ELEC 278 Assignment (S001)

Presented to: Dr. Jianbing Ni and Mr. Nicholas Mertin

Final Report

ELEC 278 – Fundamental of Information Structures

Electrical and Computer Engineering

Faculty of Engineering and Applied Science

Queen's University

Prepared by Hendrix Gryspeerdt

I, (Hendrix Gryspeerdt), attest that all of the materials which I am submitting for this assignment are my own and were written solely by me. I have cited in this report any sources, other than class materials, which I used in creating any part of this assignment. Furthermore, this work adheres to the policy on generative artificial intelligence as documented in the instructions.

Date of Submission: 6th December, 2023

Executive Summary

The goal of this assignment was to implement the necessary functions to complete a simple spreadsheet program that runs in a terminal. The finished program fulfills the following functional requirements (FRs):

1. A user can navigate between cells and modify or clear the value of each.
2. When a user enters a new value, it is interpreted as either text, a number, or a formula.
    a. A value which starts with an equals sign (=) is always interpreted as a formula; if it does not form a valid formula, then the user is shown an error.
    b. A valid formula consists of one or more numbers or cell coordinates (e.g., B7) separated by addition signs (+).
    c. A value which consists of one or more decimal digits (0 through 9) with at most one decimal point is interpreted as a number.
    d. A value which does not start with an equals sign and is not a number is interpreted as text.
3. When a user navigates to a cell, a textual representation of its value is shown in an editable field (i.e., the content field at the top of Excel's interface), and the user can edit the value of the cell based on that representation. When the cell contains a formula, this should be a representation of the formula itself, not the computed value (which is shown in the cell itself).
4. When the value of a cell changes, the displayed contents of all formula cells which (directly or indirectly) depend on it are updated.

This report contains detailed explanations on the data structures and algorithms used to meet the FRs and the following non-functional requirements (NFRs):

1. For each algorithm implemented, it should not be possible to achieve a better time complexity using only the tools taught in this course.
2. Any dynamically allocated memory should be freed once it is no longer needed.
3. The code must be sufficiently organized and commented for a software engineer with appropriate background knowledge to maintain.

For Example, to handle FR4, a topological sort of the dependency graph was computed to determine the order in which to update dependent formula cells. Each of the functional requirements were tested by writing tests in the file tests.c and running those tests using testrunner.c. Extra features were added to extend the basic requirements of the FRs and those are detailed in the section titled Extra Features.

1 Design Proposal

My approach to solving this problem was to do the following steps in order:

1. Understand what each function the interface uses does, understanding what the constraints would be on the arguments and output values of each function.
2. Decide on the basic data structure to store the cells of the spreadsheet. This ended up being a simple 2D array with the same dimensions as the spreadsheet user interface.
3. Decide what each cell needs to store for itself so that a user could navigate between cells and modify or clear the (textual) value (this is to achieve FR 1). For this I needed to store the strings for the editable input text and the display text of a cell.
4. Then, I expanded the data model for each cell, to store the type of input the cell contains, distinguishing between formula inputs and non-formula inputs, also to distinguish between the data type stored in the cell, whether it was a number or a string or an error in the case of an invalid formula. This was the first step in achieving FRs 2 and 3.
5. Then I wrote a function to compute the value of a cell, based on its text input. To get this done I had to write code to:
    a. Parse a number from a string that handles possible invalid inputs.
    b. Update the cell display text with a decimal number containing 1 digit after the decimal point.
    c. Parse and compute the value of a formula that handles errors in the case of an invalid formula. For the formula, I decided to support not just addition, but subtraction as an extra feature. I also decided to display different error messages if the formula was invalid due to a syntax error or due to an invalid cell reference.

   This completed FRs 2 and 3.
6. To achieve FR 4, I determined that both the cell data structure and the function to compute cells had to be updated significantly.
    a. I first defined some terminology used to relate cells ***A***, ***B***, and ***C***.
        i. ***B*** is ***child*** of ***A*** if: ***B*** is a cell containing a formula referencing ***A***
        ii. ***C*** is a ***descendant*** of ***A*** if: ***C*** is a cell containing a formula that references: ***A*** or any ***descendant*** of ***A***. Note that a ***child*** of ***A*** is also a ***descendant*** of ***A***.
        iii. ***A*** is a ***parent*** of ***B*** if: ***B*** is a ***child*** of ***A***.

b. Given a cell ***A***, to know which formula cells must be updated when ***A*** is updated, we need a method of determining all the ***descendants*** of ***A***. It is the ***descendants*** of ***A*** that must be updated. The simplest way to determine this is for each cell to maintain a list of their ***children*** and to recursively visit the cell's ***children's children*** to determine all the ***descendants***.

c. Another situation I considered was this: Given cells ***A*** and ***B***, if ***B*** was originally a ***child*** of ***A***, and upon an update ***B*** is no longer a child of ***A***, then ***B*** should be removed from the ***children*** of ***A***. The simplest way to remove a cell from each of its ***parents' children*** list is for each cell to maintain a list of its ***parents*** and, upon an update, to iterate over that list removing itself from each ***parent's children*** list.

d. So, I updated the data structure stored in a cell to contain a list of ***parents*** and ***children***.

e. And each time a cell receives an input from the user, I decided to clear its list of parents.

f. Then to update **parent** and **child** relationships, I decided to do this when computing a formula cell's value, since in that calculation you iterate over all the cells referenced in the formula anyway. To avoid adding duplicate **parent-child** relationships I decided to maintain a flag on each cell which marked whether it was added as a parent during the current formula computation and cleared the flag after the formula was computed.

    i. The above solution to avoiding duplication of ***parent-child*** relationships makes the compute cell function unparallelizable since each cell is being flagged globally. An alternate solution that would maintain the possibility of running the function in parallel could be to perform a direct search on the list of ***parents*** to see if a ***parent*** was already added.

g. Then, after a cell ***A*** has its value updated, we need a way of updating each of its **descendants** in the correct order. To update a ***descendant B*** correctly, you must ensure that each of the ***parents*** of ***B*** have been updated beforehand. In other words, the **farthest descendants** of ***A*** must be updated last, the ***second farthest descendants*** of ***A*** can be updated after the ***farthest descendants***, and so on until the ***children*** of ***A*** and ***A*** itself can be updated.

h. The valid ordering can be computed by creating a stack to store the update order and by doing a post-order depth first search on the graph representing ***A*** and its ***descendants***. A cell is added to the update order stack only once each of its ***children*** have also already been added.

i. To figure out this algorithm, I got inspiration from a YouTube video about the Topological Sort Algorithm[1]. This algorithm for a valid update ordering is in fact a Topological Sort of the dependency graph stemming from cell ***A***.

j. This algorithm also lends itself to finding circular dependencies. If during the algorithm you begin to depth first search on a cell and you end up encountering that same cell during the depth first search, then you can conclude that cell must be causing a circular dependency. This means that you should skip depth first searching down circular dependent cells and a cell is added to the update order stack when either all of its children have been either: (a) added to the update order or (b) are causing a circular dependency.

k. To update the cells once a valid order has been determined, you pop each cell from the update order stack and if it was not flagged as a circular dependency, then you compute its value.

