

Assignment 3: CPSC 449

November 25, 2016

Abstract

The goal of this assignment is to assess your ability to put together your knowledge of functional programming, and build an actual functional program. There are two parts to this assignment, explained below: a design component and an implementation component. The design component simply stresses the functional programming ideology that a well designed program should be broken down into composites of very simple functions. The implementation component stresses making sure that you can tie down the loose ends and obtain a working program.

1 Problem Description

Most of you have heard of the children's board game "Connect 4" https://en.wikipedia.org/wiki/Connect_Four. It is a two player game where players take turns placing "pieces" into a column of a grid with 7 columns and 6 rows. pieces stack on top of each other. The goal is to create a connection of four pieces in any direction: horizontally, vertically, or diagonally. If you're not familiar with this game you can play it online: <https://www.mathsisfun.com/games/connect4.html>.

Now, as a functional program we could easily write a program to play connect four. However, in functional programming, whenever we see an opportunity for generalization, we take it! Hence, we will be playing connect $n - m - k$!! Connect $n - m - k$ has the same goal as connect four: you want to connect n pieces, horizontally, vertically, or diagonally in a grid with m rows and k columns. For example, if you ever want a delusional confidence boost, try playing connect $1 - 1 - 1$.

Let's talk about the data types we will use to represent the game. First, there are two players, traditionally yellow and red.

```
data Piece = Yellow | Red
```

These will represent a piece in the board. The "board" is represented as a list of columns of pieces. For this, we don't really need a new type, a simple type alias will suffice.

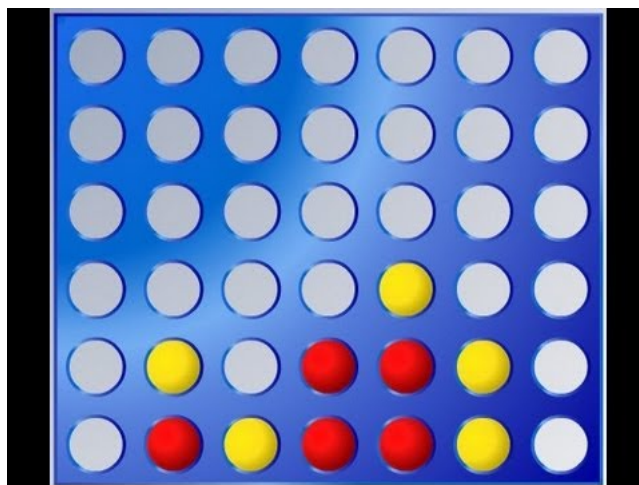


Figure 1: Connect 4-6-7. Source https://www.youtube.com/watch?v=5e2hTZIE0_g

```
type Column = [Piece]
type Board = [Column]
```

You must use these types in your program (you don't have to alias them, but it is a good idea for documentation's sake).

Let's give an example board representation. Consider the following board in Figure 1. We will assume it is connect 4 – 6 – 7

This board is represented as

```
bdFig1 =
  let
    c1 = []
    c2 = [Red, Yellow]
    c3 = [Yellow]
    c4 = [Red, Red]
    c5 = [Red, Red, Yellow]
    c6 = [Yellow, Yellow]
    c7 = []
  in
    [c1, c2, c3, c4, c5, c6, c7]
```

We chose to model the board pictured with columns from left to right, and we list the pieces in a column from bottom to top. One cautionary word: the 6, 7 are determined from the shape of the board; however, the number 4 or the winning number of pieces was chosen arbitrarily. This game could very well be connect 3 – 6 – 7 as well. If it were connect 2 – 6 – 7, then yellow would have 3

wins and red would have 6, hence this board configuration would be nonsense in connect 2 – 6 – 7.

Your goal in this assignment, is ultimately to implement the functions in the design component, that could be used to automate playing connect $n - m - k$.

One final thing, the state of the board: the board itself and whether the last player was Red or Yellow, as well as the dimensions of the game.

```
data BoardState = BS
  {theBoard :: Board,
   lastMove :: Piece,
   numColumns :: Int,
   numRows :: Int,
   numToConnect :: Int}
```

2 Design Component

The design component will be read by your instructor and TAs. We will be looking to ensure that your design typechecks, and makes sense. The goal of this segment is to break down a few big goals into smaller goals by implementing big functions as “composites” of smaller functions. This is often called top-down functional design.

