The code for a game board and some functions related to the game. The **Board** data type seems to represent a game board as a 2D list of **Player** values, with a pair of **Row** and **Column** values as metadata. The **Player** data type represents the players in the game, which seem to be named **Yara** and **Kamal**. The functions provided include **newBoard**, which creates a new game board with all spaces empty, **isDraw**, which checks if the board is in a draw state, **showBoard**, which displays the board in a human-readable form, **isLegalMove**, which checks if a given move is valid, **makeMove**, which modifies the board to reflect a given move, **winner**, which returns the winning player if the game is over and a player has won, and **playerTurn**, which returns the player whose turn it is. There are also several helper functions for checking for wins in different directions.

**play** is the main function that handles the gameplay. It has a loop that continues until either player Yara or Kamal wins or the game is a draw. If the game is over, it prints the appropriate message. If the game is not over, it reads the next column for the current player to place their piece, makes sure the move is legal, updates the board if the move is legal, and then displays the updated board. The function **checkMove** is used to check if a move is legal and **readInteger** reads in the column number from the user. The **main** function initializes a new empty board and starts the game by calling **play**.

The **Board** data type is defined as a constructor with two fields: a 2D list of **Player** values, and a tuple of **Row** and **Column** values.

The **Player** type is defined as a sum type with three constructors: **Yara**, **Kamal**, and **None**.

There are several functions defined for this game:

* **newBoard** creates a new board with the given number of rows and columns and fills it with **None** values.
* **isDraw** checks if the board is in a draw state (all cells are filled and there is no winner).
* **showBoard** displays the board in a user-friendly way.
* **isLegalMove** checks if a given move is legal (the cell is not already filled).
* **makeMove** allows a player to make a move on the board.
* **winner** checks if either player has won the game.
* **playerTurn** alternates between the two players.

There are also several helper functions defined for checking the different win conditions:

* **checkColumn** checks if a given column is filled with the same player's value.
* **checkVertical** checks all columns for a win condition.
* **playerSwapping** alternates between **Yara** and **Kamal**.
* **getItem** gets the value at a given row and column of the board.
* **rightCheckDiagonalsItem** and **rightCheckDiagonalsIter** check for a win condition on the board's right-leaning diagonals.
* **leftCheckDiagonalsItem** and **leftCheckDiagonalsIter** check for a win condition on the board's left-leaning diagonals.
* **fillRow** fills a given row of the board with the given player's value.
* **fillColumn** fills a given column of the board with the given player's value.

**Board.hs**

**module Board(**

**Row,**

**Column,**

**Player(..),**

**Board,**

**newBoard,**

**isDraw,**

**showBoard,**

**isLegalMove,**

**makeMove,**

**winner,**

**playerTurn,**

**)**

**------------------------------------------------------------------------------------**

**where**

**import Data.List**

This is the module declaration. It defines the names that are exposed by the module and the names that are imported from other modules. In this case, the module exports the types Row, Column, and Player, as well as the data type Board, and the functions newBoard, isDraw, showBoard, isLegalMove, makeMove, winner, and playerTurn. It also imports the Data.List module, which provides various functions for working with lists.

This is the header of the **Board** module, which exports several types and functions related to a game board and the game of Connect Four.

**type Column = Int**

**type Row = Int**

**data Player = Yara | Kamal | None deriving (Eq, Show)**

**data Board = Con [[Player]] (Row, Column) deriving (Eq)**

These lines define the types and data structures used in the module. The **Column** and **Row** types are just synonyms for **Int**, representing a column or row index on the game board. The **Player** data type is a simple enumeration with three possible values: **Yara**, **Kamal**, and **None**. The **Board** data type is a custom data type that represents a game board as a 2D list of **Player** values, with a pair of **Row** and **Column** values as metadata. The **deriving (Eq)** and **deriving (Eq, Show)** clauses at the end of each data type definition allow the values of these types to be compared for equality and displayed as strings, respectively.

These lines define the **Column** and **Row** types as synonyms for **Int**. This means that a **Column** value is simply an **Int** value and can be used interchangeably with one. Similarly, a **Row** value is also an **Int** value.

This line defines the **Player** data type, which represents the players in the game. It has three possible values: **Yara**, **Kamal**, and **None**. The **deriving (Eq, Show)** clause allows the **Player** type to be compared for equality (using the **==** operator) and converted to a string representation (using the **show** function).

