1 How to use the program

To run the parser, simply use:

```
stack run -- [name of file].gridlock
```

The files are expected to be stored within /allGridLockGames, and all the provided passing/failing files have been copied there.
To run the TUI application, `stack run`
And for the command-line version, `stack run - command-line`

2 Using `State` to keep track of game changes

Since the parsing of the game files required to check if each move was valid, I decided to use `State` to keep track of the grids after each move is applied, which also made compiling each `Grid` for the `GameRecord` a lot easier.
For file parsing, a function `processMove` updates the `State` (and passes the current `Grid` as a state value) if the current move is valid, otherwise passing the appropriate error - this function is `traverse`d through to get all the grids throughout the game (and errors, if any).
For the playable games, `processMove` is still used, but now called when needed after user input, and the current `State` of the grid is used to display the game.

3 Command-line game

For the additional functionality, I decided to start by making a playable version of Gridlock in a command-line style.
I tried to use similar principles to parsing to take in user input for the set-up of the game, e.g. ensuring the grid size inputs are positive integers. Then as stated in section 2, the players are then asked for their next move, coordinates and colour, which is turned into a `Move` so it can update the `State` using `processMove`. The game allows players to re-input if any information is invalid.
The program checks if the game is over after each user input, and displays the winner once it is.

4 Goal Patterns

While testing the command-line version of the game, I came up with the idea to add 'goal patterns' to the game of Gridlock to make it more interesting. The idea is that both players get a 2x2 pattern they should attempt to match somewhere inside the grid. This means that not only should players worry about making valid moves and 'locking' the other player, but also to fill in their patterns and stop the other player from doing the same. Winning as normal now gets 2 points, and extra points are awarded for each correctly filled in cell of the pattern (the program finds the windox in the larger grid which shows the best score).

I created a method to get randomised patterns based on the player's name (turned into an integer by its ASCII values) and the size of the game grid, which makes around $(c+1)^2$ patterns for c colours in the game. This means that patterns with 1/2 colours become repetitive, so I made a list for these to make gameplay more fun.

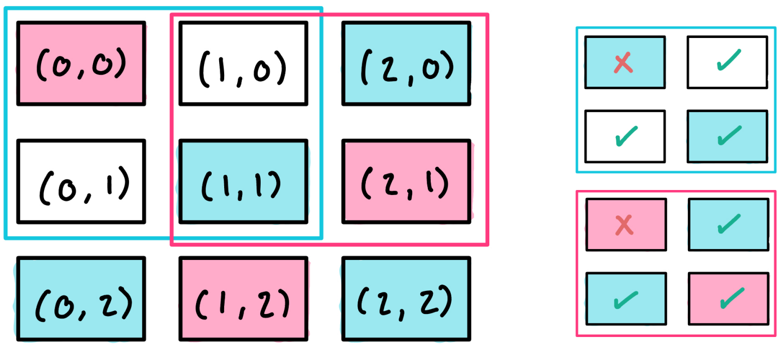

Figure 1: Both players get 3 points.

Unfortunately, the random method sometimes gives incomplete patterns (i.e. they are impossible to get since there are too many empty cells for the game to be over), but there's not enough to employ a complicated way of avoiding them (only 3 in 16 for 3-colour patterns), and it allows for a different gameplay - it could encourage the player to be more on the offensive and ensure the other player does not get their pattern.

5 TUI using `Brick`

Once I completed the command-line game and added the patterns, I decided to try adapting it to play as a TUI game using `Brick`. I mainly used the `Brick` user guide to help through this process, and took inspiration from the Halatro front-end code.
For Halatro, I mainly looked at to see how the code was structured and the functions being used. I knew that the game would have some similarities to Halatro, so I sometimes looked at the code to see how a specific part of the app was done. No code was copied.
As I got more used to the `Brick` library, the process became more intuitive and I started to understand where I didn't need to use extra `Brick` features, such as not needing the `[Modifier]` or `Location` of any events, which made coding a lot easier.

6 List of resources read and/or used

In rough order of access, (links are embedded when appropriate).

1. Haskell Tutorial - 12 - Writing Parsers From Scratch, James Hobson.
2. The State Monad: a tutorial for the confused? Brandon Simmons' website.
3. Haskell: State and StateT examples, Vlad Posmangiu Luchian.
4. Haskell - Tutorial 7 - GUI Programming, James Hobson - *not used, but helped understand GUI/TUI more.*
5. Building Terminal User Interfaces with Haskell, FP Complete Corporation - *outdated now, but useful for initialising the TUI App.*
6. Brick User Guide, Jonathan Daugherty - *used as a 'tutorial' to build the TUI app using Brick.*
7. The Halatro Coursework, Alex Dixon