Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Module: CSU44012,
Functional Programming
Date: December 2023
Name: Claire Gregg,
Student Number: 20332015

Minesweeper-o-Matic

Assignment Description: The final project for CS4012 requires you to build an auto-playing implementation of the game minesweeper; use Threepenny to build the user interface and provide as complete an automatic player as you can.

Note that all of my source code is available on my GitHub at https://github.com/clairegregg/Minesweeper-O-Matic.

1. Phase 1

1.1 - Assignment Brief

1. Model the game of Minesweeper in Haskell. With no UI or interaction model this part of the project is simple enough.
2. Build a UI for the program from phase 1 using Threepenny GUI. You can specify the details of how the user interacts to play the game (i.e. what the game looks like and exactly what controls are used to select locations and clear or flag them). It's probably best to keep this simple.

The following are necessary

- The program must distribute mines randomly around the grid (*complete)*
- Allow the user to uncover and flag mines (interactively) (*complete)*
- Detect the endgame condition (*complete)*

1.2 - Modelling Minesweeper

I enjoyed this part of the project most.

1.2.1 - The Data Structure

The basic data structure can represent any game of MineSweeper

- The **Game** is represented as a combination of a **Map** and a **GameState**.
- The **GameState** tracks whether the game is currently in play, or has been won or lost.
- The **Map** tracks the game itself as a 2D array of **Squares**.
- A **Square** represents a Minesweeper tile, which contains **Contents**, and is in one of three states:
    - **Unflipped**
    - **Flagged** (which must be unflagged before being revealed)
    - **Revealed**
- **Contents** consist of either
    - **Mine** or
    - **Empty X** - where X is the number of bombs adjacent to the tile

1.2.2 - Generating the model

To generate the model, the following are passed in:

- *Width* of the map
- *Height* of the map
- *Random stream* of numbers in the range (0, w*h) (infinite length)
- *Difficulty* - the percentage of the map which will be bombs

(*Difficulty * (w*h))* unique positions are taken from the random stream, and converted into indices into the 2D array of the map. The map is initialised to have all **Unflipped (Empty 0)** squares, then the bombs are placed at the generated indices, and the Empty squares around each bomb are incremented.

1.2.3 - Interaction with the Model

2 actions are required for this game:

1. *Flip a square* - if it is flagged or revealed already, do nothing; if it is unflipped, reveal it!
2. *Flag a square* - if it is flagged already, unflag it; if it is revealed, do nothing; if it is unflipped, flag it!

These actions are implemented by simply swapping out the top level constructor of a Square (**Unflipped**, **Flagged**, **Revealed**), which is selected by indexing into the Map.

Expanding flips of **Empty** Squares

If a square's contents pattern matches to **Empty 0**, flipping that tile should result in *all surrounding squares being flipped*. This is a standard feature in most Minesweeper games, so I made sure I implemented it. For this, I just implemented a new function which is called when an **Empty 0** square is flipped, which flips all surrounding squares.

