Final Assignment – ELEC278

David Laeer
Queen’s University
Submitted on Dec 4th 2023

I, David Laeer, 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 the 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.

Executive Summary

The spreadsheet application produced in the context of the final assignment for the ELEC278 course fulfills or exceeds all given functional and non-functional requirements. It has been extensively checked for memory safety and undefined behavior.

Data structures and algorithms used to implement the application include structs and arrays for storing spreadsheet data, as well as binary trees, stacks, and queues to implement the parsing and evaluation capabilities for spreadsheet expressions. There are no known better alternatives in terms of space and time complexity for all algorithms implemented in this program.

Additionally, minor changes have been made to the user interface, such as adding support for spaces in text fields, and eliminating a source of crashes related to input string handling. The CMakeLists.txt file was also edited to include new files, 4 of which were added to the base template provided by the instruction team. Link options may have to be modified on windows systems, where it may not be necessary to specify linkage against the math library ("m" under target_link_libraries). Generative AI was not used during any part of the creation of this project.

Table of Contents

Design Proposal and General Approach

The general design philosophy used when approaching the problem of designing a CLI-based spreadsheet program reflects several goals and design philosophies, within the constraints and programming paradigms imposed by the C programming language. In no particular order, these include:

1. Memory safety: The program should handle ownership of memory in a safe, consistent manner, with references to allocated memory being either ‘owning’ or ‘non-owning’. This minimizes the risks and challenges associated with manual memory management in the C programming environment.
2. Exception handling and safety: Errors are to be handled in a consistent, safe manner, ensuring the program behaves in an expected manner from the end-user’s point of view. Erroneous or unexpected input should not cause errors, crashes, safety issues, or other undesirable behavior, and should be handled in a user friendly manner.
3. Undefined behavior: Such behavior is undesirable and is to be avoided entirely.
4. Efficiency: Algorithms and data structures used in the design of this program should leave no room for wasteful memory usage or computation.
5. User-friendly design: The program should be easy to use and user-agnostic, meaning that users who have never been exposed to the program’s user interface should face no difficulty when first using the software. The user interface should be easy to use and follow a minimalist design approach.
6. NFR (Non-Functional Requirement) and FR (Functional Requirement) compliance: The program should comply with all given FRs and NFRs, and should implement additional features when feasible given the existing time constraints.
7. Additional Features: The parsing algorithm used by this software has capabilities beyond those outlined in the list of FRs.

1. Memory Safety

Some of the most common issues encountered by developers using the C programming language stem from C’s manually managed memory model: Invalid pointers and memory leaks. Invalid pointers occur when a pointer holds a memory address which is not associated with an area of dynamic program memory which has been explicitly allocated (acquired) by the program, while memory leaks occur when access to an area in memory which has been explicitly allocated is ‘lost’, meaning when there are no pointers holding that memory block’s address. To address these two issues, data structures using dynamically-allocated memory in this program fall into two categories[1]: non-owning and owning references (pointers)[2]. In essence, ownership indicates where the responsibility of managing the lifetime of the underlying memory lies. Non-owning pointers or references are generally temporary in nature, such as when data is passed by reference to a function. While this function may or may not modify the underlying data (as indicated by the “constness” of the passed pointer), the onus is on the caller to manage the lifetime of the memory associated with the pointer. This is fundamentally different to a pointer which “owns” the underlying data. An owning pointer’s lifetime is classified into three parts: creation, use, and destruction. Upon entering execution scope (or upon program start in the case of globally scoped pointers), owning pointers must either be assigned to a valid area in memory (using malloc() or equivalent), or must be explicitly set to NULL. This ensures that NULL-checks can be implemented throughout the program and can be relied upon for checking the state of the pointer, where a pointer with a value of NULL explicitly is in a non-owning state. This prevents the first issue outlined above, where references to invalid memory blocks may be used, causing program crashes. During the lifetime (use) of owning pointers, they may be passed by value to functions accepting non-owning pointers: this demonstrates the use of safe non-owning pointers by helper functions. They may not however be assigned new values (memory addresses) without first going through a new lifetime cycle, i.e. without first being destructed. Destruction of an owning pointer involves freeing the memory “owned” by that pointer, then assigning said pointer to the NULL pointer macro. In short, an owning pointer may be in one of two states: “dead”, having a value of NULL and owning no memory, or “alive”, having a non-NULL value. Only “live” pointers may be passed by copy (not by reference) to functions accepting non-owning pointers. As an aside, function accepting pointers-to-pointers accept these as owning pointers, and are responsible for proper destruction/creation if they choose to modify the memory address stored in the pointer.

