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 4 on string reversal and parentheses matching with a stack and with recursion, and study up to page 353 of Chapter 7 before you begin. The project name for turnin and the executable file name to be built by your build script is proj4.

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 Differences From Section 7.2

1. Most of the testing will be done on expressions (1) whose numbers are only single digits, with no decimal points, and (2) with no blanks.

2. Rather than read and process a single expression (which `main()` on page 350 does), your program should read and process one expression after another, line by line, until a blank line is entered.

3. Each expression’s length can be limited to 200 characters. Each 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 on page 350.

4. The expressions will use 4 kinds of parentheses pairs: ( and ), { 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 go on to process the next expression (or blank line).

5. 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 go on to process the next expression (or blank line).

6. 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 on pages 345-348:

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

Type a fully parenthesized arithmetic expression:

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:

```c++
// 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.
```
7. (Refinement: 10 points) Add single letter variables and a C-like assignment operator “=”, so that values calculated during previous expression evaluation can be used in later expressions during the same program run. This gives your calculator 52 “memories”. Example:

Type a fully parenthesized arithmetic expression:
\( (3+(A=4)) \)

\( (A=4)= \)
\( 4= \)
\( 4 \)

\( (3+(A=4))= \)
\( 3+4= \)
\( 7 \)

That evaluates to 7
Type a fully parenthesized arithmetic expression:
\([9+A]\)

\([9+A]=\)
\(9+4=\)
\(13\)

That evaluates to 13
Type a fully parenthesized arithmetic expression:
\([9+B]\)

EXPRESSION SEMANTIC ERROR: UNDEFINED VARIABLE
Type a fully parenthesized arithmetic expression:
\(A\)

That evaluates to 4
Type a fully parenthesized arithmetic expression:

This exercise will lead you to invent symbol tables.

8. (Refinement: 10 points) Handle multidigit and fractional (decimal) atomic expressions. (To input negative numbers like -3, you still must type something like \{0-3\}).

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

2 The Future

The next project will be based on this one: The program should also build the expression tree, print the tree, and use it to evaluate the expression and print it in prefix and suffix form. That uses recursion!
Lab 5 Assignment.

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.

Next, create another program that inputs a line of the 4 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 above. 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:

```
(][{{()}}}()
(()()
)
```

About halfway through the period, the instructor will discuss strategies for formulating solutions to these two programming problems each using a recursive function.

For the rest of the lab, re-implement the above two assigned programs using a (separate) recursive function in each.

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