Documentation
(Minesweeper-o-Matic)

Although I feel like my source files are suitably commented, I believe it's good to explain what I was really trying to do and why I modelled Minesweeper the way I did, in a different text file (this document).
I organised my source code in 4 different files -

1. CustomDataTypes.hs — contains all the custom data types i created
2. Configs.hs — contains all the settings (like number of rows, size of a cell, etc)
3. MinesweeperFuncs.hs — contained the **non-UI** functions used in the program
4. Main.hs — contained the **UI** functions (which usually wrapped the non-UI functions and called them to make things work)

I'll explain them as briefly as I can, in the order above, from simple files to more detailed files

---

CustomDataTypes.hs

This file just contained the data types i created to model minesweeper. This is where it all began.

What attributes does a cell have in Minesweeper?
Well, a Cell can have a Mine, or a number value indicating the number of mines around it. I made a data type 'Value' to indicate what the cell contained (Mine or Num 2, for example)
Secondly, a cell can be revealed, unrevealed, or flagged (a special type of unrevealed cell which we believe contains a Flag). Flagged is important to include as it "stops" the user from revealing that cell, which is good, because that cell probably has a mine!
Last, as a programmer, I need to keep track of my cells, and differentiate them with some sort of "primary key" or a unique ID. The most natural thing I could think of, which would also help me in my design of the rest of the program, was the combination of it's row number and column number. This would be unique for each cell, and this was the third attribute I added to my cell

So, in summary, my cell has a tuple indicating the row and column it was in, a value, indicating what the cell actually contained, and the status of the cell, to indicate whether it was shown, hidden or flagged.

A board could be a 2d list of Cells. But hey, since we have numbers to state which row and column each cell lies in, we don't really need a list of lists! That could complicate things, so I chose to represent a board by just a list of cells.

There were 3 other self-explanatory data types:

1. GameStatus - the current status of the game (ongoing, win or loss)
2. clickMode - I came up with this later. I didn't manage to get right click to work for threepenny, as it was supposed to open up some kind of context menu. So i used left click for revealing as well as flagging mines. It was the clickMode that decided what my left click meant at that point, and that was modified by a button.
3. playerType - the type of player who was playing i.e. Human or AI

That's all you need to know for this file.

Configs.hs

This file contained the "settings" or "constants" used throughout the program.

Honestly, this was one of the latter files I created. I used numbers throughout my program, and this was the result of my "program cleanup". I defined all the "constants" here:

1. rows - Number of rows the minesweeper board must contain
2. cols - Number of columns the minesweeper board must contain
3. mines - Number of mines in the board
4. cellWidth - width of a cell in minesweeper (for the UI)
5. cellHeight - height of a cell in minesweeper (for the UI) (this was initialised to cellWidth so as to keep the cell as a square…I could have just used cellSize, but cellWidth and cellHeight makes the program much more general and adjustable, in case someone wants rectangles)
6. xGap - spacing between horizontally adjacent cells
7. yGap - spacing between vertically adjacent cells
8. resultSpace - I left some space at the bottom of the board, for the "You win/lose" messages
9. revealMsg - (explained below)
10. flagMsg - (explained below)
11. canvasWidth - calculating canvas width based on number of rows, cell dimensions and gaps between cells
12. cellHeight - ditto.

Back to revealMsg and flagMsg.
Recall that in the first file, I had a click mode. I had a button for changing this mode from revealing to flagging, and vice versa. When in reveal mode, the button must say "Change to flag mode". Likewise, when in flag mode, the button must say "Change to reveal mode". This is what revealMsg and flagMsg was, respectively.

Finally, we can move on to the real stuff.

MinesweeperFuncs.hs

This file contained the non-UI functions of the program.

Before I start explaining this, I'd like to make something clear. This file aren't the "basic implementations" or the first few functions I defined (although most of the basic functions reside in this file). This file contains some files I used for the AI as well, which is the last step of the process. I just wanted to make keep my UI functions and non-UI functions in separate files, so I could be clear as to which functions I could use directly, and which functions' results had to be wrapped within UI, or used without the "do" notation.
These functions dealt with the "programatic" part of my solutions, and left the UI alone.