1.2.4 - Win/Loss checking

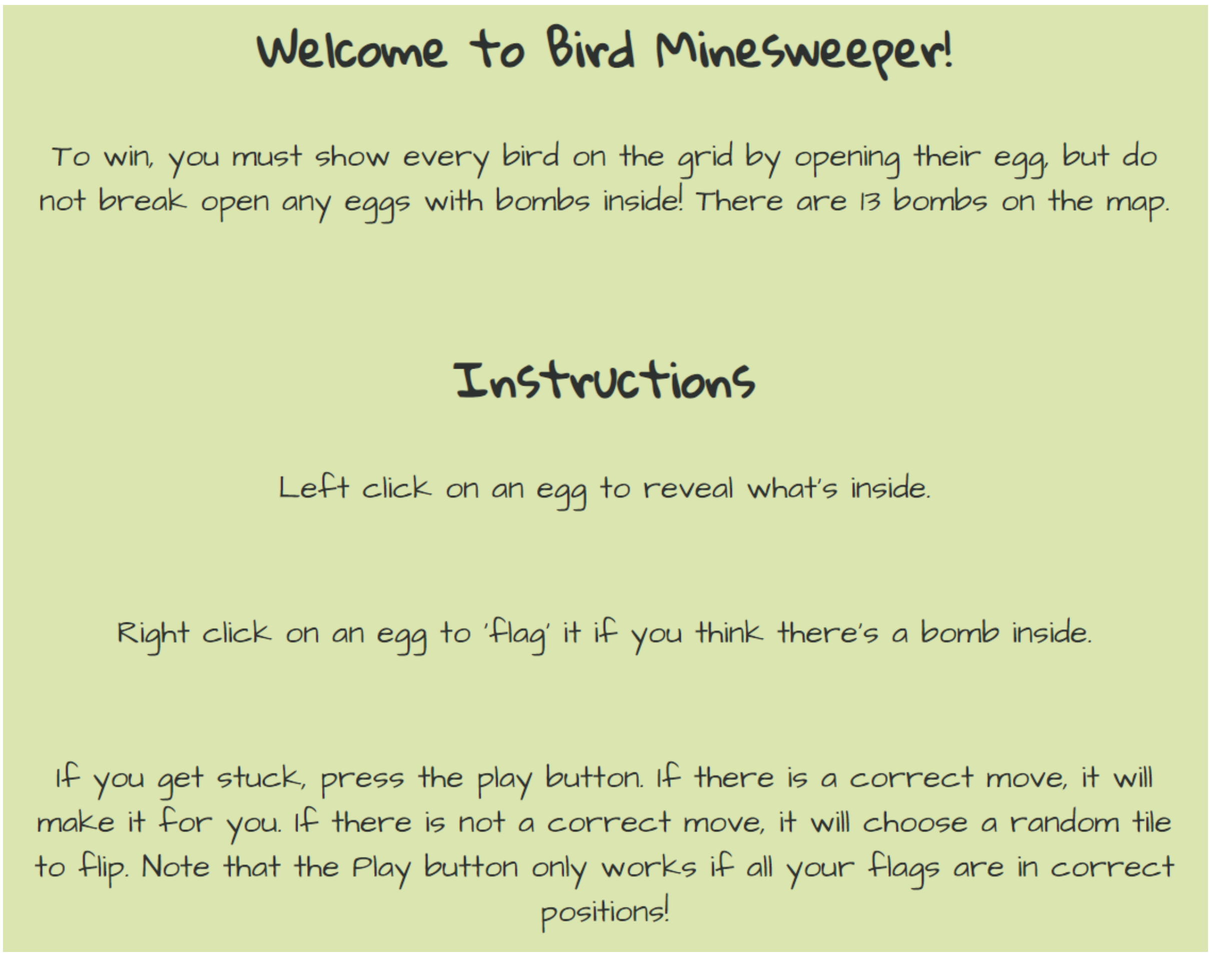
Whenever a square is flipped, a check is made if the game has been won/lost. Checking loss is easy - if the square which was intended to be flipped now pattern matches to **Revealed Mine**, the game has been lost. Importantly, this must pattern match to **Revealed**, as otherwise the player will lose when they try to flip a flagged mine, which is not intended. Win checking requires looping through the whole map, and checking if every tile is in a valid "win condition". For a **Mine**, the tile can be **Flagged** or **Unflipped**, and an **Empty** tile must be **Revealed**. An improvement to this win checker could be made to allow for the game to be won if all mines are flagged, regardless of if all **Empty** tiles are revealed.

1.3 - Building a UI using Threepenny

1.3.1 - Elements of the UI

1.3.1.1 - Description

I included instructions for the game as an element of my Threepenny website:



Welcome to Bird Minesweeper!

To win, you must show every bird on the grid by opening their egg, but do not break open any eggs with bombs inside! There are 13 bombs on the map.

Instructions

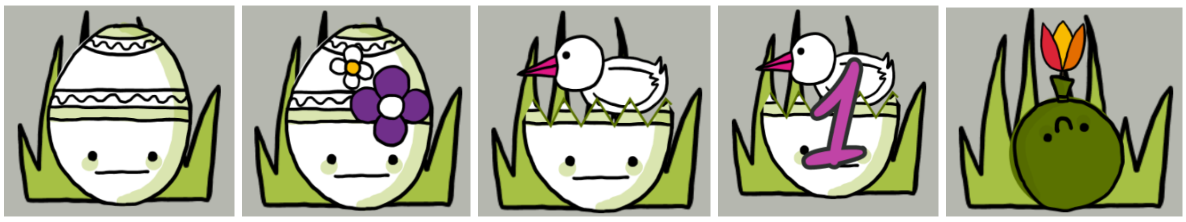
Left click on an egg to reveal what's inside.

