Eric Sheeder
Steve Hamrick

1. showBoard is given the list of moves that have been made so far in the game. For each cell of the board, it calls showCell (for a total of 9 times). showCell returns either an empty space, “X”, or “O”. showCell works by calling another function, findPlayer, on the list of moves that have been made, along with which cell it is associated with (TL, TM, etc.). findPlayer is a maybe type: it returns Nothing if no one has made a move in the cell and returns just Player if someone has made a move in the cell.

   To determine what it should return, findPlayer filters the list based on the second value of the tuples in moves, which is the Position. If the Position exists in the list of moves, then findPlayer returns the head of the (Player, Position) tuple. If the Position does not exist in the list (filter returns null), then it returns Nothing. showCell then uses this to determine whether it should give showBoard a blank space, “X”, or “O”.

   showBoard gets called sneakily in main through the use of the line that says “print game”. When print is called, it calls the show method for the given object. In lines 51-54, the show method is defined for a Finished or Unfinished board by calling the showBoard method on the list of moves.

2. To start a new game, just call the main function. This initializes an empty list of moves, stores it in the game variable, and starts playing the game.

3. playGame accepts the player’s move. getPosition determines is the position is valid or not. If not, then it will tell the player to re-enter a valid position while giving them a list of possible positions. playGame then tries to apply the given move on the current game by using the move function. move is a Maybe Game type. If move returns a valid game with the new move on it, playGame will continue playing the game with the new board state (if the game is still an unfinishedGame) or return the final board state (if the game is a finishedGame). If move returns. If move returns nothing, then playGame will tell the player they made an invalid move and then call itself again on the board state before the move was made.

   To show a list of valid moves, the program uses [minBound..maxBound] of the Position data. Since Position is bounded, minBound is the first value (TL), maxBound is the highest value (BR), and [minBound..maxBound] returns a list of values between the first and last value, inclusively. Printing this list shows all of the valid moves.

4. First, move checks if the game is empty. If it is, it returns an Unfinished game (using the Left type of Game) with a single tuple of (One, Position), where position is where player one made their move.

If the game is not empty, then moves checks to see if the move is valid. If it isn't, it returns Nothing. If it is, it determines whose turn it is it using the whoseTurn function, and creates a list with the new move prepended to the list of moves. If the list is empty, whoseTurn returns One. Otherwise, it looks at the first tuple of the list and returns Two if that player is One or One if that player is Two.

move then calls the isFinished function on the new list to see if the game is finished. isFinished takes an Unfinished game and returns a Boolean on whether the game is finished or not. To do this, it looks at the first tuple in the list (which is the player who made the most recent move), gets a list of all of the moves made by that person using filter, and then uses map to get a list of all moves made by that person.

Once ifFinished has this list, it looks at the list of winningPatterns: for each list in winningPatterns, it removes all of the moves that the player has made that are in that list. At the end of this, it checks the lists in winningPatterns for any empty lists. If an empty list exists, it means that the player made all of the moves in that pattern, and thus won. If the game is considered finished, then the move function returns a Finished game (using the Right type of Game).

If the game is not considered finished, then the move function returns an Unfinished game with the latest move now in the list.

5. The first case statement in playGame first checks the type that move returned. If move returned Just Game, then it goes to a second case statement. If move returned Nothing, then invalidMove is called instead. For the second case statement, it checks to see if Game is the Left Game type, which is Unfinished. If it is unfinished, it continues playing the game with this game board. If the type is a Right Finished Game, then it returns this game to main, where it is printed.

6. You could add another guard in the move function to see if the game has 9 moves in it, in which case you would return a finishedGame. You would also need to change the whoWon function. To do this, you could have whoWon call isFinished to see if somebody actually won the game. If isFinished returns true, you would return as normal. If it returns false, you would return "nobody."