# COP290 Rust Lab: Spreadsheet + Extension

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

April 25, 2025

## 1 About the Project

We have developed a versatile and feature-rich spreadsheet application in Rust that supports a command-line (terminal) interface and a web-based interface (mainly for display). This dual-interface design allows users to interact with spreadsheets in the environment that best suits their preferences, either directly in the terminal or through a graphical web interface in the browser.

### Terminal Interface

The terminal interface provides a keyboard-driven experience for managing a grid of integer-valued cells. Users can:

- Assign values to cells directly or via the following types of formulas:
    1. Constants or references to other cells
    2. Arithmetic expressions (addition, subtraction, multiplication, division)
    3. Range-based functions such as `SUM`, `AVG`, `MIN`, `MAX`, and `STDEV`
- Reference both one-dimensional and two-dimensional cell ranges in formulas
- Navigate large spreadsheets efficiently using commands (`w`, `a`, `s`, `d`) and jump to specific cells
- See execution time for each command and receive clear error messages for invalid inputs.

The terminal version emphasizes responsive parsing, real-time dependency updates, and robust cycle detection to prevent circular references.

### Terminal and Web Extension

Inside the Terminal, feature additions are:

1. Track and reverse changes using undo/redo functionality
2. Save to and load spreadsheets from JSON format for saving custom nodes, which allows preservation of dependencies among cells.
3. A web based display module for data analysis. It allows unit-changes only.

The web extension brings a modern, graphical spreadsheet experience to the browser, leveraging Rust compiled to WebAssembly (WASM) for performance and safety. Its key features include:

- Intuitive cell editing and formula entry through a graphical user interface
- Mouse-based range selection and navigation
- Dynamic charting and data visualization, including pixel art/image generation from spreadsheet data (heat maps)
- Undo/redo functionalities
- Theme switching (light/dark mode)
- JSON import/export for seamless data transfer between terminal and web versions
- Real-time updates and error feedback
- Shared backend logic with the terminal version for consistency

The extension is structured with a clear separation between backend logic (grid management, dependency tracking, formula evaluation) and frontend components (UI rendering, user interaction), communicating efficiently via browser APIs and WASM bindings.

## 2 Built With

The main data structures and concepts used in the Rust implementation are

1. **Dependency Graph**: The spreadsheet is modeled as a dependency graph, where each cell is a `Node`. E.g., if Cell1 depends on Cell2, a directed edge is formed from Cell2 to Cell1. This structure enables automatic propagation of updates and robust cycle detection. This dependency is stored in the Node, which gets stored in the JSON files used for serialization.
2. **Vectors (`Vec`)**: Rust's dynamic arrays are used for storing the grid of cells, lists of dependents, and for efficiently managing collections whose size may change at runtime.

3. **Strong Typing and Enums**: Custom structs and enums (such as `Node`, `Coordinates`, and `Operation`) provide clarity and safety in representing cells, positions, and supported operations.

4. **Efficient Evaluation**: The backend uses topological sorting to update dependent cells in the correct order whenever a value changes, ensuring consistency and performance.

5. **Error Handling**: Rust's type system and explicit validity flags in each node ensure that errors (such as invalid references, cycles, or division by zero) are detected and reported without crashing the program unless necessary.

6. **Extensible Parsing**: A modular parser interprets user commands, supporting a wide range of expressions and functions, and can be easily extended for new features.

**Additional Features:**

- Dual interface: Both terminal and web-based interfaces (with the web extension supporting advanced features like charts and pixel art)
- Undo/Redo: Track and revert changes to the spreadsheet
- JSON Import/Export: Save and load spreadsheet states
- Data Visualization: Generate images and charts from spreadsheet data (web interface)

The Rust implementation leverages safe and efficient data structures from the standard library, ensuring both performance and maintainability.

## 3 Encapsulation

The project is organized into well-defined modules to support both terminal and web-based spreadsheet interfaces, while also handling efficient command parsing and the backend logic. Below is a description of the main directories and files, based on the current repository structure:

- `terminal/`: Implements the terminal-based spreadsheet interface and its backend logic.
  - `backend.rs`: Core spreadsheet logic, including grid management, dependency updates, evaluation, and cycle detection.
  - `graph.rs`: Data structures and methods for the dependency graph and cell nodes.
  - `functions.rs`: Definitions for arithmetic and range functions (SUM, AVG, MIN, MAX, STDEV, etc.) and operation enums.

