Kodable Game

In this project the goal is to code a clone of a children game called *Kodable*.

With Kodable kids learn core programming concepts such as sequencing,

conditionals, functions, and loops.

Before Getting Started

Expecting you have the installation of **Haskell stack** before running the program, if you do not have Haskell stack please go to https://docs.haskellstack.org/en/stable/install_and_upgrade/ download the package

To successfully run the import System.Random, input the followings commands

```
stack --resolver=lts setup
stack --resolver=lts install random
stack --resolver=lts ghci
```

https://stackoverflow.com/questions/42201587/how-to-install-system-random-maybe-cabal-issue-in-version-8-of-stack

***If you don not want to do the above please use **temp.hs** instead of Main.hs. However, **the new feature shuffle will be lost.** ***

Getting Started

After entering the ghci first load the Main.hs file with the command **:l Main.hs** then input **main** to run the game.

```
Prelude> :l Main.hs
[1 of 1] Compiling Main             ( Main.hs, interpreted )
Ok, one module loaded.
*Main> main
WELCOME!!!
Please Input Command (load/play/quit/check/solve/shuffle/help)
```

Setting of the Game

The game map is using

‘@’ represents the ball.

‘-’ represents a path block that the ball could roll on.

‘*’ represents the grass (obstacles).

‘p’ represents the tile of the path’s block color is the special color : pink.

‘o’ represents the tile of the path’s block color is the special color : orange.

‘y’ represents the tile of the path’s block color is the special color : yellow.

‘b’ represents the bonus (the stars in Kodable).

‘t’ represents the target point.

The direction is using

“Right” “Left” “Down” “Up”

If setting for the condition for the color block, then add “o_”,”p_”, ”y_”.

e.g., “o_Right”

Basic Features

Loading Game Map

Before playing the game or do others function of the games, the player is required to load the game map for the game first.

```
*Main> main
WELCOME!!!
Please Input Command (load/play/quit/check/solve/shuffle/help)
play
PLEASE LOAD THE GAME MAP FIRST
Please Input Command (load/play/quit/check/solve/shuffle/help)
```

In the above case, the player starts the game immediately without loading the game map. Therefore, the system will ask the player to load the game map first.

```
"load" -> do newFileName <- getFileName
             tempGameMap <- readMap newFileName
             let gameMap = allRemoveSpace tempGameMap
             case fileName of
                "" -> do putStrLn "Read map successfully!"
                         putStrLn "Initial:"
                         printBoard gameMap
                _ -> do putStrLn "New map is updated"
                        putStrLn "Initial:"
                        printBoard gameMap
             operate newFileName
```

In the load game function, the game will first check the file exists or not. If the file name exists, the system will grab the information inside the game map and return the face of the game map to the users. Then store the file name for the next step to run.

```
getFileName :: IO String
getFileName = do putStrLn "Please Input Map File Name"
                 fileName <- getLine
                 fileExist <- doesFileExist fileName
                 case fileExist of
                    True -> do putStrLn "Map File Is Found!"
                               return fileName
                    False -> do putStrLn "Map File Does Not Exist!"
                                getFileName

-- link of teaching using readFile
-- http://zvon.org/other/haskell/Outputprelude/readFile_f.html
-- link of teaching the usage of lines to split \n into list
-- https://stackoverflow.com/questions/37656419/how-i-can-split-n-in-haskell
readMap :: String -> IO [String]
readMap fileName = do putStrLn "Reading Map"
                      gameMap <- readFile fileName
                      return $ lines gameMap
```

The getFileName is to check the filename is exists or not. It will keep on asking the player to input the file name until the player inputs the file name is exists.

The readMap will turn the text file into list of string, which is list of lists, then return the result to the system.

Checking the Map is Solvable or Not

After the player have load the file into the program, the player can use the command **check** the game map is solvable or not. The programming will first spot out locations of bonus is map and the location of the end point to form a list. Then,

the programming will clear all the symbol of bonus and end point in the map to start the checking progress.

```
checkMap :: [String] -> Bool
checkMap gameMap = checkALLpath (emptyMap gameMap []) (bonusEndList gameMap)

checkALLpath :: [String] -> [(Int, Int)] -> Bool
checkALLpath oneMap location = case length location of
                                0 -> True
                                _ -> do let command = shortestPathCommand (addMap oneMap (location !! 0) 'b')
                                        case command of
                                             [] -> False
                                             _ -> checkALLpath oneMap (tail location)
```