2 Implementation

The code I wrote directly reflects the design I outlined in the previous section. My code is heavily commented enough to explain all the implementation details. In this section I will go over the time complexity of the functions that I implemented.

2.1 DATA_TYPE

This enum is used in conjunction with cell_t and it stores the type of data stored in a cell. Below is the struct in C:

```
// Data type stored in a cell.
typedef enum _DATA_TYPE
{
    NUMBER,
    STRING,
    ERROR,
} DATA_TYPE;
```

2.2 cell_t

This is the struct that specifies the data that each cell stores so that each function can be carried out optimally. Below is the struct in C:

```
struct _cell_t
{
    char *text_input;
    char text_display[CELL_DISPLAY_WIDTH + 1];
    ROW r;
    COL c;
    DATA_TYPE data_type;
    number num;
    cell_pos_t parents[NUM_ROWS * NUM_COLS];
    size_t num_parents;
    cell_pos_t children[NUM_ROWS * NUM_COLS];
    size_t num_children;
    bool already_added;
    bool in_process;
    bool processed;
    size_t children_processed;
    bool circular_dependency;
};
```

The constants NUM_ROWS, NUM_COLS, and CELL_DISPLAY_WIDTH were given for the purposes of this assignment.

In the code, I decided to make a typedef for this cell struct for improved readability. See model.h for detailed comments on each member of the struct.

The ***parents*** and ***children*** are arrays of type cell_pos_t, for example, when I store a **parent** I also store the position of the current cell in that parent's **children** array. This allows for constant time lookups of parents and children. This facilitates constant time deletion of a cell from a parent's children array while maintaining parents and children stored in a contiguous block of memory without the need for shifting. More on the reasons for this is explained in update_cell.

I decided not to make the ***parents*** and ***children*** arrays statically allocated as opposed to dynamically allocated since the size of the spreadsheet is fixed for this assignment. This meant that each cell could theoretically only have at most NUM_ROWS * NUM_COLS ***parents*** and ***children***. If I had to make those arrays dynamically allocated, I would have done so using a similar implementation to the stack_t.

2.3 cell_pos_t

This struct is simply to store a pointer to a cell and a position. It is used as the elements of the ***parents*** and ***children*** arrays in cell_t.

2.4 is_operator

I will just paste the code as it is short and self-explanatory:

```
/**
 * Checks if the character is an operator. Currently only supporting '+' and '-'.
 *
 * @param op The character to check.
 *
 * @return true if the character is an operator, otherwise false.
 */
bool is_operator(char op)
{
    switch (op)
    {
    case '+':
    case '-':
        return true;
    default:
        return false;
    }
}
```

2.5 parse_number

This function parses a number given a text input. An optional argument can be passed in which is the last character after parsing the number if the text input was valid. This function returns false in the case of invalid input, infinity values, or nan values.

The time complexity of this function is O(n) where n is the number of characters parsed when calling strtof.

2.6 cell_update_display_num

This function is used to update the cell_t::text_display of a cell based on the value of cell_t::num. This function was made so that if the *number* type is expanded to not just be an alias for a float, but instead of a struct for perhaps an arbitrary precision number it easy to expand the types of number that can be displayed in a cell or even optionally specify the level of precision desired to be shown instead of just a fixed 1 decimal place. See model.c for the function implementation.

The time complexity of this function is $O(CELL\_DISPLAY\_WIDTH)$ since snprintf writes at most CELL_DISPLAY_WIDTH characters from the format string to the text_display and then pads the rest of text_display with null characters.

2.7 update_cell

This is the function that recomputes the values in a cell based on its current text_input.

If the new_input is false, this function takes O(n) time where n is the number of characters in the input. There is additional runtime for resetting the display text, however that is O(1) since CELL_DISPLAY_WIDTH is taken to be a constant.

If the cell contains new input, then all its parents are removed which takes an additional O(m) time where m is the number of parents. So then the time complexity becomes O(m + n).

The removal of parents done here in O(m) time is accomplished by the use of storing both the cell and position in the ***parents*** and ***children*** arrays which enables constant time lookups up the current cell in its ***parents children*** arrays.

If the text_input does not start with an ‘=’ then the function ends early.

If the text_input did start with with an ‘=’ then the last for-loop runs, and the formula is parsed. As each valid cell reference is identified the parent child relationship is added.

Invalid numbers, characters or cell references cause the loop to exit early and those errors have the highest priority. If the referenced cell is not of type number, then the error is set to say that the cell is referencing not a number (ERR:REFNAN). In this case the formula calculation continues so that the remaining parents can be calculated and perhaps those parents would lead to a circular dependency which I have decided is a more important error to display.

2.8 table

This is the 2D static array that stores all the cells in the spreadsheet. Initialization is below:

```
// Stores all the cells in the spreadsheet.
static cell_t table[NUM_ROWS][NUM_COLS];
```

2.9 model_init

Time complexity is $O(NUM\_ROWS * NUM\_COLS)$ since there are two nested loops, the outer loop iterates from 0 to $NUM\_ROWS - 1$ and the inner loop iterates from 0 to $NUM\_COLS - 1$.

In this function I initialize each cell in the spreadsheet to an empty state. I decided that an empty cell had a data_type STRING and the input and display texts were empty strings.

2.10 get_textual_value

This function was implemented as a single return statement:

```
char *get_textual_value(ROW row, COL col)
{
    return strdup(table[row][col].text_input);
}
```

See the doc comment in model.h for more information.

2.11 set_cell_value

This is a function which is called by the interface. This function updates the cell with the new input by calling update_cell. It also updates all of the cell’s ***descendants*** using an iterative version of the algorithm described towards the end of the Design Proposal.

Let’s call the graph representing the cell at (row, col) and it’s ***descendants*** be called ***G***.

The worst-case time complexity of this function is O(n+e + n*m) where n is the number of nodes in the graph ***G*** and e is the number of **parent-child** relationships (edges) in the graph ***G***. M is the worst-case length of the input of to each cell in the spreadsheet.

Considering, the pair of nested loops starting at line 314, the outer for loop iterates once for each node in the graph, and once for each time a node is revisited. There are n nodes in the graph and each node gets revisited once for each of its children. The inner loop iterates over each child of a node only once during the entire process. So, the time complexity of the nested loops running is O(n+e).

Considering the last while loop where all the cells are updated. That loop iterates once for each node to be updated, and in the case where each node receives an update each iteration takes O(m). So, the time complexity of running the final while loop is n*O(m) = O(n*m).

Adding the two time complexities together we get O(n+e) + O(n*m) = O(n+e + n*m) = O(n*(m + 1) + e) = O(n*m + e).

In this function I am also using a stack that stores pointers to cell_t. This is the same stack that I programmed for lab 3. I just modified the data type to be stored on the stack from char to cell_t*.

2.12 stack_t

This is the same stack that I implemented as the array-stack in lab 3. It is just modified to store elements of type cell_t* instead of char.