This line defines the **Board** data type, which represents a game board. It is defined as a **Con** constructor that takes a two-dimensional list of **Player** values and a tuple of **Row** and **Column** values representing the size of the board. The **deriving (Eq)** clause allows the **Board** type to be compared for equality (using the **==** operator).

**newBoard:: (Row, Column) -> Board**

**newBoard (r, c) = Con (replicate c []) (r, c)**

This is the **newBoard** function, which creates a new game board with the given number of rows and columns. It does this by using the **replicate** function to create a list of **c** empty lists, which will represent the columns of the board, and then wraps this list in a **Board** value with the given **r** and **c** values as metadata.

This is the **newBoard** function, which takes a tuple of **Row** and **Column** values representing the size of the board and returns a new, empty **Board** value of that size. It does this by using the **replicate** function to create a list of **c** empty lists (i.e., a list of **c** empty rows) and then using the **Con** constructor to create a new **Board** value with this list and the given size tuple.

**checkColumn :: [Player] -> Player -> Bool**

**checkColumn [] player = False**

**checkColumn (x:y:z:q:ls) player | x==y && x==z && x ==q && x == player = True**

**| otherwise = checkColumn (y:z:q:ls) player**

**checkColumn (x:xs) player = False**

This is the **checkColumn** function, which checks if the given player has won in a vertical direction. It does this by taking a list of **Player** values, which represents a column on the game board, and a **Player** value representing the player to check for. It then checks if there are four consecutive occurrences of the player's value in the list. If there are, it returns **True**, otherwise it returns **False**.

This is the **checkColumn** function, which takes a list of **Player** values representing a column on the game board and a **Player** value representing the player to check for, and returns **True** if the player has four in a row in that column, **False** otherwise. It does this by pattern matching on the input list. If the list is empty, it returns **False**. If the list has at least four elements, it checks if all four elements are equal to the player and returns **True** if they are, **False** otherwise. If the list has fewer than four elements, it returns **False**.

**checkVertical :: Board -> Player -> Bool**

**checkVertical (Con [] (r, c)) player = False**

**checkVertical (Con (x:xs) (r, c)) player = (checkColumn x player) || (checkVertical (Con xs (r, c)) player)**

This is the **checkVertical** function, which checks if the given player has won in a vertical direction on the given game board. It does this by recursively checking each column on the board using the **checkColumn** function defined earlier. If **checkColumn** returns **True** for any column, then **checkVertical** returns **True**, otherwise it returns **False**.

This is the **checkVertical** function, which takes a **Board** value and a **Player** value as input and returns **True** if the player has won in the vertical direction (i.e., has four in a row in any column), **False** otherwise. It does this by pattern matching on the board value. If the board value is an empty board (i.e., has no columns), it returns **False**. If the board value has at least one column, it checks if the player has four in a row in the first column using the **checkColumn** function, and if not, it recursively checks the rest of the board using the **checkVertical** function.

**playerSwaping :: Player -> Player**

**playerSwaping player = if player == Yara then Kamal else Yara**

This is the **playerSwapping** function, which swaps between the two players **Yara** and **Kamal**. It does this by taking a **Player** value as input and returning the other player if the input is **Yara**, or **Yara** if the input is **Kamal**. If the input is **None**, it returns **None**.

**getItem :: Board -> Int -> Int -> Player -> Player**

**getItem (Con xs (r, c)) a b player | b < length col = col !! b**

**| otherwise = player**

**where**

**col | a < length xs = xs!!a**

**| otherwise = []**

This is the **getItem** function, which gets the **Player** value at a given position on the game board. It does this by taking a **Board** value, and two **Int** values representing the row and column indices to get the value from. It also takes a **Player** value as a default return value in case the indices are out of bounds. It then gets the column at the given index and gets the element at the given row index from the column. If either of these indices are out of bounds, it returns the default **Player** value.

This is the **getItem** function, which takes a **Board** value, two **Int** values representing a row and column index on the board, and a **Player** value representing a default value to return if the index is out of bounds. It returns the **Player** value at the specified row and column index on the board, or the default value if the index is out of bounds.