Right click on an egg to 'flag' it if you think there's a bomb inside.

If you get stuck, press the play button. If there is a correct move, it will make it for you. If there is not a correct move, it will choose a random tile to flip. Note that the Play button only works if all your flags are in correct positions!

This tells the user what the controls are, how many bombs are on the map (calculated by multiplying width*height*difficulty and flooring it), and explains how the *play* button works (this will be explained in Phase 2).

1.3.1.2 - Tiles

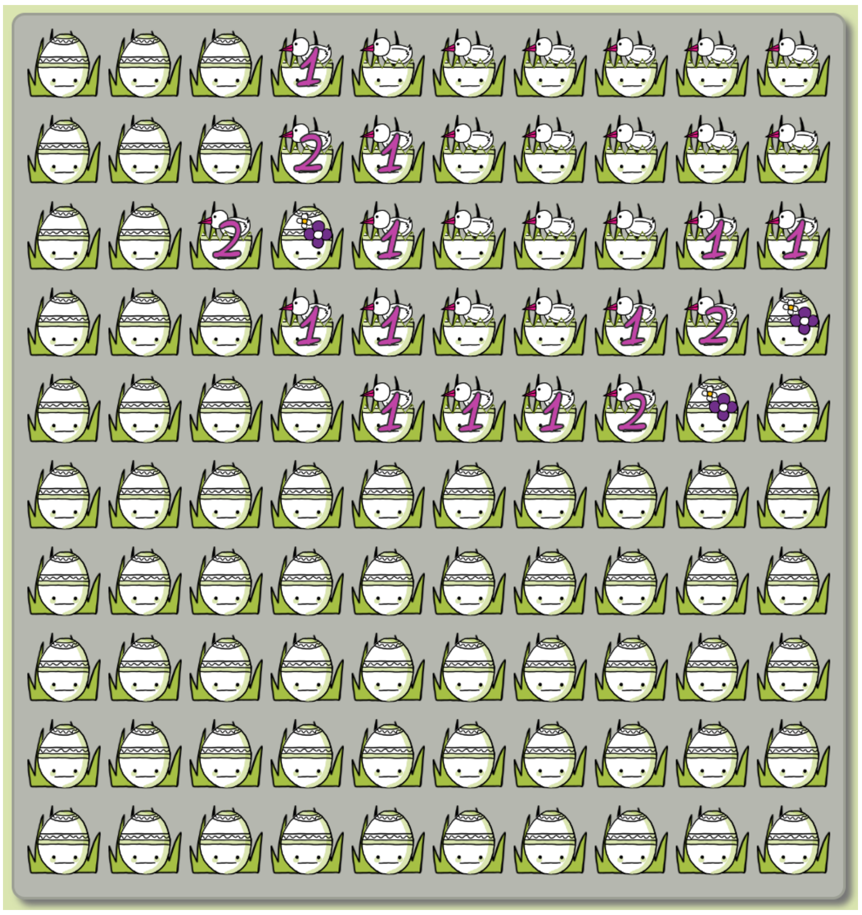


These are the representations for each square:

1. **Empty _**
2. **Flagged _**
3. **Revealed (Empty 0)**
4. **Revealed (Empty 1)**
5. **Revealed Mine**