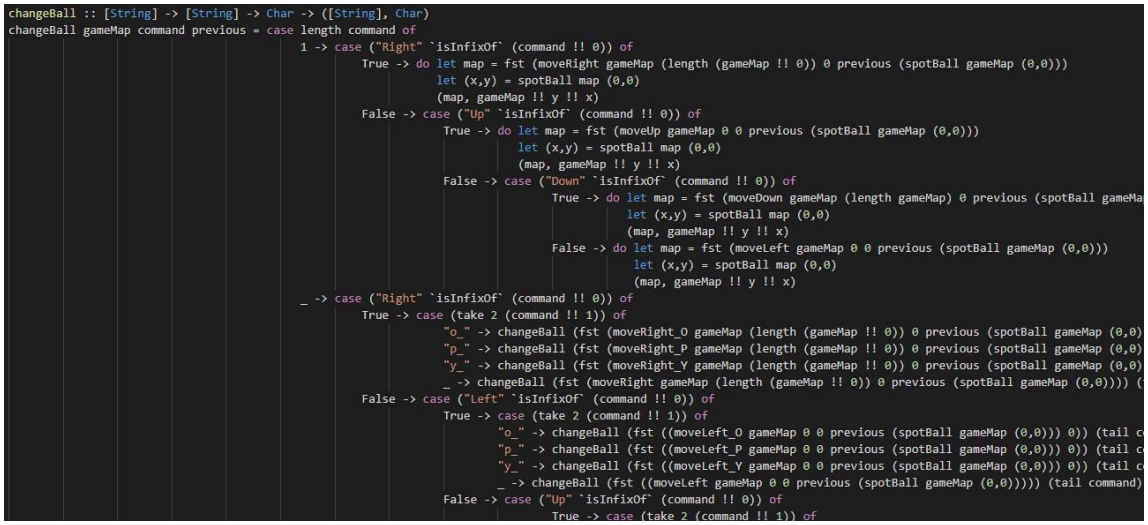
In checkALLpath function it will take the empty map and the list of bonus and end point to check the starting point (which the ball located) can reach those bonus and end point.

The addMap function will add back the symbol to the empty map with the location. After that it will run the function of shortestPathCommand it finds out the command which can use the less basic steps (Left, Up, Right, Down) to reach the point from the bonus. If the function returns empty list command, then which meant there does not have a solution for the starting point to reach that point. As the map will be solvable, as there has a way for starting point to reach any bonus or the end point.

Getting the Answer of the Game

The player input the command **solve** can get the answer of command to get the correct path to run the map. Same as the check it will create an empty game map which without the bonus and the end point, then build a list that have all the location of the bonus.

The program will first find out the bonus have the shortest way for the ball to get first, then store the command and use the command to control the ball to move. After the ball move, the program will find the bonus have the shortest path for the ball to take from the remaining bonus list and do until all the bonus list is empty.



```
changeBall :: [String] -> [String] -> Char -> ([String], Char)
changeBall gameMap command previous = case length command of
                                      1 -> case ("Right" `isInfixOf` (command !! 0)) of
                                              True -> do let map = fst (moveRight gameMap (length (gameMap !! 0)) 0 previous (spotBall gameMap (0,0)))
                                                         let (x,y) = spotBall map (0,0)
                                                         (map, gameMap !! y !! x)
                                              False -> case ("Up" `isInfixOf` (command !! 0)) of
                                                        True -> do let map = fst (moveUp gameMap 0 0 previous (spotBall gameMap (0,0)))
                                                                   let (x,y) = spotBall map (0,0)
                                                                   (map, gameMap !! y !! x)
                                                        False -> case ("Down" `isInfixOf` (command !! 0)) of
                                                                   True -> do let map = fst (moveDown gameMap (length gameMap) 0 previous (spotBall gameMa
                                                                              let (x,y) = spotBall map (0,0)
                                                                              (map, gameMap !! y !! x)
                                                                   False -> do let map = fst (moveLeft gameMap 0 0 previous (spotBall gameMap (0,0)))
                                                                               let (x,y) = spotBall map (0,0)
                                                                               (map, gameMap !! y !! x)
                                      _ -> case ("Right" `isInfixOf` (command !! 0)) of
                                              True -> case (take 2 (command !! 1)) of
                                                        "o_" -> changeBall (fst (moveRight_O gameMap (length (gameMap !! 0)) 0 previous (spotBall gameMap (0,0)
                                                        "p_" -> changeBall (fst (moveRight_P gameMap (length (gameMap !! 0)) 0 previous (spotBall gameMap (0,0)
                                                        "y_" -> changeBall (fst (moveRight_Y gameMap (length (gameMap !! 0)) 0 previous (spotBall gameMap (0,0)
                                                        _ -> changeBall (fst (moveRight gameMap (length (gameMap !! 0)) 0 previous (spotBall gameMap (0,0)))) (
                                              False -> case ("Left" `isInfixOf` (command !! 0)) of
                                                        True -> case (take 2 (command !! 1)) of
                                                                  "o_" -> changeBall (fst ((moveLeft_O gameMap 0 0 previous (spotBall gameMap (0,0))) 0)) (tail c
                                                                  "p_" -> changeBall (fst ((moveLeft_P gameMap 0 0 previous (spotBall gameMap (0,0))) 0)) (tail c
                                                                  "y_" -> changeBall (fst ((moveLeft_Y gameMap 0 0 previous (spotBall gameMap (0,0))) 0)) (tail c
                                                                  _ -> changeBall (fst ((moveLeft gameMap 0 0 previous (spotBall gameMap (0,0))))) (tail command)
                                                        False -> case ("Up" `isInfixOf` (command !! 0)) of
                                                                   True -> case (take 2 (command !! 1)) of
```