2. Exception Handling/Safety

Functions whose tasks are not guaranteed to succeed are required to implement mechanisms to handle failure to complete, which must include a way of notifying the caller of the failure of the required task. In other languages such as C++ or Java, exception handling is explicitly supported by the language standard, effectively providing two paths for function return mechanisms: the non-exceptional path (return

[1] This approach is inspired from the concepts of “smart” (owning) pointers and RAII (Resource Acquisition Is Initialization) used in the C++ programming language.

[2] The terms “pointer” and “reference” are used interchangeably throughout this report

by value/reference) and the exceptional path, consisting of raising an exception resulting in the unwinding of the call stack until the exception is handled. However, C does not permit this type of exception handling, and the programmer must therefore use careful design procedures when writing functions which may result in a failure to evaluate. The approach used in the design of this software involves modifying the function signature, wherein the function which is subject to failure returns a “bool” value, indicating success or failure. The function then takes an extra parameter in the form of a pointer, which is to be passed as the address of the variable where the result is to be stored upon success. As seen in section 1, this pointer parameter is a non-owning pointer whose underlying value may be modified by the called function, in order to store the result.

3. Undefined Behavior

The C programming language standard specifies that the machine readable output of the C compiler must not adhere to the source code input in a line-by-line fashion, as long as the program behaves as would be expected based on the source code AND there is no undefined behavior (UB) present in the program. If there is undefined behavior present in the source code, the program may not be expected to do anything – i.e. anything is allowed to happen. This is obviously highly undesirable and does not align with the goals outlined above. Examples of undefined behavior include accessing an array out of bounds, signed integer overflow, or calling “free()” on a pointer which is not associated with properly allocated memory. Such undefined behavior is therefore to be avoided at all costs, and several tools such as clang-tidy or IntelliSense can provide warnings if UB is thought to occur.

4. Efficiency

Given the limited nature of memory and compute cycles, the software developed in the context of this assignment must make every effort to use the most efficient algorithms available to solve the task at hand, and to limit the use of memory to what is required of the program and no more. This stems from the concept that a user’s time and resources are more valuable than those of the developer, since there may be 10,000 users for every developer. Thus, it makes sense for the developer to spend additional time improving his or her programs’ efficiency if it results even in a minor improvement for the end user. The following data structures are used in the implementation of the software:

- Binary Tree: a type of binary tree is used to implement the evaluation of equations in the spreadsheet. Each cell containing an equation holds a pointer to the root node of a binary tree representing the equation in that cell. This eliminates the need for repeated parsing, as well as simplifying the use of references to other cells.

- Stack: stacks are used in the implementation of the shunting yard algorithm, a simple and efficient way of converting equations from infix notation to postfix notation.
- Queue: queues are used in the implementation of the shunting yard algorithm, specifically to reverse the direction of the input stack using “push_back” and “pop_front” operations.
- Array: a 2-dimensional array is used to store cell data, since this data structure corresponds most closely to the “natural” way of storing the data.
- Struct: structs are used extensively throughout the program, both to represent nodes in the binary tree, and to store cell data. This aggregate data structure allows the program to follow a somewhat object-oriented design with constructor-destructor patterns, further enhancing memory safety as described in point 1 above.
- Enum: enums are used extensively within struct definitions to represent flags representing the struct’s current state. This allows for variable behavior at runtime based on the state, while representing state in a readable, simple manner.

5. User-friendly Design

While the code defining the user interface has already been provided by the ELEC278 teaching team, minor improvements have been made at the discretion of the student. Furthermore, user experience (UX) is influenced by the efficiency and responsiveness of the software, thus falling under item 4 listed above. This also encompasses certain program behavior, such as auto-capitalization of characters used in formulas referencing other cells.

