Five In a Row

Instructor: Eric Walkingshaw
Oregon State University
Xu, Zheng (zhengxu) Tien-Lung,Chang (changti)

1. Introduction

Our project is to implement board game called “Five in a Row” (which is also known as “Gomoku” in Japanese). It is a two players game that is suitable for all ages. The principle for this game is the first player who make a coherently sequence of five stones will win the game. The sequence can be vertically, horizontally or diagonally. The user can choose to play against another user or the A.I.. The program will check if there is a winner at the end of every turn. Although there are many people different kinds of version of “Five in a Row”, most of the people implement it in a other programming language. The difference between our game and others is the language we are using is Haskell, which is a more elegant and simplify programming language. That is, the pattern and flow will be more clearly than other and also more efficiently.

2. Implementation

To implement the game “Five in a Row”, we define couple data types and functions.

2.1 *Types*

There are three main data type, Cell, Player and Board in our game.

The Cell data type represents the state of the stone. The Cell can be “Black”, “White” or “Blank”. In the initial state, all the Cell will be “Blank”. As the game start, the first player will use the “Black” stone as a mark for the position that he input and the second player will be “White”. The data type is shown below:

```
data Cell = Black
          | White
          | Blank
```

The Player data type represents the player. If we choose to play with real person, each round the player will put in stone Black or White. If we choose to play with A.I., each round the player will put in stone Black or White as well. The data type is shown below:

```
data Player = Human Cell
            | AI    Cell
```

The game could be a single player game or a duo game. So we create Mode data type. If we choose to play with A.I., it means there is only one real person in this game. In this case, the mode should be Single. Otherwise, the mode is Duo. The data type is shown below:

```
data Mode = Single
          | Duo
```

The Board data type represents the board. The data type is shown below:

```
data Board a = Board [[a]]
```

2.2 *Functions*
There are seven main functions in our code, showBoard, updateBoard, checkFive, isGood, Helper, whichMode and gameLoop.

showBoard: The function for showBoard is to print out the current statement of the board. We will show the current status of the board at the end of every turn, so that the player can play the game much easier.

updateBoard: This function is used to update the board at the end of every turn. After the player put a stone on the board, the updateBoard function will update the current board.

checkFive: This function is used to check if one of the player wins the game at the end of current turn.

isGood: This function is used to check whether the input position is valid or not. If the position is not blank then the position is invalid. When an invalid input is detected, the player will be asked to input a new position.

helper: This function is a high-order function that help to identify the player for several functions. We have several functions to modify the state of player and cell. The higher order function can help us to make our codes more clear and well organized.

whichMode: This function is used to set the game mode based on the player's choice. We have three different modes. 1) Play against AI (Black); 2) Play against AI (White); 3) Play against another player.

gameLoop: The function gameLoop is the main function to control the logic of whole game. It will help the game continues running in a correct way.

3. Design description

We applied three ideas into our project, higher-order function, data-type encoding and refactoring.

3.1 *Higher order Function*

The Higher order function that we implement in our game is called playerHelper. This function gives a pattern to the other function and a technique for implementing name bindings. The function is shown below:

```
helper :: t -> t -> Player -> t
helper a _ ( Human Black ) = a
helper _ b ( Human White) = b
helper a _ ( AI Black ) = a
helper _ b ( AI White) = b
```

The playerHelper takes two data type which has same type and a Player data type, it return a data type which is the same as the first two data type. The playerHelper is used in three functions, currentPlayer, checkPlayer and nextPlayer. Because all of the three functions will use the Player data type and generate a different data type, we decide to bind these three function to a higher order function. The function are listed as below:

```
currentPlayer :: Player -> IO()
currentPlayer = helper (putStrLn "BLACK's turn: ") (putStrLn "WHITE's turn: ")

cellColor :: Player -> Cell
cellColor = helper Black White

nextPlayer :: Player -> Mode -> Player
nextPlayer (Human o)     Duo    = helper (Human White) (Human Black) (Human o)
nextPlayer (Human Black) Single = helper (AI White) (Human Black) (Human Black)
nextPlayer (Human White) Single = helper (Human White) (AI Black) (Human White)
nextPlayer (AI Black) _ = helper (Human White) (AI Black) (AI Black)
nextPlayer (AI White) _ = helper (AI White) (Human Black) (AI White)
```

By using the Higher order function, we can minimum our code and let the source code looks more elegant. The reason that we decide to use higher order function is because it is easy to extend with new cases and we can reuse features of host language.

3.2 *Data-type encoding*

From the course, we learned that the tradeoffs of data-type encoding and type-class encoding. We choose to use data-type encoding instead of using type-class encoding. As we know, it is easy to add new operations by using data-type encoding. At first, we have no idea how many functions we need. If we want to add a new operation, using data-type encoding is more convenient. The extensibility of data-type encoding meets our requirement. This is the reason why we choose data-type encoding rather than type-class encoding.

3.3 *Refactoring*

During the implement of our idea, we did a lot of codes review. What we did about refactoring includes changing the layout, using generalization to factor out the repeated patterns in functions, refactoring data types and so on. Here are some examples show what we did.

For the gameLoop function, we do the two things to refactoring the function. First, we break the whole function into several small piece of code to make the function much easier to read. Second, we do the pattern matching to delete the code which repeated too many times. As the following code shown, the original code has two similar part and the code is unreadable.