2.13 How did I ensure that dynamically allocated memory was freed after use?

To ensure that there was no unnecessary build up of leaked memory during the running of the program I employed the following techniques:

- Each time I initialized a stack by calling stack_new, I made sure to call stack_free in the same function call.
- I the only time I allocated memory using malloc was for the text_input strings in model_init, this is so that each text_input always contains an allocation to a string
- Each time set_cell_value is called and the char *text is passed in, I always free the existing text_input for the cell and replace it with the text string that was passed as an argument to the function.
- The only memory that is not deallocated by the time of program end is the text_input strings for each cell in the table. There is exactly one text_input string allocated for each cell at any given time and it is always freed and immediately replaced once every time the function set_cell_value is called.
- All other memory that I am using is statically allocated so there is no issue with having to free that memory.

3 Testing

Please see the file tests.c for all the comprehensive tests that were run. Also, note that the string testing may result in a failure if the randomly generated string happens to start with an equals sign or it contains an number either in standard notation or scientific notation.

3.1 Testing for FR1

Please see the tests.c file from lines 15-67 for the tests that were run for FR1. The following tests also test this.

3.2 Testing for FR2

3.2.1 Testing for FR2.a and FR2.b

Please see the tests.c file from lines 69-89 for the testing of FR2.a and FR2.b. On those lines I tested inputs consisting of an equals sign, followed by a single arbitrary character value to ensure that a string starting with an equals sign was always interpreted as a formula no matter what character followed it.

Also, see lines 24, 25, 28, 34, 35, 36, for some more normal testing of formula parsing.

3.2.2 Testing for FR2.c

Please see the tests.c file from lines 90-104 for the specific test done for FR2.c. Here I tested parsing random floats from zero up to 500 thousand and simulated the user inputting 3 decimal places of accuracy. I then check that the display text in the cell contained the same number but with only 1 digit after the decimal place.

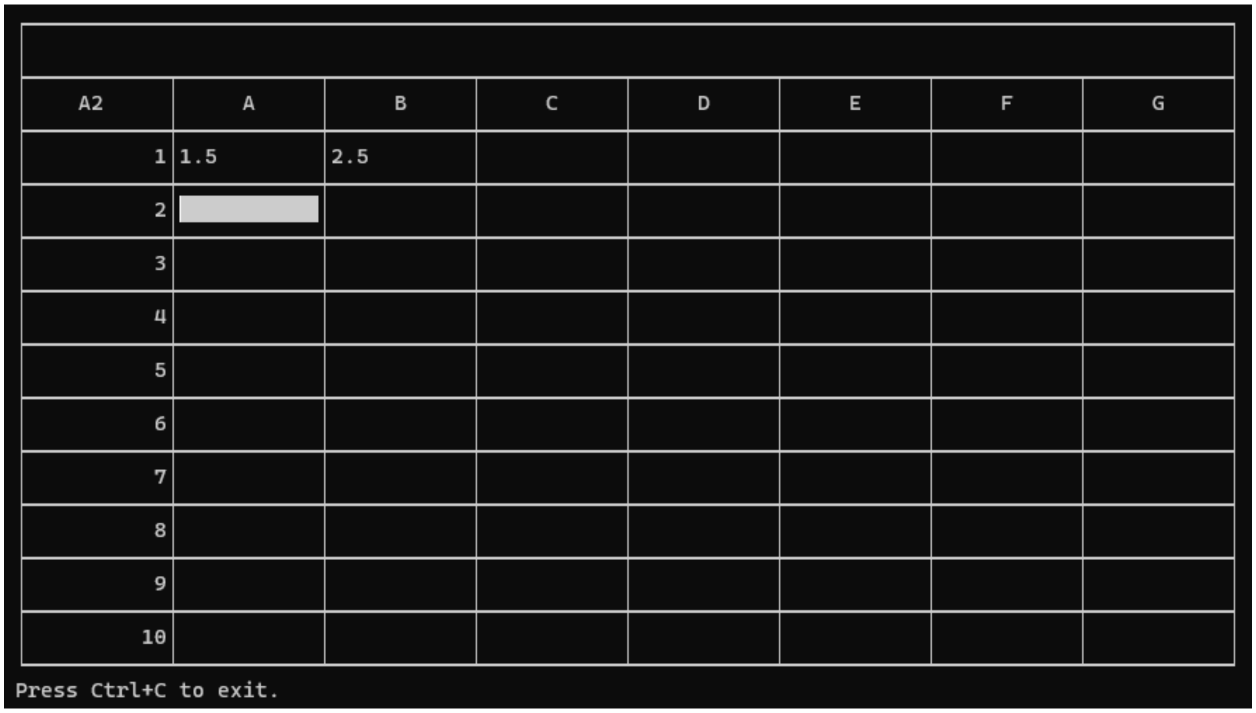
3.2.3 Testing for FR2.d

Please see lines 105-124 in tests.c for the code for the test run here. Here I simulated the user inputting 100 strings with lengths ranging from 0-1000 containing random (non-whitespace and non-control) characters. This test does fail sometimes since the random string may start with an equals sign (=) or be just a pure number. This is however a rare case and if the test does fail, the output in the console it to be true.

3.3 Testing for FR3

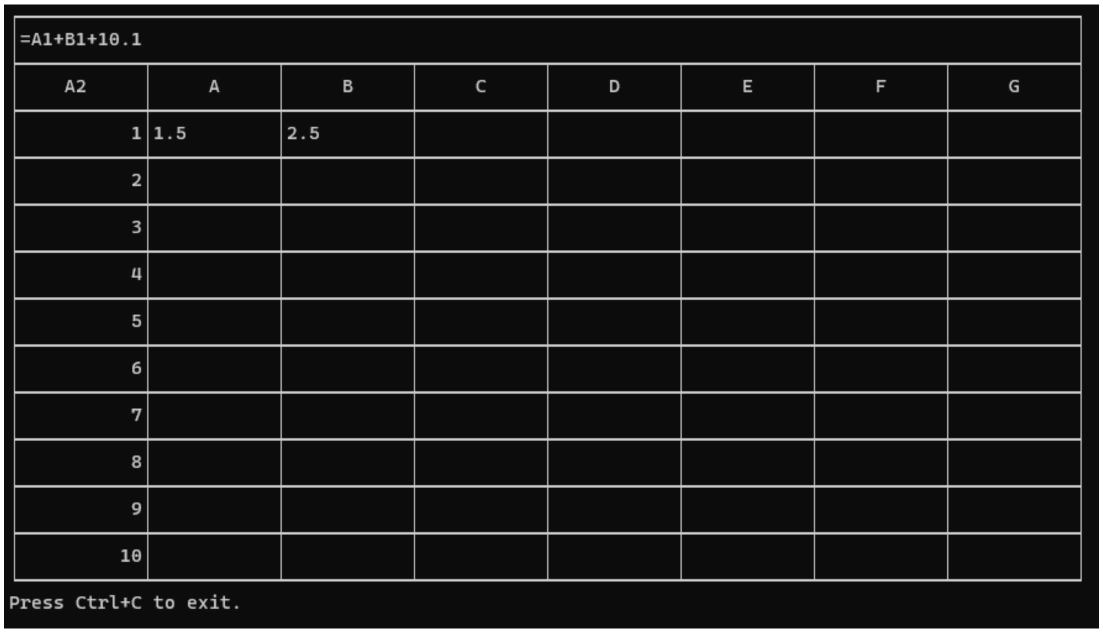
FR3 was also tested in the tests.c file, however I will include some screenshots here for demonstration:

Starting with cells A1 and B1 containing 1.5 and 2.5 respectively.