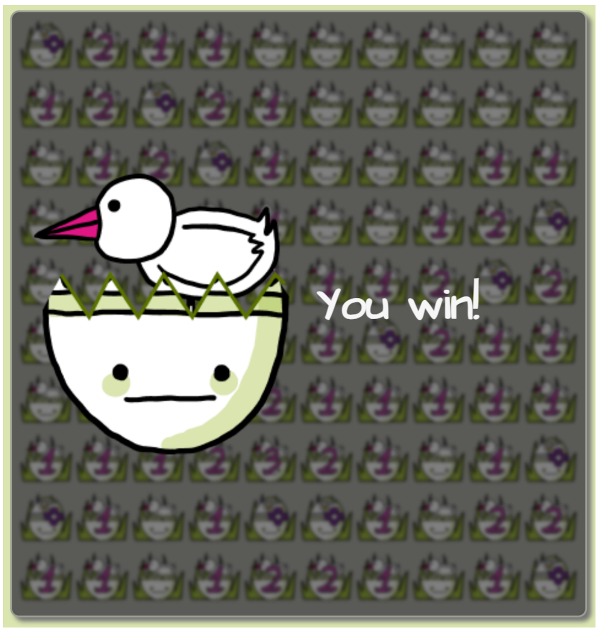
Each Threepenny tile object takes in: their (x,y) coordinates in the map; and an **IORef** to the game model. On click, it calls the game model to flip the tile at those coordinates. On right click, it calls the game model to flag the tile at those coordinates.

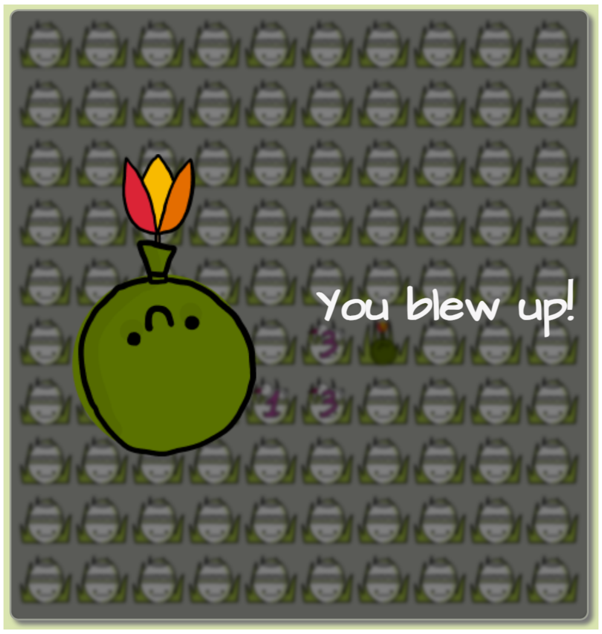
1.3.1.3 - Grid



The grid takes in width, height, an **IORef** to the game model, and the end game display **UI Element**. It creates a 2D array of **Tile**s. When it is clicked or right clicked, it updates all of the **Tile's** visuals based on the changed game model. The end-game display is also modified based on the current game condition (**Play**, **Won**, **Lost**).

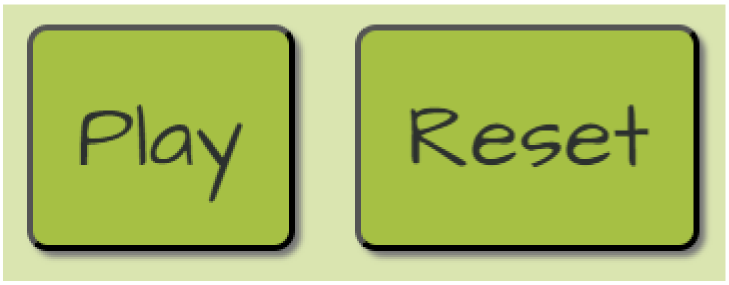
1.3.1.4 - End game display





These end-game displays appear when the game state is either won or lost. They are implemented as a div, whose CSS includes “**display: none**” when the game is in **Play**, and “**display: block**” when the game is **Won** or **Lost**. This means when the game is not in **Play**, it cannot be clicked.

1.3.1.5 - Buttons



The *Play* button allows for the autoplayer to play a single move. The reset button creates a new randomised game, and writes it to the Game **IORef**. After either button is pressed, a click on the grid is simulated using JavaScript to force a rerender. I’ll discuss this more in 1.3.3 - Problems faced - A discussion of FRP.

1.3.2 - Problems faced - A discussion of FRP

The main problem I dealt with was trying to do things Threepenny does not yet implement! These ended up being allowing right click as an action, and simulating a click on an element. Both were easily solved using Threepenny’s inbuilt **runFunction**, which allows you to run JavaScript. However, simulating a click using JavaScript could have been avoided if I set up event streams using FRP.