After we do the refactoring, we define a new function loopfunc to eliminate the repeat part. The gameLoop become much shorter than the original one.

```
gameLoop :: Board Cell -> Player -> Mode -> IO ()
gameLoop (Board x) (AI o) m =
  do showBoard (Board x)
     currentPlayer (AI o)
     col <- randomRIO(1,15) >>= (\x -> return x)
     putStr "Col: "
     print col
     row <- randomRIO(1,15) >>= (\x -> return x)
     putStr "Row: "
     print row
     loopfunc (Board x) col row (AI o) m

gameLoop (Board x) (Human o) m =
  do showBoard (Board x)
     currentPlayer (Human o)
     c <- getCol
     r <- getRow
     let col = read c :: Int
     let row = read r :: Int
     loopfunc (Board x) col row (Human o) m
```

We also define a new function for the input and the input exception, which is in the original gameLoop function. You can see the detail listed below:

```haskell
getCol :: IO String
getCol = do
  putStr "Col: "
  c <- getLine
  if isNumber c
    then return c
    else do
      putStrLn "Please enter a valid position."
      getCol

getRow :: IO String
getRow = do
  putStr "Row: "
  c <- getLine
  if isNumber c
    then return c
    else do
      putStrLn "Please enter a valid position."
      getRow
```

To check if one of the player win the game, we break the function into two part. One is checkFive and the other is checkList. The checkList function count the same cell on every row and column and the checkFive get the information collect from checkList. The new checkFive function is listed below:

```haskell
checkFive :: [[Cell]] -> Cell -> Bool
checkFive xs x | xs == [] = False
               | otherwise =
     let
       checkLoop lst cnt | cnt >= 5          = True
                         | lst == []         = checkFive (tail xs) x
                         | head lst == x     = checkLoop (tail lst) (cnt+1)
                         | otherwise         = checkLoop (tail lst) 1
     in
       checkLoop (head xs) 1
```

As for function showBorad, We change the layout of the codes to make it clear.
Old version:

```haskell
showBoard :: Board Cell -> IO ()
showBoard (Board (x:xs)) =
    let
        colMark [] idx = putStrLn ""
        colMark (y:ys) idx | idx < 10 = do putStr ((show idx) ++ "  ")
                                           colMark ys (idx+1)
                           | idx >= 10 = do putStr((show idx) ++ " ")
                                            colMark ys (idx+1)
        rowMark [] idx = return ()
        rowMark (y:ys) idx | idx < 10 = do putStr ((show idx) ++ " ")
                                           showCell y idx
                                           rowMark ys (idx+1)
                           | idx >= 10 = do putStr ((show idx) ++ "")
                                            showCell y idx
                                            rowMark ys (idx+1)
        showCell [] idx     = print idx
        showCell (y:ys) idx | y == Blank = do putStr " * "
                                              showCell ys idx
                            | y == Black = do putStr " O "
                                              showCell ys idx
                            | y == White = do putStr " X "
                                              showCell ys idx
    in
        do
            putStr "   "
            colMark (x:xs) 1
            rowMark (x:xs) 1
            putStr "   "
            colMark (x:xs) 1
```

Final version:

```haskell
-- Print the column number
colMark :: [[Cell]] -> Int -> IO ()
colMark [] idx = putStrLn ""
colMark (y:ys) idx | idx <  10 = putStr ((show idx) ++ "  ") >> colMark ys (idx+1)
                   | idx >= 10 = putStr ((show idx) ++ " " ) >> colMark ys (idx+1)
-- Print the row number
rowMark :: [[Cell]] -> Int -> IO ()
rowMark [] idx = return ()
rowMark (y:ys) idx | idx <  10 = putStr ((show idx) ++ " ") >> showCell y idx >> rowMark ys (idx+1)
                   | idx >= 10 = putStr ((show idx) ++ "" ) >> showCell y idx >> rowMark ys (idx+1)
-- Print the cells
showCell :: [Cell] -> Int -> IO ()
showCell [] idx = print idx
showCell (y:ys) idx | y == Blank = putStr " * " >> showCell ys idx
                    | y == Black = putStr " O " >> showCell ys idx
                    | y == White = putStr " X " >> showCell ys idx
showBoard :: Board Cell -> IO ()
showBoard (Board (x:xs)) = putStr "   " >> colMark (x:xs) 1 >> rowMark (x:xs) 1 >>
                           putStr "   " >> colMark (x:xs) 1
```

As for function updateBorad, I add a new function cellColor to use generalization to factor out the repeated patterns.
Old version:

```haskell
updateBoard :: Board Cell -> Int -> Int -> Player -> Board Cell
updateBoard (Board x) col row (Human Black ) =
    Board (take (row-1) x ++ [newRow] ++ drop row x) where
      newRow  = take (col-1) (x !! (row-1)) ++ [Black] ++ drop col (x !! (row-1))
updateBoard (Board x) col row (Human White) =
    Board (take (row-1) x ++ [newRow'] ++ drop row x) where
      newRow' = take (col-1) (x !! (row-1)) ++ [White] ++ drop col (x !! (row-1))
updateBoard (Board x) col row (AI Black ) =
    Board (take (row-1) x ++ [newRow] ++ drop row x) where
      newRow  = take (col-1) (x !! (row-1)) ++ [Black] ++ drop col (x !! (row-1))
updateBoard (Board x) col row (AI White) =
    Board (take (row-1) x ++ [newRow'] ++ drop row x) where
      newRow' = take (col-1) (x !! (row-1)) ++ [White] ++ drop col (x !! (row-1))
```

Final version:

```haskell
updateBoard :: Board Cell -> Int -> Int -> Player -> Board Cell
updateBoard (Board x) col row player =
    Board (take (row-1) x ++ [newRow] ++ drop row x) where
      newRow  = take (col-1) (x!!(row-1)) ++ [cellColor player] ++ drop col (x!!(row-1))
```