TCSS 342 Winter 2022

Homework #6B

Spreadsheet

Due: Electronic submission (e-mail) and written report (in lecture), 5:00pm Monday, March 7.
In-class demo: Tuesday, March 8 and Thursday, March 10.

IMPORTANT: Read this handout completely and read it several times before you type in any code! It is vital that you understand what the assignment is and plan your strategy for implementation *before* spending hours programming. If you do not plan, many of those hours on the computer will be a wasted effort.

Learning Objectives

- Designing and implementing a medium size programming project.
- Understanding the inner workings of a "real" application.
- Experience with stacks, queues, trees, and graphs.

Background

A *spreadsheet* is just a two-dimensional grid of cells. Each *cell* contains a formula that specifies how to compute the value of the cell. Every cell has a name consisting of a letter followed by a nonnegative integer (e.g., "A0" or "C14"). A formula can be as simple as an integer constant (e.g., "5"), or it can refer to other cells (e.g., "B1 + C3").

Or, it can be a combination of the two (e.g., "2 * D3"). Here's a sample 8x8 spreadsheet with the formulas corresponding to those cells.

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
0			23		B2+C2			
1	4			B3+2				
2				2*5				
3		A1+B1	B3+A4		3+C0			
4	A1-D4			-5				
5								
6								
7								

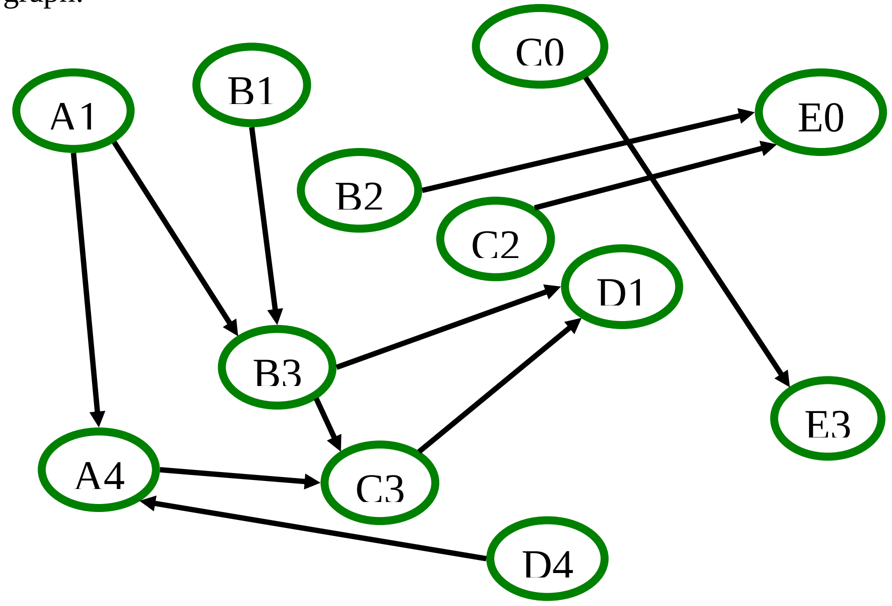
Note that every cell's formula consists of integer constants, cell references, and operators. The operators we will use in our spreadsheet are just the integer operations +, –, *, and /. Any cell that does not have a formula is assumed to have "0" as its formula.

Here is the same spreadsheet with the values computed for each cell based on their formulas above. (All empty cells have value 0.)

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
0			23		0			
1	4			6				
2				10				
3		4	13		26			
4	9			-5				
5								
6								
7								

There are two main questions to ask. How do you decide what order to evaluate the cells? How do you evaluate a cell given its formula?

To determine what order to evaluate the cells, you first need to construct a *dependency graph*. Call the dependency graph G = (V,E). The vertices, V, in the graph are the cells of the spreadsheet. (There are 64 cells in the above spreadsheet, and therefore 64 vertices in the graph.) There is edge in E between vertices *v* and *w* if the formula of the cell corresponding to vertex *w* contains a reference to the cell corresponding to vertex *v*. For example, since the formula for cell C3 contains a reference to B3, then there will be an edge in E that goes from the vertex for B3 to the vertex for C3. Here is part of the dependency graph:


$$\begin{tikzcd}
A1 \arrow[d] \arrow[rd] & B1 \arrow[d] & B2 \arrow[r] & E0 & C2 \arrow[l] & C0 \arrow[d] \\
A4 \arrow[r] & B3 \arrow[d] \arrow[r] & D1 & & & E3 \\
& C3 \arrow[ru] & D4 \arrow[llu, bend left] & & &
\end{tikzcd}$$

All other nodes in the graph are vertices with no edges going in or coming out of them.