Of course, I'll try to be as brief as possible, while focusing on the higher level logic.

Choosing the Mines:

The first part of this file was about choosing random mines, where I threaded the generators carefully so as to use a new one for every Int generation. Recall that my cell could be identified by the (row, column) tuple. This is exactly what I had to generate. I tried generating a random Integer within certain bounds at first, then a random tuple, and then a list of random tuples, and that was (almost) all I needed!
I say almost, because the tuples had to be unique. Imagine creating 10 mines in one cell, that'd make for a much easier game! This is why I had to keep a track of what was being added into the current list of tuples separately, and before adding in tuples, I'd have to check whether it was a member of the current tuple list. And that's all about generating random mines.

Initialising the Board:

This took me twice as long to think about. Creating the board was easy, I just generated a row of numbered cell data types, and had a higher order function to generate the row n times, where rows = n from the Configs file. Initialising the row, column tuples was easy for this.
Also, obviously, at the start of a game, all cells would be hidden. So I initialised the status of all cells to Hidden.
What about the value? Well, we had the Mines, so we could assign that, and that's what I did while creating the board. However, for the other cells, I needed the mines first, to fill in the values with real number types corresponding to the mines, to indicate how many neighbouring mines there were. I initialised all the non-mines' cells' values to (Num 0) while creating the Board. After initialising the board, I went through the cells, and whenever I found a mine, I incremented the value of all neighbouring non-Mine cells, in the 9-box grid around the mine. That worked well, and initialised my board with values appropriately.
(I used the gridMap function for this, which was like fmap, but for the neighbour cells of the mine instead of the entire board).

Simple Boolean Funcs:
I created functions which informed things about cells, such as isMine (is this cell a mine), isFlagged, isHidden, etc. Quite useless, right? Why didn't I just use pattern matching instead? This helped to make the program simpler, which isn't a great reason. The actual reason was so that I could use these (Cell -> Bool) functions as an argument for the 'filter' function later on, to get all the Hidden, or flagged, or mine cells (using their corresponding methods), with ease!

Grid-based Functions:
As mentioned earlier, gridMap took the board, a Cell (Cell 1), a function (Cell -> Cell), and updated all the neighbouring cells of Cell1 in the board, using that function. I was quite limited because of the function type, and I created this quite early, hoping to use it for most things later on, but it didn't quite work out. So this was used by just the fillBoard function.
gridList however, was very useful. It gave the row, col tuples of the neighbouring cells. These tuples could be out of range. This is why I had another function called inRange, and a function ValidGridList used gridList, and filtered it deepening on whether the tuples were in range or not. getNeighbourCells returned a list of neighbouring cells using validGrid list.

Revealing/Flagging Cells:
Only those Cells which were hidden could be revealed (shown).
Likewise, only those Cells which were hidden could be flagged, and only Cells that were flagged could be unfledged and set to hidden.
There's a point where I want to reveal all the cells in the game, not matter whether they're hidden or flagged. This is at the end of the game! So i Included another method called forceRevealCell, which set the status of the Cell to 'Shown', regardless of what it's status was at first.

Miscellaneous Utility Functions:
'isMember' and 'replaceElem' are self-explanatory.
'findCell' takes the board, a row, col tuple, and returns the cell from the board if it exists.
'safeCellsRevealed' goes through the board and checks if all the non-mine cells are revealed. (This is useful for the endGame detection)
updateGameStatus checked the cell currently clicked (or chosen) and the board. If the cell clicked was flagged, game is ongoing. If the mine was clicked, the player lost. If all the safe cells were revealed, the player wins. Else, the game is ongoing.

AI Logic Functions:

I used these functions for my AI solver. Unless this has to be very long, it's a good idea to talk about these functions in high level.

'getHiddenCorner' returned the unrevealed (hidden) corner cells of the board.

