Plot 4 in Haskell

Srineet Sridharan

October 5, 2002

```
> import Maybe
> import Array
> import Char
> import List
> import GraphicsUtils
> import GameTreeSearch
```

1 Introduction

This document is the literate source of the game Plot 4 written in Haskell. It contains the entire source code of the program.

2 Plot 4

Plot 4, also called Connect 4, is a simple game. You have a grid (in our case with seven columns and six rows). There are two players, one having red coins and the other having yellow coins. The players take turns alternately to drop a coin in any column. The player who gets four of his coins in a line (vertical or horizontal or diagonal), wins.

This program provides four player modes.

1. Human
2. Low Skilled Computer
3. Medium Skilled Computer
4. High Skilled Computer

3 Motivation

I wrote this program when I got fascinated by game tree search. I wanted to try out the algorithm and see what results it shows.

The game tree search routines have been implemented as a separate library (there's just one function there as I write this, but I plan to add more). The wonderful higher order nature of Haskell enables one to capture computational patterns in a higher order function. Therefore, the game tree search functions in the library can be used for any game with two players taking alternate turns.

4 Haskell Graph Library

For all the graphics in the game, I used the Haskell Graphics Library by Alaistar Reid (http://www.haskell.org/graphics/). Its a real nice and easy library for building simple images - easy to setup and use.

Only those features in the library which work on both Microsoft Windows, and X11 are used.

5 Building the source

The source is made up of two files.

1. Plot4.lhs contains the main functions and data structures for the game.
2. GameTreeSearch.lhs contains the functions for game tree search.

If the Haskell Graphics Library is installed in the directory HGL_DIR, the command to build the source is:

```
ghc -i$(HGL_DIR)\lib\win32 -package concurrent
        -package win32 --make plot4.lhs -o plot4.exe
```

I named the output as plot4.exe on Windows, the name an be changed to suit other tastes and platforms.

Also, it'll have to be

```
-i$(HGL\_DIR)\lib\X11
```

instead of win32, for Unix platforms.

6 The Board

This section discusses the representation of the board, and gives the related useful functions.

The entire grid is represented by the type Board. It is a two dimensional array (array of arrays). The first index is the column, and the second is the row. The top left cell of the board has column number 1, and row number 1. The column number increases as you go right, and the rown number increases as you go down. The array entries are of type Maybe Player. If there's no coin, the cell will contain Nothing, otherwise it'll be Just PRed or Just PYellow.

```
> type Board    = Array Int (Array Int (Maybe Player))

> bNumRows, bNumCols    :: Int
> bNumRows  = 6
> bNumCols  = 7
> bCentreCol    = (bNumCols + 1) `div` 2
```

The function initBoard gives an empty board.

```
> initBoard :: Board
> initBoard = let
>   colArray    = listArray (1, bNumRows) (repeat Nothing)
>   in
>   listArray (1, bNumCols) (repeat colArray)
```

The functions below are for accessing individual cells of the board. bCell gives the entry at a certain cell in the given board. putBCell returns a new board updated with a new provided entry, in a provided position.

```
> bCell :: Board -> (Int, Int) -> Maybe Player
> bCell bd (col, row)   = bd ! col ! row

> putBCell  :: Board -> (Int, Int) -> Player -> Board
> putBCell bd (col, row) p  =
>   bd // [(col, (bd ! col) // [(row, Just p)])]
```

colTop is a function that returns the top vacant cell in the given column. If the collumn is full, it'll return 0, otherwise it'll return a number between 1 and bNumRows.

```
> colTop    :: Board -> Int -> Int
> colTop bd col = let
>   fullCellsInCol  =
>       [row' | (row', Just _) <- assocs (bd ! col)]
>   in
>   case fullCellsInCol of
>       [] -> bNumRows
>       _  -> (head fullCellsInCol) - 1
```

isBoardFull is a function that checks whether the all the cells in the board are filled.

```
> isBoardFull   :: Board -> Bool
> isBoardFull bd    =
>   (and . map (isJust. bCell bd))
>       (zip [1..bNumCols] (repeat 1))
```

isValidMove is a function that checks whether inserting a coin in any column “move” is legal - it will be legal if that column is not already full.

```
> isValidMove   :: Board -> Int -> Bool
> isValidMove bd move   = colTop bd move > 0
```