changeBall function will help to change the ball location in the map with the corresponding command list. In each move it will check the incoming command have contain a condition or not which “o_”, ”p_” or ”y_” in the head of the next command. If there is such command exits, then will control the ball to stop at those special color block.

```haskell
-- shotest path command
shortestPathCommand :: [String] -> [String]
shortestPathCommand gameMap = do let map = (path (gameMap,'-') "")
                                 case length map of
                                    0 -> []
                                    _ -> do let command = shortest map (map !! 0)
                                            let command2 = moreShorterPath command [] ""
                                            (take 1 command) ++ command2
-- shortest list2 (list2 !! 0)
-- Right p_Right then change to Right
-- https://www.programming-idioms.org/idiom/39/check-if-string-contains-a-word/858/haskell
moreShorterPath :: [String] -> [String] -> String -> [String]
moreShorterPath command stack previous
    | return = stack
    | continue = moreShorterPath (tail command) stack (command !! 0)
    | otherwise = moreShorterPath (tail command) (stack ++ [command !! 0]) (command !! 0)
    where
        return = length command == 0
        continue = previous `isInfixOf` (command !! 0)

shortest :: [String] -> String -> [String]
shortest input command
    | return = commandConvertor (words (input !!0)) []
    | return1 = commandConvertor (words command) []
    | change = shortest (tail input) (input !! 0)
    | otherwise = shortest (tail input) command
    where
        change = length (words command) > length (words (input !! 0))
        return = length input == 1 && (length (words command) > length (words (input !! 0)))
        return1 = length input == 1
```

