COP290 C Lab

Arjun Sammi: 2023CS50163; Popat Nihal Alkesh: 2023CS10058; Viraaj Narolia: 2023CS10552

March 2025

1 Frontend

1.1 Input Parser

The input parser takes as input the input command, the number of rows, the number of columns, and a pointer to a `struct parsedInput`:

```
typedef enum ops { FIX, SLEEP, ADD, SUB, MUL, DIV, MIN, MAX, STDEV, SUM, AVG, DISABLE_OUT,
    ENABLE_OUT, SCROLL } ops;
typedef enum inputType {Invalid, Movement, Assignment, Display} inputType;
struct parsedInput
{
    inputType inpType;
    ops operation;

    bool val1Type; // 1 if address, 0 if int
    int val1Col;
    int val1Row;
    int val1Int; // May carry error code (if input is invalid)

    bool val2Type;
    int val2Col;
    int val2Row;
    int val2Int; // May carry error position (if input is invalid)

    int targetCol;
    int targetRow;
};

void parse_input(char* inp, struct parsedInput* parsed_out, int R, int C)
```

The function will update the struct at the pointer given to the correct value. `targetcol` & `targetRow` contain the target cell details.

- For FIX type (direct value), `val1Int` contains the value.
- For arithmetic operations `val1` and `val2` contain the required information.
- For range operations, `val1` and `val2` contain the range endpoints
- For single letter commands, `val1Int` contains 1,2,3,4,5 for w,a,s,d,q respectively.
- For display commands, the command is in `operation`, and the target cell is stored if required.
- In case of an error during parsing, `val1Int` stores the code (0: Invalid Syntax, 1: Address out of range, 2: Range query is not in increasing order). `val2Int` stores the position (character index) at which error occured.

1.2 Display

```
#define windowWidth 10
#define windowHeight 10
void display_window(Cell** data, int currR, int currC, int R, int C);
```

The `display_window` command takes the `data`, `currC` & `currR`, and `C` & `R`, and then prints a window of the sheet such that `currC` and `currR` is the first cell displayed (top left). The logic for which cell should be at the top left is handled outside. The window height and width defined in the header file. If the window extends outside the sheet (near the right or bottom edge), then it shows as much as possible, though whether or not this would be allowed is handled outside the function by setting `currR` and `currC` appropriately.

2 Evaluation

2.1 Data structures used

1. AVL Tree (self balancing BST):
2. Hash Table
3. Linked List
4. Stack

2.2 Overview

To store the sheet, we have created two structs, '`Cell`' and '`Cell_func`'. In these structs, we have used **Bitfields** to optimize memory usage. We are storing the parents (dependencies) of the current cell in an AVL tree whose pointer is a member of the struct Cell (AVL tree was used to optimize removal of old dependencies when an operation is modified in O(logn) time complexity).

```
 struct Cell {
    int col_name: 8;
    int row_num: 8;
    int value;
    int valid: 16;
    Cell_func *func;
    AVL* children;
};
struct Cell_func {
    union {
        int value1;
        Cell* Cell1;
    };
    union {
        int value2;
        Cell* Cell2;
    };
    bool flag1, flag2;          // FOR BOTH FLAGS: 1 MEANS CELL, 0 MEANS VALUE/NONE
    ops op;
};

// MAIN FUNCTION THAT WOULD BE CALLED BY MAIN FUNCTION
int evaluate(Cell** data, Cell *cell, Cell_func* old_func, int R ,int C);

// FOR RECALCULATING VALUE OF CELL AND ITS DEPENDENTS
update_children(Cell** data, Cell* cell, int C);
```

```
ll_Node* topological_sort(Cell* current_cell);
int dfs(Cell* current_cell, HashTable* visited, HashTable* recStack, Stack *stack);

// ADDING NEW DEPENDENCIES TO AVL TREE OF CELL
void update_parent_avls(Cell** data, Cell *cell, int C);

// IT JUST TAKES IN A SINGLE CELL AND RECALCULATES ITS VALUE USING cell->func
void calculate(Cell** data, Cell* cell, int C);

// USED TO REMOVE DEPENDENCIES THAT WERE CAUSED BY OLD OPERATION old_func
void remove_old_dependencies(Cell** data, Cell_func* old_func, Cell* cell, int C);
```

2.3 Pathway

1. Clear old operation (if any)
2. Initialize dependencies due to new operation
3. Recalculate the cell and its dependents
4. Loop detection

2.3.1 Clear old operations

When there is an operation assignment, we must check if there was earlier some other operation assigned to the cell, that is, removing old dependencies from AVL.