Below are functions that give the list of positions of all collinear cells in the board. The main function is collinearCells, which uses the other functions given below. So we have a list of list of positions. Each member list in this list of lists, contains positions of collinear cells.

```
> columnPts   :: [[Point]]
> columnPts   = map (\col -> zip (repeat col) [1..bNumRows])
>                 [1..bNumCols]

> rowPts  :: [[Point]]
> rowPts  = map (\row -> zip [1..bNumCols] (repeat row))
>           [1..bNumRows]

> diags1Pts :: [[Point]]
> diags1Pts = let
>   genDiag (a, b)  = (takeWhile (\(x, y) -> x <= bNumCols &&
>                                             y <= bNumRows).
>                     iterate (\(x, y) -> (x + 1, y + 1)))
>                     (a, b)
>   diags'  = map genDiag (zip [1..bNumCols] (repeat 1))
>   diags'' = map genDiag (zip (repeat 1) [2..bNumRows])
>   in
>   diags' ++ diags''

> diags2Pts :: [[Point]]
> diags2Pts = let
>   genDiag (a, b)  = (takeWhile (\(x, y) -> x <= bNumCols &&
>                                             y >= 1).
>                     iterate (\(x, y) -> (x + 1, y - 1)))
>                     (a, b)
>   diags'  = map genDiag (zip [1..bNumCols] (repeat bNumRows))
>   diags'' = map genDiag (zip (repeat 1) [2..bNumRows])
>   in
>   diags' ++ diags''

> collinearCells    :: [[Point]]
> collinearCells    =
>   columnPts ++ rowPts ++diags1Pts ++ diags2Pts
```

7 Player

This section gives the representation of the players, controlling red and yellow coins, and related functions.

Given below is the data declaration for Player which is straight forward.

```
> data Player   = PRed  | PYellow
>   deriving Eq
```

otherPlayer is a simple function, and does what it name says.

```
> otherPlayer   :: Player -> Player
> otherPlayer PRed  = PYellow
> otherPlayer PYellow   = PRed
```

playerColor gives the color of the coin for a player. The color is the actual RGB value.

```
> playerColor   :: Player -> RGB
> playerColor PRed  = red
> playerColor PYellow   = yellow
```

Given below is the data declaration for the various player modes. The data type Mode is made an instance of show, and the instance is given. isHuman is a simple function that tells whether a mode is human or not.

```
> data Mode = MHuman
>           | MCompLow
>            | MCompMid
>            | MCompHigh

> modes = [MHuman, MCompLow, MCompMid, MCompHigh]

> instance Show Mode where
>   show MHuman = "Human"
>   show MCompLow    = "Low Skilled Computer "
>   show MCompMid    = "Medium Skilled Computer "
>   show MCompHigh   = "High Skilled Computer "

> isHuman   :: Mode -> Bool
> isHuman MHuman    = True
> isHuman _ = False
```

8 The Game State

This section describes the representation of the state of the game at any given point. The data type is called GameState. It contains the following fields.

- gNextTurn: The player to make the next move.
- gNextCoinCol: This is just an (unnecessary?) field that contains the column above which the next coin to be played hovers. Currently, its always the middle column!
- gBoard: The state of the board.
- gModes: The modes of the two players.

Note that we allow the modes of the players to be changed at any point in the game. So the player modes have to be made part of the game state.

```
> data GameState    = GS { gNextTurn    :: Player,
>                          gNextCoinCol :: Int,
>                          gBoard       :: Board,
>                          gModes       :: (Mode, Mode)
>                        }
```

initState is just a function that returns the default initial state. It accepts the player modes as an argument.

```
> initState :: (Mode, Mode) -> GameState
> initState modes   =
>   GS { gNextTurn  = PRed,
>        gNextCoinCol   = bCentreCol,
>        gBoard = initBoard,
>        gModes = modes}
```

nextPlayerMode returns the mode of the player whose turn is next.

```
> nextPlayerMode    :: GameState -> Mode
> nextPlayerMode (GS {gNextTurn = PRed, gModes = (a, _)}) = a
> nextPlayerMode (GS {gNextTurn = PYellow, gModes = (_, b)}) = b
```

makeMove is a function that updates the game state with the given move that consists of dropping a coin in column col. The board is updated. The player who is to play next changes.

```
> -- col must not be already full
> makeMove  :: GameState -> Int -> GameState
> makeMove gs col   = let
>   bd' = putBCell (gBoard gs)
>                  (col, colTop (gBoard gs) col)
>                  (gNextTurn gs)
>   in
>   gs { gBoard = bd',
>        gNextTurn = otherPlayer (gNextTurn gs),
>        gNextCoinCol = bCentreCol}
```