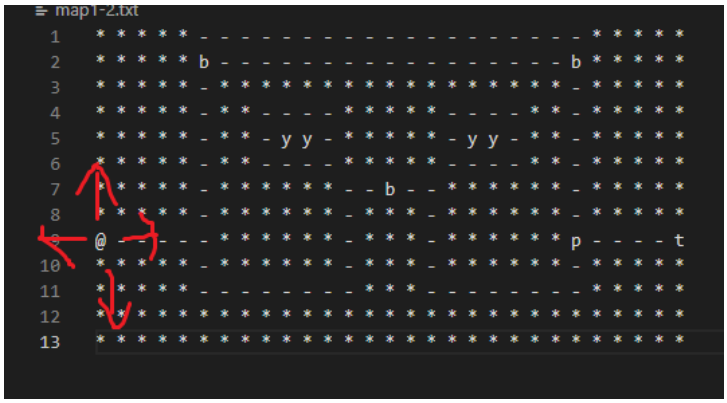
shortestPathCommand will take the shortest command list for getting that location in the map. Also it will remove some duplicate command such like “p_Right” “Right” can directly use one “Right” instead of using two command.

```
path :: ([String], Char) -> String -> [String]
path (gameMap,previous) stack = case (bonusInMap gameMap 0 == 1) of
                                True -> case (validDown gameMap (length gameMap) (spotBall gameMap (0,0))) of
                                        False -> case (validRight gameMap (length (gameMap !! 0)) (spotBall gameMap (0,0))) of
                                                False -> case (validLeft gameMap 0 (spotBall gameMap (0,0))) of
                                                        False -> case (validUp gameMap 0 (spotBall gameMap (0,0))) of
                                                                False -> []
                                                                True -> path (solveUp gameMap (spotBall gameMap (0,0)) 0 '-') (stack
                                                        True -> case (validUp gameMap 0 (spotBall gameMap (0,0))) of
                                                                False -> path (solveLeft gameMap (spotBall gameMap (0,0)) 0 '-') (sta
                                                                True -> (path (solveLeft gameMap (spotBall gameMap (0,0)) 0 '-') (sta
                                                True -> case (validLeft gameMap 0 (spotBall gameMap (0,0))) of
                                                        False -> case (validUp gameMap 0 (spotBall gameMap (0,0))) of
                                                                False -> path (solveRight gameMap (spotBall gameMap (0,0)) (length (g
                                                                True -> (path (solveRight gameMap (spotBall gameMap (0,0)) (length (g
                                                        True -> case (validUp gameMap 0 (spotBall gameMap (0,0))) of
                                                                False -> (path (solveLeft gameMap (spotBall gameMap (0,0)) 0 '-') (st
                                                                True -> (path (solveLeft gameMap (spotBall gameMap (0,0)) 0 '-') (sta
                                        True -> case (validRight gameMap (length (gameMap !! 0)) (spotBall gameMap (0,0))) of
                                                False -> case (validLeft gameMap 0 (spotBall gameMap (0,0))) of
                                                        False -> case (validUp gameMap 0 (spotBall gameMap (0,0))) of
                                                                False -> path (solveDown gameMap (spotBall gameMap (0,0)) (length gam
                                                                True -> (path (solveDown gameMap (spotBall gameMap (0,0)) (length gam
                                                        True -> case (validUp gameMap 0 (spotBall gameMap (0,0))) of
                                                                False -> (path (solveDown gameMap (spotBall gameMap (0,0)) (length ga
                                                                True -> (path (solveDown gameMap (spotBall gameMap (0,0)) (length gam
                                                True -> case (validLeft gameMap 0 (spotBall gameMap (0,0))) of
                                                        False -> case (validUp gameMap 0 (spotBall gameMap (0,0))) of
                                                                False -> (path (solveDown gameMap (spotBall gameMap (0,0)) (length ga
                                                                True -> (path (solveDown gameMap (spotBall gameMap (0,0)) (length gam
                                                        True -> case (validUp gameMap 0 (spotBall gameMap (0,0))) of
                                                                False -> (path (solveDown gameMap (spotBall gameMap (0,0)) (length ga
                                                                True -> (path (solveDown gameMap (spotBall gameMap (0,0)) (length gam
                                False -> [stack]
```

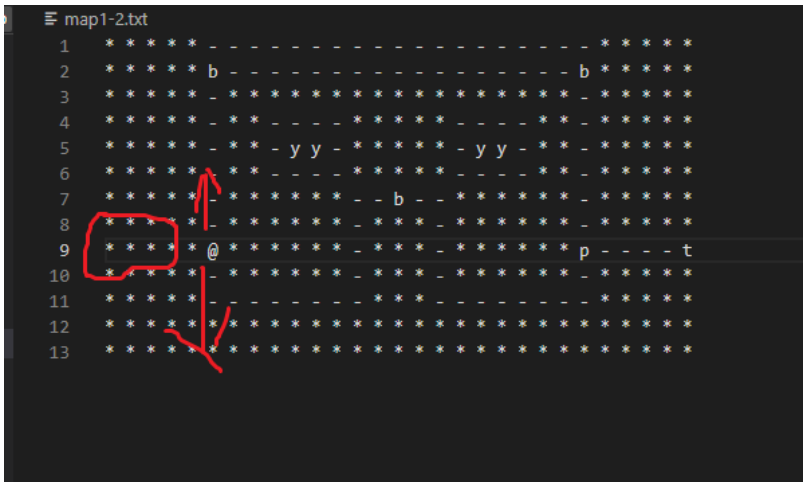
path is the function that return all the possible method to grab the bonus. Since the there will have only one bonus in the empty map. It will be starting from the ball to run through whole map until there does not have a bonus in map, which the bonus is grabbed by the ball. The path that the ball goes through will become obstacle, therefore the ball will not be able to walk back to the path that it comes. Such that if the ball goes up, it will no longer be moving back. In such case, the ball will try to move four way out and check it is possible to go that way or not, until there is a deadlock (no way go).

