```
     CS 3360 - Design and Implementation of Programming Languages

          PROJECT 3: FUNCTIONAL PROGRAMMING WITH HASKELL
             (File $Date: 2018/11/17 18:59:37 $)
```

Due: December 4, 2018

This assignment may be done in pair. If you work in pair, however, you need to fill out the contribution form (see the course website).

The purpose of this assignment is to understand the concepts of functional programming and have a taste of it in Haskell [1].

You are to to write a Haskell program for playing Connect Four games. The Connect Four game is a two-player connection game in which the players take turns dropping their discs from the top into a seven-column, six-row vertically suspended grid [Wikipedia]. The pieces fall straight down, occupying the next available space within the column. The aim of the game is to connect four of one's own discs next to each other vertically, horizontally, or diagonally before the opponent.

As in the AspectJ project, you will use the model-view-control (MVC) pattern. Your Haskell program must consist of two modules, one for the game model (M) and the other for a console-based UI (VC). To write the second module, you will need to learn about the basic I/O operations in Haskell (refer to Haskell tutorials such as https://www.haskell.org/tutorial/io.html). In Haskell, an I/O function may call non-I/O functions, but a non-I/O function may not call I/O functions. You shouldn't include any I/O function in the first module (M).

Do not use any library function other than the standard Prelude functions that are automatically imported into every Haskell module (see Part II below for an exception).

Part I. (65 points) Develop a Haskell module named Board to model a Connect Four board and two players. As said earlier, the idea is to separate the model part of your program from the UI part to be developed in Part II below. Thus, no UI (especially, I/O) function should be defined in this module. The following functions are suggested to be written in this module. They will be useful in writing the UI module in Part II.

1. (8 points) Creating a board and players.

```
   mkBoard m n

     Return an empty mxn board, where m and n are positive numbers
     denoting the numbers of columns and rows, respectively. A 1-based
     index will be used to denote and access a specific column and row
     of a board. But, it is up to you to come up with a concrete
     representation of a board, e.g., a nested list.
   mkPlayer = 1
     Return the first player. You may choose your own representation,
     e.g., 1 as done above.

   mkOpponent = 2
     Return the second player (the opponent). You may choose your own
     representation, e.g., 2 as done above.
```

2. (20 points) Checking a board and dropping a disc

```
   dropInSlot bd i p
     Drop a player p's disc in a slot (column) i of a board bd. The
     specified slot is assumed to have an empty place to hold the
     dropped disc (see isSlotOpen below).

   isSlotOpen bd i
     Is a slot (column) i of a board bd open in that it can hold an
     additional disc?

   numSlot bd
     Return the number of columns of a board bd.

   isFull bd
     Is the given board bd full in that there is no empty place?
```

3. (25 points) Determining the outcome

```
   isWonBy bd p
     Is the game played on a board bd won by a player p?
```

4. (12 points) Converting a board to a string for printing

```
   boardToStr playerToChar bd

     Return a string representation of a board bd. It is a
     higher-order function. The first argument (playerToChar) is a
     function to convert a player to a character representation, e.g.,
     'O' or 'X'. A formatted sample return value is shown below
     (assuming that one player is mapped to 'O' and the other to 'X'
     by the playerToChar function) .

     ". . . . . . .\n
      . . . . . . .\n
      . . . . . . .\n
      O . . . . . .\n
      O X . . . . .\n
      O X O X O X ."
```

Part II. (35 points) Develop a Haskell module named Main that provides a console-based UI for playing a Connect Four game. Define the following functions.

1. (15 points) Reading user inputs and printing outputs.

```
   playerToChar p
     Return a character representation of a player p. It returns a
     Char value. This function is used to print the current state of a
     board (see the boardToStr function above).

   readSlot bd p
     Read a 1-based index of an open slot of a board bd for a player p
     to drop her disc. The function reads inputs from the standard
     input (stdin) and returns an IO value such as IO(Int) or
     IO(Integer).

   The following IO functions may be useful.

   putStr, putStrLn - print a string to the standard out
   getLine - read a line from the standard in
   reads:: [(Integer, String)] - parse an Integer from a string

   For example, the following IO function reads lines from stdin
   until a positive Integer is read.

   getX = do
     putStrLn "Enter a positive value?"
     line <- getLine
     let parsed = reads line :: [(Integer, String)] in
       if length parsed == 0
       then getX'
       else let (x, _) = head parsed in
         if x > 0
         then return x
         else getX'
     where
       getX' = do
         putStrLn "Invalid input!"
         getX
```

2. (20 points) Playing a game

```
   main
     Main function to play a Connect Four game by two players.
     It returns an IO() value. The dimension of the board is 7x6, and
     user inputs are read from the standard input (see the readSlot
     function below) and outputs like the board state and the game
     outcome are printed on the standard output. For Haskell I/O, you
     will need to import the System.IO module.
```

Part III. (20+ bonus points) You may earn bonus points by implementing the following features. However, your bonus points count only when you complete all the previous functions for regular points.

1. (5 points) Let a player be able to quit a game by entering a special value, say -1, for the slot index in the readSlot function above.

2. (15+ points) Support a strategy game to play a game against a

computer by implementing a computer move strategy. An easiest way to add this feature is to define a strategy function, say genRandomSlot, and use it in place of the readSlot function when it is the opponent's turn. Use the System.Random module to generate a random value, e.g., x <- randomRIO(1,7) to generate a random number between 1 and 7, inclusive.

TESTING

You code should run with the Hugs Haskell interpreter available from https://www.haskell.org/hugs/. That is, your code should be compilable with GHC 6.6 or higher. If you use the Haskell Platform available from https://www.haskell.org/platform/, do not use any ghci-specific features.

WHAT AND HOW TO TURN IN

Submit your source code files along with any supporting documents through the Assignment Submission page found in the Homework section of the course website. The page will ask you to zip all your files and upload a single archive file. The zip archive file should include only a single directory named YourFirstNameLastName which contains all your source code files and other support files needed to run your program. You should submit your work by 11:59 pm on the due date.

If you work in pair, make only one submission through the Assignment Submission page by specifying both names during the submission; make sure to include the contribution form in your submission.

GRADING

You will be graded, in part, on how clear your code is. Excessively long code will be penalized: don't repeat code in multiple places. Your code should be well documented and sensibly indented so it is easy to read.

Be sure your name is in the comments in your code.

REFERENCES

[1] Alejandro Seraano Mena, Beginning Haskell, A Project-Based Approach, Apress, 2014. Free ebook through UTEP library.