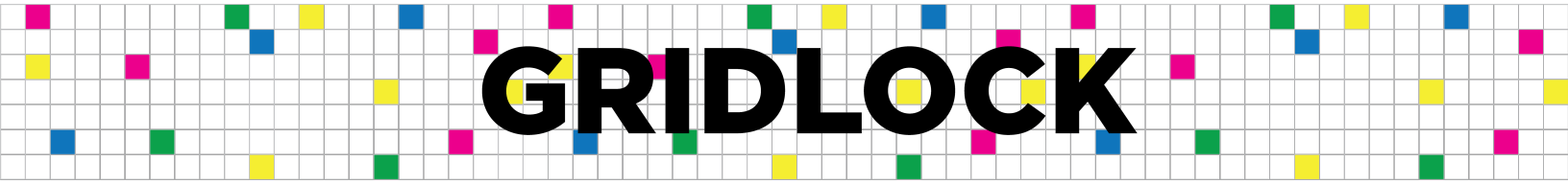


Gridlock is a simple colouring game played between two players. Here are the rules:

- The game is played on a grid made of M by N cells. The grid is initially empty.
- On a player's turn, they may colour in any cell of the grid, so long as they do not colour it the same colour as an adjacent cell.
- The game ends when no more moves can be made, and the player who made the last move wins.

An Example Game

Let's consider a game played on a 3x3 board between Amy and Bill.

The board is initially empty:

Amy starts by colouring the top-left cell red:

Bill colours the top-right cell red:

Amy colours the middle cell green:

Bill colours the bottom-left cell red:

Amy colours the bottom-right cell red:

Bill now has no available moves, so Amy wins.

Your second coursework assignmnent is as follows:

> Write a Haskell program that allows the user to explore individiual games of Gridlock by parsing written records of Gridlock games.

This document lays out what is expected from your program. **Please read this specification in its entirety before embarking on a submission.**

If you have questions which are not resolved by this specification, which may be of interest to other students, please contact me. For specific programming queries related to your own coursework submission, contact your lab tutor.

Minimum Requirements

A **minimal** submission can do the following things:

- Draw a Gridlock grid as a nicely formatted string.
- Convert a file in `.gridlock` format into a parsed `GameRecord` structure.
- Write a program which tries to parse a `.gridlock` file and:
  - Displays the game in some human-readable way if it represents a valid game.
  - Displays an error if the file is not well-formed.

> The submission must also include a very short report titled `report.pdf` which explains how to use your program, any interesting architectural choices made, and your personal experience with writing the code. The report should be roughly 1 page of A4, but this is not strict.
>
> Append to the report a full list of any resources read and/or used.

Any program that meets the requirements above will pass.[1] Any program that meets the requirements and is implemented to a high standard will receive a strong mark.

[1] Barring issues of plagiarism or other serious deficiencies, of course.

Submission details

The coursework is due Thursday Week 11 at midday.

Before submitting, make sure your project meets the **minimum requirements**.

To submit your work, perform **ALL OF THE FOLLOWING STEPS, IN ORDER**:

- Make a backup of your project, just in case.
- Delete the `.stack-work` directory, which contains build artifacts. (It is easy to do this directly from the file manager inside Visual Studio Code.)
- Compress the **whole project folder** into a `.zip` or `.tar.gz` archive.
- Check that the archive contains all your files.
- Upload the archive to the Tabula submission portal in the usual way.

It is your responsibility to ensure your submission is complete and contains the work that you intended to submit. Failure to follow the above steps will **not** be considered mitigating circumstances and delayed submissions as a result will earn the appropriate late penalty.

Exercises

Ex. 1. Read this document in full.

Ex. 2. Do the final lab sheet. It will give you a good grounding in how to write a parser in Haskell.

Ex. 3. Grab a copy of the coursework template:

```
git clone https://cs141.fun/gridlock
```

Look through `Gridlock.Types` in the `src` folder to see the data types you will be working with. Pay special attention to `Grid` and `GameRecord`.

Ex. 4. Open the `Gridlock.DrawGrid` module, in the `src` folder.

Implement `drawGrid` as specified on Page 7 of this document. Use `stack test` to check your work.

Ex. 5. Open the `Gridlock.Parser` module, in the `src` folder.

Implement `parseGame` as specified on Page 5. Use `stack test` to check your work.

Ex. 6. Open the `Main` module, in the `app` folder.

Implement `main`. An example of what your program could do is given on Page 7.

Consider adding any extra functionality, or replacing it with a more featureful or interesting program. The application will be tested by hand, so you can add or modify its behaviour however you want.