6. NFR and FR Compliance

The software complies with all given Functional and Non-Functional Requirements. Additionally, it implements features beyond the FRs, such as additional arithmetic operators, which are explained in greater detail below, in the implementation section.

FR Compliance

- A user can navigate between cells and modify or clear the value of each: this FR is fulfilled. Cells can be navigated using a scroll wheel or up-down-left-right keys, and their values can be edited by typing into the edit window.

- When a user enters a new value, it is interpreted as either text, a number, or a formula: this FR is fulfilled. The software detects which type of input was entered (basic string, number, or formula), and acts accordingly.
- When a user navigates to a cell, a textual representation of its value is shown in an editable field. This FR is fulfilled: all cells internally store a string representation of the input they were last assigned. This means formulas and numbers are stored as character strings which are displayed when they enter "edit mode".
- When the value of a cell changes, the displayed contents of all formula cells is updated. This FR is fulfilled: all formula cells are updated when any one cell is modified.

NFR Compliance

- For each algorithm implemented, it should not be possible to achieve a better time complexity: the algorithms used in the implementation of this software (shunting yard algorithm, binary tree-based expression evaluation, queue operations, stack operations) are to be implemented in the most efficient manner possible, according to the manner they were taught in the context of ELEC278.
- Dynamic memory allocation/de-allocation: dynamic memory is to be freed when no longer need, such that no memory leaks stem from those parts of the program that were implemented. Memory leaks are detected with a tool such as Valgrind MemCheck.
- Comments: each function and structure definition is accompanied by a comment block explaining its purpose and possible side effects / failure modes.

Implementation

The general layout of this program is as follows: a 7x11 2-dimensional array of "cell" data structures is used to store all data pertaining to the cells in the spreadsheet. Equations are represented as binary trees, where each node consists of a "treeNode" struct. The shunting yard algorithm is used to convert tokenized strings from infix to postfix notation, which can then be used to construct the corresponding binary tree. This use of a binary tree to represent expressions also made it trivial to implement additional features beyond the FRs – this spreadsheet application supports operators +, -, *, and /.

The Cell Array

Spreadsheet data is represented internally as a 2-dimensional, globally allocated array of “cell” structures. The definition of struct cell follows below.

```
typedef struct cell {
   double numval;
   treeNode *expression;
   char *strval;
   CELL_TYPE type;
   bool isValid;
} cell;
```

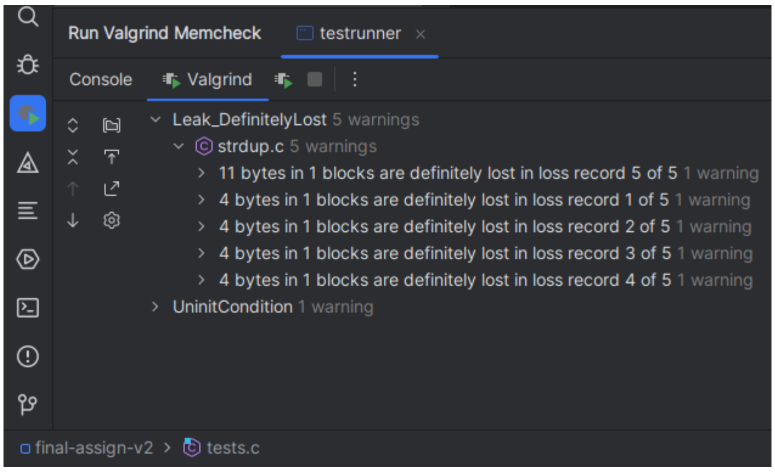
The CELL_TYPE enumerated variable is a flag holding one of “NONE”, “STR”, “EQN”, “NUM”, indicating the type of data being stored in the cell (the naming convention of which is fairly self-explanatory). This helps tie back into the concept of ownership, where a non-corrupted cell’s strval pointer is guaranteed to refer to valid memory if its type variable holds any value but “NONE”. The cell data structure is initialized with a constructor “init_cell(ROW row, COL col)”. This constructor allocates “MAXLEN” (256) bytes to the strval pointer using calloc, setting the type to STR, isValid to false, and expression to NULL. Memory allocated to strval is freed by the cell’s destructor “clear_cell(ROW row, COL col)”. Additionally, the “treeNode *expression” variable holds a pointer to the root node of a binary tree representing the equation for the given cell, if the cell is of type “EQN”. Otherwise it has a NULL value. Finally, “bool isValid” is set to true if the cell may be referenced by another cell’s equation, i.e. if it is of type “EQN” and holds a syntactically correct equation OR if it is of type “NUM”. The variables in “struct cell” are ordered from largest to smallest according to the number of bytes occupied by each variable. This ensures that the compiler can use less padding bytes when setting the struct’s layout, improving space efficiency. For example, “double”, “treeNode *”, and “char *” variables all have a size of 8 byte (on a 64-bit system), and are placed first. “CELL_TYPE” and “bool” variables have a size of 1 byte, and are placed last. This requires 32 bytes per struct, as opposed to 40 with a sub-optimal layout.