Before diving in, we will make use of a common trick in functional programming: using undefined!

In Haskell, there is a function called undefined, that has any type you want. It is predefined, and you can use it. You could define your own as

```
fix :: (a -> a) -> a
fix f = f (fix f)

undefined :: a
undefined = fix id
```

Both of these functions are pure nonsense, but they can help with designing programs because undefined can literally have any type it wants. At any rate, it is better to use the builtin undefined.

Using undefined, we can give partial implementations of programs, by breaking big functions into little functions, and leaving the little functions defined as undefined, so that we can come back and fill them in later.

For example, we could give a candidate design for matrix multiplication as:

```
type Matrix a = [[a]]
multiplyMatrices :: Matrix Double -> Matrix Double -> Matrix Double
multiplyMatrices xs ys = multMat xs (transpose ys)

multMat [] _ = []
multMat (x:xs) ys = map (\a -> dotProduct x a) ys : multMat xs ys
```

```
dotProduct :: [Double] -> [Double] -> Double
dotProduct = undefined

transpose :: Matrix a -> Matrix a
transpose = undefined
```

That said, defining multMat as part of the design is actually a bit heavy since there is case distinction and recursion going on there. A better thing to do at the design phase is to write multMat as a foldr.

```
multMat :: Matrix Double -> Matrix Double -> Matrix Double
multMat xs ys = foldr (\x zs -> map (dotProduct x) ys : zs) [] xs
```

Basically, you want to write “simple” functions that don’t explicitly make a choice of cases (i.e. pattern matching, case, or if-then-else). And this is a judgment call: basically if you can figure out the implementation in a minute and it is just functions composed with and applied to other functions, then it probably goes in the design component.

Enough about philosophy! Your job is to design the following functions that can be used in building a connect $n - m - k$ program.

```
makeMove :: BoardState -> Int -> Maybe BoardState
```

makeMove takes in a BoardState and a desired column number into which to drop a piece. The color of the piece being dropped depends on the color of the lastMove recorded in the current board state. Ultimately, we would like to return a new board state – the result of dropping a Piece into a column, as well as updated lastMove. However there is something that could go wrong: the number of columns is recorded in our game state and playing a piece into a column could, potentially, cause a column to have so many pieces that it exceeds the dimension – hence we return Just the new board state when there is no issue, otherwise we return Nothing. **NOTE:** the columns are indexed from 1! So makeMove bs 1 would attempt to put a piece in the first column.

By the way, this already suggests a great place for composition! You check that the move is legal, and then you make the move!

```
checkWin :: BoardState -> Maybe Piece
```

This function is really simple. You examine the board, and see if a player has a win as in there are numToConnect consecutive pieces of a single color in a horizontal, vertical, or diagonal line.

If we assume that the game is played correctly, there can only be one winner. Hence, under this modest precondition: you may without loss of generality, search if Red has a win and then search if Yellow has a win (mmm more composition). You can of course break these two checks into four simpler checks (hint: I use four simpler checks here).

However, given our assumption, we can make yet a further simplification! You only really need to check that the player that moved last won. It is your choice how you check, my implementation made this simplification.

These two functions are all that there really is to connect $n-m-k$. We don't know who makes a move: maybe it is a human, or maybe it is a computer. At any rate, we have the game logic at this point. Two players could now submit a move, the move is checked for consistency, and then whether or not there is a win is checked.

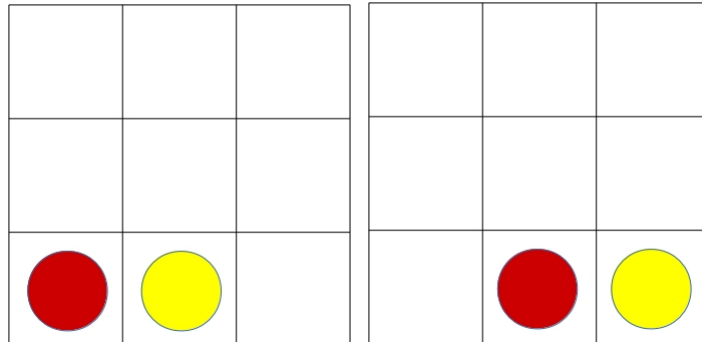
It's almost like, well thought out types completely determines the solution we come up with. Your goal is to place the design part of this program into its own haskell source file "\$Name.hs"