9 Game Ended?

This section contains the function that checks whether the game has ended. The game is said to end when either the board is full, or if some player has got four coins in a straight line. The function returns a value of type (Maybe (Maybe ([Point], Player))).

It behaves as follows.

- It returns Nothing if the game has not ended.
- It returns Just Nothing if the game has ended because the board is full - nobody's won.
- It returns Just (Just (the-four-collinear-cells, the-winner)) if somebody wins.

findWin is the function that given a list of collinear cells, determines whether any four adjecent positions have the coins belonging to the same player. isGameEnd calls findWin on each list of collinear positions in collinearCells.

```
> isGameEnd :: Board -> (Maybe (Maybe ([Point], Player)))
> isGameEnd bd  = let
>   mw  = (listToMaybe.mapMaybe (findWin bd)) collinearCells
>   in
>   if isJust mw
>     then Just mw
>     else if isBoardFull bd
>            then Just Nothing
>            else Nothing
```

findWin is the function that, given a list of collinear cells, determines whether any four adjecent positions have the coins belonging to the same player. It works by finding the board entry at each position, then grouping them based on the board entry, and checking whether there's any coin containing group, of length greater than four (which indicates a win). If it finds such a group, it returns the four points in the group, and the player to whom those coins belong. Otherwise it just returns Nothing.

```
> findWin   :: Board -> [Point] -> Maybe ([Point], Player)
> findWin bd pts    = let
>   ptPlayers   = (dropWhile (\xs -> length xs < 4) .
>                  filter (\xs -> (not.null) xs &&
>                                 (isJust.snd.head) xs) .
>                  groupBy
>                      (\(_, mp1) (_, mp2) ->mp1 == mp2))
>                  (zip pts (map (bCell bd) pts))
>   in
>   case ptPlayers of
>        [] -> Nothing
>        _  -> Just ([pt | (pt, _) <- head ptPlayers],
>                  (fromJust.snd.head.head) ptPlayers)
```

10 All the Pictures

This section contains functions that creates values of type Graphic (which is part of the Haskell Graphic Library), and represents the pictures drawn on screen. The functions are quite simple, and just build the pictures by combining the graphics elements at various positions.

Given next are some constants. cellSide is the length of the side of a board cell. coinDia is the diameter of a coin. hoverCoiny is the y-position for the coin that floats above the board, indicating the next turn. bLeftX and bTopY are the pixel positions of the top left corner of the board. gridColor and myTxtColor are the RGB values of the colours used for the grid, and for all the text, respectively.

```
> cellSide, coinDia, hoverCoinY, vCoinFall :: Int
> cellSide      = 40
> coinDia       = cellSide
> hoverCoinY    = 30
> vCoinFall     = cellSide ‘div‘ 5

> bdTopY, bdLeftX  :: Int
> bdTopY    = 80
> bdLeftX   = (winX - (bNumCols * cellSide)) ‘div‘ 2

> gridColor     = RGB 200 140 100
> myTxtColor    = RGB 240 170 170
```

The functions below are used to map the (col, row) co-ordinates for a board cell to the pixel values on the screen display.

```
> colToX    :: Int -> Int
> colToX col    = bdLeftX + (col - 1) * cellSide

> rowToY    :: Int -> Int
> rowToY row    = bdTopY + (row - 1) * cellSide

> colRowToXY    :: (Int, Int) -> (Int, Int)
> colRowToXY (c, r) = (colToX c, rowToY r)
```