It does this by pattern matching on the **Board** value to extract the two-dimensional list of **Player** values and the size tuple, and then using a **where** clause to define a local variable **col** that is the column at the specified index, or an empty list if the index is out of bounds. It then checks if the column index is within bounds of the column list and returns the value at that index if it is, or the default value if it is not.

**rightCheckDiagonalsItem :: Board -> Player -> Int -> Int -> Bool**

**rightCheckDiagonalsItem board player a b = c1 == c2 && c1 == c3 && c1 == c4 && c1 == player**

**where**

**player' = playerSwaping player**

**c1 = getItem board a b player'**

**c2 = getItem board (a+1) (b+1) player'**

**c3 = getItem board (a+2) (b+2) player'**

**c4 = getItem board (a+3) (b+3) player'**

This is the **rightCheckDiagonalsItem** function, which takes a **Board** value, a **Player** value, and two **Int** values representing a row and column index on the board, and returns **True** if the player has four in a row in a diagonal starting at the specified index and going in the right-down direction, **False** otherwise. It does this by using the **getItem** function to get the **Player** value at each of the four indices in the diagonal and then checking if they are all equal to the player. It uses the **playerSwapping** function to get the other player as a default value for the **getItem** function to use if the index is out of bounds.

**rightCheckDiagonalsIter :: Board -> Player -> Int -> Int -> Bool**

**rightCheckDiagonalsIter board@(Con xss (r, c)) player a b | a>c-4 && b > r - 5 = False**

**| a>c-4 = rightCheckDiagonalsIter board player 0 (b+1)**

**| otherwise = rightCheckDiagonalsItem board player a b || rightCheckDiagonalsIter board player (a+1) b**

This is the **rightCheckDiagonalsIter** function, which is a helper function for the **rightCheckDiagonals** function. It takes a **Board** value, a **Player** value, and two **Int** values representing a row and column index on the board and returns **True** if the player has four in a row in any diagonal starting at the specified index and going in the right-down direction, **False** otherwise. It does this by using the **rightCheckDiagonalsItem** function to check if the player has four in a row in the diagonal starting at the current index, and if not, it advances the index by one column and recursively checks the rest of the board using the **rightCheckDiagonalsIter** function. If the index goes out of bounds of the board, it advances to the next row and resets the column index to zero.

**rightCheckDiagonals :: Board -> Player -> Bool**

**rightCheckDiagonals board player = rightCheckDiagonalsIter board player 0 0**

This is the **rightCheckDiagonals** function, which takes a **Board** value and a **Player** value as input and returns **True** if the player has won in the right-down diagonal direction (i.e., has four in a row in any diagonal starting at any index and going in the right-down direction), **False** otherwise. It does this by calling the **rightCheckDiagonalsIter** function with the board, player, and indices of the top-left corner of the board. The **rightCheckDiagonalsIter** function will then check all diagonals starting at that index and going in the right-down direction, and if any of them has four in a row for the player, it will return **True**. If none of them do, it will return **False**.

**leftCheckDiagonalsItem :: Board -> Player -> Int -> Int -> Bool**

**leftCheckDiagonalsItem board player a b = c1 == c2 && c1 == c3 && c1 == c4 && c1 == player**

**where**

**player' = playerSwaping player**

**c1 = getItem board a b player'**

**c2 = getItem board (a-1) (b+1) player'**

**c3 = getItem board (a-2) (b+2) player'**

**c4 = getItem board (a-3) (b+3) player'**

This is the **leftCheckDiagonalsItem** function, which takes a **Board** value, a **Player** value, and two **Int** values representing a row and column index on the board, and returns **True** if the player has four in a row in a diagonal starting at the specified index and going in the left-down direction, **False** otherwise. It does this by using the **getItem** function to get the **Player** value at each of the four indices in the diagonal and then checking if they are all equal to the player. It uses the **playerSwapping** function to get the other player as a default value for the **getItem** function to use if the index is out of bounds.