3 Implementation Component

Here you will fill in the undefined functions from the design part. In addition, you will have to define the following functions:

```
columns :: BoardState -> [[Maybe Piece]]
rows :: BoardState -> [[Maybe Piece]]
diagonalsForward :: BoardState -> [[Maybe Piece]]
diagonalsBackward :: BoardState -> [[Maybe Piece]]
```

These four functions return the list of columns, rows, and diagonals of a board. The reason we return `[[Maybe Piece]]` is to disambiguate. Suppose we just returned `[[Piece]]`. Then the output of `rows`, in the following two example boards, would be ambiguous, because either way it would be just `[[], [], [Red, Yellow]]`:



Where as when returning `[[Maybe Piece]]`, the output is unambiguous. With the two examples above the outputs are, respectively:










- `[[Nothing, Nothing, Nothing], [Nothing, Nothing, Nothing], [Just Red, Just Yellow, Nothing]]`

- `[[Nothing,Nothing,Nothing],[Nothing,Nothing,Nothing],[Nothing,Just Red,Just Yellow]]`

Columns returns the columns from left to right, pieces bottom to top. Rows returns the rows from top to bottom, pieces left to right.

Now, for diagonals forwards and diagonals backwards. The distinction is the direction: diagonals forward is like a forward slash through the board `/`. Diagonals backward is like a backward slash through the board `\`. For the diagonals, you should start with the leftmost possible diagonal.

For diagonals forward, you should collect along each diagonal left to right, bottom to top. Thus if the board in `3-3-3` is (remember each list is a column):

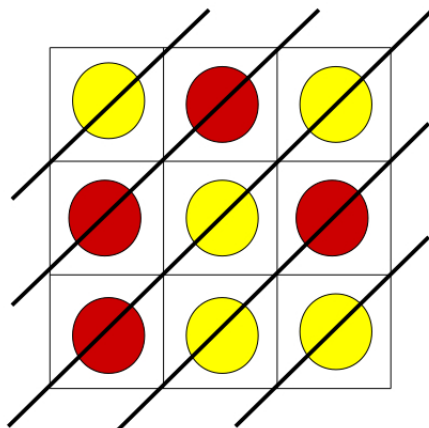
i.e.

```
bdEx333 =
  let
    c1 = [Red,Red,Yellow]
    c2 = [Yellow,Yellow,Red]
    c3 = [Yellow,Red,Yellow]
  in
    [c1,c2,c3]
```

then diagonals forward will return

```
[
  [Just Yellow],
  [Just Red,Just Red],
  [Just Red,Just Yellow,Just Yellow],
  [Just Yellow,Just Red],
  [Just Yellow]
]
```

And you can visualize this as:



Similarly, the diagonals backward should collect the elements along a diagonal left to right, but this time top to bottom. Thus, if we draw lines similarly, and start with the left most line on the same board we get

```
[
  [Just Red],
  [Just Red,Just Yellow],
  [Just Yellow,Just Yellow,Just Yellow],
  [Just Red,Just Red],
  [Just Yellow]
]
```

You are required to implement these functions with the types specified here. You are not required to use these in the implementation of the two functions in the design, section; however, it will probably be quite useful. Finally, you must of course, complete the implementation of the two functions in the design component section. You must submit a working implementation of all 6 required functions, together with any helper functions indicated in your design into a Haskell file "\$NamePart2.hs".

4 Evaluation

As already stated, there are two components to the grading.

1. Design
2. Implementation

For design, we will be typechecking your design file \$Name.hs. We will also be looking at it to ensure that some amount of thought went into this. For example,

a design file that defined

```
makeMove = undefined  
checkWin = undefined
```

will receive very low marks! At the same time, if the design segment consists of a full blown implementation, then it will also receive low marks. Justify each function you put into the design segment with a comment that states why you think it's part of the design and not part of the implementation: i.e. state why it's a simple function composed of other functions.

For the implementation component, it is much simpler. We will be considering \$NamePart2.hs. If the program does not compile, it's a zero. Next, a battery of HUnit and QuickCheck tests will be run on your program to ensure that it works correctly. This aspect of the grade will be automated. Points will be deducted for each test that is the program fails.