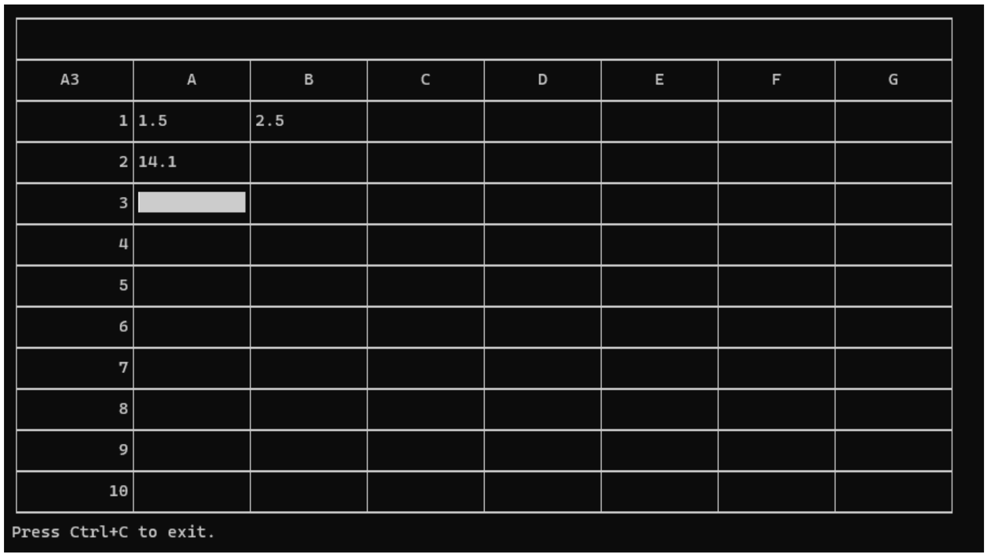
```
 A2   | A    | B    | C | D | E | F | G
    1 | 1.5  | 2.5  |   |   |   |   |
    2 |      |      |   |   |   |   |
    3 |      |      |   |   |   |   |
    4 |      |      |   |   |   |   |
    5 |      |      |   |   |   |   |
    6 |      |      |   |   |   |   |
    7 |      |      |   |   |   |   |
    8 |      |      |   |   |   |   |
    9 |      |      |   |   |   |   |
   10 |      |      |   |   |   |   |
Press Ctrl+C to exit.
```

Now typing a formula into cell A2, the expected display text should be 14.1.



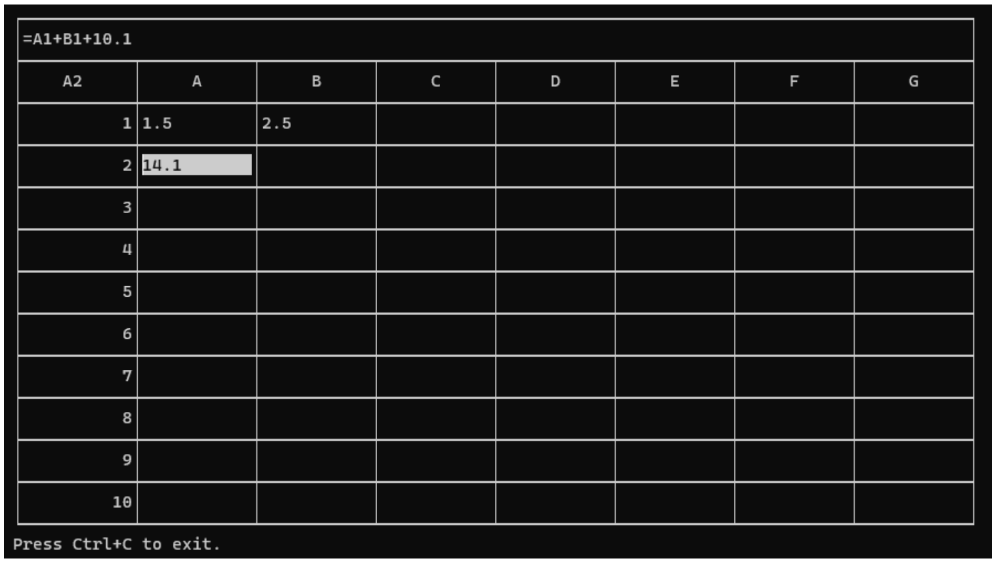
```
=A1+B1+10.1
   A2  |  A     |  B     |  C  |  D  |  E  |  F  |  G
     1 | 1.5    | 2.5    |     |     |     |     |
     2 |        |        |     |     |     |     |
     3 |        |        |     |     |     |     |
     4 |        |        |     |     |     |     |
     5 |        |        |     |     |     |     |
     6 |        |        |     |     |     |     |
     7 |        |        |     |     |     |     |
     8 |        |        |     |     |     |     |
     9 |        |        |     |     |     |     |
    10 |        |        |     |     |     |     |
Press Ctrl+C to exit.
```

After pressing enter we see that it is indeed 14.1.



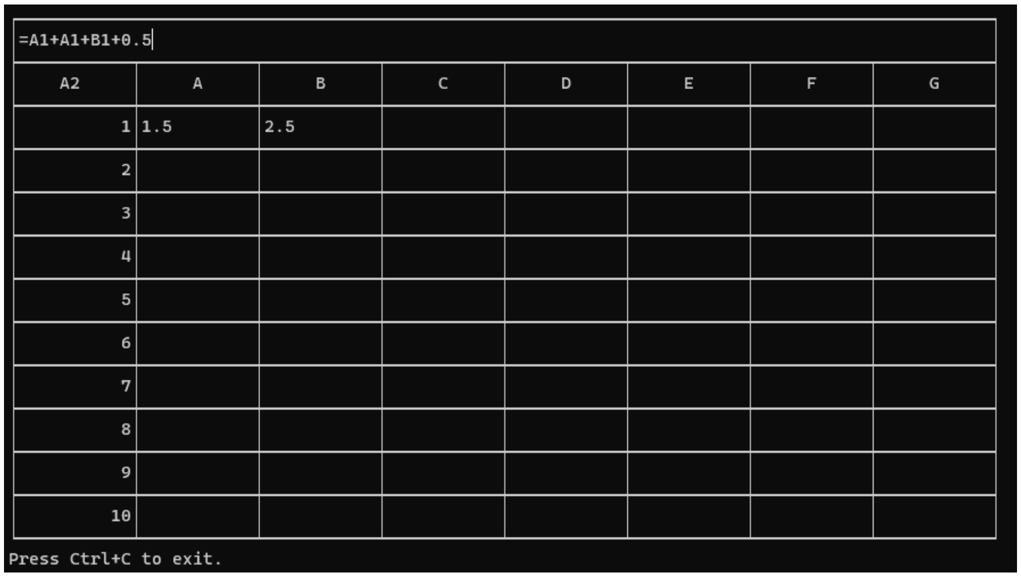
```
   A3  |  A     |  B     |  C  |  D  |  E  |  F  |  G
     1 | 1.5    | 2.5    |     |     |     |     |
     2 | 14.1   |        |     |     |     |     |
     3 |        |        |     |     |     |     |
     4 |        |        |     |     |     |     |
     5 |        |        |     |     |     |     |
     6 |        |        |     |     |     |     |
     7 |        |        |     |     |     |     |
     8 |        |        |     |     |     |     |
     9 |        |        |     |     |     |     |
    10 |        |        |     |     |     |     |
Press Ctrl+C to exit.
```