**leftCheckDiagonalsIter :: Board -> Player -> Int -> Int -> Bool**

**leftCheckDiagonalsIter board@(Con xss (r,c)) player a b | a >= c && b < (r-4) = leftCheckDiagonalsIter board player 3 (b+1)**

**| a >= c && b >= (r-4) = False**

**| otherwise = leftCheckDiagonalsItem board player a b || leftCheckDiagonalsIter board player (a+1) b**

This is the **leftCheckDiagonalsIter** function, which is a helper function for the **leftCheckDiagonals** function. It takes a **Board** value, a **Player** value, and two **Int** values representing a row and column index on the board and returns **True** if the player has four in a row in any diagonal starting at the specified index and going in the left-down direction, **False** otherwise. It does this by using the **leftCheckDiagonalsItem** function to check if the player has four in a row in the diagonal starting at the current index, and if not, it advances the index by one column and recursively checks the rest of the board using the **leftCheckDiagonalsIter** function. If the column index goes out of bounds of the board, it resets the column index to three (since we are only interested in diagonals starting at indices with a column index of three or greater) and advances the row index by one. If the row index goes out of bounds of the board, it returns **False**.

**leftCheckDiagonals :: Board -> Player -> Bool**

**leftCheckDiagonals board@(Con xss (r, c)) player | c > 3 = leftCheckDiagonalsIter board player 3 0**

**| otherwise = False**

These functions define a helper function for checking for wins in the left diagonal direction on the game board. The **leftCheckDiagonals** function takes a **Board** value and a **Player** value as input and returns **True** if the player has won in the left diagonal direction, **False** otherwise. It does this by calling the **leftCheckDiagonalsIter** function, which iterates over the rows and columns of the board and checks each possible four-in-a-row sequence using the **leftCheckDiagonalsItem** function. The **leftCheckDiagonalsItem** function takes a **Board** value, a **Player** value, and two **Int** values representing the row and column indices to check, and returns **True** if the player has won with the four values at those indices, **False** otherwise.

This is the **leftCheckDiagonals** function, which takes a **Board** value and a **Player** value as input and returns **True** if the player has won in the left-down diagonal direction (i.e., has four in a row in any diagonal starting at any index with a column index of three or greater and going in the left-down direction), **False** otherwise. It does this by checking if the board has at least four columns, and if so, it calls the **leftCheckDiagonalsIter** function with the board, player, and indices of the top row and column index three. The **leftCheckDiagonalsIter** function will then check all diagonals starting at that index and going in the left-down direction, and if any of them has four in a row for the player, it will return **True**.

**fillRow :: Board -> Int -> Board**

**fillRow board@(Con xss (r, c)) n = if not (n == (-1))**

**then fillRow (makeMove board n None) (n-1)**

**else board**

This is the **fillRow** function, which takes a **Board** value and an **Int** value as input and returns the same **Board** value with the specified column filled with **None** values from top to bottom. It does this by calling the **makeMove** function to place a **None** value in the specified column and then recursively calling itself with the modified **Board** value and the next column index, until the column index goes out of bounds.

**fill :: Board -> Int -> Board**

**fill board@(Con xss (r, c)) 0 = board**

**fill board@(Con xss (r, c)) n = fill (fillRow board 6) (n-1)**

This is the **fill** function, which takes a **Board** value and an **Int** value as input and returns the same **Board** value with the specified number of columns filled with **None** values from top to bottom. It does this by calling the **fillRow** function to fill the rightmost column with **None** values and then recursively calling itself with the modified **Board** value and the next column index, until the specified number of columns have been filled.

**transboard :: Board -> Board**

**transboard board@(Con xss (r,c)) = ps2b ( transpose ( b2ps (fill board 6))) r c**

This is the **transboard** function, which takes a **Board** value as input and returns a new **Board** value with the rows and columns of the input **Board** transposed. It does this by first calling the **fill** function to fill the board with **None** values, then calling the **b2ps** function to convert the **Board** value to a list of lists of **Player** values, then calling the **transpose** function from the **Data.List** module to transpose the rows and columns, and finally calling the `ps2.

**b2ps :: Board -> [[Player]]**

**b2ps (Con xss (r, c)) = xss**

This is the **b2ps** function, which takes a **Board** value as input and returns a list of lists of **Player** values representing the rows of the input **Board** value. It does this by simply extracting the rows from the **Board** value.

**ps2b :: [[Player]] -> Row -> Column -> Board**

**ps2b xss r c = (Con xss (r,c))**

This is the **ps2b** function, which takes a list of lists of **Player** values, a **Row** value, and a **Column** value as input and returns a new **Board** value with the specified number of rows and columns and the rows of the input list of lists of **Player** values. It does this by simply constructing a new **Board** value with the input list of lists of **Player** values and the specified number of rows and columns.