Setting and Getting Cell Values

Setting cell values is fairly straightforward: if the cell has type NONE, memory is allocated to the cell’s “strval” field, and the type is set to str. The input string is then copied into the cell’s “strval”. Then, the input string is checked for a leading ‘=’ character, and if one is found, an attempt is made to parse the inputted equation. If this is successful, the cell’s type is set to EQN, and its “numval” field is set to the result obtained from calling evaluate on the binary tree. If not, the cell’s “isValid” field is set to false. Finally, if no ‘=’ character is detected, the input is checked for a valid numerical representation. If the input constitutes a valid number string, the cell’s type is set to NUM and the input is parsed, setting the cell’s “numval” field to the result. Parsing is done with the built-in “strtod” function (why reinvent the wheel, after all). Getting string representations of the cell’s value for display in the spreadsheet cells is fairly simple: if the cell has type “STR”, the “strVal” field is returned. If the cell has type “NUM” or “EQN”, a temporary string buffer is allocated (using the alloca() function for improved memory safety), and “sprintf” is used to convert the cell’s “numval” field to a string, by printing into the temporary buffer. The string is then returned.

Cell Memory Ownership

An initialized cell (CELL_TYPE not NONE) owns the buffer pointed to by strVal. This buffer is released by the clear_cell function, which also sets the the cell’s type to NONE. Similarly, the “isValid” field indicates whether the cell owns data pointed to by its “expression” field. This memory is allocated when the cell’s type becomes “EQN”, and is released when the cell’s type is reset, or when “clear_cell()” is called. Memory leaks were eliminated using Valgrind Memcheck. While the testRunner program still leaks some memory, this memory comes from “interface.c”, and is out of scope for the student’s project.



```
Leak_DefinitelyLost 5 warnings
  strdup.c 5 warnings
    11 bytes in 1 blocks are definitely lost in loss record 5 of 5 1 warning
    4 bytes in 1 blocks are definitely lost in loss record 1 of 5 1 warning
    4 bytes in 1 blocks are definitely lost in loss record 2 of 5 1 warning
    4 bytes in 1 blocks are definitely lost in loss record 3 of 5 1 warning
    4 bytes in 1 blocks are definitely lost in loss record 4 of 5 1 warning
UninitCondition 1 warning
```

Note: the “UninitCondition” warning was investigated, and is likely a false positive.

This ensures that the program wastes no memory, and also verifies that (more importantly) no illegal memory reads or writes occur, which would lead to program crashes.