Ex. 7. Write your short report. Make sure to include any resources you used, and any interesting architectural decisions you made.

Ex. 8. Submit and relax :)

Gridlock Format

We write down games of Gridlock using a special format. Here is a written version of the same game described on Page 1:

```
GRIDLOCK
Players: Amy, Bill.
Grid: 3x3.
Colours: red, green.
Amy plays red at (0,0).
Bill plays red at (2,0).
Amy plays green at (1,1).
Bill plays red at (0,2).
Amy plays red at (2,2).
Bill cannot move.
Amy wins!
```

Let's break down the notation piece by piece.

```
GRIDLOCK
```

Every game starts with the word `GRIDLOCK` to show that it is indeed a game of Gridlock!

```
Players: Amy, Bill.
```

There are always exactly two players in a game of Gridlock. The players are listed in the order they play (so Amy will go first).

```
Grid: 3x3.
```

The size of the grid is given as MxN, where M is the width (number of columns) and N is the height (number of rows). M and N can be any non-negative integer, including zero, and the grid does not have to be square.

```
Colours: red, green.
```

The colours that can be used to colour in the grid are listed here, separated by commas. There must be at least one colour. The set of available colours is `red`, `green`, `yellow`, `blue`, `magenta`, and `cyan`.

```
[player] plays [colour] at [coordinate].
```

Each move is given in order. (For a 0x0 board, there will be zero moves.) The player's name is followed by the colour they used, and then the coordinate they coloured in. Coordinates are given as (x, y), where x is the column and y is the row. The top-left cell is $(0, 0)$.

A move **must** be legal. That is, the cell must be empty, the colour must not be the same as any adjacent cell, it must be that player's turn, and they must play a colour from the list of available colours within the grid boundaries.

```
[player] cannot move.
[player] wins!
```

The game ends when a player cannot move. The player who made the previous move (i.e. the other player) wins the game. A GRIDLOCK file always ends with someone winning as given above.

Invalid Games

There are may ways that a Gridlock file may be invalid. The skeleton code for the coursework includes many examples that fail in different ways. Here are some invalid cases that your parser should be able to handle. **The specific error thrown is not too important**, but it is nice to have a helpful error message.

- One or more of the required pieces of information ("GRIDLOCK", Players, Grid, Colours) is missing.
- The grid size is invalid (e.g. negative or not a whole number).
- The list of colours is empty.
- A player tries to play a colour that is not in the list of available colours.
- They try to play a colour in a cell that is already coloured, or out of bounds.
- They try to play a colour that is the same as an adjacent cell.
- They try to play a colour when it is not their turn.
- They try to play a colour when the game is already over.
- There is no end condition given when the game is over.
- The formatting is wrong (e.g. something is misspelled or a full stop or comma is missing).

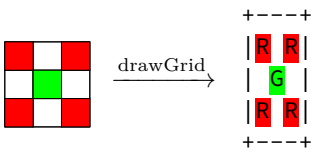
Drawing the Grid

This is a simple warmup to get you started and to make sure you are happy using the `Map` data type, which is what is used to represent the Grid structure. No monads will be needed for this part, it's a "normal" function definition :)

The `Gridlock.DrawGrid` module contains a function `drawGrid` for you to implement. A grid should be drawn as a series of coloured cells, with the colour of each cell represented by a single character.

We want to draw a grid in the following very specific way: Each cell should be represented by a single character, coloured the same as the cell. We also want to put a border around the outside of the grid. Horizontal edges should use `-` and vertical edges should use `|`. The corners should be represented by `+`. Empty cells should be represented by a space.

For example, the final state of the game described on Page 1 should be drawn as follows:



The module `Gridlock.ColourSquares` includes a function `square` that takes a `Colour` and gives you back the character used to represent a cell of that colour. Make use of the `square` function in your implementation—it will be much simpler! There are some (non-exhaustive) tests in the testing suite to help you check your implementation.

The Application

Ada implements a minimal frontend (maybe 10 lines) that has the following behaviour.

- The user runs the program by giving it the name of a Gridlock file as an argument on the command line. (`stack run -- mygame.gridlock`)
- The program reads in `mygame.gridlock` as a string (in the `IO` monad), and uses the parser she wrote to parse it into a `GameRecord` structure.
- If there was an error parsing the file, or the game is invalid somehow, Ada's program prints an error message. Otherwise, it uses `drawGrid` to print out the grids in order.

Assuming Ada commented her code sensibly, used good abstractions for her parser, and briefly explained her approach in the report, she may well get a very good (i.e. first-class) mark on the coursework. However, there is plenty of scope to be creative and add more functionality to the program, and that would be worth a higher mark.