**check :: Board -> Player -> Bool**

**check board player = (leftCheckDiagonals board player) || (rightCheckDiagonals board player) || (checkVertical board player) || (checkVertical (transboard board) player)**

This is the **check** function, which takes a **Board** value and a **Player** value as input and returns a **Bool** value indicating whether the input **Player** has won the game by having four of their pieces in a row either horizontally, vertically, or diagonally on the input **Board**.

The function does this by checking each of the four possible ways that the input **Player** can win: by having four of their pieces in a row horizontally, by having four of their pieces in a row vertically, by having four of their pieces in a row diagonally from top left to bottom right, and by having four of their pieces in a row diagonally from top right to bottom left. It does this by calling the **leftCheckDiagonals**, **rightCheckDiagonals**, **checkVertical**, and **checkVertical . transboard** functions, respectively, and combining the results using the **||** operator, which returns **True** if any of the input values are **True**, and **False** otherwise.

**winner :: Board -> Int**

**winner board | check board Yara = 1**

**| check board Kamal = 2**

**| otherwise = 0**

This is the **winner** function, which takes a **Board** value and a **Player** value as input and returns the **Player** value of the winner, if there is one, or **None** if there is no winner. It does this by calling the **checkVertical**, **checkHorizontal**, **leftCheckDiagonals**, and **rightCheckDiagonals** functions to check if the player has won in any of those four directions, and if so, it returns the player. If the player has not won in any of those directions, it returns **None**.

This is the **winner** function, which takes a **Board** value as input and returns an **Int** value indicating which player, if any, has won the game.

The function does this by checking if either player has won the game by calling the **check** function with each player as the input. If the **check** function returns **True** for player **Yara**, the function returns **1** to indicate that **Yara** has won. If the **check** function returns **True** for player **Kamal**, the function returns **2** to indicate that **Kamal** has won. If neither player has won, the function returns **0** to indicate that the game is still in progress or that it is a draw.

**isDraw:: Board -> Bool**

**isDraw (Con [] z) = True**

**isDraw (Con (x:xs) (r, c)) = length x == r && isDraw (Con xs (r, c))**

This is the **isDraw** function, which takes a **Board** value as input and returns a **Bool** value indicating whether the game represented by the input **Board** value is a draw.

The function does this by checking if all of the rows of the **Board** value are full. It does this by recursively checking each row of the **Board** value until it reaches an empty row or the end of the rows. If it reaches an empty row, the function returns **False** to indicate that the game is not a draw. If it reaches the end of the rows and all of the rows are full, the function returns **True** to indicate that the game is a draw.

**showBoard:: Board -> String**

**showBoard board@(Con xss (r,c)) = (displayBoard board 1) ++ trailer (c-1)**

This is the **showBoard** function, which takes a **Board** value as input and returns a **String** value representing the game board as a printable string.

The function does this by calling the **displayBoard** function to generate the main body of the board and the **trailer** function to generate a string representing the column numbers at the bottom of the board. The **displayBoard** function takes a **Board** value and an integer representing the row number to start at as input, and it returns a **String** value representing the rows of the board starting from the given row number. The **trailer** function takes an integer representing the number of columns in the board as input and returns a **String** value representing the column numbers at the bottom of the board.

**trailer :: Int -> String**

**trailer n | n == -1 = "+\n"**

**| otherwise = "+-" ++ trailer (n-1) ++ " " ++ (show n)**

This is the **trailer** function, which takes an integer representing the number of columns in the board as input and returns a **String** value representing the column numbers at the bottom of the board.

The function does this by recursively generating a string that consists of alternating **+-** characters and the column numbers, starting from the rightmost column and working towards the leftmost column. When the column number reaches **-1**, the function returns a **+** character to indicate the end of the row.

d**isplayCell :: [Player] -> Int -> String**

**displayCell [] player = " |"**

**displayCell board 1 | head board ==Yara = "Y|"**

**| otherwise = "K|"**

**displayCell board player | player > (length board) = " |"**

**| otherwise = (displayCell (tail board) (player-1))**

This is the **displayCell** function, which takes a list of **Player** values representing a row of the board and an integer representing the column number of the cell to display as input, and it returns a **String** value representing the given cell.

