COMP1100 Assignment01 Report

Name : Renhao(Cameron) Tan UID: u6211458

COMP1100 assignment01 is about designing a programme for a battleship game. For now, I would like to explain my functions by dividing them into 3 parts, which are Preliminaries, Part A and Part B. These 3 parts refer to the same contents as those in the assignment instruction.

Preliminaries

In this stage, we are required to write 3 functions, which are **cStateToCell**, **shipLength** and **coordInBound**.

The purpose of **cStateToCell** function is to judge the status of the board map. If there is a **Hit** at one point in the board matrix, this point will be marked as "o" while generating the board map. It can help the players or programmers to understand the game in a much easier way. If there is a **Miss** or **Unchecked** at the point in the board matrix, this point will be marked as "x" or " ", respectively.

The next function is **shipLength**. This function is to calculate the length of the 5 types of ship. It takes a **ShipType** parameter and returns the length of this type of ship.

coordInBound is a judging function. The system will feed a coordinate to it, and the function can tell the validity of the coordinate. Since the game is working on a 10×10 map, the valid coordinates are from (0, 0) to (9, 9). The function is to judge whether the horizontal coordinate and the longitudinal coordinate are both in the range of 0 to 9 or not. If the coordinate satisfies the statement above, the function will return a Boolean

value, **True**. On the contrary, it returns a **False**.

Part A - Board Generation

The core of this part is the **placeShip** function, whose target is to generate a board matrix and make sure all 5 types of ships are created and located validly in the map. **placeShip** is more complex than the 3 functions in the previous part. It takes 4 parameters, which are **gs**, **(x, y)**, **dir** and **st**. The data type of **gs** is **GenShips**. This data type also contains 3 types of data, which are **gsShips**, **existingShips** and finished. Since the function is required to generate a board matrix and make sure all types of ships are located, the first task is to check the coordinate we feed to the **placeShip** function is valid to locate the ship or not. At this moment, we can call the function **validPlacement** to judge if the corresponding point of the coordinate we input is valid or not. **validPlacement** also takes 4 variables, and the types of these variables are exactly same with those are fed to **placeShip**. In that case, we can just input **gs**, **(x, y)**, **dir** and **st** to **validPlacement**, and the function will return a Boolean value. If the ship cannot be located at the point we input, which **validPlacement** returns a **False**, we do not need to do anything and make sure **placeShip** return the original **gs**. However, when the coordinate we choose is valid to locate a ship, we need to update the element in **existingShips**, a variable in **gs**. **existingShips** is a list whose elements are **ShipType**, and the purpose of it is to store the types of the ships which have been successfully located in the board matrix. Since **validPlacement** can guarantee that it the new **ShipType** value does not exist in **existingShips**, what I am supposed to do next is to update the elements in **existingShips**, so the same type of ships will not be located in the board matrix more than once. In this case, we can call the function **updateList** to add the new **ShipType** variable at the end of **existingShips**. The definition of **updateList** is below:

```
updateList :: [a] -> Int -> a -> [a]
updateList list n x = take (n) list ++ [x] ++ drop (n + 1) list
```

As we can see in the definition, **updateList** is a function to replace the original (n + 1) 'th element of the list with a new element(**x** in **updateList**) we input in the function. Therefore, we can call this function to add the new **ShipType** variable at the end of **existingShips**. So the code I wrote to judge this condition is:

```
| validPlacement gs (x, y) dir st = GenShips (gsShips gs) (updateList
(existingShips gs) (length (existingShips gs)) st) (finished gs)
```

*This piece of code is written in one row

In this case, actually, the **Int** I input to **updateList** is "`length (existingShips gs)`". This **Int** variable represents the length of **existingShips**, which represents I would like to replace the next element of the last element of the original **existingShips**. The next element of the last element of the original **existingShips** is actually nothing. In this case, **updateList** adds the element we input to the end of the original **existingShips**.

There is also another method to update existingShips by using the operator "++". We can change the code "`(updateList (existingShips gs)`" to "`((existingShips gs) ++ [st])`". This method is to add the new **ShipType** element at the end of **existingShips** directly.

Since **validPlacement** guarantees that the same type of ships will not be located in the board map more than once, we can deduce that when the length of **existingShips** reaches 5, all 5 types of ships have been successfully located in the board matrix and board generation is finished. At this moment, we ought to change the variable **finished**(a variable in **gs**) to **True** to end this part.

Part B - Gameplay

In Part B, it is required to focus on writing the function **transitionState**. This is the most important and challenging part in the whole assignment. The function **transitionState** take a variable, whose type is **State**, and a coordinate. **State** is a data type containing

multiple variables, which is similar to **GenShips**. **State** consists of 4 variables - **board**, **ships**, **condition** and **numMoves**. Both **board** and **ships** are matrices. **board** is a matrix of **Cell**(**Hit**, **Miss** or **Unchecked**), and **ships,** whose purpose is to record the presence of the 5 types of ships generating in Part A, is a matrix consisting of Boolean values. **condition** is a variable to judge whether the game is over or still playing. **Won**, **Lost** or **Playing** can be a value of **condition**. The last variable of **State**, **numMoves**, is an integer representing how many moves the player goes.