Once we have the dependency graph, we can then perform a topological sort on it to determine an order to evaluate the formulas in the cells. Note that if we evaluate the cells in order of their topological numbers, then we can be assured that when we evaluate a formula that has a reference to another cell, that other cell's formula will already have been evaluated. Also note that this only works if there are no cycles in the dependency graph. For example, if A0's formula were "B0+3" and B0's formula were "A0+1", then we would have a cycle in the dependency graph, and it would be impossible to evaluate the formulas. In our spreadsheet, we will disallow the user from creating formulas that would result in a cycle in the dependency graph. (What does Microsoft's Excel spreadsheet do when you try to create a cyclic dependency?)

Once we know what order to evaluate the formulas in the cells, we then have to determine how to evaluate a cell's formula. This is fairly easy. We compute its formula using the values in the cells it references, if there are any.

The Assignment

Your task in this project is to write a spreadsheet ADT and a program that allows the user to assign formulas to cells, view the formulas in the cells of the spreadsheet, and view the values computed for all the cells in the spreadsheet.

You will be randomly assigned to a group of three or four people.

The **three-person assignment** has a GUI for the user to see and modify the spreadsheet.

The requirements for three-person groups and four-person groups are the same. In a four-person group, it will be slightly harder to divide the work evenly, but the overall workload for each person in the group (assuming you can work together as a group) should be smaller.

Data Structures

You will need to decide how to represent the spreadsheet. You will likely want to implement it as a class, where the two-dimensional array of cells is a private data member. The cells themselves will be a class, containing some representation of a formula. We will need to use the concept of a *token* to represent the various parts of a formula.

We will need to define the following classes:

```
public class LiteralToken extends Token {
    private int value;
}

public class CellToken extends Token {
    private int column; // column A = 0, B = 1, etc.
    private int row;
}
```

```
public class OperatorToken extends Token {
    public static final char Plus  = '+';
    public static final char Minus = '-';
    public static final char Mult  = '*';
    public static final char Div   = '/';
    public static final char LeftParen  = '(';

    private char operatorToken;
}
```

A cell might be represented like this:

```
public class Cell {
    private String formula;
    private int value;
    // the expression tree below represents the formula
    private ExpressionTree expressionTree;

    public void Evaluate (Spreadsheet spreadsheet);
}
```

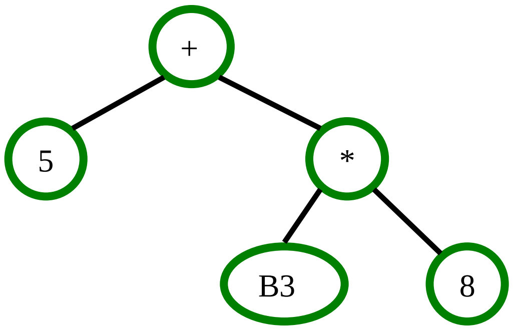
The nodes of the ExpressionTree (which is a binary tree, NOT a binary search tree) will look like this:

```
public class ExpressionTreeNode {
    private Token token;

    ExpressionTreeNode left;
    ExpressionTreeNode right;
}
```

And an ExpressionTree would look like this:

```
public class ExpressionTree {
    private ExpressionTreeNode root;

    public void makeEmpty();
    public void printTree();
    public int Evaluate(Spreadsheet spreadsheet);
}
```

As an example of what the expression tree would look like, suppose we had a formula such as "5 + B3 * 8". This would be represented by the following expression tree:



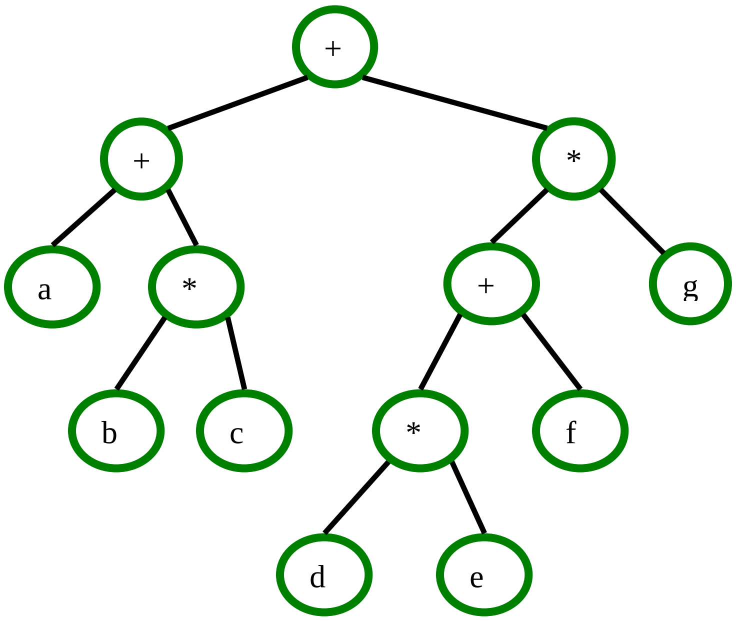
The spreadsheet will be a class with a (private) two-dimensional array of cells. (At first, you will want it to be a small array so that it is easy to test. Say, 4x4. Later, you will want to make it bigger. Use a static final variable to make it easy to change the number of rows and columns of the spreadsheet.)

Algorithms

You will need to implement topological sort over the cells to determine what order to evaluate them. Instead of numbering the cells (as presented in class and in the book), it is better to evaluate the cell's value as you do the topological sort. If you discover a cycle in the graph, then you should stop the topological sort, print an error message, and *set the formula of the cell that was changed back to what it was before the change*. (It turns out that the cells that were computed first in the new topological sort will have the same value that they had in the old topological sort (why?), so we don't have to go back and recompute them if a cycle is discovered.)

Handling the expressions is a little more complicated. Code is provided (in util.java) that will take a string that represents an infix expression, such as “5 + B3”, and produce a stack of tokens of a specific form. It will turn the infix expression into a postfix expression, following the algorithm described on pages 456-459 of Weiss. It is likely helpful to read the entire section 11.2 (pages 454-469). Thus, in our example, the stack, from bottom to top, will look like this: 5 B3 +. Note that util.java does not compile.

The expression tree that results from the postfix expression a b c * + d e * f + g * + :

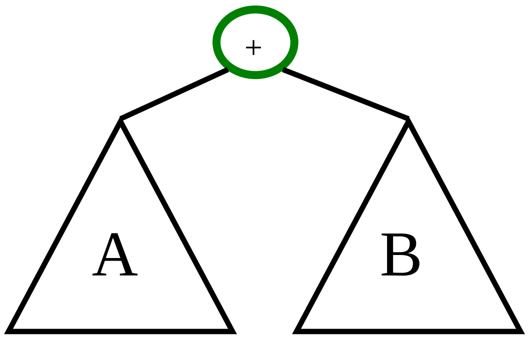


Handling the expressions is involved, but pretty easy. To get an expression tree from the postfix expression on the stack, pop an item off the stack. If it is a literal or cell, then we create an ExpressionTreeNode from the token and stop. If it is an operator, then we must generate a right ExpressionTree and a left ExpressionTree from what is left on the stack. We continue doing this until the stack is empty. Here is some code that expresses that algorithm:

```
// Build an expression tree from a stack of ExpressionTreeTokens
void BuildExpressionTree (Stack s) {
    root = GetExpressionTree(s);
    if (!s.isEmpty()) {
        System.out.println("Error in BuildExpressionTree.");
    }
}

ExpressionTreeNode GetExpressionTree(Stack s) {
    ExpressionTreeNode returnTree;
    Token token;

    if (s.isEmpty())
        return null;

    token = s.topAndPop();  // need to handle stack underflow
    if ((token instanceof LiteralToken) ||
        (token instanceof CellToken) ) {

        // Literals and Cells are leaves in the expression tree
        returnTree = new ExpressionTreeNode(token, NULL, NULL);
        return returnTree;
    } else if (token instanceof OperatorToken) {
        // Continue finding tokens that will form the
        // right subtree and left subtree.
        ExpressionTreeNode rightSubtree = GetExpressionTree (s);
        ExpressionTreeNode leftSubtree  = GetExpressionTree (s);
        returnTree =
           new ExpressionTreeNode(token, leftSubtree, rightSubtree);
        return returnTree;
    }
}
```

Be sure you understand how the postfix expression gets transformed into an expression tree using the algorithm described above.