The function does this by recursively generating a string that consists of a **|** character followed by the player symbol (either **Y** or **K**) or a space character, depending on the value of the player at the given column in the row. If the column number is greater than the length of the row, the function returns a **|** character followed by a space character.

**displayRow :: [[Player]] -> Int -> String**

**displayRow [x] player = displayCell x player**

**displayRow (x:xs) player = displayCell x player ++ displayRow xs player**

This is the **displayRow** function, which takes a list of lists of **Player** values representing the rows of the board and an integer representing the row number of the row to display as input, and it returns a **String** value representing the given row.

The function does this by calling the **displayCell** function to generate the cells of the row and concatenating the results together.

**displayBoard :: Board -> Int -> String**

**displayBoard (Con xss (r, c)) n | n >= (c-1) = "|" ++ displayRow xss n ++ "\n"**

**| otherwise = displayBoard (Con xss (r, c)) (n+1) ++ "|" ++ displayRow xss n ++ "\n"**

This function takes a **Board** value and an integer representing the row number of the row to display as input, and it returns a **String** value representing the entire board.

The function does this by recursively calling itself to generate the rows of the board and concatenating the results together. It stops when it has generated all the rows of the board.

**isLegalMove:: Board -> Column -> Bool**

**isLegalMove (Con [] (r, c)) col = True**

**isLegalMove (Con xss (r, c)) col | col >= c = False**

**| otherwise = (kk < r)**

**where**

**kk = length k**

**k = (xss !! col)**

This function takes a **Board** value and an integer representing a column number as input, and it returns a **Bool** value indicating whether it is legal to make a move in the specified column.

If the board is empty (that is, it has no rows), then any column number is considered a legal move, so the function returns **True**.

Otherwise, if the column number is greater than or equal to the number of columns in the board, then it is not a legal move, and the function returns **False**.

Otherwise, if the column number is valid, the function checks the number of elements in the specified column. If this number is less than the number of rows in the board, then there is room to make a move in the column, and the function returns **True**. Otherwise, the column is full, and the function returns **False**.

**makeMove :: Board -> Column -> Player -> Board**

**makeMove board@(Con xss (r, c)) column player = Con (a ++ b ++ c') (r, c)**

**where a = (firstHalf xss column)**

**b = [((xss !! column) ++ [player])]**

**c' = (secondHalf xss column)**

The **makeMove** function takes in a **Board**, a **Column**, and a **Player**. It returns a new **Board** with the **Player** added to the specified **Column**.

It does this by using pattern matching to break the board into three parts:

* **a** is the first part of the board, up to but not including the column specified
* **b** is a single element list containing the specified column, with the **Player** appended to the end of it
* **c'** is the second part of the board, after the specified column

It then uses the **Con** constructor to create a new **Board** by combining these three parts, and returns it.

**firstHalf:: [[Player]] -> Int -> [[Player]]**

**firstHalf xss n = take n xss**

The **firstHalf** function takes a list of lists (**xss**) and an integer (**n**) as input and returns the first **n** elements of **xss** as a list.

For example, if **xss = [[1,2,3],[4,5,6],[7,8,9]]** and **n = 2**, **firstHalf xss n** will return **[[1,2,3],[4,5,6]]**.

**secondHalf:: [[Player]] -> Int -> [[Player]]**

**secondHalf xss n = drop (n + 1) xss**

The **secondHalf** function takes a list of lists of **Player**s, **xss**, and an integer **n** and returns the list of lists of **Player**s obtained by dropping the first **n + 1** elements of **xss**. For example, if **xss = [[Yara, Yara, Kamal], [Yara, Kamal, Yara], [Yara, Yara, Yara]]** and **n = 1**, then **secondHalf xss n = [[Yara, Yara, Yara]]**.

**playerTurn :: Board -> Player**

**playerTurn (Con board (r, c)) = if rc > yc then Kamal else Yara**

**where**

**rc = length (filter (==Yara) longBoard)**

**yc = length (filter (==Kamal) longBoard)**

**longBoard = concat board**

This code defines a function **playerTurn** that takes in a **Board** and returns the **Player** whose turn it is to make a move. It does this by concatenating the rows of the **Board** into a single list called **longBoard**, then counting the number of occurrences of **Yara** and **Kamal** in **longBoard**. If there are more occurrences of **Yara**, then it is **Kamal**'s turn, otherwise it is **Yara**'s turn.

