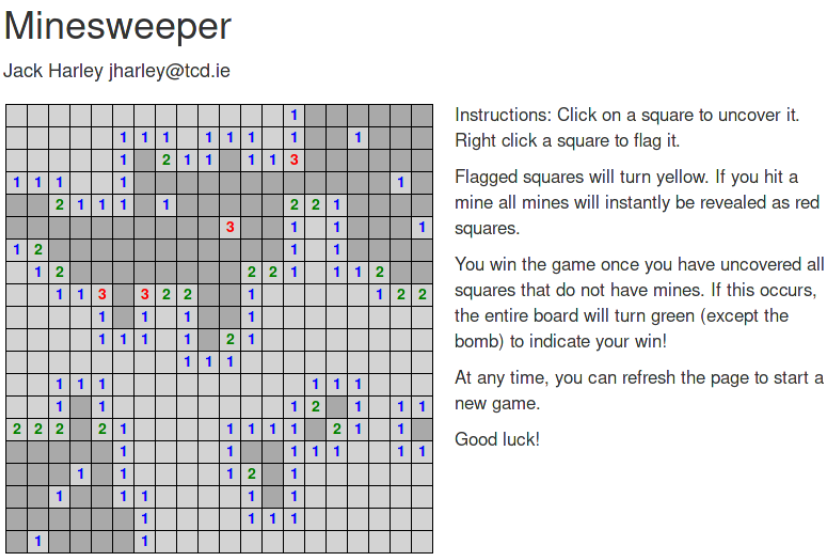
CSU44012 Topics in Functional Programming
Assignment #2
Minesweeper

Jack Harley jharley@tcd.ie — Student No. 16317123

January 2021



Contents

1 Introduction

I have implemented a fully functional Minesweeper game in Haskell with the Threepenny GUI serving the interface.

Stack successfully builds and executes the solution binary, serving the GUI at `http://localhost:8023`.

I attempted to integrate it into an Electron application, so that the interface would launch automatically into an Electron window (embedded Chrome, see `https://www.electronjs.org/`) but unfortunately realized partway through that the effort required to get it working was likely to be disproportionate to the improvement in functionality.

2 Design and Implementation

I will cover a few of the more important functions in some detail in this section. The comments included in the source files should be sufficient to explain the simpler function.

2.1 Basic Minesweeper Model

The model for the game is implemented in Minesweeper.hs.

I modelled the game board as an ADT with 4 fields:

```
1 data Board = Board { size :: Int, mines :: Grid, uncovered :: Grid, flagged :: Grid }
```

size defines the horizontal and vertical length in squares of the game grid (all grids are squares). **mines**, **uncovered** and **flagged** hold a data structure indicating the squares that respectively have mines, have been uncovered by the user and have been flagged by the user.

The Grid type is a 2D list of Booleans with the outer list denoting rows and the inner list denoting columns:

```
1 type Grid = [[Bool]]
```

For example, you could determine whether the 2nd row down, 4th column across has a mine with the following expression: (N.B. rows and columns are 0-indexed)

```
1 (mines !! 1) !! 3
```

And indeed this is how the hasMine function is implemented:

```
1 hasMine :: Board -> Square -> Bool
2 hasMine b (r,c) | validSquare b (r,c) = (mines b !! r) !! c
```

Throughout my implementation, particular squares are referred to with a 2-tuple defining first the row and then the column as 0-indexed integers, with (0,0) being the square at the top left of the board:

```
1 type Square = (Int, Int)
```

2.2 Creating a Game

A fresh game board is initialised with the createBoard function:

```
createBoard :: Int -> Float -> StdGen -> Board
createBoard size mineRatio rng = Board size
                                       (seedGrid rng mineRatio (createGrid False size))
                                       (createGrid False size)
                                       (createGrid False size)
```

The function requires a size (number of squares in both horizontal and vertical directions), a "mine ratio" and a random number generator instance. It then produces a Board type with three initialised grids. The uncovered and flagged grids are initialised with all False values (since the user will not have uncovered or flagged any squares yet).

The mines grid is initialised with all False values, but then the seedGrid function is used to randomly seed mines into the grid by making a random decision with probability of the provided mineRatio for every square on the grid.

For example, with a mine ratio of 0.1, every square will have a one in ten chance of having a mine, and after the decisions have been made for every square, roughly one tenth of the grid will have mines.

seedGrid works by splitting the random number generator repeatedly, one instance for each row of the grid, and then calling seedList on each row. The seeded rows are then joined back together at the end of the recursion. The full source for seedGrid, seedList and seedList' can be found in the appendix and project files.

2.3 Handling Game Moves

2.3.1 Uncover

Uncover is triggered in the UI by left clicking on a square. It triggers the following function:

```
uncover :: Board -> Square -> Board
uncover b (r,c) | not $ validSquare b (r,c) = b
                | isUncovered b (r,c) = b
                | hasMine b (r,c) = let Board s m u f = b
                                      in Board s m (createGrid True s) f
                | otherwise = let Board s m u f = b
                                  (rowsA, row : rowsB) = splitAt r u
                                  (cellsA, _ : cellsB) = splitAt c row
                                  newRow = cellsA ++ True : cellsB
                                  newRows = rowsA ++ newRow : rowsB
                              in uncoverAdjacentsIfSafe (Board s m newRows f) (r,c)
```

The first guard handles cases where the provided Square is not valid (lies outside the edge of the Board), in this case, the Board is returned unchanged.

The second guard handles cases where the provided Square is already uncovered, and again the board is returned unchanged.

The third guard handles cases where a user clicks on a mine. In this case the game has ended and the player has lost, therefore the function simply replaces the uncovered Grid with an all True Grid. The user will therefore immediately see the entire grid, including all of the mines.

The final guard handles normal cases where the Square clicked is safe. It reconstructs the uncovered Grid, replacing the Square at (r,c) with a True status. It then also calls the uncoverAdjacentsIfSafe function on the modified Board:

```
uncoverAdjacentsIfSafe :: Board -> Square -> Board
uncoverAdjacentsIfSafe b (r,c) | adjacentMines b (r,c) == 0 = uncoverAll b $ adjacentSquares (r,c)
                               | otherwise = b
```

uncoverAdjacentsIfSafe checks if the newly uncovered Square has 0 adjacent mines, and if so, uncovers all of them. This can trigger recursion where large parts of the Board will be uncovered.

2.3.2 Flag

Flag is triggered in the UI by right clicking on a square. It is intended to be used when a user wants to mark a square they think has a mine. It triggers the following function:

```
flag :: Board -> Square -> Board
flag b (r,c) | not $ validSquare b (r,c) = b
             | isUncovered b (r,c) = b
             | isFlagged b (r,c) = b
             | otherwise = let Board s m u f = b
                               (rowsA, row : rowsB) = splitAt r f
                               (cellsA, _ : cellsB) = splitAt c row
                               newRow = cellsA ++ True : cellsB
                               newRows = rowsA ++ newRow : rowsB
                           in Board s m u newRows
```

Flag works similarly to uncover.

If the square is not valid, already uncovered or already flagged then the board is returned unchanged. We could combine these cases into a single guard but I think readability is better with them separately.

Then we use the same procedure to replace the flagged Grid with a new grid, with the right clicked square's status changed to True.

2.4 Rendering the UI

3 Reflection