1. (10) How the game is displayed? Trace `showBoard` (use moves `[(One, TL), (Two, TR)]`), explain uses of Just/Nothing, explain how/when showBoard gets called, etc.

showBoard takes a list of "Move" objects and returns a string composed of the rows of a board, separated by newlines. The first, third, fifth, and final lines of the board are static decoration. The second, fourth, and sixth lines are dynamicaly generated based on the contents of each of the cells. Each cell is displayed by a call to "showCell" with the first argument as the cell's position and the second as the list of moves passed into showBoard. ShowCell takes the position and the list and passes them to findPlayer, which returns either Nothing, or the player that is in the specified position, depending on whether the position is filled. The type signature for findPlayer specifies "Maybe Player," which means the function will return either a "Nothing" or a "Just Player" (Player encapsulated by Just) object. The "otherwise" clause in findPlayer's guard does just that. Back in showCell, a case statement returns a space, "X" or "O", depending on which "Maybe Player" object it receives from findPlayer. This string is then appended to the string being formed in showBoard, and once all the showCell calls have been completed and the board string has been formed, is returned from showBoard to be printed by another function.

A step-by-step trace of program flow for the call "showBoard [(One, TL), (Two, TR)]" follows:

1. moves = [(One, TL), (Two, TR)]
2. starts to form unnamed string: "+---+---+---+\n"
3. appends to string: "| ", calls showCell TL moves
4. in showCell: case (findPlayer moves TL)
5. in findPlayer: check each tuple in the moves list, if the second element of the tuple is "TL", append it to a list.
6. Assign "move" to the list containing all moves matching "TL". The value of move is now [(One, TL)].
7. Check if move is null. It's not; move on.
8. Return the first element of the firt tuple in "move": "Just One".
9. Since findPlayer returned "Just One", showCell returns "X".
10. showBoard appends the "X" to its string, then adds a " | ", and calls showCell again this time passing "TM moves".
11. Another case statement and call to findPlayer, which this time returns Nothing, as "TM" is not in "moves".
12. showCell returns a space, which is appended to the showBoard string.
13. showBoard appends another " | ", and calls "showCell TR moves", appending the return value "O".
14. The process of calling showCell and appending characters continues, with the rest of the showCell calls returning spaces.
15. showCell completes and returns a string representing the current state of the board.

2. (2) Explain how we start with a new game

We start with a new game because main calls playGame passing in the return value of a call to newGame, which is always the same: an "Unfinished []".

3. (10) Explain how the program accepts/validates a player's move. How does it show the list of valid moves?

The program accepts and validates a player's move in the getPosition function. This function attempts to read a line of input from the user, and uses a case statement to decipher the input. The case statement checks the input against “Left (SomeException e)” to handle input exceptions, then prints an error message and recursively calls getPosition if an exception exists. If there was no exception, it checks the input against “Right rawPosStr,” then passes the new value of rawPosStr into “reads”, which returns a list of tuples representing the matched and unmatched portions of the input. (Note: Left and Right are defined by the exception handling module Control.Exception. They encapsulate values, which is necessary to ensure that functions always receive the return type they are expecting, something that is vital in a functional programming language.) This list is then checked in a nested case statement, which compares it agains the empty list “[]”, printing an error and calling itself recursively if this matches, and “[(goodPos,_)]”, returning goodPos if this matches. This makes sure that reads returned the expected tuple, meaning there was a good position specified. The function prints the valid moves by calling printValidMoves, which itself calls validMoves, a function that simply returns a list (range) of positions from minBound to maxBound.

4. (18) Explain how `move` alters the game state.

Move accepts an “Unfinished” game and a position as input, and returns a “Maybe Game” (a Nothing or a Just Game.) It is called by playGame, which assigns its return value to “nextGame”. The move function does not alter global or member variables; it is true to the style of functional programming, accepting input, performing operations, and returning results. Internally, move returns “Nothing” if the move it is passed is invalid, a “Just Finished” game if the isFinished function determines that the provided move completes the game, or a “Just Unfinished” game otherwise.

5. (6) Explain the case statement in `playGame`.

The case statement in playGame compares the value of nextGame to “Just goodGame” and “Nothing” to decide whether to run “do invalidMove” or another case statement. This allows the function to act sanely even if goodGame is not a game, e.g if the last call to move returned a Nothing. The second case statement compares “goodGame” to each of its two possible values, “Left unfinishedGame” and “Right finishedGame”. If it matches unfinishedGame, it recursively calls playGame until the game is finished; else, it returns finishedGame.

6. (2) Code does not appear to stop when the game is a draw. What would you need to modify? (you don't have to write code, just think at a high level where the problem is).

To make the game stop when there is a draw, you would have to add a check to isFinished that would return True if all the positions contained a Player, but none of the winningPatterns cases were matched.