screenPic is the function that returns the picture for the current screen. As can be seen, it just combines other functions like gridPic, instrPic etc. If the game has ended with a win, a thick black line is drawn joining the centres of the four coins that are in a line.

```
> screenPic :: GameState -> Graphic
> screenPic gs  = let
>   coinRad = coinDia ‘div‘ 2
>   coinCentre (col, row)   =
>       (colToX col + coinRad, rowToY row + coinRad)
>   winPic  =
>       case isGameEnd (gBoard gs) of
>           (Just (Just (pts, _))) ->
>                   withRGB black $
>                   thickLine 6 ((coinCentre.head) pts)
>                                ((coinCentre.last) pts)
>           _   -> emptyGraphic
>   in
>   winPic `overGraphic`
>   instrPic gs `overGraphic`
>   gridPic `overGraphic`
>   bdCoinsPic (gBoard gs) `overGraphic`
>   hoverCoinPic (gNextTurn gs) (gNextCoinCol gs)
```

coinPic draws a coin in the appropriate colour.

```
> coinPic   :: Player -> Point -> Graphic
> coinPic player (x, y)    =
>   withRGB (playerColor player) $
>   ellipse (x, y) (x + coinDia, y + coinDia)
```

gridPic draws the vertical and horizontal thick lines of the grid.

```
> gridPic   :: Graphic
> gridPic   = let
>   foov col    = thickLine 3
>                     (colToX col, rowToY 1)
>                     (colToX col, rowToY (bNumRows + 1))
>   fooh row    = thickLine 3
>                     (colToX 1, rowToY row)
>                     (colToX (bNumCols + 1), rowToY row)
>   in
>   withRGB gridColor $
>   foldl overGraphic emptyGraphic
>       (map foov [1..(bNumCols + 1)] ++
>        map fooh [1..(bNumRows + 1)])
```

bdCoinsPic draws all the coins on the board. This is superimposed over gridPic.

```
> bdCoinsPic    :: Board -> Graphic
> bdCoinsPic bd = let
>   pics = [coinPic (fromJust mpl) (colToX col, rowToY row) |
>                           col <- [1..bNumCols],
>                           row <- [1..bNumRows],
>                           let mpl   = bCell bd (col, row),
>                           isJust mpl]
>   in
>   foldl overGraphic emptyGraphic pics
```

instrPic displays the text instructions about what the user has to do next.

```
> instrPic  :: GameState -> Graphic
> instrPic gs   = let
>   coinRad = coinDia `div` 2
>   commonInstr = text (10, winY - 20)
>       "Press C to change modes. Esc to Quit."
>
>   instr (m@MHuman)    =
>                     (text (10, winY - 40)
>                       (show m ++
>                       " Player: Press any digit from 1 to "
>                       ++ (show bNumCols))) `overGraphic`
>                     commonInstr
>   instr m = text (10, winY - 40)
>                (show m ++ " Player: Press any key")
>            `overGraphic` commonInstr
>   colNumbers   =
>       withRGB gridColor $ foldl overGraphic emptyGraphic
>                    (map
>                     (\i -> text
>                            (colToX i + coinRad,
>                               rowToY (bNumRows + 1) + coinRad)
>                            (show i))
>                            [1..bNumCols])
>   in
>   withTextColor myTxtColor $
>       colNumbers `overGraphic`
>       (instr (nextPlayerMode gs) `overGraphic`
>       commonInstr)
```

hoverCoinPic draws the floating coin, that represents the next turn.

```
> hoverCoinPic  :: Player -> Int -> Graphic
> hoverCoinPic pl col  = coinPic pl (colToX col, hoverCoinY)
```

thinkingPic shows a text that says "Thinking..." over the screenPic. This is used to tell the user that the computer player is busy calculating the next move.

```
> thinkingPic   :: GameState -> Graphic
> thinkingPic gs  =
>   withRGB myTxtColor $
>       text (winX `div` 3, 4 ) "Thinking..." `overGraphic`
>       screenPic gs
```

playAgainPic and quitPic just add the corresponding text above the normal screenPic.

```
> playAgainPic  :: GameState -> Graphic
> playAgainPic  gs = withRGB myTxtColor $
>   text (winX `div` 3, 4 )
>        "Play again? (y/n)"
>   `overGraphic`
>   screenPic gs
```

```
> quitPic  :: GameState -> Graphic
> quitPic  gs = withRGB myTxtColor $
>   text (winX `div` 3, 4)
>        "Do you really want to quit? (y/n)"
>   `overGraphic`
>   screenPic gs
```

startPic is the first screen that shows the name of the game.

```
> startPic  :: Graphic
> startPic  = let
>   cx  = winX `div` 2
>   cy  = winY `div` 2
>   in
>   withTextColor myTxtColor $
>   text (cx - 30, cy - 80) "P L O T  4" `overGraphic`
>   text (cx - 70, cy - 30) "by Srineet Sridharan"  `overGraphic`
>   text (cx - 45, cy + 0) " August 2002"  `overGraphic`
>   text (cx - 130, cy + 70)
>       "Implemented in Haskell: www.haskell.org"
```