Upon navigating back to cell A2 we see that the edit text is the same as was inputted before:



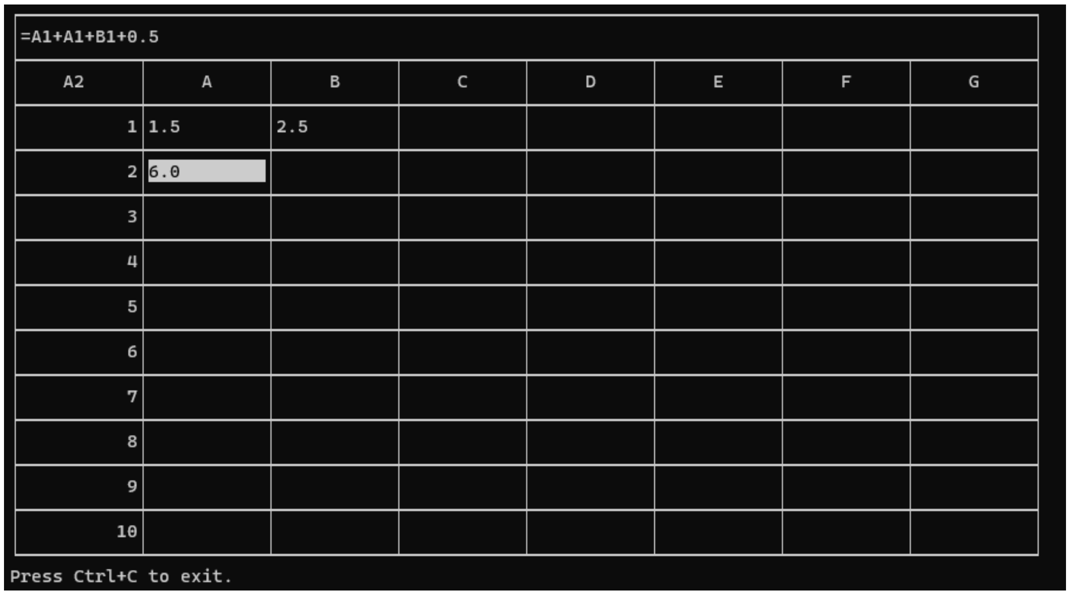
```
=A1+B1+10.1
A2    | A    | B    | C | D | E | F | G
    1 | 1.5  | 2.5  |   |   |   |   |
    2 | 14.1 |      |   |   |   |   |
    3 |      |      |   |   |   |   |
    4 |      |      |   |   |   |   |
    5 |      |      |   |   |   |   |
    6 |      |      |   |   |   |   |
    7 |      |      |   |   |   |   |
    8 |      |      |   |   |   |   |
    9 |      |      |   |   |   |   |
   10 |      |      |   |   |   |   |
Press Ctrl+C to exit.
```

Then we can edit that text and a different value should appear in the display of cell A2.



```
=A1+A1+B1+0.5
A2    | A    | B    | C | D | E | F | G
    1 | 1.5  | 2.5  |   |   |   |   |
    2 |      |      |   |   |   |   |
    3 |      |      |   |   |   |   |
    4 |      |      |   |   |   |   |
    5 |      |      |   |   |   |   |
    6 |      |      |   |   |   |   |
    7 |      |      |   |   |   |   |
    8 |      |      |   |   |   |   |
    9 |      |      |   |   |   |   |
   10 |      |      |   |   |   |   |
Press Ctrl+C to exit.
```

Upon pressing enter, the value displayed in cell A2 should be 6.0 (1.5+1.5+2.5+0.5=6.0).



```
=A1+A1+B1+0.5
  A2  |  A  |  B  |  C  |  D  |  E  |  F  |  G
    1 | 1.5 | 2.5 |     |     |     |     |
    2 | 6.0 |     |     |     |     |     |
    3 |     |     |     |     |     |     |
    4 |     |     |     |     |     |     |
    5 |     |     |     |     |     |     |
    6 |     |     |     |     |     |     |
    7 |     |     |     |     |     |     |
    8 |     |     |     |     |     |     |
    9 |     |     |     |     |     |     |
   10 |     |     |     |     |     |     |
Press Ctrl+C to exit.
```

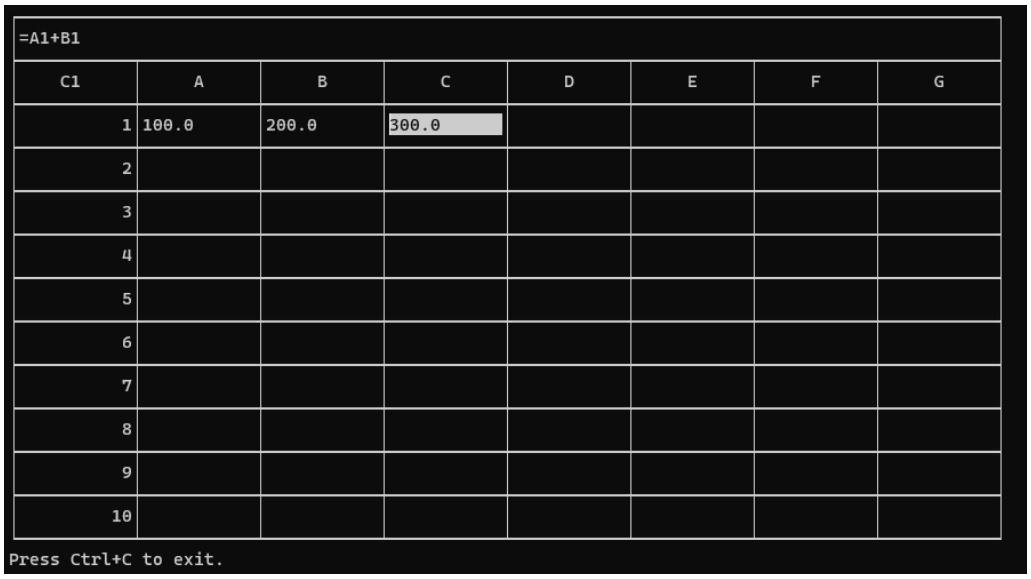
Which is indeed the case. This completes testing for FR3.