**Game.hs**

**import Board**

**play :: Board -> IO ()**

This function takes a **Board** as input and returns an **IO ()** action. An **IO ()** action is a computation that may perform I/O and returns no useful result. In this case, the **play** function will perform I/O by printing messages to the console and reading input from the user.

**play board | winner board == 1 = putStrLn "Yara wins!"**

**| winner board == 2 = putStrLn "Kamal wins!"**

**| isDraw board == True = putStrLn "Ta3adol"**

**| otherwise = do**

**c <- readInteger (playerTurn board)**

**ok <- checkMove board c**

**let board1 = if ok then makeMove board c (playerTurn board) else board**

**putStrLn ("\n" ++ (showBoard board1))**

**play board1**

This function is a recursive function that plays the game of Connect Four. The function has an input of **board**, which represents the current state of the game board. The function then checks if either player has won the game, or if the game is a draw. If any of these conditions are met, the function prints out the appropriate message. If none of these conditions are met, the function continues with the game.

First, the function reads in an integer input representing the column that the current player wants to make their move in. It then checks if the move is legal by calling the **checkMove** function. If the move is legal, the function makes the move by calling the **makeMove** function and updating the board to the new board state. If the move is not legal, the board remains unchanged.

The function then prints out the updated board and continues the game by calling itself with the updated board as input. This continues until either player wins, or the game is a draw.

This is the main loop of the game. It uses pattern matching on the result of the **winner** and **isDraw** functions to determine the current state of the game. If either player has won or the game is a draw, it prints the appropriate message and returns. Otherwise, it continues with the next move.

To make a move, the **readInteger** function is used to prompt the current player (determined by the **playerTurn** function) to enter a column number. The **checkMove** function is then called to check whether the chosen move is legal. If it is legal, the **makeMove** function is used to update the board with the new move. Otherwise, the board is left unchanged.

Finally, the updated board is printed using the **showBoard** function, and the **play** function is called recursively to continue the game loop.

**checkMove :: Board -> Column -> IO Bool**

**checkMove b c = do**

**let x = isLegalMove b c**

**putStr (if not x then "mamno3!\n" else "")**

**return (x)**

This function is used to check if a move is valid on the given board. It takes a **Board** and a **Column** as input and returns a boolean value indicating whether or not the move is legal.

The function first checks if the move is legal using the **isLegalMove** function, which checks if the given column is not full. If the move is not legal, it prints an error message "mamno3!". Finally, the function returns the boolean value indicating whether the move is legal or not.

This function takes a **Board** and a **Column** (representing a player's chosen move) as input and returns an **IO Bool** action. It uses the **isLegalMove** function to check whether the move is legal, and if it is not legal it prints "mamno3!" and returns **False**. Otherwise, it returns **True**.

**readInteger :: Player -> IO Int**

**readInteger player = do**

**putStr "\ndoor "**

**putStr (show player)**

**putStrLn ": "**

**x<-getLine**

**return (read x)**

The function **readInteger** takes in a **Player** and returns an **IO Int**. It first prints a string asking the user to enter an integer and then reads in an integer from the user using **getLine**. It then parses this input as an integer using **read** and returns the result wrapped in the **IO** monad.

This function takes a **Player** (either 1 or 2) as input and returns an **IO Int** action. It prompts the player to enter a column number, reads their input from the console, and returns it as an integer.

**main :: IO ()**

**main = do**

**let board = newBoard (6, 7)**

**putStrLn ("\n" ++ (showBoard board))**

**play (board)**

The **main** function is the entry point of the program. When the program is run, this function is called first.

In this case, the **main** function first creates a new board with 6 rows and 7 columns using the **newBoard** function and binds it to the variable **board**. Then it prints the board using **putStrLn** and passes it to the **play** function to start the game.

The **main** function is the entry point of the program. It creates a new board using the **newBoard** function and prints it out using the **showBoard** function. Then it starts the game loop with the **play** function.

Summary:

**Board.hs**

This code defines a module **Board** that provides functions for creating and manipulating a board game.

The **Row** and **Column** type synonyms define the type for row and column indices in the board, respectively.

The **Player** data type defines the possible values for a player in the game: **Yara**, **Kamal**, or **None**.