IDEA OF BUILDING THE PATH

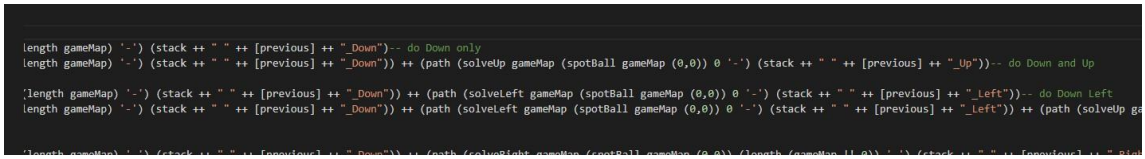
Just thinking the ball is the water and will flood the whole game map, which like the fire simulator in NetLogo. The ball will go through the whole game map.



The first steps the ball only can go Right, as other three way is blocked. The function will accumulate the command Right and move the ball to right.



After the ball move to Right, the path that's the ball come from will be change to '*' which is blocked the ball move to Left. Since there is two way for the ball to go. It will pass the function into two which can move Up and Down and continue the loop.



As the game map only exist one bonus it will loop until it grabs the bonus and **return the command list which can get to the bonus** or until it inside a deadlock which is no longer to move in any direction then it will **return an empty command representing cannot move to the bonus**

```haskell
solveDown :: [String] -> (Int, Int) -> Int -> Char -> ([String], Char)
solveDown gameMap (ballX, ballY) max condition
    | return_o = (gameMap, condition)
    | return_p = (gameMap, condition)
    | return_y = (gameMap, condition)
    | continue = solveDown (replaceColumn (ballY + 1) (replaceRow ballX '@' (gameMap !! (ballY + 1))) (replaceColumn ballY (replaceRow ballX
    | otherwise = (gameMap, condition)
    where
        return_o = condition == 'o'
        return_p = condition == 'p'
        return_y = condition == 'y'
        continue = (ballY + 1 /= max) && (gameMap !! (ballY + 1)!! ballX /= '*')
solveUp :: [String] -> (Int, Int) -> Int -> Char -> ([String], Char)
solveUp gameMap (ballX, ballY) min condition
    | return_o = (gameMap, condition)
    | return_p = (gameMap, condition)
    | return_y = (gameMap, condition)
    | continue = solveUp (replaceColumn (ballY - 1) (replaceRow ballX '@' (gameMap !! (ballY - 1))) (replaceColumn ballY (replaceRow ballX '
    | otherwise = (gameMap, condition)
    where
        return_o = condition == 'o'
        return_p = condition == 'p'
        return_y = condition == 'y'
        continue = (ballY /= min) && (gameMap !! (ballY - 1)!! ballX /= '*')

solveLeft :: [String] -> (Int, Int) -> Int -> Char -> ([String], Char)
solveLeft gameMap (ballX, ballY) min condition
    | return_o = (gameMap, condition)
    | return_p = (gameMap, condition)
    | return_y = (gameMap, condition)
    | continue = solveLeft (replaceColumn ballY (replaceRow (ballX - 1) '@' (replaceRow ballX '*' (gameMap !! ballY))) gameMap) (ballX - 1, b
    | otherwise = (gameMap, condition)
    where
```

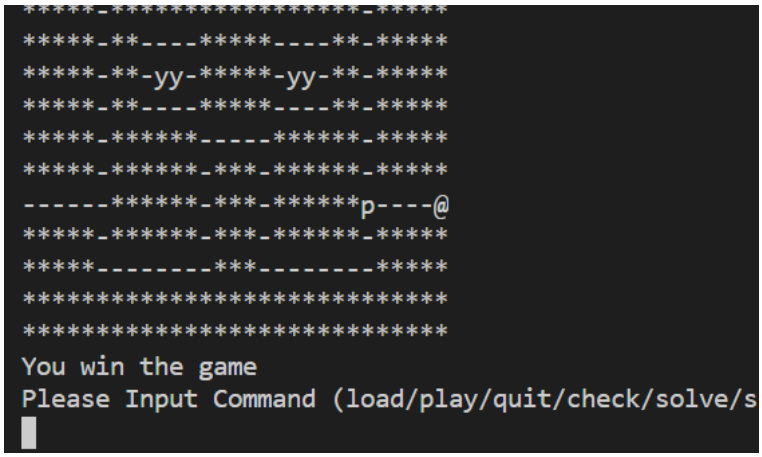
solveUp, solveDown, solveRight, solveLeft will change the path to '*'. It will stop when the next steps are '*' or stepping into a color block and return the current map and the steps it is standing on.

