# Phase 1

## Modelling the game

### The Board

* The game board is modelled as a 2D plane with an X plane (limit given by bWidth) and a Y plane (limit given by bHeight)
* xy coordinates used to navigate around the board modelled as XYCors type which is just a tuple of Ints
* Board itself implemented as a List of Lists of Cell objects AKA a List of rows of Cell objects
* A cell is a square/tile on the game board and has a state (VisState), contents (Mine or Empty), and position on the board (coords)
* The game board can be created initially as an empty board (there are no mines anywhere) with boardInit or ready to play straight away with generateMinefield
* The random generation of mines is done in generateMines by generating a random number to use as an X coordinate, again for a y coordinate, and pairing them together to give a coordinate for a mine. This continues recursively until the number of mines (quantMines = 60) has been satisfied

### The Game

* Calculating what cells are adjacent and their properties is key to the game running smoothly. Some coordinate geometry revision (credited in the --comments) and logical thinking lead to isAdjacent which is simple but powerful and is used throughout the program
* Revealing and Flagging:
  + Players automatically start in revealing mode and can switch to flagging or questioning by clicking their respective buttons at any point. A flagged cell is not “locked” once flagged and can still be revealed to allow players to correct any mistakes that they catch. The same goes for questioned cells and vice versa.
* Game Won:
  + In order to be complete, all the empty(mineless) cells in a row must be visible. We don’t need all the mines to be flagged to win so we don’t waste time checking those. The game is still “clickable” after this point and *losable* (in that the game lost message will appear) if the player reveals a previously flagged mine.
* End Game:
  + The program checks row by row for any game-ending cells (visible mines) and returns true if it finds even 1, causing a game lost message to be displayed. The game is still “clickable” after this point but not winnable. I left it this way mainly out of curiosity so players can uncover any cells they were questioning to see if their suspicions were correct.

# Phase 2

## The Bot

### Unambiguously safe moves

* These moves are only possible when there is enough visible cells on the board so that at least one cell has either…
  + as many hidden adjacents as it does mine-having adjacents, meaning that those hidden adjacents must contain mines. At this point the unambiguous safe move is to flag those cells one at a time. => Implemented in getDefiniteMines

OR

* + as many visible adjacents as it does mine-having adjacents, meaning that the mine quota has been satisfied and the hidden adjacents must all be empty. At this point the unambiguous safe move is to reveal those definitely empty cells one at a time. => Implemented in getDefiniteEmpties
* The bot first moves to identify and flag unflagged definite mines until that is no longer possible because definite mines become undeterminable without revealing more cells. At that point the bot looks to identify cells that are definitely empty that are adjacent to the definite mine cells it just flagged. If there are no more definitely empties then we move into ambiguous territory…

### Ambiguous least dangerous moves

* When the bot can no longer identify definite mines or empties given what is currently visible, it must start revealing cells so that there are more visible cells to consider when identifying definites. Choosing which cell to reveal is done by checking each cell’s probability for having a mine based on the numbers in the cells that surround it. I called this *the number of mines in the adjacents’ adjacents* (=> Implemented by numAdjAdj). Think of it as adjacents squared or if the immediately adjacents are 1 degree of separation then here we are looking at 2 degrees. We get the sum of *the number of mines in the adjacents’ adjacents* and reveal whichever one gives us the lowest sum value. This is not perfect and the game can still be lost but, while possible, it is very unlikely that this bot will end the game.

#### Special Case: The beginning of the game

* The beginning of the game has no visible cells, so the bot is in total ambiguity and will always choose the centre (centre-right in reality because bWidth is even) cell to reveal because statistically that gives the best chance for a large clearing. If that move does not aid in identifying definites, the bot will continue moving around diagonally, avoiding areas with high numbers of adjacent mines until it can happily determine some definites.
* Using the bot at the beginning of the game is the most likely case where the bot will cause an EndGame scenario because of the sharp ambiguity.

# Reflection

The process of designing the program was frustrating because I still automatically try to solve problems in an object-oriented way rather than functionally. In many ways I did solve problems with heavy influences from OO programming; I created new *classes of objects* using type and data for Cell, VisState, etc. I just couldn’t connect things as I would’ve in OO programming and that meant I had to think of a lot of workarounds including dealing with IO, monads, and transformers. In general I don’t think Haskell is best suited for this but there are pros and cons:

One of my favourite things in Haskell is being able to pass functions around as arguments and currying (although I do hate currying for readability purposes). The lazy evaluation is also a positive in that there are so many long lists and lists of lists of lists in my implementation that it is highly efficient and the user playing the game is unaffected by the length of these lists as they grow.

The cons are the inflexibility of the type system when we start building up and combining things. For example I spent an entire day infuriated with myself trying to fix an issue of converting Event (Int, Int) to the expected type Event Cell. It should’ve been as simple as applying getCellAt which takes cell coordinates as (Int, Int) and returns the cell at those coordinates but it was extremely difficult and complex only because of the fact the Threepenny Events were involved. At one hopeless point I committed to git with the message “Seriously considering giving up and dropping out” and that was no word of a lie.

The software development process itself I also found atrocious. Testing and debugging in particular are near impossible using VSC, even with a few Haskell extensions installed. There is no proper Haskell debugger with a nice interface, that I could find, that is similar to those available for Java or C++. Even to just be able to step through the program would’ve exponentially improved my experience. Printing the values of things at different points is also terrifying because you have to start thinking about IO and monads and “deriving Show” when you’re already at a complete loss with your code and exhausted. My pet peeve with Haskell is really it’s readability, and that is not the fault of the language but more of the online programmers who use single letter names for things and inconsistent naming conventions.

The most useful part of the language I found, and I have to say it is a redeeming quality is that in the type error messages, stack/ghc tells you what the expected type was and what the type it found was. That was my saving grace for many things.

# Final Note

Overall I am genuinely sorry to say that I did not enjoy this project. I think it would’ve taken me half the time if I had done it in Java, C++, or even Javascript and that made all the error solving in Haskell just that more sour because I kept dreaming of how much easier it would be for me to do in an OO language that I am way more comfortable with. That is not to say I hate Haskell though by any means. I genuinely find it fascinating and I have loved being able to study and learn about something in programming outside of OO. The theory and possibilities and advantages that Lazy evaluation give are very cool. But that is all in theory, in practice (meaning when I practice it) I find myself feeling utterly incapable and drifting towards hate. To end this on a positive note though I will say that the feeling of relief when you finally get a working program after days of errors is unmatched.