2.3.2 Initialize dependencies due to new opeartion

Similar to previous function, this only includes inserting new dependencies to AVL of the current cell.

2.3.3 Recalculate the cell and its dependencies

We have used **Topological sort** algorithm to implement recalculations. The idea behind this was that topological sort, when applied on a directed graph, gives us a linked list that contains nodes in a particular order, such that there are no back-edges, i.e. there won't exist any cell whose dependencies come after itself in order. This is crucial as, when some cell is updated, we would like to recalculate its dependents such that we would only have to calculate them once and get correct values.

For topological sort, we need some data structure that can store all the cells visited in front edges, as well as possibility of some back edges (loop in the logic of operations, where we would reject the latest operation and revert the changes). For the same, we have used two hash tables that would store pointers to visited cells. This was done to ensure optimized insertion and deletion functions. We have also used stack data structure, that was used for DFS (Depth first search) algorithm that is used in topological sort.

Topological sort would return a linked list in the desired order. The order given by it would then be followed while recalculating values.

Note: There are three possible recalculation time errors, Division by zero error, Loop error, and negative input in sleep operation. In loop detection and negative input in sleep operation, the latest operation would be rejected, and in Division by zero error, the affected cells would store 'ERR'.

2.3.4 Loop Detection

As mentioned above, the 'calculate' function would use topological sort which would further use DFS. In this process, we are using two hash tables, of which one is used primarily for loop (circular dependency) detection. In that case, old function would simply be restored.

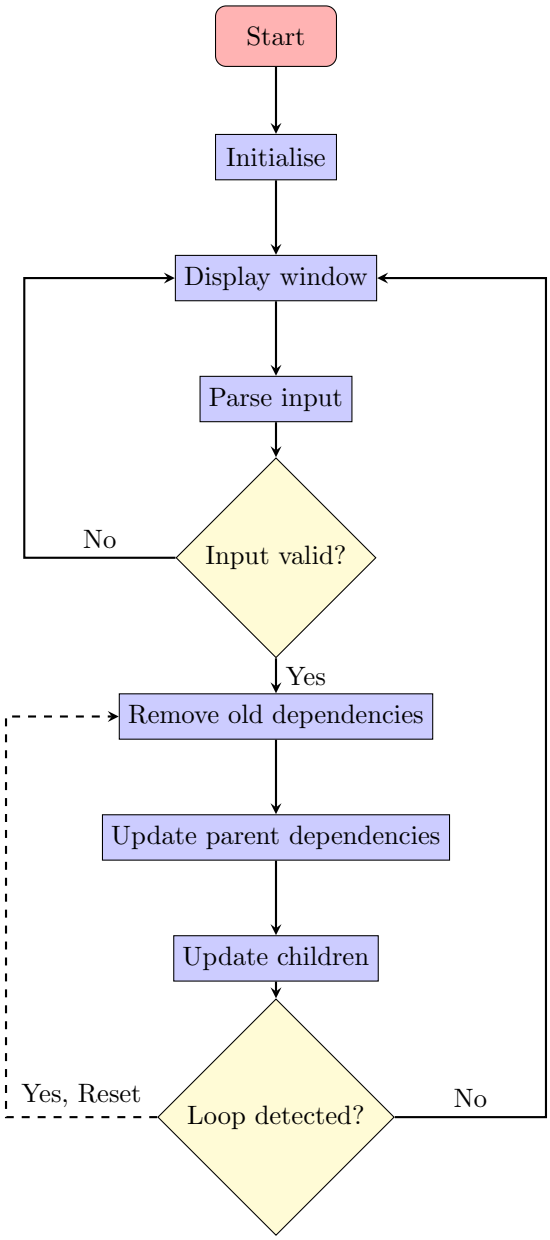

Figure 1: Program Control Flow

3 References

1. https://github.com/Pianissimo-3115/lab1_2023CS10058_2023CS10552_2023CS50163
2. https://drive.google.com/file/d/1scgtT82MF7pSDNsDceoqfxziVYxi8K4N/view?usp=sharing