COP290 Semester 2024-2025 Sem II

C Lab: Spreadsheet program

You will create a spreadsheet program that manages a grid of cells; each cell contains a value. Users can set cell values directly (e.g., A1=3) or they can set cell values via formulas (e.g., B1=A1+1). Your spreadsheet program shall support user interactions with a simple command-line interface described below.

Interface

Running `make` compiles the spreadsheet and produces an executable called `sheet`. Running `./sheet R C` starts the program with a spreadsheet of size `R x C` with `1 <= R <= 999 and 1 <= C <= 18,278 (=26*26*26+26*26+26)`, i.e, cells can be from A1 to ZZZ999. Each cell can only hold integers. All cells are initially 0.

When we run the program it shall show the sheet and wait for a user input. After the user provides an input, the program updates the sheet, shows the updated sheet, and waits for another user input. Following shows an example (we have made the changes bold and blue just for readability of this document, your program shall not output bold and blue text):

```
$ ./sheet 2 2
      A     B
1     0     0
2     0     0
[0.0] (ok) > A1=2
      A     B
1     2     0
2     0     0
[0.0] (ok) >
```

Before “>”, the program shows the time it took to run the last command within square brackets in seconds, and the status of running the last command within parentheses. `(ok)` is reserved for successful command runs. If the user provides an invalid command, it shows an error string instead. Example below:

```
$ ./sheet 2 2
      A     B
1     0     0
2     0     0
[0.0] (ok) > random user input not expected by the program
      A     B
1     0     0
2     0     0
[0.0] (unrecognized cmd) >
```

Valid user inputs

Control inputs

The program shows a maximum of 100 (=10x10) cells at any given time. To support large sheets, users can scroll the sheet with:

- `w` for up,
- `d` for right,
- `a` for left, and
- `s` for down.

Each scroll command scrolls the sheet by 10 rows/columns.

`q`: exit the program.

Formula inputs

All formulas are of the form: `Cell=Expression.`
`Cell`. Examples: A1, ZA222.
`Expression` can be a

- `Value`:
  - `Constant`. Example: 23, 41.
  - `Cell`. Example: C1, Z22.
- `Arithmetic expressions`:
  - `Value+Value`
  - `Value-Value`
  - `Value*Value`
  - `Value/Value`
- `Function:`
  - `MIN(Range)`
  - `MAX(Range)`
  - `AVG(Range)`
  - `SUM(Range)`
  - `STDEV(Range)`
  - `SLEEP(Value)`
- `Range` can be
  - One-dimensional. Example: `A1:A20` (cells in column A from rows 1 to 20), `A1:F1` (cells in columns A to F in row 1). `A1:A1` is a valid range with just the cell `A1`. `A9:A1` is not a valid range.
  - Two-dimensional. Example: `A1:D10` (cells in the 2D grid from columns A to D and rows 1 to 10). `D10:A1` is not a valid range.

Note that ranges are inclusive, i.e, it includes both first and last cells.

All expressions evaluate to integer values. The program drops digits after the decimal point when necessary. For example, `A2=5/2` sets `A2` to 2. `SLEEP(2)` sleeps the program for 2 seconds and returns 2. Other expressions are hopefully self-explanatory.

You must handle formulas without any spaces, e.g., `A1=B1+C1`. You can optionally choose to allow spaces in formulas, e.g., `A1 = B1 + C1`.

Putting it all together, the following shows an example user interaction with the sheet:

```
$ ./sheet 2 2
      A     B
1     0     0
2     0     0
[0.0] (ok) > A1=2
      A     B
1     2     0
2     0     0
[0.0] (ok) > B1=A1+1
      A     B
1     2     3
2     0     0
[0.0] (ok) > A2=MAX(B1:A1)
      A     B
1     2     3
2     0     0
[0.0] (Invalid range) > A2=MAX(A1:B1)
      A     B
1     2     3
2     3     0
[0.0] (ok) > B2=SLEEP(2)
      A     B
1     2     3
2     3     2
[2.0] (ok) > q
```

Error Handling

You should think about errors that can happen while a user interacts with your spreadsheet and handle them appropriately. The following is a non-exhaustive list of errors:

- Incorrectly running the program. Example: `$ ./sheet 2 hello`
- Invalid input. Example: `> hello`
- Operations on invalid cells. For example, if the size of the sheet was 2x2, `> A3=3`
- Undefined calculations. Example: `> A1=2/0`
- Circular formulas. Example: `> A1=A1+1`

Recalculations

The spreadsheet shall support *automatic recalculations*. Let us take another look at the above user interaction: A1=2, B1 = A1 + 1, and A2 = MAX(A1:B1). Now, if the user sets A1 to 5, the program needs to recalculate B1 which becomes 6. Since both A1 and B1 have changed, the program needs to recalculate A2 which also becomes 6.

The spreadsheet shall only do *necessary* recalculations, i.e, it does not recalculate unrelated formulas. In the example above, when we change the value of A1 to 5, we skip recalculating `B2=SLEEP(2)`. By doing only necessary recalculations, the sheet shall update faster.

The program needs to track what recalculations are required after each update and accordingly trigger recalculations. Analyze the following interaction carefully. Specially note that after B1=1; A1=5 does not trigger recalculations for B1 and B2!

```
$ ./sheet 2 2
      A     B
1     0     0
2     0     0
[0.0] (ok) > A1=2
      A     B
1     2     0
2     0     0
[0.0] (ok) > B1=A1+1
      A     B
1     2     3
2     0     0
[0.0] (ok) > A2=MAX(A1:B1)
      A     B
1     2     3
2     3     0
[0.0] (ok) > B2=SLEEP(B1)
      A     B
1     2     3
2     3     3
[3.0] (ok) > A1=1
      A     B
1     1     2
2     2     2
[2.0] (ok) > B1=1
      A     B
1     1     1
2     1     1
[1.0] (ok) > A1=5
      A     B
1     5     1
2     5     1
[0.0] (ok) q
```

Deliverables

1. Source code
    - Well-documented C source code files with clear separation of logic into functions and files.
    - Include comments explaining the purpose of each function and major code sections.
    - All .c and .h files submitted must be written by your group. Code copied from the internet and from each other counts as plagiarism.
2. Test suite
    - Test cases, also written in C, that automatically run your program and test against expected output.
3. Demo video (< 3 minutes)
    - Each project member demonstrates a different behavior of the program. The speaker needs to have their camera on.
4. Github link
    - We will judge contributions of individual members using code commits to your github repository. Make sure that each member commits their contributions regularly to the repository. Please keep your repository private until the end of the course. Add sourabhcop290 as a collaborator to your repository.
5. Report (2-4 pages)
    - A concise document written in LaTex that explains the design decisions, challenges faced, and structure of the program.
    - The document also discusses edge cases and error scenarios that your test suite covers.
    - The document shows a diagram of your software explaining key files, data structures, and functions.
    - Gives the video demo link and github link.
6. Makefile:
    - Running `` `make` `` should produce a "`sheet`" executable file that can be run as `./sheet 10 10`.
    - Running `` `make test` `` runs all your test cases.
    - Running `` `make report` `` should produce a "`report.pdf`" from the LaTex source.

Submission Instructions

- Compress all files into a single .zip file and upload it to Moodle.
- Include entry numbers of all project members in the archive name: <EntryNumber1>_<EntryNumber2>_<EntryNumber3>.zip