Arithmetic Expression Trees (Part 1)
Cpt S 322 Homework Assignment
by Evan Olds

<u>Submission Instructions</u>: (see syllabus)

<u>Assignment Instructions</u>:

Read each step's instructions *carefully* before you write any code.

<u>Note:</u> This is a standalone <u>console application</u> and integration into the spreadsheet will happen in a future homework assignment. Make sure you implement the code in your logic engine DLL and have the console application be a separate compiled entity that references this DLL.

In this assignment you will create the ExpTree class, in which you will implement an arithmetic expression parser that builds a tree for the expression. This tree can then be used for evaluation of the expression. The parsing aspect of this assignment is simplified and you will extend the parser in the next homework. But the entire evaluation functionality, including setting variable values, will be implemented in this assignment.
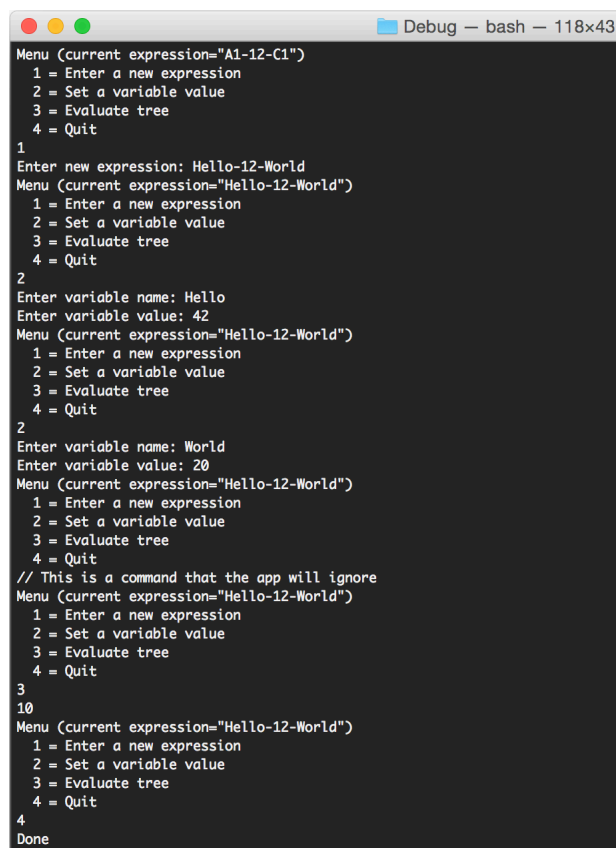
Create a class named **ExpTree** in the **CptS322** namespace. Do not create a class named ExpressionTree, expTree, EXPTREE, etc. Make a class name that is an EXACT match including casing. This assignment is likely to be graded with an automated grader and it will not give you credit if it cannot find the class. Create this class in your logic engine DLL. The console app for this assignment just references your engine DLL and provides a simple menu for testing. Include the following functions in the ExpTree class, again with EXACT signatures matching:

- ExpTree(string expression)
    - Implement this constructor to construct the tree from the specific expression
- void SetVar(string varName, double varValue)
    - Sets the specified variable variable within the ExpTree variables dictionary
- double Eval()
    - Implement this member function with no parameters that evaluates the expression to a double value

Create a console application that presents a menu when run. As a second reminder, this app references your engine DLL. This menu must contain the following options:

1. The option to enter an expression string. You may assume that only valid expressions will be entered with no whitespace during grading. Simplified expressions are used for this assignment and the assumptions you are allowed to make are discussed later on.
2. The option to set a variable value in the expression. This must prompt for both the variable name and then the variable value.
3. The option to evaluate to the expression to a numerical value.
4. The option to quit.
5. Should the user enter an "option" that isn't one of these 4, simply ignore it. As trivial as this may seem it is vital should the assignment be graded with an automated grading app.
   - On that note, also avoid use of Console.ReadKey() as it can be problematic when grading with an automated app. Simply fall through to the end of main after the quit option is selected.

The screenshot below shows what this might look like:

Requirement Details (10 points total):

Support simplified expressions (2 points):

- For this assignment all expressions will be simplified next to what you'll have to support in the next homework. As stated earlier, this assignment is primarily about getting the correct evaluation logic in place for the tree.
- Assume expressions will NOT have any parentheses
- Assume expressions will only have a single *type* of operator, but can have any number of instances of that operator in the expression
    - Example 1: expression could be "A+B+C1+Hello+6"
    - Example 2: expression could be "C2-9-B2-27"
    - Example 3: expression could NOT be "X+Y-Z" because that has two different types of operators: + and –
- (In the next assignment you'll have to support all valid arithmetic expressions)
- Support operators +,-,*, and / for addition, subtraction, multiplication, and division, respectively. Again, only one type will be present in an expression that the user enters.
- Parse the expression that the user enters and build the appropriate tree in memory

Tree Construction and Evaluation (5 points):

- Build the expression tree correctly internally. Non-expression-tree-based implementations will not be worth any points.
- Each node in the tree will be in one of three categories:
    - Node representing a constant numerical value
    - Node representing a variable
    - Node representing a binary operator
- As discussed in class the above three types should match nicely with a design of a central base class for the node and then 3 inheriting classes.
    - No node should store any more data than it needs to. As just one example of what this means, the constant value and variable nodes never have children so their class declarations shouldn't have references to children (nor should they be inheriting such declarations from a parent class).
- Your expression class must have a public evaluation function that takes no parameters and returns the value of the expression as a double.
    - public double Evaluate()

Support for Variables (2 points):

- Support correct functionality of variables including multi-character values (like "A2").
- Variables will start with an alphabet character, upper or lower-case, and be followed by any number of alphabet characters and numerical digits (0-9).

- A set of variables is stored per-expression, so creating a new expression will clear out the old set of variables.
- Have a default expression, something like "A1+B1+C1" would be fine, so that if setting variables is the first action that the user chooses then you have an expression object to work with.

Clean, REUSABLE code (1 point):

- Do <u>not</u> have things like the variable dictionary declared as a local variable in Main(). It must be a member of the ExpTree class. Such implementations would not allow you to easily bring your code into the WinForms app for the later homework assignment.
- Have clean, well-documented code and an easy to use interface for grading.