11 Mode Selection

getModeChanged is the function that displays the screen for player mode selection when that key for changing modes is pressed. The user is presented with the list of modes for each player, and is asked to choose the mode. The function returns Nothing if the user hits escape. Otherwise, it returns the modes of the two players.

```
> getModeChanged    :: Window -> IO (Maybe (Mode, Mode))
> getModeChanged w  = let
>   redPic  = withRGB red $ text (10, 60) "Player Red:"
>   yellowPic   = withRGB yellow $ text (10, 60) "Player Yellow:"
>   choices startx starty = foldl overGraphic emptyGraphic $
>       map (\(num, mode) -> text (startx, starty + 20 * num)
>                           (show num ++ ". " ++ (show mode)))
>           (zip [1..] modes)
>   redChoices  = withRGB myTxtColor $
>                (commonPic ‘overGraphic‘
>                  redPic ‘overGraphic‘
>                  choices 10 80)
>   yellowChoices   = withRGB myTxtColor $
>                (commonPic ‘overGraphic‘
>                  yellowPic ‘overGraphic‘
>                  choices 10 80)
>   titlePic    = text (winX ‘div‘ 3, 20) "S E L E C T  M O D E"
>   aimPic      = text (10, winY - 100)
>       "Aim of the game:"  ‘overGraphic‘
>       (text (10, winY - 80)
>           "Get four of your coins in a line. (vertical / horizontal / diagonal)")
>   instrPic    = text (10, winY - 30)
>       ("Press any digit from 1 to " ++ (show.length) modes ++ "...")
>   commonPic   =
>       titlePic ‘overGraphic‘ aimPic ‘overGraphic‘ instrPic
>   in do
>   setGraphic w redChoices
>   k1 <- getKeyWhere w (\k -> isDigitKey k &&
>                 (digitKeyToInt k ‘elem‘ [1..(length modes)]))
>   if isEscapeKey k1
>     then return Nothing
>     else do
>       setGraphic w yellowChoices
>       k2 <- getKeyWhere w (\k -> isDigitKey k &&
>               (digitKeyToInt k ‘elem‘ [1..(length modes)]))
>       if isEscapeKey k2
>           then return Nothing
>           else return $
>               Just (modes !! ((digitToInt.keyToChar) k1 - 1),
>                     modes !! ((digitToInt.keyToChar) k2 - 1))
```

modeSelect is the initial mode selection screen that appears. It is different from getModeChanged in that, if the user presses escape, he is presented with an option to quit the game. In getModeChanged, if the user presses escape, the function just returns nothing, and the game will continue with the previous mode setting.

```
> modeSelect    :: Window -> IO (Maybe (Mode, Mode))
> modeSelect w  = let
>   quitPic = withRGB myTxtColor $
>               text (winX `div` 3, 4)
>                "Do you really want to quit? (y/n)"
>   in do
>   jmodes  <- getModeChanged w
>   case jmodes of
>      Nothing -> do
>          setGraphic w quitPic
>          k <- getKeyWhere w (\k -> isYesOrNoKey k)
>          if isEscapeKey k || isYesKey k
>           then return Nothing
>           else modeSelect w
>      _ -> return jmodes
```

12 The Play Loop

play is the main function for the game. Its a loop where the screen is displayed, input is taken, the move is presented, and the game moves to the next turn.

Notice the use of the function alphaBeta from the Game Tree Search library, to do alpha beta pruning, with depth corresponding to the player mode. minusInfinity and plusInfinity are passed as alpha and beta. The function returns the best move to make, and also the value of the move.