As we can see in the instruction of assignment, **transitionState** is supposed to upgrade the **State** variable after player going every single move, and change **condition** of the variable **state** after the game is over. However, the data type, **Condition**, cannot be checked by using Boolean expression "==". Here is the definition of **Condition** the data type:

```
data Condition = Won | Lost | Playing
   deriving (Show)
```

So it is necessary to write a new function called **judgeCondition** to judge the value of **condition** the variable. What is more, case expression should be used in the new function. The code of **judgeCondition** is below:

```
judgeCondition :: Condition -> Bool
judgeCondition con = case con of
   Won     -> True
   Lost    -> True
   Playing -> False
```

The function returns **True** while **con** is **Won** or **Lost**, and returns **False** while **con** is **Playing**. Since we can judge **condition** by calling **judgeCondition**, we can also return the original **State** variable in **transistionState** when **condition** the variable is **Won** or **Lost**, which meets the first requirement in Part B.

The second requirement told us to return the original **State** variable when move counter equals to 20 and change **condition** to **Lost**. To satisfy this requirement, we only need to change the **condition** when **numMoves** is 20. The code is below:

```
| numMoves state == 20 = State (board state) (ships state) Lost 20
```

The next requirement asks us to check whether the coordinate feeds to **transitionState** is valid for the board map. If it is not valid, return the original **state** variable. We can do the same thing in the function **coordInBound** by checking the new coordinate is in the range of (0, 0) to (9, 9).

The forth and fifth requirement is more complicated than all the previous ones. For now, we need to figure out in what kind of circumstance the ships are all sunk. Personally speaking, we can check the number of **Hit** in **board** the matrix. Since we know all of the length of the 5 types of ships, we can deduce that all ships are sunk when the summation of the length of all ships in matrix equals to the number of **Hit** in **board** the matrix. Since Haskell cannot check how many **Hit**s are there in a matrix, we need to write several functions to make this come true. First of all, the data type **Cell** is similar to **Condition**, so we cannot judge whether the **Cell** variable is same with a specific value or not by using "==". Instead of using "==", case expression can check the value of **Cell**. The new function is called **judgeHit**, using the same method in that we used in **judgeCondition**. The new function returns a **True** when the input is a **Hit**. Otherwise, it returns a **False**. After this, we can finally try to count the number of **Hit** in **board** the matrix. In this case, we can call the built-in function "filter" in Haskell, since it can be fed a "a -> Bool" type function and take every element of a list which can get a **True** by input the element in the "a -> Bool" type function, and build a new list consists of these elements which meets the requirement. If I feed **judgeHit** function as the filter condition, the system can return a new list only contain the element **Hit**. Then the length of the new list will be the number of **Hit** in **board** the matrix. The function counting number of **Hit** is called **countHit**. The code of the function is below:

```
countHit :: Board -> Int -> [Cell]
countHit b n
    | n == 9 = filter judgeHit (b !! n)
    | n >= 0 && n < 9 = filter judgeHit (b !! n) ++ countHit b (n + 1)
```

The variable **b** is a **board** matrix. The expression "b !! n" can get the n' th element in **b**. In this case, **b** is supposed to be a list. It is necessary to clarify that matrix is actually a list whose element are lists. If we type a command of "`countHit (board state) 0`" in terminal of IntelliJ, the **countHit** function will output a list contains all the **Hit** in **(board state)** the variable. Then we can get the length of the list, which is actually the number of **Hit** in **board** the matrix. So we can be sure that if there is a ship presented at the coordinate we input and the number of **Hit** equals to 16, we can just update target coordinate of **board** the matrix as a **Hit**, and change the condition to **Won**. If the number of not is less than 16, we can just update the target coordinate of the board as a **Hit** and move on. Since I mentioned above, matrix is a list whose elements' type is list, we can call the updateList twice to change a single element in matrix. For example:

```
updateList (board state) y (updateList ((board state) !! y) x Hit)
```

x and **y** refers to the horizontal and longitudinal coordinate, respectively. In this case, we change the **Cell** variable at **(x, y)** to **Hit**. However, in the updateList function, it only takes Int instead of Integer, the coordinate x and y is actually Integer. So we need to call **fromIntegral** function, which it can change the Integer to Int in **updateList** function. So we can change the previous example code like this:

```
updateList (board state) (fromIntegral y) (updateList ((board state) !!
(fromIntegral y)) (fromIntegral x) Hit)
```

*The code are written in one row

While the number of **Hit** is less than 16, the code can be written in this way:

```
| ((ships state) !! (fromIntegral y)) !! (fromIntegral x) = State (updateList
(board state) (fromIntegral y) (updateList ((board state) !! (fromIntegral y))
(fromIntegral x) Hit)) (ships state) (condition state) (numMoves state)
```

*The code is written in one row

So far, we have satisfied the forth requirement of instruction, then we can meet the fifth one easily, because it is similar as the forth instruction. We need to mark a **Miss** if there is no ship located at the target coordinate and add one to move counter. So the code is similar to the previous one, and it works similar with the last one:

```
| not (((ships state) !! (fromIntegral y)) !! (fromIntegral x)) = oneMoreMove
(State (updateList (board state) (fromIntegral y) (updateList ((board state) !!
(fromIntegral y)) (fromIntegral x) Miss)) (ships state) (condition state)
(numMoves state))
```

*The code is written in one row

The function **oneMoreMove** is a function takes a **State** variable and returns a **State** variable after adding one to the **numMoves** variable.

We have finished all the code right now. However, it occurs an error in the Pipeline of gitlab. Finally, I found out that the coordinate feed to **transitionState** could be the same as the coordinates being fed into it before, so I need to make sure the **Cell** of **board** the matrix is **Unchecked**. Otherwise, I need to add 1 to the move counter no matter the **Cell** variable is **Hit** or **Miss**. The new code should be like this:

```
| judgeChecked (((board state) !! (fromIntegral y)) !! (fromIntegral x)) =
oneMoreMove state
```

*The code is written in one row

After adding this statement, the new file can pass all the tests on Pipeline. The battleship game is successfully programmed in Haskell.