Assessment Criteria

This coursework is broadly assessed on the same criteria as the first. In particular, the language used about justification and elegance in the first coursework spec still applies.

Functionality (40%)

The **functionality** mark is a subjective assessment of your program. It does not take into account the quality of the code itself; just the end result. Included in the "functionality" mark are the *complexity* of the final program (how much it does); the relative *difficulty* of the task that has been undertaken; and the *competence* with which that task has been executed.

A relatively simple program with few capabilities, competently executed, will be worth a decent grade for functionality. Exotic, creative project ideas will be rewarded. Instances of several people producing very similar "extensions" will be treated with scrutiny.

Justification (30%)

In this coursework, appropriate justification should be included both in-code and in the short report that accompanies your program. Individual functions should be commented appropriately regarding the choices made at that level. The accomapnying document should include information about the program itself, any libraries used, broader architectural decisions, and your personal experience of writing the program.

More emphasis is placed on the *quality* of your documentation than the *quantity*. Thorough but brief documentation will be marked with leniency; waffle and bluster will not. Evidence of your own understanding should be shown.

Elegance (30%)

Elegance marks are derived entirely from the code itself. Elegant code will:

- make use of programming patterns discussed in lectures;
- abide by the principle of *separation of concerns*;
- make use of appropriate data structures;
- be easy to read;
- **not** feature long, repetitive blocks of code.

Code meeting all these expectations will do well for elegance.

Plagiarism and Cooperation

For an open-ended coursework such as this, you will naturally want to seek help, advice and guidance from all sources, including your friends, members of the Computing Society, and the wider programming community beyond the university. This is encouraged!

However, the maxim of "talk together, code alone" still applies, as do the guidelines laid out in the first coursework specification:

- **You must not paste code written by any other person into your implementation, even if you make changes to it afterwards.**
- **You must not share your own implementation publicly or privately with any other person, inside or outside of the univeristy,** with the exception of lab tutors and myself.

If you have made use of outside resources (books, websites, other people's unit tests, etc.), you **must** name those resources in your `report.pdf`; explain how you used them; and give a link for where to access them.

While you may discuss general principles and overarching approaches, you **should not** ask other members of the CS141 course to help you directly implement your code. If someone asks for assistance of this type, **you should not give it**. If you need a prod in the right direction, ask one of your your lab tutors in the labs, or speak with Alex.

For more information about cheating and plagiarism, see the relevant section in the DCS student handbook. If you would like to report any potential cases of cheating or you have a question about the requirements laid out here, I will be very happy to talk to you.

Possibly Asked Questions

How can I do well in this coursework?

The purpose of this coursework is to assess your ability to produce real, functional computer programs. The skills of a good programmer are many and varied, but a good submission for this coursework will definitely succeed at the following:

- Making use of **available resources** (books, blog posts, documentation) which are relevant to the program you are developing;
- Thorough **explanation** of the high- and low-level design decisions made while writing your program;
- The use of **idiomatic functional programming techniques** to write elegant, readable, and reliable code.

In order to get a superior (>80%) mark on this coursework, you should go *above and beyond* the minimum listed in this document. So long as the minimum requirements are met, the sky is the limit!

Can I use ChatGPT/Copilot/Claude/etc to... No.

In this coursework, the use of generative tools is **explicitly forbidden**, for all use cases. That includes, but is not limited to:

- Generating ideas for how to approach exercises
- Explaining exercises to you
- Reporting on how an attempted solution could be improved
- Generating code
- Generating documentation

You are allowed to use the tools built into the Haskell Language Server in VS Code, including `hlint` (the blue squiggles), which will sometimes suggest meaningful simplifications or improvements to your code.

Strong positive evidence of AI tool usage (for example, if you are seen using it to solve the coursework) then it will be treated as a form of academic misconduct and with the same set of penalties.

It is fine to use AI tools while working on the lab exercises though, and that might help you to understand the coursework better. But remember, Alex and the lab tutors are always available to answer any questions you have, and to point you in the right direction.

How do I know if my solution is good? How can I get help?

As a hint, you should answer the following questions while producing your code:

- In what situations is a Gridlock file *invalid*?
- How can I detect whether a file is invalid?
- What error should I print in those cases?
- How will I display each move to the user?
- How will the user interact with the program, if at all?
- What other functionality shall I add to my program?
- How can I show off skill with functional programming (especially abstractions like monads)?

If you have concerns, or you are not sure of how to progress, speak with your lab tutors (via email or in person during lab sessions) or contact me directly (via email, after a lecture, or preferably during office hours.) We are happy to take a look over your work and offer guidance and advice.

You are not alone!