Rust Lab: Extended Spreadsheet Program Documentation

Mannat (2023CS10138)
Yuvraj Sahu (2023CS50678)
Pearl Bugalia (2023CS10489)

April 2025

1. Features Included in the Implementation

- **General Formula Support**
  - Now the spreadsheet support general expressions as formula like $A1+B1+C1$, $A1+Max(A1:A5)$ Please use Max instead of MAX and similarly for all other funtions
  - Cell assignments can be done via both the formula bar at the top of the Cell using the syntax *cell name* = *arithmetic expression*
  - Also selecting the Cell and then entering the formula inside the cell itself using the syntax = *arithmetic expression*
  - To switch between entering formula inside the execution bar and inside the Cell box click on the Black bar at the top left corner of the window.
  - To scroll in the window use the arrow buttons like → at the corners of the window.
  - We can also enter w,a,s,d in the formula bar/execution box for scrolling.
- **User Interface**
  - We have added a user interface into our project using the fltk crate to display the contents of the Spreadsheet and also the formula inside the cell
  - To make the contents of a Cell bold enter syntax *b cellname*
  - To make the contents of a Cell bold enter syntax *i cellname*
  - Cell which have error values are displayed in Red Colour and pop message appears to prompt the user to correct itself.
- **Common Functionalities**

  - Cut command for single cell syntax *dc cellname target_cell_name*
  - Cut command for Range *d cell_range target_cel_range*
  - Copy command for single cell syntax *yc cellname target_cell_name*
  - Copy command for Range *y cell_range target_cel_range*
  - Undo command press Undo button on the top right corner or enter undo on the formula bar
  - Redo command press Redo button on the top right corner or enter redo on the formula bar

- **Graph**
  - Forecast - Click on the *Plot Graph* button on the top right corner and then a pop up window appears, in that window select forecast and in the input bar enter the following syntax *numberofinstances X_Range Y_Range name.png* use Spaces please
  - Plot - Click on the *Plot_Graph* button on the top right corner and then a pop up window appears, in that window select either *plot line* ,*plot scatter* or *plot histogram* depending on the type of plot you want and in the input bar enter the following syntax
    * *Plot_histogram* : *Range name.png* use Spaces please
    * *Plot_line* : *Range name.png* use Spaces please
    * *Plot_scatter* : *X_Range Y_Range name.png* use Spaces please

2. Features Not Included in the Implementation

The following features from the proposal were not implemented due to various constraints:

- **Advanced Formatting**:
  - **Underline Formatting**: While bold and italics were implemented, underline formatting (`u <cell_name>`) was not included.
  - **Range-Based Formatting**: Applying formatting (e.g., bold, italics) to a range of cells was not implemented.
- **Advanced Graph Features**:
  - Support for additional graph types (e.g., bar charts, pie charts) was not included.
  - Customization options for graph styles (e.g., colors, labels) were not implemented.
- **Dynamic Cell Dependencies**: Basic dependency tracking was implemented, but dynamic updates to dependencies (e.g., when formulas are modified) were not fully supported.

- **Concurrency**: Although Rust's concurrency features were leveraged in some areas (e.g., `SLEEP_CHANNEL`), full parallel processing for large datasets or graph generation was not implemented.
- **User Interface**: A graphical user interface (GUI) for interacting with the spreadsheet was not included. The system relies on command-line input.

3. Could We Implement Extra Extensions Over and Above the Proposal?

Yes, additional extensions could be implemented to enhance the system:

- **Advanced Formatting**: Add support for underline formatting and range-based formatting commands.
- **Graph Customization**: Allow users to customize graph styles (e.g., axis labels, colors, gridlines).
- **Dynamic Dependency Updates**: Automatically update dependent cells when a formula or value changes.
- **Enhanced Error Handling**: Provide detailed error messages and suggestions for resolving issues.
- **GUI Integration**: Develop a graphical interface using libraries like `fltk` or `egui` for better user interaction.
- **Export Functionality**: Add support for exporting the spreadsheet to formats like CSV or Excel.