'getNumFlagsAndCells' returned a triple, with the num value of a cell, the amount of neighbouring flagged cells, and a list of the hidden neighbouring cells. With these 3 pieces of information, we could make strong deductions:

1. If the num Value of a cell was equal to the amount of it's neighbouring flagged cells, it meant that all the other neighbouring hidden cells would have to be mine-free (if the flagged cells actually contained mines). So all these hidden neighbouring cells would be safe cells. This corresponded to the 'findSafeCell' and 'findSafeCells' method.
2. The difference between the num value of the cell (which indicating the amount of neighbouring mines) and the neighbouring flagged cells represents the amount of neighbouring mines left to be found and flagged. If this was equal to the amount of hidden neighbouring cells, it meant that all those hidden neighbouring cells would have to contain the rest of the mines! This corresponded to the 'findMine' and 'findMines' functions.

'getNum0Opening' was a function that found a hidden cell which was a neighbour of a cell with a value (Num 0). Since a cell with the value (Num 0) has no mines around it, every hidden neighbouring cell is bound to be a safe choice.

'findHiddenEdge' found a hidden cell that was on the edge of a board

'findHiddenCell' found the first hidden cell it could on the board.

'getAllShownCells' and 'getAllNum0ShownCells' were just helper function, used by the AI functions above, and returned all the cells that were opened, and all the opened cells with the value of (Num 0), respectively.

That's all this file contains. The next files will have GUI methods where I'll present a solid structure of the AI strategy I used.

Main.hs

The program started off with my ‘main’ function (of course), which started the Gui and called ‘uiSetup’. uiSetup was the function where I setup the canvas, buttons and their stylings, and more importantly, events for each of these buttons. First, there would be an empty canvas with two buttons below, prompting the user to choose whether a human should play or an AI. In either case, the board would be filled and the reset button was displayed (which, onClick, could easily restart the game, right from the beginning where we choose the type of player). For an AI player, a play move button would be presented, which would cause the AI to make a move on every press. The human player was supplied with a much-needed change clickMode button, to symbolise whether the human was trying to reveal or flag a cell on the canvas. Since I had measurements of xGap, yGap, cellWidth and cellHeight in my configs files, I could easily map the onCanvas-click coordinates to the cell that was clicked on.
Lastly, I also used some IORefs in my uiSetup function. At first, I would pass them around, but that destroyed the purity of the program, and so I edited my other UI functions so that they returned a UI Board instead of updating the boardRef by themselves. I received these UI wrapped Board and game status values through these functions and updated the references in UI itself, thus conserving the purity of the program as much as I could.
Other than the Board and gameStatus, the other IORefs in this function were:

1. coord - points of mousemove on canvas, to capture which coordinates of the canvas the user clicked on.
2. g - The Standard generator. I used newStdGen instead of getStdGen so that ‘g’ would contain a new generator on clicking the reset button, and not the same StdGen again, which would place the mines in the same position as last time - which in turn would make the reset button a lot less useful.
3. the player type was kept as an IORef, so as to consider whether clicks on the canvas had to be reacted to or not. I reacted to canvas clicks only if the player was a Human AND if the gameStatus was still Ongoing.
4. clickMode - which is useful for the human player as described earlier (Flag or reveal cells on click)

‘createBoardUI’ and ‘createRowUI’ simply created the board UI after the player type was chosen,

‘revealResponse’ was called when a human tried to reveal a cell, and ‘flaggedResponse’ was called when a human tried to flag a cell. They called the programmatic parts of the program in MinesweeperFuncs.hs, and then reflected the change in the UI of the board, if appropriate.

The ‘playAI’ function was where my main logic resided. My AI worked in this order:

1. Search for a Num 0 cell that’s opened, and open a hidden neighbouring cell around it if it exists. This is guaranteed to be a safe move, so this was the move I prioritised first.
2. If no hidden neighbours to Num 0 was found, I tried to find an obvious mine using the findMine function in the MinesweeperFuncs.hs file, where the logic of this function was described earlier. If such a cell was found, I made the AI flag it.
3. If not, I tried to find an obvious safe move next, using the ‘findSafeCell’ function in MinesweeperFuncs.hs, whose logic was also described earlier.