2. Phase 2

2.1 - Assignment Brief

Add a 'play move' button to the game from phase 1. Pressing this button will attempt to play a good, safe, move. I will note that Minesweeper is known to be NP-complete, which means that a well written and efficient solver is, well, hard! That link contains some very good commentaries and links on solving minesweeper. Check out his PDF presentation linked there.

For this part it is necessary that

- The program play a move automatically if there is an unambiguously safe move available (*completed)*

It is desirable that

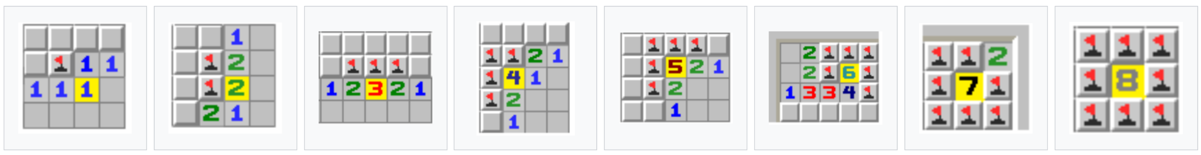
- If a safe move is not available the program will play the least dangerous move available (*not completed, efforts went to other parts of the project)*

2.2 - The Auto-Player's rules

I did these with help from https://www.minesweeper.info/wiki/Strategy, so all images are from that.

2.2.1 - Basic Bomb Finding

(Image taken from the wiki)

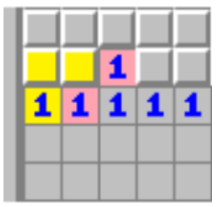


If an **Empty X** tile has exactly X unrevealed tiles next to it, then all of these are mines and can be flagged.

2.2.2 - Basic Empty Finding

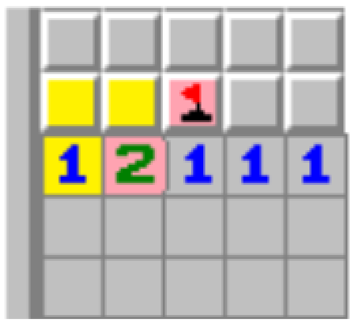
If an **Empty X** tile has exactly X flagged tiles next to it (and we assume the flags are correct), then any other unflipped tiles must be empty. This is implemented similarly to basic bomb finding.

2.2.3 - 1-1 Pattern



Given there are two **Empty 1** tiles next to each other, where one is up against "an edge" (the edge can also be revealed tiles), and the tile above the 1 against the edge is *also* against an edge, then the tile diagonally above the other 1 is **Empty,** and should be flipped. This works in any orientation.

2.2.4 - 1-2 Pattern



Given there is an **Empty 1** next to an **Empty 2**, the tile diagonally above the 2 is a bomb and should be flagged.

2.3 - Ambiguous Situations

According to the minesweeper rules, the rules described above should solve all logically solvable situations. However, there are some situations where there is no certain solution, although there are some probabilistically better choices. However, I was unable to implement any of these with the time I had available, so when there is no logical solution available, the play button chooses randomly (and tells the user that is what it has done!).

3. Reflection

How suitable was Haskell as a language for tackling each phase of the project?
I found Haskell to be great for the model building and the autosolver parts of the project - the pattern matching really suits building a game like this, and especially suits literally matching to a pattern in the game! However, I will admit I found Threepenny a bit lacking in some functions which are standard in web development, even ones which make sense in a pure context.

What was your experience of the software development process (including things like testing and debugging)?
I found the software development process alright. It is quite different to programming in other languages, but to me includes more puzzle solving, which I appreciated. Debugging was easy enough, especially once I was working with UI objects, as print statements can be included in UI functions. Unfortunately, I didn’t get a chance to do many actual proper tests given the time limit, but the simple manual tests I did were fine.

What features of the language proved useful?
My favourite parts of the language were definitely pattern matching - I found them very useful in being able to describe the project as well as helpful in coding. Once I was using library monads, I found them very useful, especially concerning Monad Transformers (mostly liftIO!)

What would I do differently?

If I were doing this project again, I would definitely try to lean more into making my game model a Monad (so that it would be easier to sequence actions), and try to use FRP to make the website code more pleasant.

How did I find the project?
I really enjoyed this project! I had a lot of fun designing the model behind it, and making it look nice. I only wish I had more time to work on it.