Also notice that thinkingPic is shown before using alphaBeta. And there is a case expression on the move returned returned by alphaBeta, in order to strictly evaluate the next move. So at that point, we know that the evaluation is done, and that we can show the screenPic where the ”thinking” caption will go away.

```
> play  :: Window -> GameState -> IO GameState
> play w gs   = let
>   play'   = do
>       (gs', k) <- getInput w gs
>       if  isEscapeKey k
>         then return gs'
>         else do
>              if isHuman (nextPlayerMode gs')
>                then return ()
>                else setGraphic w (thinkingPic gs')
>              let newCol =
>                   if isHuman (nextPlayerMode gs')
>                     then digitToInt (keyToChar k)
>                     else
>                        fst (alphaBeta otherPlayer
>                                       successors
>                                       evaluate
>                                       (gNextTurn gs')
>                                       (gBoard gs)
>                                       (minusInfinity)
>                                       (plusInfinity)
>                                       (nextPlayerDepth gs'))
>              case newCol of
>               0 -> error "Dummy error in play for strictness"
>               _ -> showMove w (gNextCoinCol gs') gs' newCol
>
>              play w (makeMove gs' newCol)
>   in do
>   setGraphic w (screenPic gs)
>   if isJust (isGameEnd (gBoard gs))
>     then return gs
>     else play'
```

getInput is the function used to get the input at each turn. If its the turn for a human player, it expects not only a number between 1 and bNumCols, but also checks whether a move can be made at that column. For a computer player's turn, any key is allowed. If the key for changing player modes is pressed, the mode change screens will be shown, and the key processing will be done from within this function.

```
> getInput  :: Window -> GameState -> IO (GameState, Key)
> getInput w gs = let
>   humanKeyCond    = \k -> isCharKey k &&
>     (isDigitKey k && (digitKeyToInt k) `elem` [1..bNumCols] &&
>      isValidMove (gBoard gs) (digitKeyToInt k)) ||
>      isModeChangeKey k
>   compKeyCond = (\k -> True)
>   getInputKey = if isHuman (nextPlayerMode gs)
>                   then getKeyWhere w humanKeyCond
>                   else getKeyWhere w compKeyCond
>   in do
>   k <- getInputKey
>   if isModeChangeKey k
>     then do
>       md  <- getModeChanged w
>       let gs' = if isJust md
>                   then gs {gModes = fromJust md}
>                   else gs
>       setGraphic w (screenPic gs')
>       getInput w gs'
>     else
>       return (gs, k)
```

showMove is the function that shows the move being made. The coin moves to the column where it'll be dropped. This is done in loop1. Then the coin drops into the column, to rest above the topmost column in the coin (done in loop2). Note that this function just shows the move, and does not update the game state.

```
> showMove  :: Window -> Int -> GameState -> Int -> IO ()
> showMove w curCol gs moveCol   = let
>
>   step    | curCol < moveCol   = (+1)
>           | otherwise         = (\x->x-1)
>
>   topY    = (rowToY . colTop (gBoard gs)) moveCol
>   player  = gNextTurn gs
>
>   loop1 col   = do
>       delayMilliSecs w 100
>       let pic = screenPic (gs {gNextCoinCol = col})
>       setGraphic w pic
>       if col /= moveCol
>         then loop1 (step col)
>         else return ()
>
>   loop2 (coinx, coiny)    = do
>       delayMilliSecs w winTick
>       let pic = instrPic gs `overGraphic`
>                 gridPic `overGraphic`
>                 bdCoinsPic (gBoard gs) `overGraphic`
>                 coinPic player (coinx, coiny)
>       setGraphic w pic
>       if coiny < topY
>         then loop2 (coinx, coiny + vCoinFall)
>         else return ()
>   in do
>      loop1 curCol
>      loop2 (colToX moveCol, hoverCoinY)
```

13 Intelligence - Using Game Tree Search

In this section we look at the functions that we provide to the Game Tree Search function.

The first is the evaluation function. When the game tree reaches the leaf, it evaluates the position, and returns the value of that position. At each level other than the leaf, a node will pick the move which is best for him (the worst for the opponent). The details of how the game tree search works is not given here.