Trying Solve

Using map1-2.txt as the sample to try the solve function can function well or not to solve the map. Then use the answer for playing the game see whether can win the game or not.

```
*****-******-***-******-*****
*****--------***---------*****
*****************************
*****************************
Please Input Command (load/play/quit/check/solve/shuffle/help)
solve
THE ANSWER:
Reading Map
Right Up Right Down Left Up Left Right Down Right Up p_Right
Please Input Command (load/play/quit/check/solve/shuffle/help)
```

```
New Direction: p_Right
New Direction:
Right
Up
Right
Down
Left
Up
Left
Right
Down
Right
Up
p_Right
*****                   *****
```



Limitation of Building shorter map

There maybe shorter method to reach finish the game if with the help of Loop or function. It is possible work out the answer have Loop with just checking out the answer which have two pair consecutives appear in a row. However, it needs find out all the possible cases to finish the map. Such that it can be tell the ball to move to the bonus which have the longest command but repetitive appear. However, it is hard to determine, that path have the use of Function or not. To concentrate the command and make comparison with other command list to reach to other command. If to do so, the program needs to list out possible cases first and then do concentration for Funciton and Loop. Following up, with the comparison job in the last.

Playing the Game

The player can play the game by inputting the **play** command after loading the map, the player can ought to choose have Function along with it or not. If the players want to stop at specific color block it needs to set condition in the next command such that “Right” -> “o_Left”, the ball will stop at orange block when moving to Right, otherwise it will move until run into an obstacle.

```haskell
loopCheck :: [String] -> Int -> Bool
loopCheck (x:xs) index···

checkDirection :: String -> [String] -> Bool
checkDirection input funciton = case (take 4 input) of
                                  "Loop" -> case (loopCheck (words input) 0) o
                                              True -> True
                                              False -> False
                                  _ -> case input of
                                         "Function" -> case funciton of
                                                         [] -> False
                                                         _ -> True
                                         "Left" -> True
                                         "p_Left" -> True
                                         "y_Left" -> True
                                         "o_Left" -> True
                                         "Right" -> True
                                         "p_Right" -> True
                                         "y_Right" -> True
                                         "o_Right" -> True
                                         "Up" -> True
                                         "p_Up" -> True
                                         "y_Up" -> True
                                         "o_Up" -> True
                                         "Down" -> True
                                         "p_Down" -> True
                                         "y_Down" -> True
                                         "o_Down" -> True
                                         _ -> False

getDirection :: [String] -> [String] -> IO [String]
getDirection stack funciton = do putStr "New Direction: "
                                 input <- getLine
                                 if input == ""
```

It will store and check the player have input correct command of directions until the player input empty command <Enter>. If the player does not input the function when starting the game, Function will not be allowed input in the direction. Also, it will check the player have input a correct command for the ball to go or not at the same time.

```haskell
oMove :: [String] -> [String] -> [String] -> Int -> Char -> Int -> IO ()
oMove command function gameMap bonus previous winbonus = case (checkWinner gameMap bonus winbonus) of
                                                            True -> do putStrLn "You win the game"
                                                                       return ()
                                                            False -> case command !! 0 of
                                                                       "Up" -> case (validUp gameMap 0 (spotBall gameMap (0,0))) of
                                                                                 True -> case (length command) of
                                                                                         1 -> do let (board, newBonus) = moveUp gameMap 0 bonus previous (spotBall ga
                                                                                                 printGetBonus newBonus bonus
                                                                                                 case (checkWinner board newBonus winbonus) of
                                                                                                   True -> do printBoard board
                                                                                                              putStrLn "You win the game"
                                                                                                              return ()
                                                                                                   False -> do printBoard board
                                                                                                               putStrLn "You lose the game"
                                                                                                               return ()
                                                                                         _ -> case (take 2 (command !! 1)) of
                                                                                                "o_" -> do let (board, newBonus) = moveUp_O gameMap 0 bonus previous
                                                                                                           printGetBonus newBonus bonus
                                                                                                           printBoard board
                                                                                                           doMove (tail command) function board newBonus 'o' winbonu
                                                                                                "p_" -> do let (board, newBonus) = moveUp_P gameMap 0 bonus previous
                                                                                                           printGetBonus newBonus bonus
                                                                                                           printBoard board
                                                                                                           doMove (tail command) function board newBonus 'p' winbonu
                                                                                                "y_" -> do let (board, newBonus) = moveUp_Y gameMap 0 bonus previous
                                                                                                           printGetBonus newBonus bonus
                                                                                                           printBoard board
                                                                                                           doMove (tail command) function board newBonus 'y' winbonu
                                                                                                "Fu" -> case (take 1 (function) !! 0) of
                                                                                                          "o" -> do let (board, newBonus) = moveUp_O gameMap 0
                                                                                                                    printGetBonus newBonus bonus
```