4. Primary Data Structures

The primary data structures used in the system include:

- `Cell`: Struct representing each cell in the spreadsheet with the following fields.
  - `value`: Stores the cell's i32 value.
  - `formula`: Contains the Expr this Cell is assigned to.
  - `is_bold`, `is_italics`: Boolean flags for formatting.
  - `dependents`, `precedents`: The spreadsheet is internally represented in a graph format with each cell having precedents and dependents forming the edges to and from each cell. This is similar to adjacency list representation but it is a HashSet¡(row,column)¿.
- `Spreadsheet`: Represents the entire spreadsheet with fields such as:

  - `rows`, `columns`: Dimensions of the spreadsheet.
  - `all_cells`: A 2D vector of `Cell` objects.

- `Expr`: A recursive data type which contains the Abstact syntax tree built from the expression this cell was assigned to with the following variants.
  - `Number`: A constant value.
  - `Cell`: A reference to another cell as (row,column)
  - `BinaryOp`: A recursive variant with operands as Expr and a char representing the operation to be performed
  - `Function`: A recursive variant with String representing the type of function to be performed and a list of operands which are Expr`SUM`, `AVG`).

- `UndoRedoStack` Helper Stack for supporting undo redo operations with the following fields
  - `undo stack` : Contains the state of the last 17 operations that have been performed in the Stack implemented using a Vec¡ CellState ¿. Rather than storing the entire snapshot of the sheet in the stack we are storing only the min info required for reverting back to the previous state for memory efficiency.
  - `redo stack` : Contains the state implemented as a Stack for supporting redo operations. On a redo operation , state is pushed onto the stack.

5. Interfaces Between Software Modules

The overall workflow can be represented using the following flowchart The architecture demonstrates clear separation of tasks between the separate modules.

  - A separate thread is created for GUI application and it is separate for the thread used for calculation and cmd execution.
  - Data is shared between the thread using Arc¡T¿ which is a thread safe pointer to heap allocated memory for multiple ownership of the data between the GUI and the cmd execution functions
  - The thread for GUI uses polling mechanism

    $$input\_text.lock().unwrap()$$

    for communicating new input to the GUI thread the GUI thread is then updated after performing each command though the *launch_gui*() function provided by *display.rs* which is given the complete shared data as input via Arc reference.