    - `parser.rs`: Parses and validates user commands.
    - `spreadsheet.rs`: Handles user interaction, grid rendering, and command execution in the terminal.
    - `types.rs`: Common types such as `Coordinates` for cell positions.
    - `mod.rs`: Module declarations for the terminal package.
- `extension/`: Contains the terminal integrated with the web-based spreadsheet implementation.
    - `backend/`: Backend logic for the web extension, mirroring the terminal backend structure.
        * `backend.rs`, `functions.rs`, `graph.rs`, `mod.rs`: Same responsibilities as their terminal counterparts, adapted for the web, for the sake of consistency.
    - `frontend/`: Frontend components for the web user interface.
        * `terminal.rs`: Terminal-like interface to manipulate and create spreadsheets to be displayed on the web as required.
        * `web.rs`: Main web UI logic.
        * `mod.rs`: Module declarations for the frontend.
    - `parser/`: Command parsing logic shared by the web extension.
        * `parser.rs`: Command parsing and validation.
        * `common.rs`: Shared parsing utilities.
        * `mod.rs`: Module declarations for the parser.
    - `mod.rs`: Module declarations for the extension package.
- `lib.rs`: Library root for shared logic.
- `main.rs`: Entry point for the purely-terminal based application (from C Lab).
- `extension_main.rs`: Entry point for the extension.

The modular structure ensures code reusability and maintainability, allowing both terminal and web interfaces to share core logic while providing interface-specific features.

## 4 Terminal Architecture

The spreadsheet application is structured around two main data abstractions: the **graph** (dependency graph of cells) and the **grid** (the table of cell values and operations). The following components define the architecture:

1. **graph**: A two-dimensional vector (`Vec<Vec<Node>>`) representing the spreadsheet's dependency graph. Each element is a `Node` struct corresponding to a cell, and the graph encodes dependencies between cells for correct evaluation order and cycle detection.

2. **grid**: The same 2D vector of `Node`s serves as both the dependency graph and the value grid, with each node storing its computed value, operation, and dependency metadata.

3. **Node**: A custom struct (`Node`) encapsulating:

   - `node_value`: The computed integer value of the cell.
   - `value1`, `value2`: Operands for the cell's operation, represented as `Coordinates` (for cell references) or as constants (with column set to `-1`).
   - `op`: The operation performed by the cell, as an `Operation` enum (e.g., Add, Sub, Sum, etc.).
   - `dependents`: A vector of `Coordinates` for all cells that depend on this cell.
   - `valid`: Boolean flag indicating if the cell's value is currently valid.
   - `visited`: Boolean flag used for traversal algorithms (e.g., cycle detection, topological sort).
   - `position`: The cell's own coordinates in the grid.

4. **value1, value2**: Instances of `Coordinates` within each `Node`, representing either:

   - The start and end cells of a range (for range operations).
   - The two operand cells (for binary operations).
   - A constant value (with column set to `-1`).

5. **Coordinates**: A struct (`Coordinates`) containing two `i32` fields: `row` and `col`, representing the position of a cell in the grid.

6. **Operation**: An enum (`Operation`) listing all supported cell operations, including arithmetic (Add, Sub, etc.), range functions (Sum, Avg, etc.), and special commands (Sleep, EnableOutput, etc.).

7. **Dependency Management**: Each `Node` maintains a list of its outward dependencies (cells that depend on it) via the `dependents` vector. Dependency edges are updated as formulas change, and cycles are detected using depth-first traversal with the `visited` flag.

8. **Evaluation and Topological Sorting**: When a cell is updated, the dependency graph is traversed in topological order to recompute all affected cells. The `evaluate_node` function handles computation, supporting both cell references and constants.

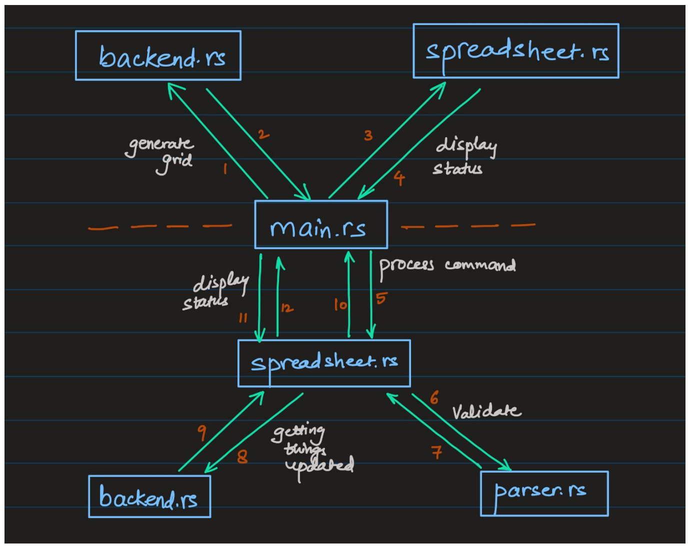


Figure 1: Interface between Modules in Terminal

9. **Parser and Frontend**: User commands are parsed into operations and operands, which are then dispatched to the backend for processing. The frontend handles grid display and user interaction.

