraverse( pnode->pleft, depth+1); }
    for(int i=0; i<depth; i++) {cout << ' ';} //depth spaces were printed.
    cout << pnode->symbol << endl ; //This node's symbol was printed.
    //you finish it!!
}
```

//optional: class TNode
//{
// private: TNode *pleft; TNode *pright; char symbol; //etc.
// public:
//    TNode( char ch );
//    TNode( char ch, TNode *pleftparameter, TNode *prightparameter);
//    void InOrderTraverse( int depth ){
//        //recursive calls within the body look like:
//        //pleft->InOrderTraverse( depth + 1 ); // etc...
//    }
//}

2.2 Developing Binary Tree Functions

It would be a good idea to develop the needed structs, classes and functions pertaining to binary trees first, independently (independently of expression parsing and evaluation). For you convenience, the only data that needs to be stored in tree nodes will be single characters. Developing this tree support software means creating a main function that uses all your the tree software in order to test and debug it. Indeed, if you do not submit an essentially working part2 program, we will check your submission for this preliminary step and we will award partial credit based on it.
2.3 Credit

Submit a build script and sources so that when we execute the build script in its own directory, it builds a linked executable file named part2 which we can run to test it. (We will also examine program sources to grade for file structure, pre- and postconditions, style and check for cheating. A fake “solution” whose output makes it appear to meet the requirements but which doesn’t build and traverse a tree is a form of cheating.)

2.4 Part 2–Assignment

Part 2 is a continuation of Part 1.

Add to your Part 1 infix expression evaluator code the functionality to build the expression tree while your infix expression evaluator is active. Up to the
That evaluates to ..
line it should input and print exactly what is specified for Part 1. Your program should also save the evaluated number in a variable so your program can automatically compare this number with the number obtained from recursive evaluation of the expression tree. It is important in scientific research that results verified during old research be reproduced and carefully compared to results of new methods that are hypothesized to do the same thing. See below.

The expression tree will be built “from the bottom up,” not using any recursive procedures, during a computation that is controlled by your stacks and the input expression.

2.5 Main Idea, Hint...

Remember the idea of putting in each number stack entry information about were the sub-expression whose value is that number, besides the number?

The idea is: Whenever a number is determined either from a constant digit, or by doing an operation, the expression tree node holding that constant or the operation symbol is allocated and filled in with data and pointers. The address of that tree node is (alongside the string index and the evaluated or converted number) stored in the number stack entry! Your code should use the node addresses retrieved from entries popped from the number stack to fill in the pleft and pright pointer fields of that new tree node. (Remember a way to convert the ASCII code ch for a digit 0, ... 9 to the corresponding integer? It is: (ch - '0').)

This idea will be dramatized in a lecture.

2.6 Continuing with Details..

After printing the value in “That evaluates to ..” your program should print:
Count of expression tree nodes is ..
where .. denotes how many nodes (for both constants and operations) there are in the expression tree your program built, as calculated by a recursive node counting traversal.

Next, print 4 more results, each preceded by the indicated label:

Pre-order Traversal:
Print each node’s operator or constant, one node per line, indented one space for each unit of
recursion depth (see the “getting started” examples), as the tree is traversed in left-to-right pre-order.

**In-order Traversal:**
Analogous the above, except for the order.

**Post-order Traversal and Evaluation:**
Analogous again, except for the order, **and**, after each operator or constant, on the same line, print a space and then `value=...`
to display the double precision floating point value obtained when that node was evaluated.

- The program should compare the number just printed with the number saved (see above) from your project 4 evaluation algorithm to check that they are equal. If they are equal, print “Values computed by the two algorithms match: Good.” Otherwise, print the copy of the stack algorithm’s answer again in a message that this program has a bug.

### 2.7 Credit

To get credit for Part 2, make your build script create the linked executable file named **part2** so we know you did it and can test it. (Submit all work, for parts 1 and 2, to project **Proj4**. For full credit, ONE build.sh script should build 2 executable programs named **part1** and **part2**.)

### 3 Test Yourself with Lab 6

Lab 6 will be assigned next week...The preview will be posted soon. It begins with using the Postscript interpreter software called Ghostscript to evaluate postfix expressions for you.

Use your Part 2 program to evaluate some infix expressions and obtain their equivalent postfix expressions. Then, translate each postfix expression into PostScript and append the PostScript “pop-and-print” operator `==`. Verify that when you input each PostScript expression to the Ghostscript interpreter (**gs**), Ghostscript computes the same value as your Part 2 program did.

### 4 Appendix—Project Plan for Developing Tree Software

Create declarations and functions described under the objectives to make a program that builds a binary tree so that each node contains one C/C++ **char** data item. The tree can be build by a sequence of function calls you write. You can code into those calls the particular letters, spaces and punctuation symbols required below. It would be a good idea to use some pointer-to-node temporary variables to hold the addresses of nodes right after they are build. Building the tree need not be done recursively.

Requirements: Build your tree so that an **in-order** left-to-right traversal prints out your full name; but, if your name has fewer than 15 characters including spaces, you must add something else (like “Sir”, “Madame”, “Albany, N. Y.”, etc.,) to make it at least 15 characters. The tree must
be built so that about half the characters are in the left subtree. (Something like this is coded on page 477.)

A more interesting option, is to prompt for any string (which could be limited in length) and then demonstrate that your program builds the tree specified above automatically from a name supplied at run time.

After the tree is built, the program must demonstrate your recursive function to count the nodes and 3 functions to traverse the tree in left-to-right pre-order, in-order and post-order. The in-order output, read line by line, should spell out your name, etc., of course. Each character must be printed on its own line and be indented by the number of spaces equal to the recursion depth of the activation that prints it (see the sample code for InOrderTraverse()).
Lab 5 Assignment–Due April 8, but should be begun NOW.

Word to the wise: Pay careful attention, and perhaps take notes because the lab assignment leads directly into the the current project.

1. Begin by writing a program that inputs a line of characters and outputs the reversal of the input line, where the reversal operation is implemented using a stack as described at the beginning of chapter 7 of Main and Savitch.

2. Next, create another program that inputs a line of the 3 kinds of parenthesis pairs described in the project, and outputs “OK” if the parentheses are properly nested and “NOPE” if not. You can assume the input is limited in size and consists only of the 8 characters described below. This program, like the first, should use a stack to implement its test.

   For this problem, a string is “properly nested” means:
   
   • Either it is empty, or, if not empty:
     • It begins with some kind of left parenthesis, that beginning parenthesis is followed by zero or more properly nested strings, and it ends with a right parenthesis of the SAME KIND as the beginning left parenthesis.

   For example, here are some properly nested strings:
   
   ([]{}(){}{})
   ((([]{}()){})))
   ()

   But here are some not properly nested strings:
   
   ([]{}(){}{})
   ()
   )()

3. Re-implement the assignment (1) above, except use a recursive function instead of a stack.

4. Re-implement the assignment 2 above, except use a recursive function instead of a stack.

   For credit, submit compilable work with a build script to project lab5

   Note: there are 4 check-off points for this lab, in addition to the one point for lab attendance.