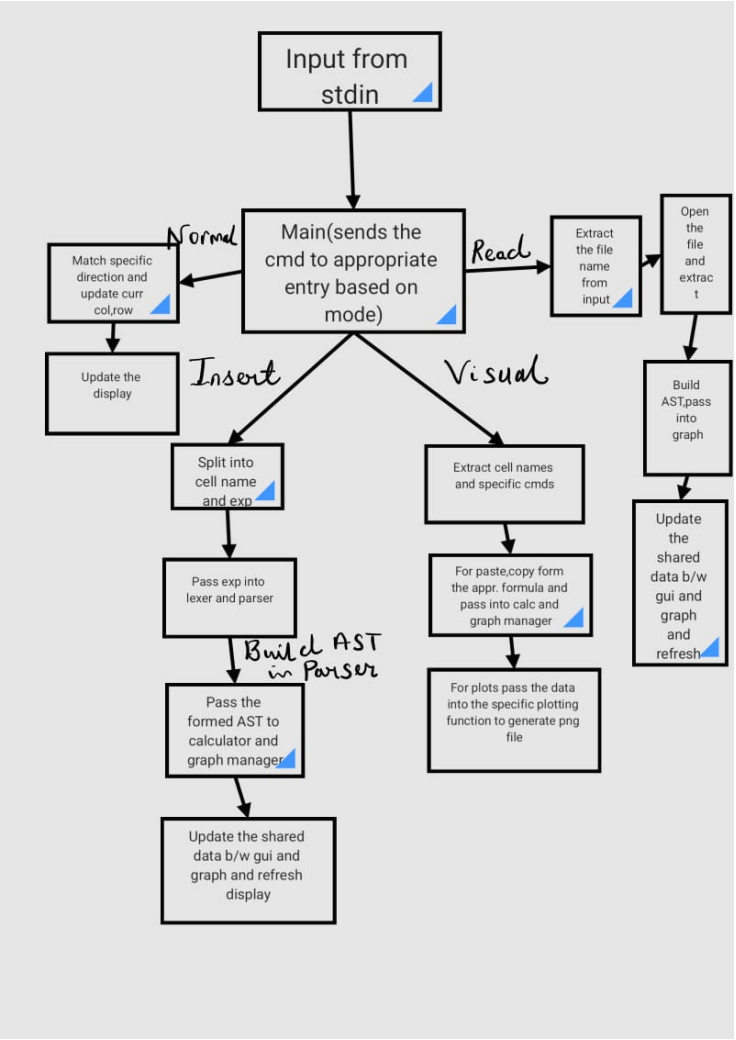

Figure 1: Overall workflow of the project

- The main function takes input from the Stdin as a String
- This input is passed along with the borrow of the current Spreadsheet state to different parser based on the current mode of the Spreadsheet.
- The mode of the Spreadsheet is communicated through the global *static mut CURRENT_MODE* which changes when instructed by the user.
- For assignment operations the input is split around = and the assigned expression is then passed into the Lexer.

- The tokenised input is then passed into the $parse\_formula()$ function provided by the parser file $formula.rs$ which is the implementation of the Grammar in $formula.lalrpop$
- The tokenised expression is then passed as a sequence to $parser\_formula$ which build the AST from the tokenised input.
- In *main* itself this built AST is then passed as input to the $assign_cell()$ function which is provided by the $graph\_extension.rs$
- The $assign\_cell()$ then updates the values and this change is communicated to the GUI through the Arc reference after which the GUI is then refreshed.
- In READ mode the filename is extracted from the cmd and in *main* this is passed as an input to $handle\_read\_command()$ given by the $read.rs$ file
- Inside $read.rs$ the comma separated values for each cell are extracted either as a value or a formula to each cell, This is then passed to $assign\_cell()$ in $graph\_extension.rs$ along with the current state of the Spreadsheet.
- In VISUAL mode the entry point is diverted to $parser\_visual()$ function provided by the $parser\_visual\_mode.rs$ which identifies the specific command and then performed operation accordingly.
- For cmds like "b" or "i" just the flag $is\_bold()$ or is_italic is updated for that specific cell
- For cut, copy, paste commands the *Expr* within the copied cell is extracted and then assigned to the target cell via $assign\_cell$ function provided by the $graph\_extension.rs$ module.
- For plotting functions the data is passed to the specific plot function like $plot\_line()$ which is provided by their specific module $plot\_line.rs$

6. Approaches for Encapsulation

Encapsulation is hiding concrete implementation details from the end user Our project uses multiple such strategies to achieve encapsulation.

- The project is separated into dedicated modules each having its own specific functionality.
- For example module $graph\_extension.rs$ manages cell dependency tracking and recalculation of dependents, module $formual.rs$ parses the tokenised expression and generates the AST etc
- Each module exposes only specific functions to interact with other modules using the *pub* keyword thus encapsulating the other logical parts of the code not using the *pub* keyword and providing only useful APIs

- We also have created encapsulation via Type Hierarchies. For example the *Cell* struct contains all the relevant info about the state of the cell but it exposes data only when called to do so. So we have used Data Encapsulation here.
- The code effectively uses the Rust ownership system for communicating info between the different modules and APIs like when we need to mutate the data we can pass mutable references but for checking states like div by zero, or cyclic dependence we can pass immutable refernces.
- Thread-Safe Concurrency Patterns- The project employs thread-safety encapsulation using Rust's concurrency primitives.This ensures that shared state is only accessed in controlled ways, preventing data races while allowing concurrent operations like UI updates and calculations to happen safely.

7. Justification for good design

- Our project extensively uses the Rust's module system separating the project into concrete parts that manage specific functions and exposing APIs for use by the other modules
- In implementing undo and redo command we made sure to make our program memory efficient by storing only the required minimum information and not take the entire snapshot of the Spreadsheet.
- For maintaining thread safe concurrency we have used Arc¡T¿ references which allow thread safe sharing of heap data.
- We also adopt a similar style to Object Oriented design where all the code manipulating the object is in one single place
- For iterating through collections we have used iterators instead of indices to prevent panic of the program.
- We also have followed the official Rust style using rustfmt (cargo fmt) and clippy (cargo clippy) to ensure formatting and lining.
- All the crates we have used are well maintained and popular crates with large and active communities.
- We also have used the Rust's ownership and borrowing system effectively to managed shared data and communicate between the different modules.
- We also avoided the use of unsafe block as far as possible, using it only when it is necessary for global variables that must live throughout the program.