Spreadsheet Formula Evaluation
Cpt S 321 Homework Assignment
Washington State University

Submission Instructions:

- Create a branch called "Branch_HW7" and work in this branch for this assignment.
- **When you are done, merge the branch back to the master.**
- Once you are done and before the deadline, tag the version that you would want us to grade with the assignment number (for example, "HW7").
- On Canvas -> Assignments -> Submit the link to your repository (a link to the tag or the branch works) by the HW deadline.
- **IMPORTANT: The HW must be tagged by the due date and a link to that tag needs to be submitted via Canvas in order to receive a grade.**

Assignment Instructions:

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

In this assignment, we will combine work from several previous homework assignments and build new features on top of that. We will link the arithmetic expression evaluator with the spreadsheet application.

Part 1: Edit the spreadsheet application from the "First Steps for Your Spreadsheet Application" homework assignment so that it has the following functionality:

1. In its normal, non-editing state each cell will display the **Value** property of the cell in the logic engine. Recall that this value represents the *evaluation* of the formula in the cell, if present. The **Text** property represents what the user actually typed into that cell (which may be a formula if it starts with '='). If there is no formula in a cell, then the **Value** just matches the **Text** property.
2. When the user starts editing a cell, they should be seeing the **Text** property of the cell. This allows easy editing of formulas for example without the need to retype them from scratch. When the user finishes editing, the cell must go back to showing the **Value**. This functionality closely resembles that of other spreadsheet applications such as Microsoft Excel.

WinForms: You'll need to use the CellBeginEdit and CellEndEdit events.

Avalonia: If you used the DataGridTextColumn for HW4, you will need to change to the DataGridTemplateColumn – for this, see Part 1 from the additional information provided for Avalonia.

We will be using the PreparingCellForEdit and the CellEditEnding events. When you select a cell in Avalonia the whole line is selected. You can simply disregard this for now - we will handle it in a later HW.

Part 2: Incorporate your expression evaluator into the spreadsheet functionality so that when the spreadsheet is updating the value for a cell, it is properly evaluating the formula. You may assume the following:

- Each cell only contains a formula to be evaluated if it starts with the '=' character. The substring to the right of this is the actual formula that your expression evaluator should be able to parse and evaluate.
- No formulas will contain whitespace. It would be *nice* if your code handled whitespace characters (by ignoring them), but it's not required.
- Every formula contains only things that were required in homework HW5 and HW6: add, subtract, multiply, divide, constants, variable names and parentheses. Some semesters will have exponents as well. In all cases, your code must allow for easily extending the application and adding more operators – i.e., **you must refactor your code to include the algorithm that uses reflection to dynamically populate handled operators**.
- All variable names will be cell names that start with a single capital letter character and then are followed by a row number. Rows start at 1 in the user interface and therefore in the formulas as well, so keep this in mind when adjusting array indices in your code (actual index in array will be 1 less).
- In HW5, we explicitly state that if variables are not set then can be default to 0. In this assignment, **you should use exceptions to handle this properly** – you should not have default values and you should not crash.
- There will be no circular references. This means that if we have some cell, say A1, it won't have A1 in its own formula. Also, it won't reference any cell that in return references back to it. You WILL have to deal with this in a future assignment, so keep that in mind. But for this assignment right now you don't need to worry about it.
- Since a cell's "Value" property is a string, and you will need a double when setting variable values during formula evaluation, consider the numerical (double) value of a cell to be:
    - The numerical value parsed if double.TryParse on the value string succeeds

There are sub-tasks within this task that require extending your expression tree code. You will need a way for the spreadsheet to enumerate all the cell names referenced in the formula. The ExpressionTree class that you built could have a "GetVariableNames" function that returns a list or array of all variable names in the expression. Alternatively, it could have a reference to a function to call when it needs to lookup a variable value. Whichever design you choose, the following must be true:

1. The world outside of your ExpressionTree class cannot edit the tree. There should be no way for the outside world to change a child reference in any node, change a variable name or value within any node, and so on.
2. The world outside of ExpressionTree must never have to deal with nodes. It's not the responsibility of objects outside this class to know how to iterate through an expression tree. Provide an easy-to-use interface for finding relevant information about the expression.

Part 3: Make sure that when a cell is changed all other cells that reference that cell in their formulas get updated. This means that the cell Text property change is not the only circumstance where you need to update its value.

- There's some work behind doing this in an efficient way. Don't just try to update all cells in the spreadsheet every time any cell changes. Come up with a more efficient design than that. **Hint: use events!**

Point breakdown (the assignment is worth 10 points):

- 5 points for implementing the correct functionality

And as usual:

- 1 point: For a "healthy" version control history, i.e., 1) the HW assignment should be built iteratively, 2) every commit should be a cohesive functionality, 3) the commit message should concisely describe what is being committed, 4) you should follow TDD – i.e., write and commit tests first and then implement and commit the functionality.
- 1 point: Code is clean, efficient and well organized.
- 1 point: Quality of identifiers.
- 1 point: Existence and quality of comments.
- 1 point: Existence and quality of test cases.

General Homework Requirements	
Quality of Version Control	• Homework should be built iteratively (i.e., one feature at a time, not in one huge commit). • Each commit should have cohesive functionality. • Commit messages should concisely describe what is being committed. • TDD should be used (i.e, write and commit tests first and then implement and commit functionality). • Include "TDD" in all commit messages with tests that are written before the functionality is implemented.

	● Use of a .gitignore. ● Commenting is done as the homework is built (i.e, there is commenting added in each commit, not done all at once at the end).
Quality of Code	● Each file should only contain one public class. ● Correct use of access modifiers. ● Classes are cohesive. ● Namespaces make sense. ● Code is easy to follow. ● StyleCop is installed and configured correctly for all projects in the solution and all warnings are resolved. If any warnings are suppressed, a good reason must be provided. ● Use of appropriate design patterns and software principles seen in class.
Quality of Identifiers	● No underscores in names of classes, attributes, and properties. ● No numbers in names of classes or tests. ● Identifiers should be descriptive. ● Project names should make sense. ● Class names and method names use PascalCasing. ● Method arguments and local variables use camelCasing. ● No Linguistic Antipatterns or Lexicon Bad Smells.
Existence and Quality of Comments	● Every method, attribute, type, and test case has a comment block with a minimum of <summary>, <returns>, <param>, and <exception> filled in as applicable. ● All comment blocks use the format that is generated when typing “///” on the line above each entity. ● There is useful inline commenting <u>in addition to comment blocks</u> that explains how the algorithm is implemented.
Existence and Quality of Tests	● Normal, boundary, and overflow/error cases should be tested for each feature. ● Test cases should be modularized (i.e, you should have a separate test case for each thing you test - do not combine them into one large test case). ● *Note: In assignments with a GUI, we do not require testing of the GUI itself.*