Program Trace

- “main” is executed. It prints a welcome message and calls “play”, passing in a list of characters representing the board.
- “play” accepts the list of characters and assigns its value to the variable “board”. It then prints the board by passing the return value of “showBoard board” to putStrLn. At this point, board = “012345678”
- “showBoard” accepts a “Board” type (simply an alias for [Char], or String) and returns a String. It combines characters representing pieces of the board with the contents of the board list (each element in the list is accessed by index using the !! operator) to create an ASCII representation of the board, each row separated by newline characters. It then returns this string to the caller.
- Next, “play” calls “validMoves board”, which in turn passes board to “winner”.
- “winner” uses a series of guards to check whether the board string matches any winning scenario, and returns a character. Since the board is still in its initial state, no winner is found, and “ ” is returned.
- The first guard check in validMoves compares the results of “winner board” with “ ”, matching on inequality. Since winner returned “ ”, program flow continues to “otherwise”, and validMoves uses a list comprehension with a filter (a call to “isValid,” which simply checks a few conditions to see if a certain “move” is valid for a given board) to return a list of integers that represent valid moves for the board. In this case, the return value will be a list of all integers in the board: [0,1,2,3,4,5,6,7,8,9].
- Execution returns to “play”, which checks the length of the returned list and checks whether it is equal to zero or there is a winner (another call to “winner” compared to “ ”). If either condition is met, it prints a message showing the game's winner. In this case, neither is met, so program flow continues to the “else” clause.
- The program prints the valid moves by issuing another call to “validMoves” and showing the result. It then prints “Play? ” and waits for user input, which is expected to be one of the integers specified in the list of valid moves.
- The program passes the input along with the board, into “playerMove”, which accepts the board and an Int. At this point, if the input was not an integer, the program crashes. If the input was an integer, and is a valid move for the board, playerMove will return a tuple of (Bool, Board), where the first element is a True or False depending on whether the position specified by the input was still a number (not “X” or “O”) on the board, and the second element is the updated board, with the specified position replaced by an “X”, “O”, or left alone, as appropriate. Let's assume we input “0” at the prompt. “playerMove” will return (True, “X12345678”).
- “play” then checks the first element of the tuple. If false, it prints a message telling the user the move was invalid, and recursively calls itself. If true, if prints “Ok”, then checks whether the game has been won by passing the new board returned by “playerMove” into “winner”. If true, it prints the winner and the board. Otherwise, it calls “bestMove”, passing in the new board, then gives the return value of bestMove to “move”, and passes the resulting Board in to a recursive call to “play”.
- Inside “bestMove”, the board is passed to scoreMoves. scoreMoves calls evaluateBoardMax and generates a list of (Int,Int) tuples that represent a single move and that move's associated score.
- evaluateBoardMin and evaluateBoardMax use what seem to be needlessly complex foldr statements to find the minimum and maximum values in a list, when they could just use something like “minimum scores” instead. I don't know why.
- At this point, I got lost in the mutual recursion and moved on to other homework.