The evaluation function chosen here evaluates a certain board position for a certain player and returns the value. This is how it works. Each set of collinear board positions is checked. In any such list of positions, if there are four adjecent cells, with atleast one cell containing a coin, and none other containing a coin of the opposite player, it is of interest. The value of this set of four cells, depends upon the number of cells filled with a player's coin. The higher the number of cells containing a player's coin, the higher their absolute value. If all four cells contain the player's coin, that means that the player has won. The sign of the value depends on the player whose coins are in this set of four cells, with respect to whom are we evaluating the board for. For example, if we are evaluating for the red player, and the coins are yellow, the sign of the value is negative, otherwise it is positive.

Now, in each list of collinear points, we process the four cells starting from the first in the list; then the four cells starting from the second in the list and so on, and keep adding the values. So it is a sliding window of size four. The advantage is that even the distribution of coins in the line gets evaluated. For example, consider a row with the leftmost two cells containing a red coin. Add two more red coins next to them to the right, and the player wins. Now consider that row, but with the two cells next to the middle cell containing two red coins. Now there are more ways in which the player can win. (If we consider the case, with three coins in the middle cells in a row, the player is bound to win, because the opponent can only plug one end, and the player will win by putting the fourth coin at the other end). Our sliding window will automatically give a greater value for the second case than the first.

The above calculation of the value for a list of collinear positions is done by the function evalSeq given below.

```
> evalSeq   :: Player -> Board -> [Point] -> Int

> evalSeq _ _ []   = 0

> evalSeq p bd pts  = let
>   pts'    = take 4 pts
>   ptss    = groupBy (==)
>             [fromJust mp | pt <- pts',
>                            let mp   = bCell bd pt,
>                            isJust mp]
>   in
>   if length pts' < 4 || length ptss /= 1
>     then 0 + evalSeq p bd (tail pts)
>     else if (head.head) ptss == p
>            then evalScore ((length.head) ptss) +
>                   evalSeq p bd (tail pts)
>            else (- evalScore ((length.head) ptss)) +
>                   evalSeq p bd (tail pts)
```

evalSeq is called for every list of collinear points and the values are added arithmetically to give the value of the board, for the given player. Notice that even diagonal cells are considered, even if there might be no way immediately to put coins that way, because there aren't enough coins in every column in the diagonal, to directly be able to put a coin in the column, to rest at that height.

The function evaluate given below evaluates the board, by using evalSeq for each list of collinear cells.

```
> evaluate  :: Player -> Board -> Int
> evaluate p bd =
>   foldr (+) 0 (map (evalSeq p bd) collinearCells)
```

The function evalScore is the score depending on the number of cells containing coins, in the chosen set of four cells. Remember the four cells will contain one or more coin of the *same* type. These values were chosen quite arbitrarily.

```
> evalScore :: Int -> Int
> evalScore 1   = 1
> evalScore 2   = 5
> evalScore 3   = 10
> evalScore 4   = 500
```

successors is the function passed to the Game Tree Search routine, to generate all the possible next positions from a given position. It returns a list of pairs, where each pair contains the move made, and the board position attained on making that move.

successors uses makeBdMove to get the board configuration on making a certain move. isValidMove is also used for validity check.

```
> successors    :: Player -> Board -> [(Int, Board)]
> successors pl bd  | (isJust.isGameEnd) bd = []
>                   | otherwise =
>   [(move, makeBdMove pl bd move) | move <- [1..bNumCols],
>                                 isValidMove bd move]

> makeBdMove  :: Player-> Board -> Int -> Board
> makeBdMove pl bd col   =
>   putBCell bd (col, colTop bd col) pl
```

nextPlayerDepth is the depth for Game Tree Search for different player modes.

```
> nextPlayerDepth   :: GameState -> Int
> nextPlayerDepth gs    = case nextPlayerMode gs of
>   MCompLow    -> 2
>   MCompMid    -> 4
>   MCompHigh   -> 6
```

14 Main

This section gives the main function. It is quite straight forward, and calls play repeatedly, unless the user says he wants to quit pressing the escape key, or if the game ends and the user does not want to play again.