The shunting yard algorithm: stacks and queues

The shunting yard algorithm is an efficient (O(n) w.r.t. to the number of elements in the given expression) algorithm for converting infix expressions to postfix notation. This software uses a simplified version of this algorithm, which cannot handle parentheses or unary operators (functions). Pseudo-code for the algorithm is given below:

```
while there are tokens left in the input queue:

        if the token is a literal value or cell reference:
                put the token in the output stack

         if the token is an operator (+, -, *, /) o1:

                while there is an operator o2 atop the operator stack AND o2 has precedence
greater than or equal to o1:
                        pop o2 from the operator stack and push it onto the output stack

                push o1 onto the operator stack

while there is an operator o1 atop the operator stack:
        pop o1 from the operator stack
        push o1 onto the output stack

return output stack
```

After this, the input and operator stack are empty and may be deleted, and the output stack contains the tokenized expression in postfix notation. The first token in the expression is at the base of the stack, so the stack must be iterated over like an array, from bottom to top. The stack implementation being used (see “stack.h”) permits this. This algorithm is implemented in the “shuntingYard(const char *input, treeNode_stack *outputStack)” function, which uses the “nextToken(const char **input, treeNode *result)” function to tokenize an input string. For simplicity, tokens are implemented using the “struct TreeNode”, such that no conversion other than setting flags is necessary when constructing the binary tree. If the expression is syntactically incorrect, the “shuntingYard” function returns false, and the “outputStack” pointer passed to it does not point to an initialized expression tree.

Binary Tree Node Structure

The struct definition used to construct the binary expression tree are as follows:

```
typedef enum nodeType {
   VALUE,
   VALUEREF,
   OPERATOR,
   TOK_OPERATOR, //used to indicate validity of precedence field in union in treeNode
struct
} nodeType;

typedef double (*nodeFunc)(double, double);

struct treeNode {
   union {
      struct {
         union { //node has either a precedence (Token mode) or two children (tree node
mode)
            struct {
                struct treeNode      *left,
                                     *right;
            };
            int precedence;
        };
        nodeFunc func;
    };
    double value;
    double *valueRef; //non-owning, points into sheet
  };
  nodeType type;
};
```

The outermost union{} is used, since a treeNode shall have EITHER a value (nodeType = VALUE) OR a pointer to a value (nodeType = VALUEREF) OR two children, left and right, as well as a pointer ("func") to an operator function. This corresponds to the three possible elements in a valid spreadsheet expression: a number literal, a cell reference, or an operator. The nextToken(), shuntingYard(), and makeExprTree() functions set these flags and member variables accordingly, such that invalid union members are never accessed.

Constructing the Binary Expression Tree

The tree is constructed in the same manner as one would interpret the postfix expression: an empty stack of treeNode objects is created. A working stack is created, and the input stack (from the shunting yard function) is iterated over, starting at the bottom:

```
for (treeNode *current = inStack.base; current < inStack.sp; ++current)
```

Although strictly speaking a stack is only accessible from the top, this is marginally more efficient than reversing the stack and popping (An alternative would be for shuntingYard() to return a queue). If the treeNode *current points to a treeNode of type VALUE or VALUEREF, that node is copied and pushed onto the working stack, and the "current" iterator is advanced. If an operator node is encountered, the algorithm checks whether the working stack contains at least 2 child nodes (all operators are binary), then pops those nodes off the working stack, appends them to the left and right members of the current operator node, and finally pushes a copy of the operator node to the working stack. For a well-formed expression, it is expected that the working stack have a size of 1 after the input stack has been processed. If any of these stack size checks fail at any point, all working memory is freed and the function returns false. Once the function completes succesfully, the working stack is NOT freed (it is now owned by the cell from which makeExprTree() was called), and the function returns true, indicating successful ownership to the caller.

Evaluating the Binary Tree

Surprisingly, evaluating a valid binary tree is far easier than parsing an expression or constructing the tree. The evaluation function is very simple, and uses a recursive post-order traversal:

```
double evaluate(treeNode *node) {
   switch (node->type) {
      case VALUEREF:
         return *(node->valueRef);
      case OPERATOR:
         return node->func(evaluate(node->left), evaluate(node->right));
      default:
         return node->value;
   }
}
```

The function checks the current node's type. If the node is of type "VALUE", it returns the union member value. The union member is guaranteed to be valid by the flag "node→type". If the node is of type "VALUEREF", the double *valueRef pointer is de-referenced and returned. Finally, if the node is of the type "OPERATOR", the function calls the function store at the function pointer "node→func", passing it two arguments of type double: the return value of the recursive call to evaluate() for the node's two children, which are again guaranteed to exist by the flag's state. The "double (*nodeFunc)(double, double) func" pointer is guaranteed to point to one of four valid functions: "add()", "subtract", "multiply", "divide", and its value is set in the nextToken function. This ensures memory safety and avoids segmentation faults. As an aside, the case "VALUE" is handled as the default case. If there were an error in the node's flag, accessing the union as a simple double value is the safest failure mode as opposed to accessing it as a pointer of some kind, which may be de-referenced and cause a program crash.