**Note:** Unlike the C implementation, which used a custom hash table for outward dependencies, the Rust version uses a vector of `Coordinates` for each node's dependents, leveraging Rust's safe and efficient collections.

## 5 Extension Architecture

We have extended the project described above to add more functions to it and make it more relevant to a real-world user. This involves having features for saving and opening a saved file in a custom format, undoing and redoing commands, and displaying the sheet in a web-based GUI for better data analytics. The features added are thus:

1. **Undo-Redo stack**: All executed operations are stored in a stack. This allows the user to run *undo* to revert the last change, and running *redo* brings the change back, along with its dependencies. This allows reverting errors, making the program more useful. The stack has an upper limit of 1000, so up to one thousand commands can be undone.

2. **Serialization**: We can save the sheet formed in a JSON file. For this, the user needs to run $save myfile.json$, by replacing 'myfile' with the desired name. This file gets saved in the project directory directly, and the sheet can be further worked upon. This feature can be used as a

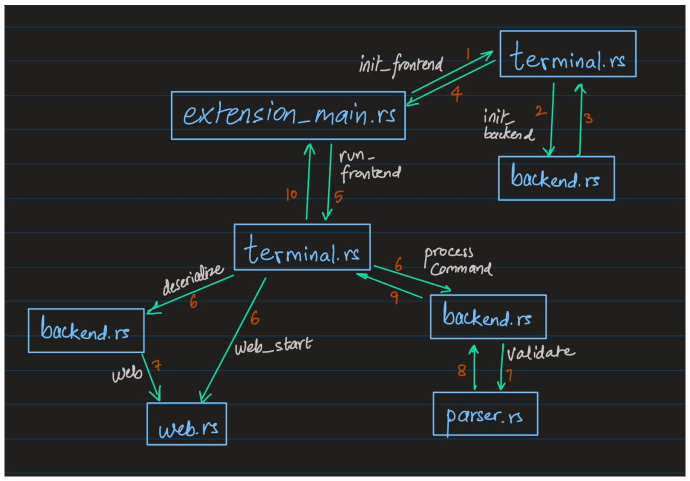


Figure 2: Interface between Modules in Extension

simpler substitute for version control, though not completely replacing it. The JSON file formed contains descriptions of all nodes in the table, thus preserving dependencies.

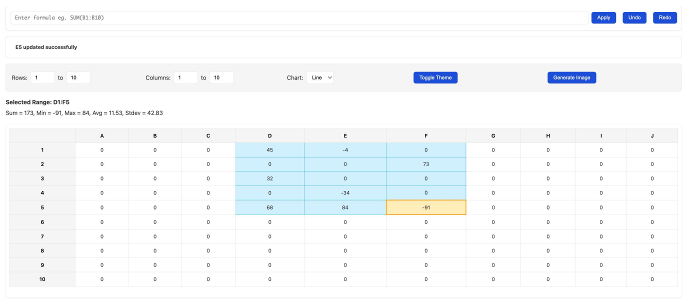
3. **Deserialization**: Once a file has been saved using the above-described command, it can be opened again later and continued to be worked upon. For this, while running the executable, additional arguments need to be passed in the terminal. The first two arguments are taken to be the dimensions (rows and columns), and the third optional argument is the file to be opened.
   e.g., for a file named 'myfile.json', the following command will work: `./"executable" 5 6 myfile.json`.
   This will open the file in the terminal interface, along with all the dependencies preserved.
   The dimensional arguments in the command serve as safety, in case the file is not accessible or does not form a sheet properly. In such a case, the dimensions passed are used to create a new sheet, completely ignoring the file.

4. **Web-based display**: We also have a web-based GUI, intended for data analysis. It allows several features and operations, as specified below. The GUI can be accessed by two means:

   (a) `web_start`: This opens the current state of the sheet in the web.

   (b) `web myfile.json`: This opens a pre-existing file in the web, rather than the current state in the terminal.