The function also opens the window, shows the mode selection screen and so forth.

```
> main  :: IO ()
> main = runGraphics (do
>   w <- openWindowEx
>           "Plot 4" Nothing (winX, winY)
>           DoubleBuffered (Just 20)
>   setGraphic w startPic
>   getKey w
>   let loop' gs = do
>          gs <- play w gs
>          if (isJust.isGameEnd.gBoard) gs
>            then do
>              setGraphic w (playAgainPic gs)
>              k <- getKeyWhere w
>                   (\k -> isYesOrNoKey k)
>              if isEscapeKey k || isNoKey k
>                then return ()
>                else loop
>            else do
>              setGraphic w (quitPic gs)
>              k <- getKeyWhere w
>                   (\k -> isYesOrNoKey k)
>              if isEscapeKey k || (isYesKey k)
>                then return ()
>                else loop' gs
>       loop    = do
>           jmodes <- modeSelect w
>           case jmodes of
>               Nothing -> return ()
>               _   -> loop' (initState(fromJust jmodes))
>   loop)
```

15 Utility Functions and Constants

Given below are the various miscellaneous utility functions and constants.
Here are the RGB values for some used colours.

```
> red, yellow    :: RGB
> red       = RGB 255 0 0
> yellow    = RGB 255 255 0
> blue      = RGB 0 0 255
> black     = RGB 0 0 0
```

Here's a delay function.

```
> delayMilliSecs    :: Window -> Int -> IO ()
> delayMilliSecs w  0   = return ()
> delayMilliSecs w  n   = do
>   getWindowTick w
>   delayMilliSecs w (n - winTick)
```

thickLine is used to draw a line of given thickness from one point to another.

```
> thickLine :: Int -> Point -> Point -> Graphic
> thickLine thickness (x1, y1) (x2, y2) = let
>   ft  = (fromIntegral thickness) :: Float
>   halfFt  = ft / 2
>   fx1 = (fromIntegral x1) :: Float
>   fy1 = (fromIntegral y1) :: Float
>   fx2 = (fromIntegral x2) :: Float
>   fy2 = (fromIntegral y2) :: Float
>   theta   = atan2 (fy2 - fy1) (fx2 - fx1)
>   theta'  = pi / 2 + theta
>   p1, p2, p3, p4  :: Point
>   p1  = (round (fx1 + halfFt * (cos theta')),
>          round (fy1 + halfFt * (sin theta')))
>   p2  = (round (fx1 - halfFt * (cos theta')),
>          round (fy1 - halfFt * (sin theta')))
>   p3  = (round (fx2 - halfFt * (cos theta')),
>          round (fy2 - halfFt * (sin theta')))
>   p4  = (round (fx2 + halfFt * (cos theta')),
>          round (fy2 + halfFt * (sin theta')))
>   in polygon [p1, p2, p3, p4]
```

getKeyWhere waits for the user to press a key. It accepts the key only if the key satisfies the validity function cond. It however always accepts the escape key.

```
> getKeyWhere   :: Window -> (Key -> Bool) -> IO Key
> getKeyWhere w cond = do
>   k   <- getKey w
>   if cond k || isEscapeKey k
>     then return k
>     else getKeyWhere w cond
```

Next are some useful funcitons used to work with keys and to see whether a digit has been pressed.

```
> digitKeyToInt :: Key -> Int
> digitKeyToInt k   | isCharKey k && (isDigit.keyToChar) k =
>     (digitToInt.keyToChar) k
>                    | otherwise = error "digitKeyToInt: invalid key."

> isDigitKey    :: Key -> Bool
> isDigitKey k  = isCharKey k && (isDigit.keyToChar) k
```

Then come the functions for checking whether some specific keys are pressed.

```
> isYesOrNoKey :: Key -> Bool
> isYesOrNoKey k  = keyToChar k `elem` ['y', 'Y', 'n', 'N']

> isYesKey  :: Key -> Bool
> isYesKey k  = keyToChar k `elem` ['y', 'Y']

> isNoKey  :: Key -> Bool
> isNoKey k  = keyToChar k `elem` ['n', 'N']

> isModeChangeKey   :: Key -> Bool
> isModeChangeKey k = isCharKey k && keyToChar k `elem` ['c', 'C']
```

Here are the window specific constants.

```
> winX, winY    :: Int
> winX  = 500
> winY  = 420
> winTick  = 20
```

The "inifinity" values used to pass as alpha and beta to the Game Tree Search routine.

```
> minusInfinity, plusInfinity   :: Int
> plusInfinity = 32000
> minusInfinity = - plusInfinity
```