Project idea

The idea for this project was to create a simple version of an auto battler like Riot Games Team Fight Tactics by using Haskell and functional programming. The plan was to have different types of pieces the player could pick up, and then use to fight against predetermined boards of enemy pieces. The player would be able to get more pieces in between combats to help strengthen their board. After a series of battles the player would either be victorious or defeated.

What should the game offer?

The game lets you choose between three different types of pieces, the Assassin, Skirmisher or Defender which have different strength when it comes to offense and defence. The Assassin deals a lot of damage but does not handle much punishment. The Defender is opposite, offering lots of health, but not so much damage, while the Skirmisher overs a balance between offense and defence. The game consists of three phases, Shopping, Preparation and Battle. In the Shopping phase the player gets to choose to either add an Assassin, Skirmisher or Defender to their collection, then in Preparation they can choose where on the board to place their pieces. After that is done, we move to battle, where the players pieces face off against a board of enemy pieces. If the player loses, they are sent back to Shopping and must try again.

How much does the program relate to the original proposal?

Due to some issues around notation and impact the Game Loop which runs the game has been broken. I don't know when this occurred and had sadly not pushed changes to git. The game was in a running state where the player could choose a piece, put it out on the board and start combat. The board could be displayed with the accurate placement of the piece, as well as enemy pieces. Sadly, movement and attacking seems to need some more work. Due to the issue, I mentioned at the start the game is not playable unless I somehow manage to fix it before the due date.

Functional techniques and how they are applied.

In Board.hs I use algebraic data types like PieceType to define different aspects of what pieces we have, their stats, how the board is structured etc. Throughout the project we use pattern matching, high-order functions, and list comprehension to reach different goals, like destructure and extract information from our algebraic types, manipulate lists or change them. We have immutable data structures, instead of functions changing our data structures functions create new ones with desired changes. In setsquare and placePiece we take their arguments in curried form to allow for practical application. These functions are also composed of other functions which allows for reuse of code and easier reasoning about their functionality. In some cases, we have pure functions which have no side effects and their output is determined only by their input. We use recursion like in the findClostestEnemyHelper function which allows us to find all adjacent squares until we find an enemy. We use the state monad to manage the game state. This allows the code to track the gamestate in a more functional way, and allows functions to access and modify a state without using mutable variables.

How suitable where the techniques to their purpose?

I would say each technique had their place in the program. It helped keep the number of lines minimal and helped with readability. State monads helped a lot with avoiding unnecessary headaches having to deal with creating a state myself.

Functionality and testing

As stated earlier the functionality of the game loop broke for some reason a few hours before the due date, as I probably changed something. As of half an hour before the due date it is not fixed. Individual parts of the program should be able to be tested using stack ghci, and giving the function you want to test the correct input, for example you can test the displayGameBoard function by first defining board as initBoard, and then running displayGameBoard and you should be able to see the board.