After the player successfully input the directions for the ball to move, the command list will pass to the doMove function to run. Every time the ball to move, the function will check the ball have grab all the bonus and reach the end point or not.

Ending

If reach the last command and still not finish all the bonus or reach the last point. The function will tell the player the result (Player lose the game). In each step, it will check the next command have set condition or not, to decide the ball stop on that specific block or just stop until run into the obstacle.

Error

In each steps the doMove function wil check the next steps is an obstacle or not. It will secure that the ball can at least move one steps.

```
validUp :: [String] -> Int -> (Int, Int) -> Bool
validUp gameMap min (ballX, ballY) ···

validDown :: [String] -> Int -> (Int, Int) -> Bool
validDown gameMap max (ballX, ballY)
    | return1 = False
    | return2 = False
    | otherwise = True
    where
        return1 = (ballY + 1 == max)
        return2 = (gameMap !! (ballY + 1) !! ballX == '*')

validRight :: [String] -> Int -> (Int, Int) -> Bool
validRight gameMap max (ballX, ballY) ···

validLeft :: [String] -> Int -> (Int, Int) -> Bool
validLeft gameMap min (ballX, ballY) ···
```

Quitting the Game

Once the player input the command **quit** in the system, the system will terminate the game and the program will return to the terminal.

```
PLEASE LOAD THE GAME MAP FIRST
Please Input Command (load/play/quit/check/solve/shuffle/help)
quit
*Main>
```

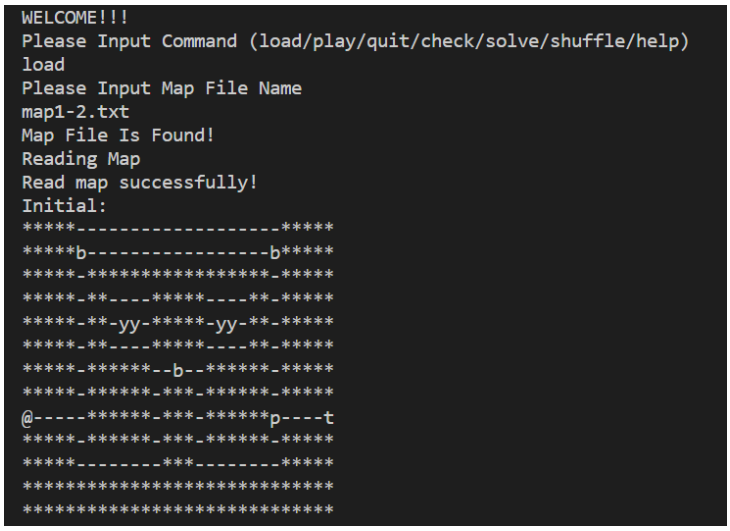
```
                    False -> putS
                operate fileName
quit" -> return ()
```