3.4 Testing for FR4

Testing for FR4 requires showing that when a value of a cell changes, the displayed contents of all formula cells which directly or indirectly depend on it are updated. This is tested comprehensively in tests.c on lines 30-34 and on lines 129-264. Below are some screenshots of this working in some simple scenarios:

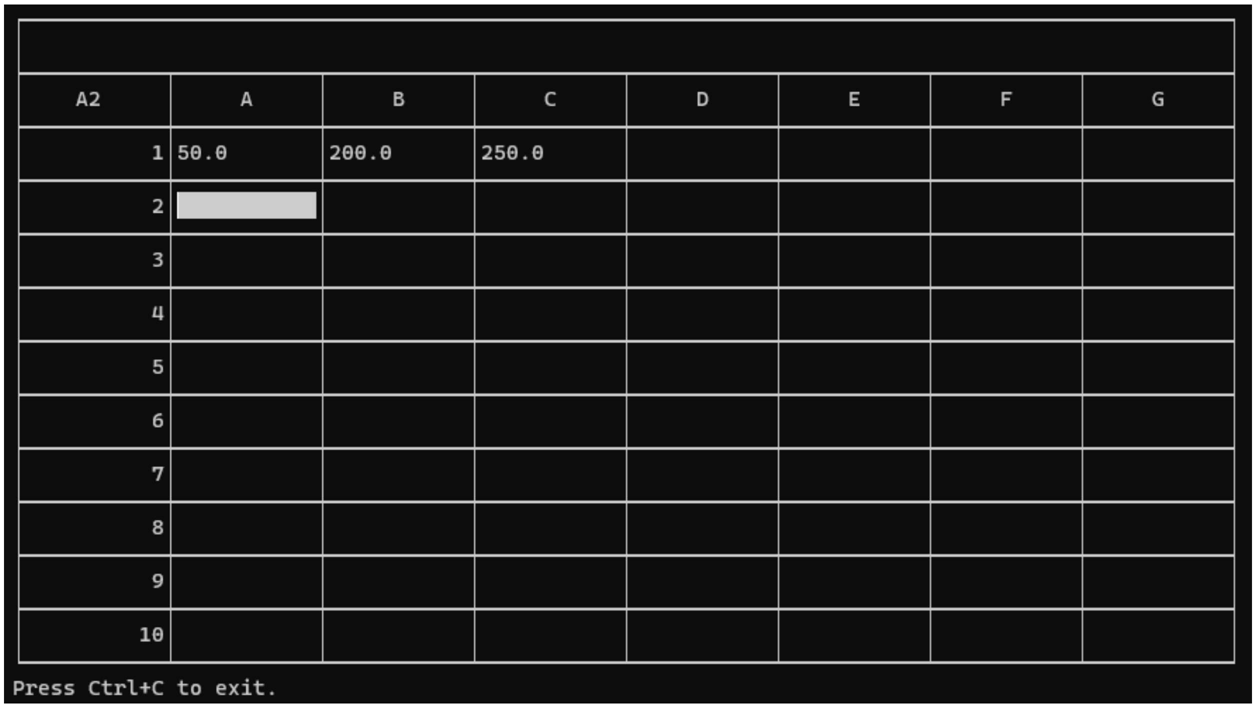
The first test that I will show involves setting cells A1=100.0, B1=200.0 and C1=A1+B1, then updating A1 by setting A1=50.0. This will result in first C1 displaying 300.0 before updating A1, to displaying 250.0 after updating A1. The images follow below:

Before updating A1:



```
=A1+B1
  C1  |   A   |   B   |   C   |  D  |  E  |  F  |  G
    1 | 100.0 | 200.0 | 300.0 |     |     |     |
    2 |       |       |       |     |     |     |
    3 |       |       |       |     |     |     |
    4 |       |       |       |     |     |     |
    5 |       |       |       |     |     |     |
    6 |       |       |       |     |     |     |
    7 |       |       |       |     |     |     |
    8 |       |       |       |     |     |     |
    9 |       |       |       |     |     |     |
   10 |       |       |       |     |     |     |
Press Ctrl+C to exit.
```

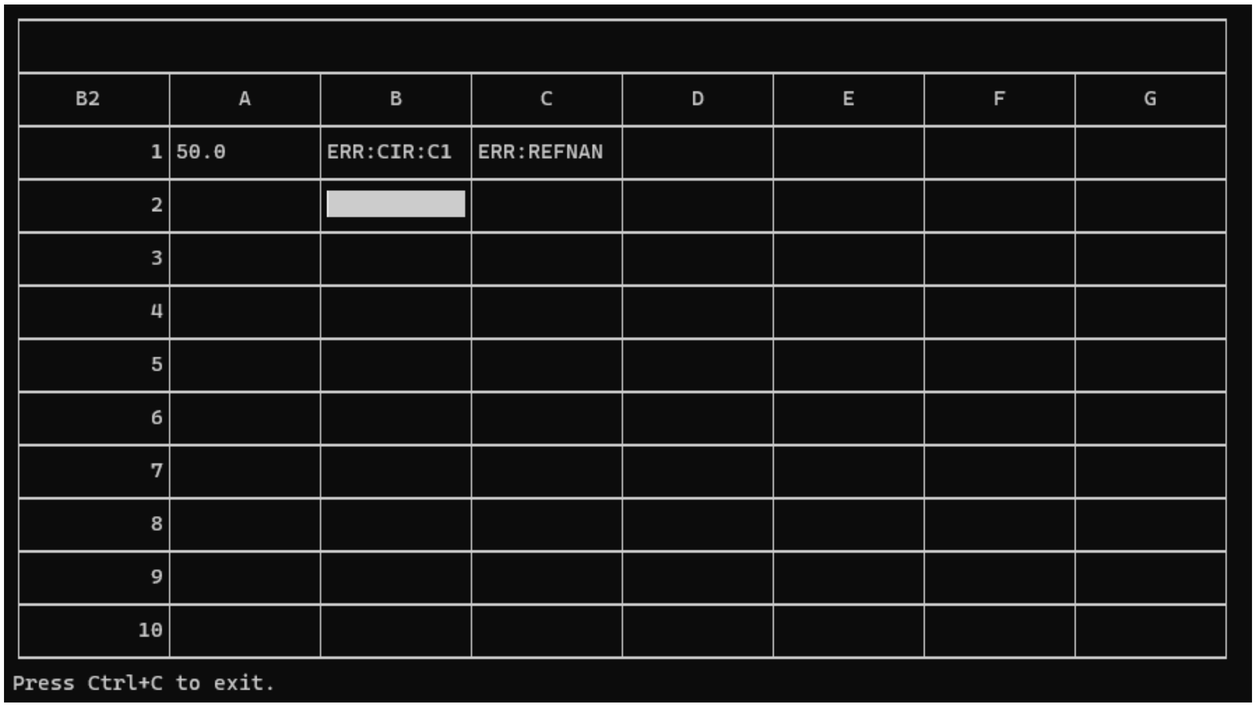
After updating A1:



```
A2          A        B        C        D   E   F   G
 1    50.0     200.0    250.0
 2
 3
 4
 5
 6
 7
 8
 9
10
Press Ctrl+C to exit.
```

