COP290 C Lab: Spreadsheet

Laksh Goel (2023CS10848)
Adit Jindal (2023CS50626)
Yuvika Chaudhary (2023CS10353)

March 2, 2025

1 About the Project

We are designing a command-line spreadsheet program that allows users to manage a grid of integer-valued cells interactively.
Users can set cell values directly or through formulas involving

1. Values such as a Constant or a cell
2. Arithmetic expressions
3. Functions such as MIN, MAX, SUM, and STDEV.

The spreadsheet supports a flexible range of cell references, including one-dimensional and two-dimensional ranges.
The program provides navigation commands for scrolling through the grid to handle large spreadsheets efficiently. Additionally, the execution time for each command is displayed, and invalid inputs are gracefully handled with error messages. The implementation requires efficient parsing, evaluation of expressions, and real-time updates, ensuring responsiveness and usability.

2 Built With

This project was built as a submission for COP290: Design Practices. The data structures and concepts used for building this Spreadsheet are:

1. **Graph** where each cell represents a Node in the graph. If there is a dependency of Cell1 on Cell2, then these two nodes are connected by a directed edge Cell2 $\rightarrow$ Cell1.
2. **Vector** is used for the lists where we don't know how many elements could be pushed into the list.
3. Concept of **Hashing** is used for saving the adjacency list of a Node. Amortized time for inserting and removing an element from HashTable is O(1). This reduces the time of execution of the program in comparison to using a Linked List.

3 Architecture

The spreadsheet comprises of two main data elements - graph and grid.

1. **graph** : (Graph) A custom struct including a matrix of Nodes, and dimensions- rows and cols.
2. **grid** : (int**) A table of values at Nodes.
3. **Node** : (struct) Custom struct containing the cell's function, inward dependencies (as Coordinates instances), ht for outward dependencies, and validity flag.
4. **value1, value2** : (Coordinates) These are present inside the Node struct, and mark the first and last cells of the range in a function, or the two cells included in the function. In case of constants in the function, these contain the constant value.
5. **ht** : (HashTable*) Pointer to HashTable, used for storing outward dependencies as Coordinates instances inside a hash table.
6. **Coordinates** : (struct) Two ints, holds the position of a cell as row and col.
7. **Hashtable** : (struct) Holds a table of Coordinates pointers, and its current and maximum sizes. It gets resized whenever the occupied space crosses 0.7 times the maximum size. Hash function used for cell (x,y) is $[((x + y) * (x + y + 1)/2) + y]\%max\_size$.

4 Test Suite

1. Cycle: Our test suite covers cases of cycle detection in test3.txt. Here the dependencies are : A1→B1→C1→E1 As soon as we try to add a dependency from E1→A1, we get an error of "Cycle not allowed". Further, changing value of A1 without disturbing dependencies works normally as we have appropriately handled cycle detection cases.
2. Err: Cases of ERR are covered in test2.txt. Our code is correctly handling division by zero errors in all cases which are checked comprehensively.
3. Parser (Out of bound/invalid inputs):
4. SLEEP : Cases of SLEEP are covered in test1.txt. Code is correctly handling cases of SLEEP of constant value, negative value, dependent values and when dependent values involved in SLEEP are changed, time is taken accordingly because every cell will take new time according to the changed values.
5. Full Sheet:

5 Challenges Faced

1. Choosing the data structure for each cell: We had to decide how to represent our cells in the program. Since we had interdependent connections between cells, Graph with each cell as a node would work well.

2. Implementing graph: Deciding what data elements to store in a node, how to reduce memory usage, and implementing functions were a challenge.

3. Parsing the commands and validating: We first implemented our program assuming valid arguments are given and parsed them. Later we extended our code to include Validation, if a valid argument is given at all.

4. How to get the sequence of nodes to iterate to update their values: To reduce time complexity, we thought of various ways to traversing graph while we had to update cells, and came up with an algorithm in which we will get a sequence of cell, iterating on which would lead to maximum O(r*c) complexity for updating cells.

5. Optimizing for time: Used concept of Hashing for storing dependencies of a node.

6. optimizing for space: Used Vector to implement a list(return list of topological sort) which had varying size, depending on outward dependencies of a Node.

7. Considering format for testing: We explored various tools like Cunit library, but failed. Since a single command runs almost all the functions, we thought of writing a single comprehensive test case where we would show working of all functions. But later we came up with a tool(check library), so we decided to further make unit tests.