The **Board** data type represents the game board as a list of lists of **Player** values. It also includes the dimensions of the board as a tuple of **Row** and **Column** values.

The **newBoard** function takes a tuple of **Row** and **Column** values and returns a new **Board** with all positions initialized to **None**.

The **checkColumn** function takes a list of **Player** values and a **Player** value and returns **True** if four consecutive values in the list are equal to the given **Player** value, and **False** otherwise.

The **checkVertical** function takes a **Board** and a **Player** value and returns **True** if there are four consecutive **Player** values in any column of the board, and **False** otherwise.

The **playerSwapping** function takes a **Player** value and returns the other **Player** value (i.e., **Yara** if the input is **Kamal**, and vice versa).

The **getItem** function takes a **Board**, row and column indices, and a **Player** value and returns the **Player** value at the specified position in the board, or the default **Player** value if the indices are out of bounds.

The **rightCheckDiagonalsItem** function takes a **Board**, a **Player** value, and row and column indices and returns **True** if the **Player** values in the four positions diagonally to the right of the given position are equal to the given **Player** value, and **False** otherwise.

The **rightCheckDiagonalsIter** function is a helper function for **rightCheckDiagonals**. It takes a **Board**, a **Player** value, and row and column indices and returns **True** if there is a sequence of four diagonal **Player** values to the right of the given position, and **False** otherwise. It iterates through the rows and columns of the board to check all possible diagonal sequences.

The **rightCheckDiagonals** function takes a **Board** and a **Player** value and returns **True** if there is a sequence of four diagonal **Player** values to the right in any position on the board, and **False** otherwise.

The **leftCheckDiagonalsItem**, **leftCheckDiagonalsIter**, and **leftCheckDiagonals** functions work similarly to their counterparts **rightCheckDiagonalsItem**, **rightCheckDiagonalsIter**, and **rightCheckDiagonals**, but they check for diagonal sequences to the left instead of to the right.

The **isDraw** function takes a **Board** and returns **True** if the board is full and there is no winner, and **False** otherwise.

The **showBoard** function takes a **Board** and returns a string representation of the board.

The **isLegalMove** function takes a **Board**, a **Player** value, and a column index and returns **True** if the move is legal (i.e., the column is not full), and **False** otherwise.

The **makeMove** function takes a **Board**, a **Player** value, and a column index and returns a new **Board** with the **Player** value added to the specified column, if the move is legal. If the move is illegal, it returns the original **Board**.

The **winner** function takes a **Board** and returns the **Player** value of the winner, if there is one. If there is no winner, it returns **None**.

The **playerTurn** function takes a **Board** and returns the **Player** value whose turn it is, based on the number of moves made so far. If the board is full, it returns **None**.

**Game.hs**

This code is an implementation of a console-based Connect Four game. The **play** function is the main game loop, which continues until a player wins, the game is a draw, or the game is otherwise terminated. At each iteration of the loop, it calls **readInteger** to get the column number from the player whose turn it is, then it calls **checkMove** to verify that the chosen column is a legal move. If the move is legal, it calls **makeMove** to update the game board with the new move. If the move is illegal, the game board remains unchanged. Finally, the game board is displayed using **showBoard** and the loop continues. The **checkMove** function returns **True** if the given column is a legal move, and **False** otherwise. The **readInteger** function reads and returns an integer from the user. The **main** function simply initializes the game board and starts the game loop.

This code defines a Haskell program that plays a simple two-player game of connect four. The game is played on a 6x7 grid, which is represented by a **Board** data type. The **play** function is the main loop of the game, which continues until one of the players wins or the game is a draw.

The first four lines of the **play** function handle the different possible outcomes of the game:

* If player 1 (Yara) has won, it prints "Yara wins!".
* If player 2 (Kamal) has won, it prints "Kamal wins!".
* If the game is a draw, it prints "Ta3adol".
* If the game is not yet over, it continues with the next move.

The **checkMove** function is used to check whether a player's chosen move is legal. If the move is illegal, it prints "mamno3!" and returns **False**. Otherwise, it returns **True**.

The **readInteger** function is used to prompt the current player to enter their move (a column number between 1 and 7). It reads their input and returns it as an integer.

The **main** function is the entry point of the program. It creates a new board using **newBoard** and prints it out. Then it starts the game loop with the **play** function.