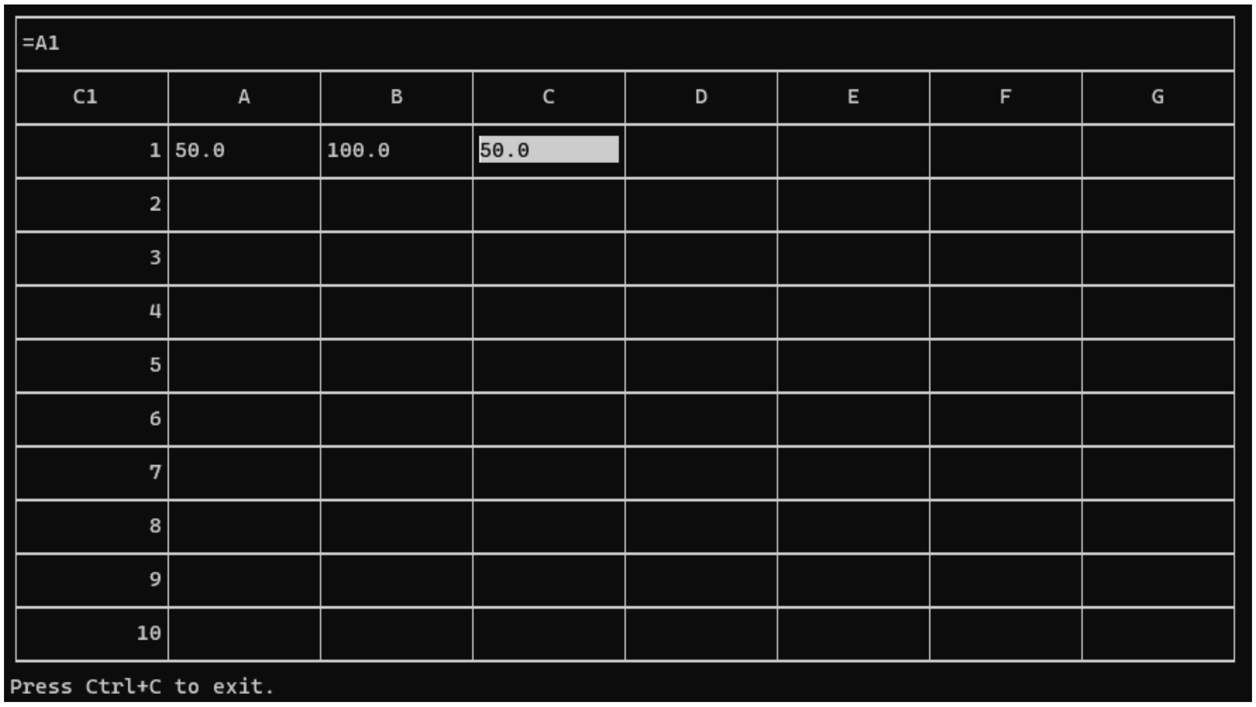
This is the expected behaviour.

To see what happens upon a circular dependency, I will then set B1=C1+A1, this will result in B1 containing ERR:CIR:C1 since it is circularly referring to cell C1, C1 containing ERR:REFNAN since B1 contains an error instead of a number, and finally A1 will remain A1=50.0, unchanged. The image below shows the expected behaviour.



```
    B2   |    A     |     B      |     C      |  D  |  E  |  F  |  G
       1 | 50.0     | ERR:CIR:C1 | ERR:REFNAN |     |     |     |
       2 |          |            |            |     |     |     |
       3 |          |            |            |     |     |     |
       4 |          |            |            |     |     |     |
       5 |          |            |            |     |     |     |
       6 |          |            |            |     |     |     |
       7 |          |            |            |     |     |     |
       8 |          |            |            |     |     |     |
       9 |          |            |            |     |     |     |
      10 |          |            |            |     |     |     |
Press Ctrl+C to exit.
```

This error could be resolved in a few different ways, one of which involves removing the reference of B1 in the formula of C1 so that C1=A1. This will result in C1=50.0, B1=100.0, and A1 remains unchanged at A1=50.0. The image below shows this behaviour:



```
=A1
    C1   |    A     |     B      |     C      |  D  |  E  |  F  |  G
       1 | 50.0     | 100.0      | 50.0       |     |     |     |
       2 |          |            |            |     |     |     |
       3 |          |            |            |     |     |     |
       4 |          |            |            |     |     |     |
       5 |          |            |            |     |     |     |
       6 |          |            |            |     |     |     |
       7 |          |            |            |     |     |     |
       8 |          |            |            |     |     |     |
       9 |          |            |            |     |     |     |
      10 |          |            |            |     |     |     |
Press Ctrl+C to exit.
```

This completes the testing for FR4.

4 Extra Features

4.1 Extra Feature Proposals

First, I thought to expand the possible numeric inputs to cells to include scientific notation and support additional operators.

- scientific notation is supported for inputting numerical values by using an included standard library function strtof.
- The subtraction operator could be supported in formula evaluation, on top of addition.
- Support for even more operators of differing precedence, operations such as multiplication, division, exponentiation, custom functions, and support for grouping with parentheses could be added if the infix formula expressions were converted to postfix considering operator precedence and then evaluated the postfix expressions.

I also chose to customize the error messages in the cell display text to better inform the user of what was causing the error.

- Specified error messages depending on the type of error and listing the circular dependent reference in case of a circular dependency.

4.2 Extra Feature Implementation and Testing

4.2.1 Specified Error Messages

I implemented this feature by setting the specific error message in a cell's display text as soon as the error was recognized. Whether the error was identified during the formula evaluation in update_cell or if it was a circular dependency that was identified while determining the update order in set_cell_value. Doing this instead of just always setting the display text to ERROR when the data_type of the cell was set to ERROR gives a better experience to the user since they better understand their errors and can correct them with ease.

Here are the following different errors that I decided to distinguish between:

- ERR:CIR:[cell reference]
    - [cell reference] would be a reference to a valid cell such as A1
    - This error tells the user that referring to [cell reference] in the cell formula is causing a circular dependency.
    - The user can fix this error by either changing the formula in the cell marked with this error, or change the formula in the labeled [cell reference] to not contain the erroring cell in it's formula
- ERR:REF
    - This tells the user that the cell with this display text contains a reference to a cell that is not present in the spreadsheet
- ERR:REFNAN
    - This tells the user that the cell containing this error message is a formula that is referring to a cell whose data_type is not NUMBER.
- ERR:SYNTAX
    - This tells the user that the formula they entered in this cell contains a syntax error.