To evaluate an expression tree, you will need to do a traversal of the tree. For example, to evaluate the following tree, you would evaluate subtree A, then subtree B, then add the two results. (What kind of traversal is this?)



Reading from and Saving to Files

For bonus, you can write methods that read and write a spreadsheet to a file. You will need to decide what the format of the file. A simple format would be that each line in a file is of the form <cell> <formula>. You would need to write code that will return a cellToken and a stack of tokens for each cell. You will need to convert the stack into an expression tree as you did for the "change cell" command.

Organizing Your Files

Be sure to organize your files in a sensible way. You will need files for a stack, a queue, an expression tree, a spreadsheet, and a variety of other files (such as one for the application itself and probably a separate one for the GUI).

What Code is Provided

util.java provides code to convert a string (representing a formula) to a stack of tokens. You will need to take these tokens and construct an expression tree, using the algorithm described above. You may take the code and alter it, simplify it, or add to it as you see fit. However, please document any changes you make (both in the code and in your report). Note that util.java does not compile.

You may use any of Mark Allen Weiss's code or any classes from the Java built-in library. The queue code, binary search tree code, and possibly some of the graph code might be useful. On the other hand, it might actually be easier to implement these data structures yourself. They are not that hard to implement if you understand them. Weiss's code contains a lot of functionality that we do not need for our purposes. His web site is http://www.cs.fiu.edu/~weiss/dsaajava/Code/

What to Turn In

Turn in all code you wrote or used. You should turn in a report (hard copy) that will act as a users manual (how to use the program) and a technical manual (describing the data structures and algorithms you used). You should also turn in one page that lists what part of the project each team member implemented.

Software Engineering Tips

Start with a small spreadsheet (4x4). If you have written your code correctly, it should be easy to make it bigger (by just changing one line of code). Put stubs in for functions the menu-driven user interface calls. Then work on changing the formula for one cell. Make sure you can print out a cell (as a formula). Then work on evaluating one cell that has no references to any other cells. Then work on the topological sort. Then you can calculate the entire spreadsheet. Then work on the GUI.

Test, test, test! Every time you add a significant feature to your code, test it before moving on to another part of the code. Work out on paper interesting spreadsheets to use to test.

Document, document, document! Put comments in your code before, during, and after you implement it. Does your code actually do what your comments say they do?

How Your Project Will Be Graded

The project counts as 80 points (equivalent to 8 regular homework problems). As usual, completeness, correctness, and style (readability, documentation) will be important criteria for your grade. The clarity of your user manual and report will also factor heavily into your grade, since they will help us understand your code. Unless there is some exceptional reason, all people in a team will receive the same grade.

Bonus (up to 15 points)
Here are some bonus points ideas. If you have some other ideas, talk to the instructor to determine how much work is involved (which will correlate with how many bonus points it will be worth). **Do not attempt bonus work until you have the "regular" part of the project working, tested, and documented.**

(4 points) Improve the user interface in some *significant* way. For example, print a useful error message any time the user enters a bad formula or bad cell reference.
(5 points) Add "Read Spreadsheet from a file" and "Save Spreadsheet to a file" functionality.
(5 points) Add exponentiation to the operations allowed in expressions. Exponentiation uses the caret ("^") and has higher priority than * or /. You will need to modify the expression parser, the code that generates the expression tree, and, of course, your expression tree evaluator.
(10 points) Sort by row (or column). The user picks a row (or column) and then the entire spreadsheet is sorted by that row (or column). Each column (or row, respectively) stays intact, but in the resulting spreadsheet, the values of the row (or column) chosen appear in ascending order.
(10 points) Implement other operators in a formula that can act on more than two operands. For example, you might define "AVG (cell1, cell2, cell3)" to be the average of the values in cell1, cell2, and cell3. The AVG function should be able to handle any number of arguments. You would have to change large parts of the code, including the expression parser. **This is hard.**