It was beyond this point that my guessing game had begun. If none of these 3 points yielded a cell, there were no obvious safe moves. Guessing:

1. Naturally, a corner cell would have less neighbouring cells, reducing the problem. Consider a corner cell with the value (Num 3). All it’s neighbouring cells would be mines. A cell in the middle of the board with a value of (Num 3) would be far less helpful.
2. If the corner cells were opened, I opened any hidden edge cells, which would have 5 neighbouring cells and opposed to 8, thus potentially making the problem easier.
3. If the above two methods failed, I had no choice but to take the trivial approach of iterating through the board to get the first Hidden cell, and then opening it. I thought of randomly choosing a cell at this point, but what was the point? There was no difference in the provability of success using these two methods, and so I took the easier one of finding the first hidden cell.

Opening a corner and edge cell might make solving minesweeper easier, but could also have it’s own issues.
If a corner cell or edge cell opened has a value of (Num 0), we get to reveal only 3 or 5 cells respectively. However, if we revealed a cell with a value of (Num 0) in the centre of the board, we’d e able to open up 8 cells instantly, which in this case, would make corner and edge cells disadvantageous. Moreover, a corner or edge cell could have a mine, making the AI lose the game, and there’s no way to say where the mines are before the game starts.

Although this guessing approach has it’s limitations, it’s usually only used in the start of the game (to open up the top left corner), and then the guaranteed safe moves seem to work almost all the time after that. These last 3 guessing strategies were only for those rarer situations where there was no safe move available. And after guessing a cell which didn’t contain a mine, there could be a lot more safer moves available. So even though the guessing technique I used was not as sophisticated as it could be, it was only applicable when the first 3 situations didn’t hold (which seemed to be rare), and moreover, would be applied when the first 3 situations didn’t hold, which after revealing a cell could quite possibly make one of the first 3 situations hold again.

After every move, the board and gameStatus (wrapped in the UI Monad) was updated, and on a win or a loss, all the cells were revealed and the game was put to an end.

‘makeMove’ handled the attempt to reveal a cell, ‘manageFlagging’ handled the attempt to flag or unflag a cell, and so on. Most of these functions have already been described at a higher level earlier in the document, although I didn’t name the functions explicitly. The Main.hs file is suitably commented, and explains what each function does, but the purpose of this document was to concentrate on the higher level logic of my code, which I feel like I’ve done adequately.

Conclusion

I do realise that I had focussed a lot on the design, and I’m happy about it.
I also think my safe move AI (first 3 conditions and actions) was quite solid and seemed to work 100% of the time.
Although the guessing part of the AI does make a certain amount of sense, I do wish I had upgraded my guessing game so that it would help me at least 90% of the time, in all minesweeper games. This wasn’t the case unfortunately, because I didn’t think upgrading the guessing part from what I already had would be proportionate to the marginal pay off, in terms of how well the AI performed, and so I spent more time in minimising code duplication, ensuring the design of the program was good so that it was easy to understand, and working on the safe-move parts of the AI to ensure a 100% accuracy rate.

However, my program lacks one design-related aspect which annoys me. I realised that every time I create or fill the board, make a move, or flag a cell, my Board gets updated! There’s this sort of “threading” going on: board -> updatedBoard1 -> updatedBoard2 and so on. This clearly screams to make a Board an instance of a Monad, just like State, and to use the bind operation to make things much easier. Unfortunately, I realised this quite late, and would have to alter my program a lot to make it work, apart from possibly paying the price of destroying the design of my program and making it hard to understand, if I hadn’t managed implemented it well given the time I had left when I realised about this sort of “threading”.

Keeping this aspect aside, I’m happy with the way I modelled my Minesweeper game and designed my AI, although the guessing part did have some scope for improvement, and overall, it was a great and fun learning experience.