   The GUI is intended only for analyzing the current state, and to try out unit changes in it and observe their effects. For this reason, it allows only



E5 updated successfully

Rows: 1 to 10 Columns: 1 to 10 Chart: Line

**Selected Range: D1:F5**

Sum = 173, Min = -91, Max = 84, Avg = 11.53, Stdev = 42.83

| | A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0 | 0 | 0 | 45 | -4 | 0 | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 0 | 0 | 0 | 73 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 32 | 0 | 0 | 0 | 0 | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | -34 | 0 | 0 | 0 | 0 | 0 |
| **5** | 0 | 0 | 0 | 68 | 84 | -91 | 0 | 0 | 0 | 0 |
| **6** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **7** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **8** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **9** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **10** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 3: Range Data Analysis

one change at a time, that is, making a second change undoes the first. It allows the user to undo the last command performed in the sheet (again, up to a depth of one).

## 6 Features of the web-display

1. Displays the table in a more interactive manner. Lets the user select the range of rows and columns to display, thus allowing focus on the required sections of data alone.

2. Allows selection of ranges of cells(1D or 2D), which displays statistics for the selected range. The statistics include the mean, sum, maximum, minimum and standard deviation.

3. Has a formula bar, which allows the user to change the value in a cell. This has an additional feature, where if a cell address appears in the formula for a cell value, then instead of having to type in the address, we can also choose to click on the cell in the table. This would print the cell address in the formula bar in position of the cursor.

4. Allows undo and redo, by a depth of 1 command.

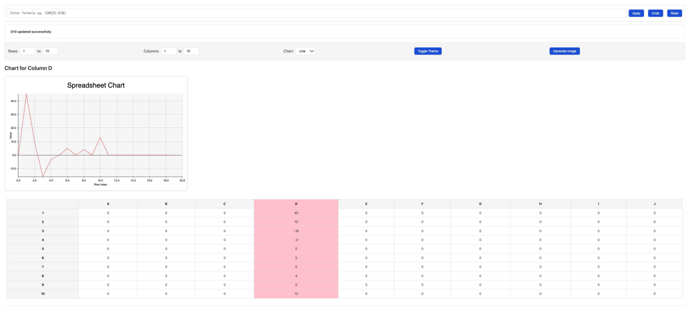
5. Creates charts for columns. Clicking on a column header generates the chart for it above the table. This chart, by default, is a line chart, but can be switched to a bar chart from a drop down list.

6. Generates a heat map of the table. Maps the values inside a cell to an RGB value, using the following formula:

   $$((n >> 16)\&\&(0xFF), (n >> 8)\&\&(0xFF), n\&\&0xFF)$$

   Any number which breaches the upper limit of numbers mappable to RGB values is mapped to the highest value, i.e., 0xFFFFFF (white). The maximum value is 16,777,215.

7. Provides option for choosing between dark and light themes.



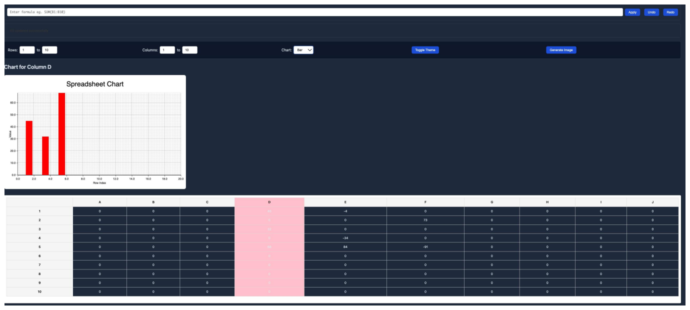
D10 updated successfully

Rows: 1 to 10    Columns: 1 to 10    Chart: Line

Chart for Column D

| | A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 45 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | -16 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | -3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 13 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 4: Line Chart



Rows: 1 to 10    Columns: 1 to 10    Chart: Bar

Chart for Column D

| | A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 45 | -4 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 73 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 32 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | -34 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 68 | 84 | -91 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 5: Dark Theme

## 7 Test Suite

Our Rust project features a comprehensive and automated test suite, leveraging Rust's unit testing framework to ensure correctness, robustness, and maintainability of both terminal and web spreadsheet implementations. Tests are organized within each module using the `#[test]` macro, following Rust best practices.