Data Structure Macros

(found in stack.h and fifo.h)

These macros were adapted from a regular implementation for the stack and fifo (queue) data structures that were developed by the student in the context of ELEC278. They were adapted as macros using token pasting, i.e. by replacing all instances of a specific typename in struct and function definitions by a generic placeholder – "T" or "type". The macro is invoked in the file where a definition of that data structure is needed for that specific type – for example, in "expression.c":

```
DEFINE_STACK(treeNode)

DEFINE_FIFO(treeNode)
```

This essentially creates a new set of struct and function declarations for every invocation of DEFINE_STACK or DEFINE_FIFO. The function declarations follow a specific pattern, to ease the programmer's mental load – for example, all functions defined by DEFINE_STACK(treeNode) follow a naming pattern of "treeNode_stack_xxxx()", where "xxxx" may be replaced by "push", "pop", "delete", or "make". The use of such a macro allows the program to avoid using "void *" to refer to create generic, reusable data structures, reducing runtime overhead and eliminating the risk of segmentation faults. The creation of this macro was inspired by the student's desire to bring something akin to the templating system found in C++ to C. Debugging and quality testing of the implementation of stack and FIFO (queue) were done with the specialized implementation, to ensure that the debugger could properly step through the functions – this is not possible with macros. Search and replace with regular expressions was used to insert the correct macro token pasting sequences into the source file.

Handling Spreadsheet References – Updating Values

Within the binary tree expressions detailed above, references to other cells are parsed in the nextToken function. This process also includes bounds checking, to avoid referring to non-existent cells (for example, A0 is an illegal reference). The treeNode structs labeled with the "VALUEREF" flag then have a pointer to double as their valid union member – this pointer is initialized to point to the "double numval" variable of the corresponding cell struct in the spreadsheet data array. Thus, fetching data while evaluating expressions is trivial – the pointer simply needs to be de-referenced. There is no risk of accessing illegal memory, since nextToken checks bounds and returns false if an illegal cell is accessed. The onus is then on the caller to check nextToken()'s return value (true or false). When a value in the spreadsheet is updated, the program iterates over all other cells and calls "evaluate()" on the cell's expression if the cell is of type expression – while this is rather inefficient, it solves the problem of circular references causing program crashes (with infinite circular recursion). This operates in O(n * m) time, where n is the number of cells containing equations and m is the average number of tokens in a given equation.

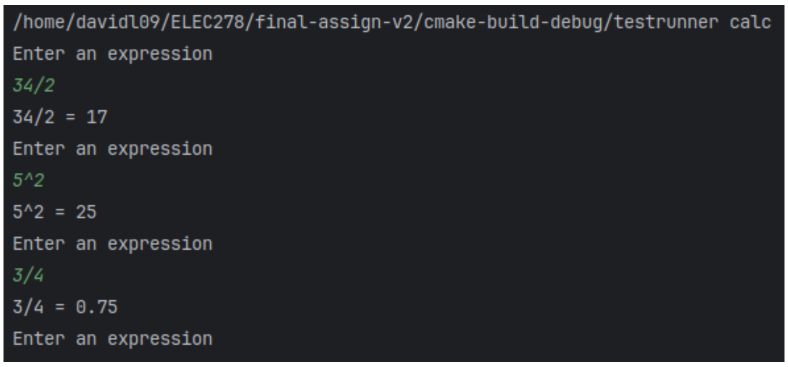
Testing

Testing may be done in two main ways:

1. Using the calculator feature in testrunner.c
2. Using the spreadsheet with various equations as show below

1. Running the calculator

To test the expression parser using the calculator, simply invoke the compiled binary produced by the “testrunner” target with a single argument, “calc”. This will run all tests as usual, however it will launch a simple calculator app following this. Example usage is shown below:

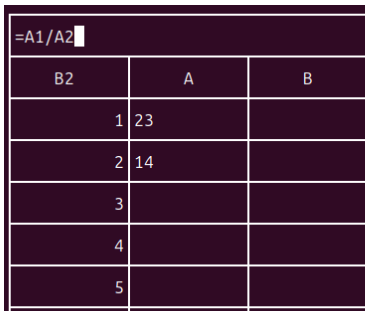


```
/home/david109/ELEC278/final-assign-v2/cmake-build-debug/testrunner calc
Enter an expression
34/2
34/2 = 17
Enter an expression
5^2
5^2 = 25
Enter an expression
3/4
3/4 = 0.75
Enter an expression
```

Results may be double-checked with a phone or pocket calculator for accuracy.

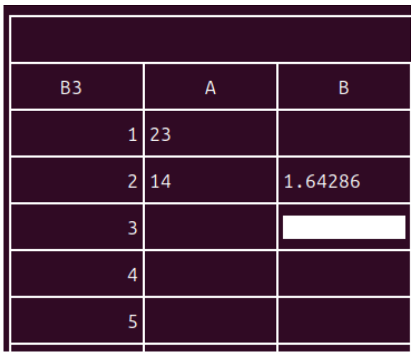
2. Running the Spreadsheet Application

To test the spreadsheet’s functionality, simply invoke the program in an appropriate CLI. For example, enter the following numbers and expressions into cells A1, A2, and B2 (Cell coordinates are case-insensitive):



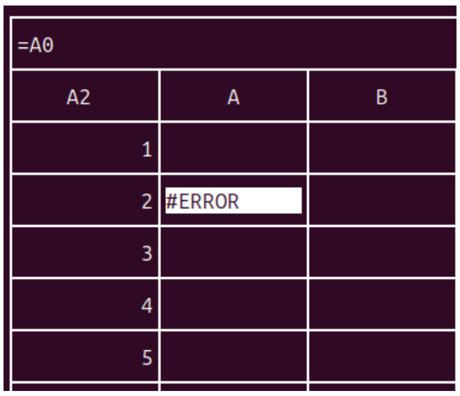
```
=A1/A2
B2  | A  | B
  1 | 23 |
  2 | 14 |
  3 |    |
  4 |    |
  5 |    |
```

The image on the right shows the expected result:



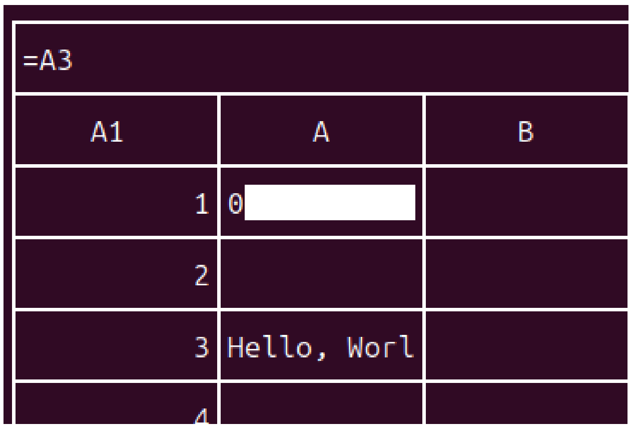
```
B3  | A  | B
  1 | 23 |
  2 | 14 | 1.64286
  3 |    |
  4 |    |
  5 |    |
```

To test the out-of-range error detection, enter “A0” or “C18” or any other invalid coordinate into a cell, as shown below:



```
=A0
A2  | A      | B
  1 |        |
  2 | #ERROR |
  3 |        |
  4 |        |
  5 |        |
```

After entering the value, the cell should display “#ERROR”. A further verification of error handling involves referring to a non-numerical cell as part of an equation: enter any string value into a cell, then refer to that cell in a different cell’s equation. The cell should display a value of 0, and no crashes or any sort of issues should occur.



```
=A3
```

A1	A	B
1	0	
2		
3	Hello, Worl	
4		