6 Links

1. Github: https://github.com/Adit-Jindal/spreadsheet-c

2. Demo: "https://drive.google.com/drive/folders/1jiLZmvZxP4VhzoxIz_2RHNDPEOpCZPVp?usp=sharin

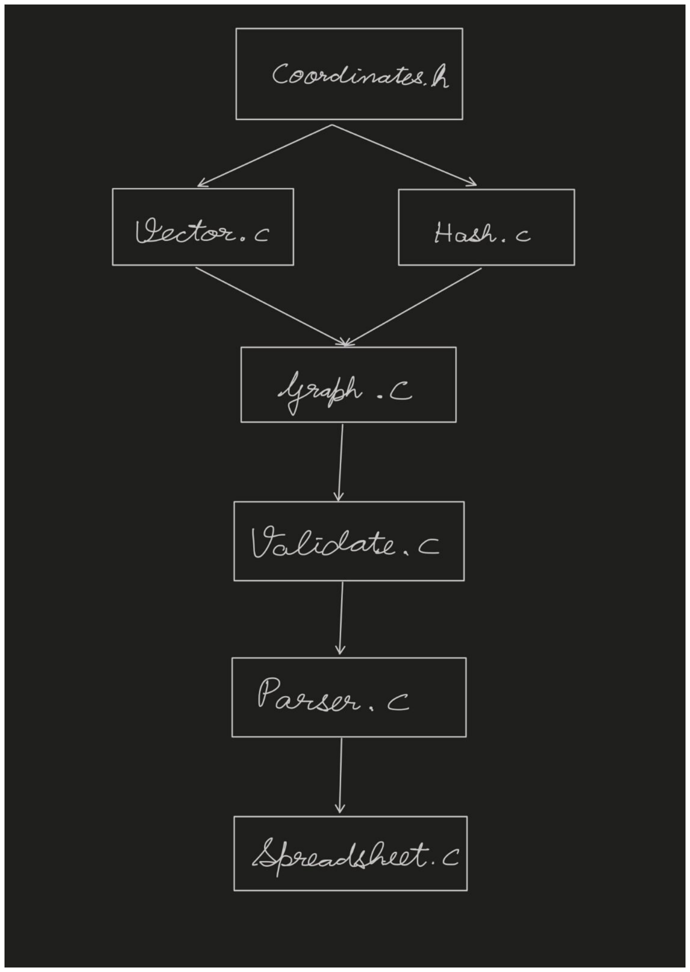

Figure 1: File Dependencies

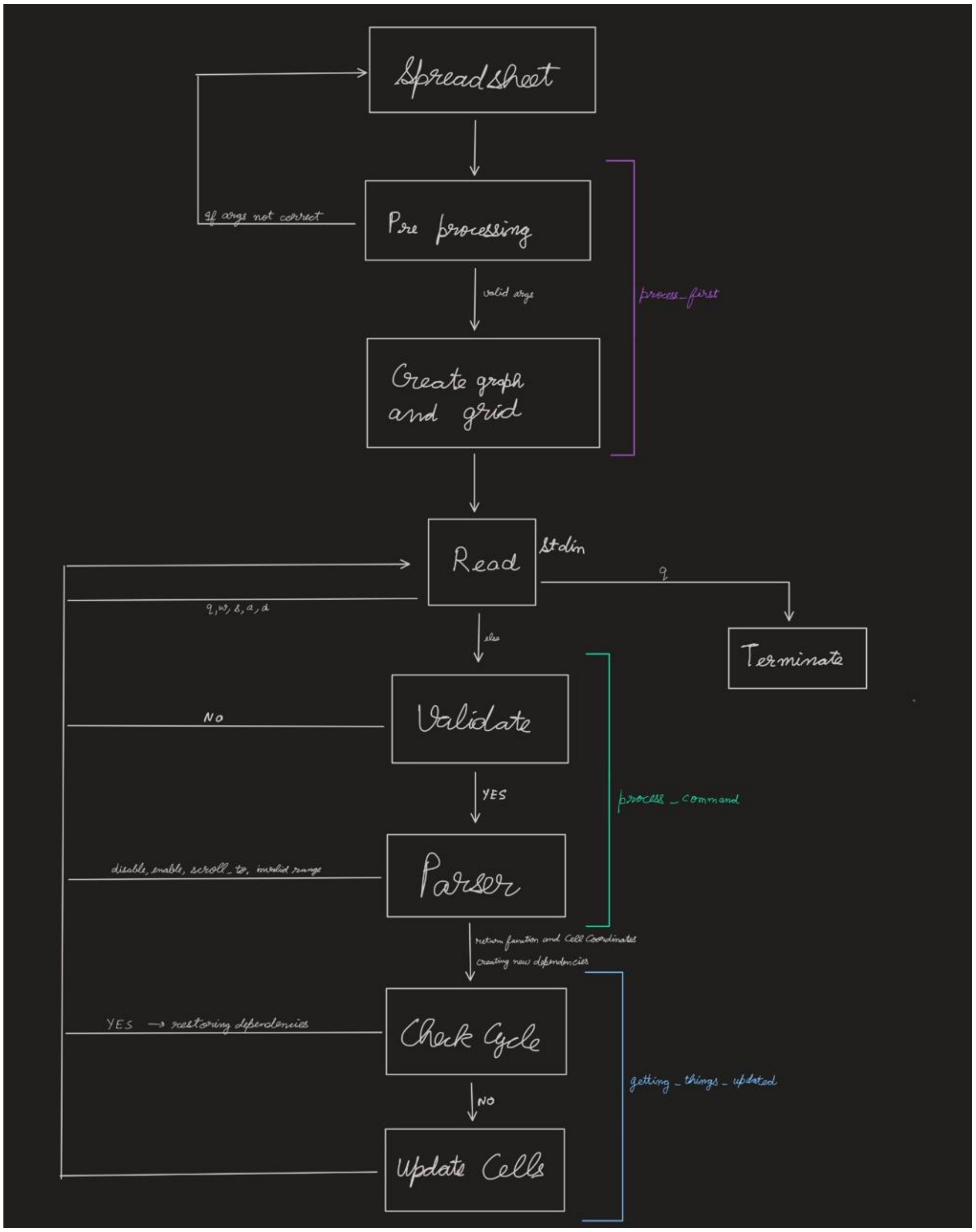

Figure 2: Flow of our program