1. **Core Functionality:**
    - **Arithmetic and Range Functions:** All spreadsheet functions (SUM, AVG, MIN, MAX, STDEV, etc.) are covered by unit tests. These tests verify correct results for single cells, rows, columns, rectangles, and edge cases such as empty or invalid ranges, as well as cells containing invalid values.
    - **Cell Operations:** Unit tests validate all arithmetic operations (addition, subtraction, multiplication, division), including correct handling of division by zero and propagation of invalid states.
    - **Node and Grid Methods:** The `Node` struct's methods (get/set value, position, validity, dependencies) are thoroughly tested for correct behavior and edge cases.

2. **Dependency Graph and Cycle Detection:**
    - **Dependency Management:** Tests confirm that adding and breaking dependency edges between cells works as intended for both binary and range-based operations.
    - **Cycle Detection:** Both direct and indirect cycles are tested. The suite ensures that attempts to create cycles (e.g., $A1 \rightarrow B1 \rightarrow C1 \rightarrow A1$) are detected and prevented, and that the spreadsheet remains consistent after such operations.
    - **Topological Updates:** Tests verify correct propagation of updates through the dependency graph, including topological sorting and resetting of traversal flags.
3. **Parser and Command Validation:**
    - **Valid Commands:** The parser is tested for correct interpretation of valid cell assignments, arithmetic expressions, range functions, and special commands (undo, redo, navigation, save, web, etc.).
    - **Invalid Inputs:** Extensive tests ensure that out-of-bound cell references, malformed commands, and invalid function names are gracefully rejected with appropriate error handling.
4. **Frontend and Navigation:**
    - **Grid Display and Navigation:** Tests cover navigation commands (w/a/s/d), scrolling, and grid rendering, including edge cases at grid boundaries and toggling output display.
    - **Frontend Initialization:** Both terminal and web frontends are tested for correct initialization, command processing, and integration with their respective backends.
5. **Web Extension Specific:**
    - **Backend Logic:** The web backend is tested for grid creation, value assignment, undo/redo functionality, serialization/deserialization, and error reporting (e.g., invalid ranges, circular dependencies).
    - **Frontend Integration:** Web frontend tests ensure correct navigation, grid display, and command execution, including edge cases and error handling.
6. **Special Features:**
    - **Undo/Redo:** Tests verify that undo and redo commands correctly revert and reapply changes, maintaining spreadsheet consistency.
    - **Serialization:** The suite includes tests for saving and loading spreadsheet state to and from files, ensuring data integrity.

- **Output Control:** Commands for enabling/disabling output are tested for correct frontend behavior.

7. **Error Handling:**

   - **Division by Zero, Invalid Ranges, and Malformed Commands:** All error cases are tested to ensure the program does not panic and provides informative feedback to the user.

The test suite is automated and can be run using `cargo test`, providing rapid feedback during development. This ensures high reliability and allows safe refactoring and extension of the codebase.

## 8 Why Few features could not be implemented?

1. **Copy Paste:** Copy pasting a single cell is easy since only a single dependency is checked for cycle. Copy pasting a range would required saving all old and new values and checking cycle for each cell, one at a time. This would take a lot of space and time of our systems. So we decided to include it as our future extensions.

2. **Find and Replace:** Finding a value in a graph is easy(but would take time). Replacing each found value with a formula would consume lot of resources since again we have to check for each cell if there is a cyclic dependency.

## 9 Why this is a good design?

Our extension has a very modular structure, as can be seen in Section 3. The only major issue which Terminal-based Spreadsheet users face is lack of visual data analysis, line graphs, bar graphs and themes. We understood the problem and made a web interface for data analysis of data present in terminal and make unit changes to analyse it. We have features of selecting a range of cells to show it's statistics like Min, Max, Avg, Stdev and Sum. We have Line and Bar graphs We have Dark theme since users have different preferences of light and dark theme.

If a user want's to open data on web and edit it right there, we have another extension where we can open a json file in web.
We can also store and open data in terminal for editing and saving for future use. Overall, we believe that it would help users for Data Analysis and a better User experience.

## 10 Challenges Faced

1. Deciding the basic framework of the extension took some time.

2. The basic framework and function signatures are necessary to be decided right from the beginning in a team-based project.
3. We had to learn how to implement web using Rust.
4. Adding features for Plotting graphs, Selecting range and Serialization took time.
5. We faced challenge in integrating our basic terminal code with our extension (terminal + web)
6. 'trunk Serve' command is not working on Baadal. We are spending a lot of time figuring that out.

## 11 Links

1. **Github**: `https://github.com/lakshgoel5/SpreadsheetRust.git`