CMPUT275—Assignment 4 (Winter 2025)

R. Hackman

Due Date: Monday March 31st, 8:00PM

Per course policy you are allowed to engage in *reasonable* collaboration with your classmates. You must include in the comments of your assignment solutions a list of any students you collaborated with on that particular question.

In this assignment and all following assignments you should test against the sample executables. The sample executables try to help you out and print error messages when receiving invalid input (though they do not catch *all* invalid input). You do not have to replicate any error messages printed, unless the assignment specification specifically asks for error messages. These messages are only in the sample executable to help you catch when you write an invalid test case.

Memory Management: In order to complete some of these questions you will be required to use dynamic memory allocation. Your programs must not leak any memory, if you leak memory on a test case then you are considerd to have failed that test case. You can test your program for memory leaks by using the tool `valgrind`.

Efficiency Requirements: While there are not strict efficiency guidelines, your code should strive to not be unnecessarily innefficient. If your solution is not comparable for large test cases in how long it takes to run compared to the sample executable then you may lose marks due to your program hitting the time limit. As such, it is good to compare your program to the sample executable not only for output but also timing purposes. The sample executable purposefully not overly optimized, so this should be an easy target to reach.

Compilation Flags: each of your programs should be compiled with the following command:

```
gcc -Wvla -Wall -Werror
```

These are the flags we'll compile your program with, and should they result in a compilation error then your code will not be compiled and ran for testing.

Allowed libraries: `stdio.h`, `stdlib.h`, `string.h`, and `assert.h`. No other libraries are allowed.

1. **Linear Sequences**

   In this question you will be writing a C program `sequences.c` that will be able to compose arithmetic operators to form a linear sequence, and be asked to print out the next item in the sequence at any time. Your program will read in pairs of operators and operands which will make up the series of arithmetic operations to apply to each integer in the sequence to form the next integer in the sequence. Your program will have the following specifications for its arguments and its input.

   - Your program will expect one command line argument which is an integer, representing the initial value of the sequence.
   - Your program will then read input from standard input until reaching EOF, correctly formed input will always be one of the following
     - One of the operator strings `add`, `sub`, `mul`, or `div` followed by one integer.
     - Or, the string `n`
   - When receiving one of the operator strings, your program will add that operator to the series of operators to apply to each number in the sequence to produce the next. The integer provided after each operator string is the right hand operand to be used with that operator when applying it, the left hand operand of each operator will always be the current integer in your series, or the result of applying a previous operation. When calculating the next integer in the series the operators must always be applied in the order they were received in input!
   - When receiveing the string `n` your program will calculate the next integer in the series, as currently defined by your operators, and print it out followed by a newline.

   So, for example, if your command line argument was `3` and your program receieves the following input

   ```
   add 1 mul 2 n n
   ```

   then your program would print out

   ```
   8
   18
   ```

   because $(3 + 1) * 2 = 8$, and $(8 + 1) * 2 = 18$.

   Deliverables For this question include in your final submission zip your C source code file named `sequences.c`

2. **Maze Game**

In this question, you will implement an ADT meant to represent a maze. For this question you have already been provided with the files `main.c` and `maze.h`. You may not modify either of these files. Even if you do modify those files and submit them they will be overwritten with the original files and if your solution required your modifications then your solution would no longer work, so do not modify them. The file `main.c` is the client code that uses your ADT, and the file `maze.h` is the header files that declares the functions that `main.c` relies on. Your job is to implement the ADT in the file `maze.c`.

Your maze will be reprsented by a two-dimensional grid of tiles, where each tile is represented by a character. The following are the types of tiles that comprise your maze:

- Start Tile - Denoted by the character `'S'`, this denotes where the player will begin in the maze when it is first created, or when it is reset. Every maze must have *exactly* one start tile. Otherwise, the start tile behaves exactly as an *Open Tile* described below.
- Goal Tile - Denoted by the character `'G'`, this denotes the "goal" or exit of the maze. When the player reaches this location its task is done. Each maze must have *at least* one goal tile.
- Open Tile - Denoted by the character `'O'`, this denotes a normal tile of the maze that the player is allowed to walk through.
- Wall Tile - Denoted by the character `'X'`, this denotes a solid wall tile of the maze that the player is not able to walk through.
- Teleporter Tiles - Denoted by any digit character (`'0'` through `'9'`), this denotes a teleporter tile. When the player walks on a teleporter tile their location becomes that of the other matching teleporter tile. You may assume that if a maze has a teleporter tile for a number there will be exactly two of that number appearing in the maze. If you are unsure about the behaviour, as always, test with the sample executable.
- Icy Tiles - Denoted by the character `'I'`, this denotes an open tile which the player may pass through but with an icy floor so the player will continue sliding in the direction they were moving until hitting a wall, hitting the edge of the maze, or landing on the next non-icy tile. If you are unsure about the behaviour, as always, test with the sample executable.

In order to implement this ADT you must define the structure type `Maze` and all of the functions declared in the provided header file `maze.h`. Each of the functions have comments describing them in the header file, but are also described here. The functions are as follows:

- `struct Maze *readMaze()` - This function read a maze in from standard input and returns a pointer to a heap-allocated `Maze` structure which you have initialized to represent the maze that was read in. When reading in a maze your function will read in lines of characters until reading in a completely blank line. Each line represents a row of the grid that makes your maze. You may assume EOF is not reached before completing the reading in of the Maze, however your function must make sure that it receives only exactly one start tile and at least one goal tile. If your function does not detect exactly one start tile and at least one goal tile then it instead returns a `NULL` pointer.
- `struct Pos makeMove(struct Maze *, char)` - This function takes a pointer to a valid `Maze` structure and a character that must be one of `'n'`, `'e'`, `'s'`, or `'w'` which represents the directions north, south, east, and west respectively. This attempts to move the player in the given direction, following the rules of the maze, and returns a `Pos` structure that

stores the players new x and y coordinates as well as mutates the `Maze` structure so that the player has made the given move. If the players new position is the goal state the function instead returns a `Pos` structure with x and y coordinates both set to `-1`.

- `void reset(struct Maze *)` - This function *resets* the maze, which simply resets the player's current position to the original starting position of that maze.
- `void printMaze(struct Maze *)` - This function prints out the maze to the screen. This prints a border around the maze where the top and bottom are enclosed in `'='` characters and the sides are enclosed in `'|'` characters. All of the tiles of the maze are printed, except for the tile on which the player currently resides where instead the character `'P'` is printed to convey that the player is located there.
- `struct Maze *destroyMaze(struct Maze *)` - This function frees all memory associated with the given `Maze` structure which was created with `readMaze`, including the `Maze` itself. Returns `NULL`.

Note: All of these functions work on a `Maze` structure which *you* get to define. Define the `Maze` structure in your `maze.c` file and make sure to include all the fields you will need to make the ADT behave as described.

Building and Testing: As you are only writing the ADT in this question and not the main file you must compile and link your file with the provided main. This can be done in the following command, assuming the files `maze.c`, `maze.h` and `main.c` are in your current working directory:

```
gcc main.c maze.c -o myMaze
```

In order to be able to test before you implement every single function, you may want to include empty implementations in your `maze.c` until you are able to finish them. That way you can test the functions you have implemented first by writing test cases that only invoke the functions you have written. For example, if you have written only `readMaze` and `printMaze` you can at least test those two functions by writing a test case that simply provides the maze and then asks to print it. Since you only have a blanke implementation for `destroyMaze` in this example your program would leak memory, but you would at least be able to see if you are able to properly read and print a maze. You may if you want even write your own main in another file only for testing, this main must not be submitted.

Deliverables: For this question include in your final submission zip your C implementation file `maze.c`. **NOTE: This file must NOT contain a main function!** This file should contain only the definitions of your `Maze` structure, as well as the definition of all of the functions declared in the `maze.h`. It may also include additional helper functions that you wrote to help with your implementation.

How to submit: Create a zip file `a4.zip`, make sure that zip file contains your C implementation files `sequences.c`, and `maze.c`. Assuming all five of these files are in your current working directory you can create your zip file with the command

```
$ zip a4.zip sequences.c maze.c
```

Upload your file `a4.zip` to the a4 submission link on eClass.