New Features

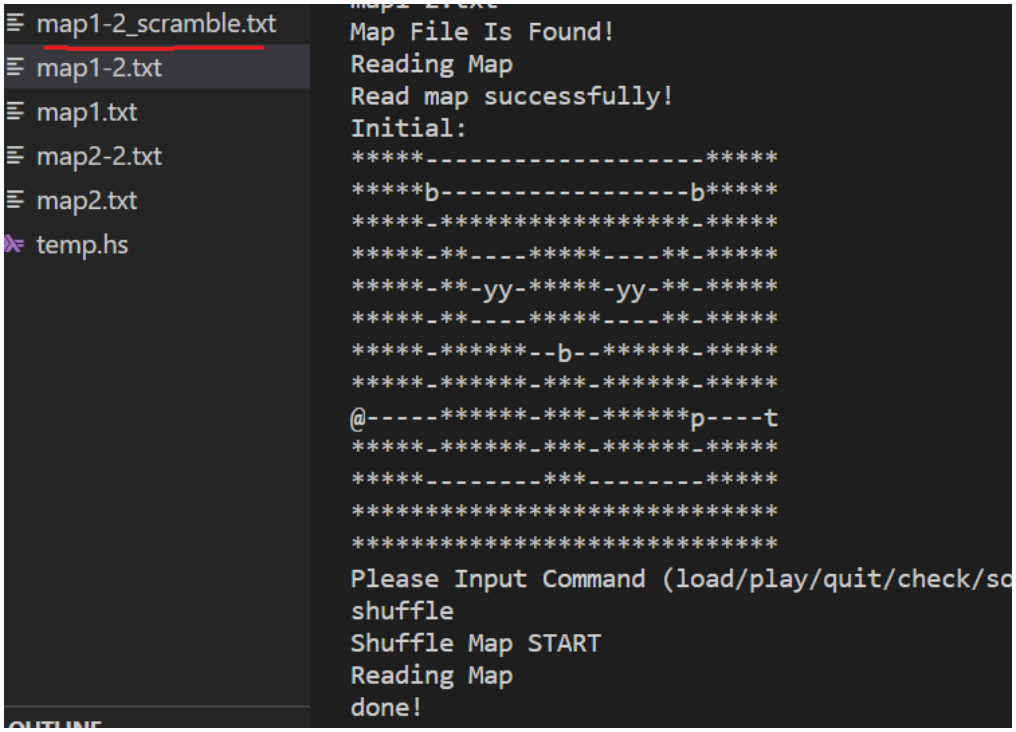
Shuffle the Game Map

The command shuffle will be provided for the player to help them create a new game map for the game base on the original game map which the user loaded in the game. The shuffled game map can be solvable or not be solvable. The player

needs to use their intelligent and smartness to determine the shuffle game map can be solved or not. If the game map can be solved, then how should the Player input the commands to reach the end as well as grabbing all the bonus at the same time.



Game map is loaded



After input the command **shuffle**, then a new text file will be generated. The new file will be based on the original file name and add the label of **scramble**. If the player wants to try the new file, the player only needs to load the new file again.

```haskell
buildFile :: [String] -> String -> IO ()
buildFile gameMap fileName = do scrambleFile <- openFile (init(init(init(init fileName))) ++ "_scramble.txt") WriteMode
                                scrambleMap <- scrambleGameMap gameMap []
                                hPutStrLn scrambleFile scrambleMap
                                hClose scrambleFile
                                putStrLn "done!"

scrambleGameMap :: [String] -> String -> IO String
scrambleGameMap gameMap stack = case length gameMap of
                                    1 -> do stack1 <- scrambleRow (gameMap !! 0) []
                                            let stack2 = stack ++ stack1
                                            return stack2
                                    _ -> do stack1 <- (scrambleRow (gameMap !! 0) [])
                                            let stack2 = stack ++ stack1 ++ "\n"
                                            scrambleGameMap (tail gameMap) stack2

scrambleRow :: String -> [Char] -> IO String
scrambleRow row stack = case length row of
                            1 -> do let stack1 = stack ++ (take 1 row)
                                    return stack1
                            _ -> do index <- randomRIO (0, length (row) - 1)
                                    let stack1 = stack ++ [row !! index]
                                    let row1 = take index row ++ drop (1 + index) row
                                    scrambleRow row1 stack1
```

Shuffle will take the original game map as base then use randomRIO to generate a random number as index to pick the elements in the row and reforming a new order. After finishing all the row then they will combine into a new Game Map for the player to play.

Detail of the Game

help function is provided inside the game, player can get the details of the command to know more how to control the game.

The detail of the command will print out to let the player know.

```
help
************* --Command Menu-- *************
load:    Player can input the file name of
         the game map for the game
check:   Player can check the map has a
         solution to solve or not
solve:   Player can get the answer to solve
         the map
play:    Player can run the map to play. If
         the player needs to use function
         such that Top Left Right. Player
         needs to input along with play as
         well. Function needs to take 3
         directions.
         The player will be asked to input
         the command to run the game map
         If the player want the ball to
         stop at color block, then the
         player need to set p_, o_, y_
         on the next direction. Such that
         Right o_Up the ball will move to
         right and stop at orange color
```