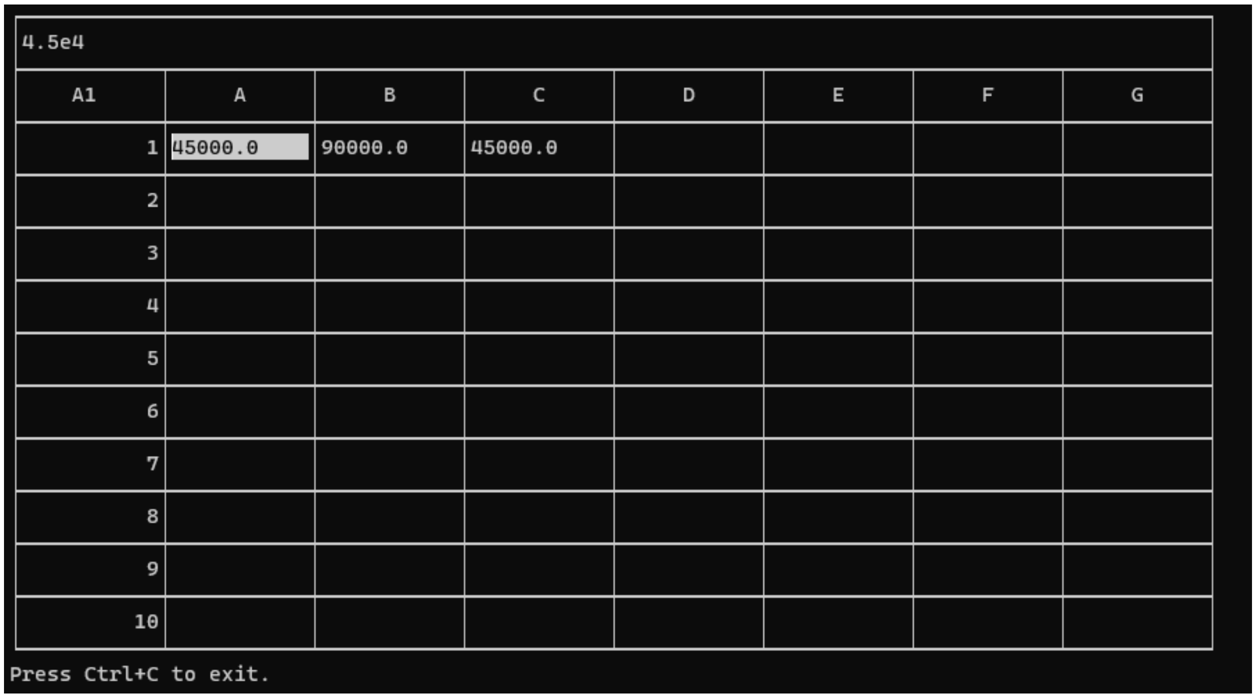
- A syntax error means the cell contains either misplaced operators, invalid operators, or any other unrecognizable strings of text.

This extra feature was tested in the previous tests.

4.2.2 Scientific notation

I implemented this feature by calling the standard library function strtof when parsing numbers.

Testing this feature is simple, picking up where we left off in the last test and by setting cell A1=4.5e4 (or 45000) we see that the result is as expected:



```
4.5e4
A1    A        B        C        D    E    F    G
 1    45000.0  90000.0  45000.0
 2
 3
 4
 5
 6
 7
 8
 9
10
Press Ctrl+C to exit.
```

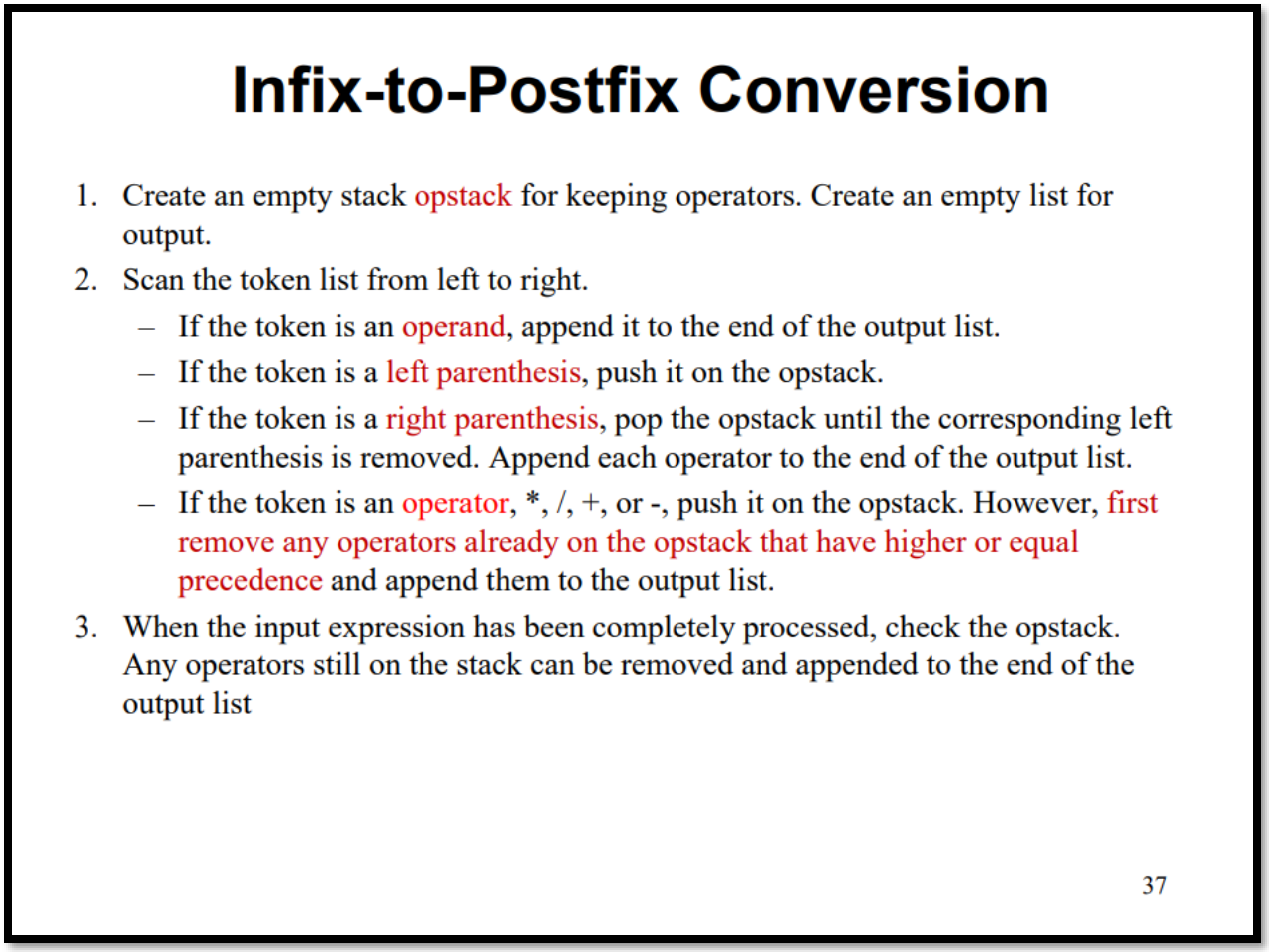
4.2.3 Subtraction and the is_operator function

I implemented this feature by creating the is_operator and adding a switch case depending on if the operator was addition or subtraction in the update_cell function. Testing for this was done in tests.c on lines 35-38.

4.2.4 Additional operators and custom functions

I did not end up implementing this feature, however, I could have implemented additional operators by first creating an array of strings containing the different operators and their precedence/priorities. Then I would convert the infix expressions to postfix using an algorithm discussed in class which employed the use of a stack.

The algorithm from Prof. Jianbing's lecture slides on Stack and Queues was as follows:



Infix-to-Postfix Conversion

1. Create an empty stack opstack for keeping operators. Create an empty list for output.
2. Scan the token list from left to right.
   - If the token is an operand, append it to the end of the output list.
   - If the token is a left parenthesis, push it on the opstack.
   - If the token is a right parenthesis, pop the opstack until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
   - If the token is an operator, *, /, +, or -, push it on the opstack. However, first remove any operators already on the opstack that have higher or equal precedence and append them to the output list.
3. When the input expression has been completely processed, check the opstack. Any operators still on the stack can be removed and appended to the end of the output list

37

References

[1] *Topological Sort Algorithm | Graph Theory*, (Oct. 19, 2017). Accessed: Dec. 03, 2023. [Online Video]. Available: https://www.youtube.com/watch?v=eL-KzMXSXXI