# EECS 368 Project 2

Table of Contents



**Documentation**

The inputs for each player is the x and y coordinate of opponent's board that they want to shoot. By default, it allocates a 5x5 board for both players and ships located in different locations. Screenshots can be found in this document and the game's full output can also be found below.

**Checklist**

✅Looking for basic Haskell know-how, in the form of a simple program of (cut down) battleship.
✅5x5 grid **and a Haskell representation of the grid**
✅1x4 boat/ship, placed at fixed location via a function (or other user).
✅user can guess via responses
✅computer checks, reports.
✅visuals help.
✅Optional: set number of rounds
✅Document what input works.
✅Document what you've tested with.
✅What examples have you used to test your code.

**Source code**

```haskell
-- Battleship by Sandy Uraz

-- Our board of ships
data Spot = Empty | Ship | Miss | Shot deriving (Show, Eq)

-- 0: Nothing here
-- 1: Ship here (hidden)
-- 2: Ship here (revealed)
-- 3: Missed shot
type Board = [Spot]

-- Pos is a type to define coordinates
-- on the  board
```

```haskell
type Pos = (Int, Int)

-- posToSpot convert a cartesian coordinate
-- (x,y) into a coordinate of a flattened matrix
posToSpot :: Pos -> Board -> Int
posToSpot (row, col) xs = (getSize xs) * row + col

-- getSize return the size of the board
getSize :: Board -> Int
getSize xs = (round (sqrt (fromIntegral (length xs))))

-- Create a list of integers square
-- the size passed, so 5 -> 25
-- This is a flattened matrix
squareList :: Int -> [Int]
squareList n = [1 .. n ^ 2]

-- convBoard converts a list of integers
-- into a list of boolean values
convBoard :: [Int] -> Board
convBoard xs = map (\x -> Empty) xs

-- makeBoard actually makes a board of
-- size nxn
makeBoard :: Int -> Board
makeBoard n = convBoard (squareList n)

-- resetBoard just resets the whole board
resetBoard :: Board -> Board
resetBoard xs = map (\x -> Empty) xs

-- placeShip will place a ship at (x,y)
placeShip :: Pos -> Board -> Board
placeShip pos xs = changeVal (posToSpot pos xs) xs Ship

-- placeShips will place multiple ships
placeShips :: [Pos] -> Board -> Board
placeShips []       xs    = xs
placeShips (x : xs) board = placeShips xs (placeShip x board)

-- changeVal puts a value at the index
```

```haskell
changeVal :: Int -> Board -> Spot -> Board
changeVal n xs val = (take n xs) ++ [val] ++ (drop (n + 1) xs)

-- checkSpot checks if a ship exists at the current
-- location (x,y)
checkSpot :: Pos -> Board -> Spot
checkSpot pos xs = xs !! (posToSpot pos xs)

-- attack lets us attack a ship, if we hit it,
-- it changes the market to "Miss", otherwise,
-- nothing happens
attack :: Pos -> Board -> Board
attack pos xs
  | (checkSpot pos xs) == Ship  = changeVal (posToSpot pos xs) xs Shot
  | (checkSpot pos xs) == Empty = changeVal (posToSpot pos xs) xs Miss
  | otherwise                   = xs

-- picCell draws a representation of the board
-- at exact position that is provided, can be used
-- do draw the whole board
picCell :: Pos -> Board -> String
picCell pos xs | (checkSpot pos xs) == Shot = "● "
               | (checkSpot pos xs) == Miss = "❌"
               | otherwise                  = "○ "

-- listToStr converts a list into a string
listToStr :: [Int] -> String
listToStr []       = ""
listToStr (x : xs) = (show x) ++ " " ++ (listToStr xs)

-- picCells concatenates positions' representations
picCells :: [Pos] -> Board -> String
picCells []       board = ""
picCells (x : xs) board = (picCell x board) ++ (picCells xs board)

-- picRow gets a string for a whole row
picRow :: Int -> Board -> String
picRow n board =
  (show n)
    ++ " | "
    ++ (picCells
```

```
          (zip (map (\x -> n) [0 .. (getSize board) - 1])
               [0 .. (getSize board) - 1]
          )
          board
        )

-- picRow gets a string for a whole rows
picRows :: [Int] -> Board -> String
picRows []       board = ""
picRows (x : xs) board = (picRow x board) ++ "\n" ++ (picRows xs board)

-- upperNums returns a string for the upper numbers
upperNums :: Board -> String
upperNums board = "y | " ++ (listToStr [0 .. (getSize board) - 1])

-- upperLines is a line delimiter between the numbers and board
upperLines :: Board -> String
upperLines board = concat (map (\x -> "-") (upperNums board))

-- picBoard gets a string for the whole board
picBoard :: Board -> String
picBoard board =
  (picRows [0 .. (getSize board) - 1] board)
    ++ (upperLines board)
    ++ "\n"
    ++ (upperNums board)
    ++ " x \n"

-- drawBoard actually draws the board
drawBoard :: Board -> IO ()
drawBoard board = putStrLn (picBoard board)

-- gameDone reports whether the game is won
gameDone :: Board -> Bool
gameDone board = (length (filter (\x -> x == Shot) board)) == 4

-- game is the recursive turn-based version of playing
game :: Board -> Board -> Bool -> IO Board
game board1 board2 player1 = do
  let win1 = gameDone board2
  let win2 = gameDone board1
```

```
  -- Check if anyone won the game
  if win1 then putStrLn "\n PLAYER 1 WON!!!" else putStrLn ""
  if win2 then putStrLn "\n PLAYER 2 WON!!!" else putStrLn ""
  -- Check the current player to print greetings and board
  if player1 then putStrLn "[Player 1 turn!]" else putStrLn "[Player 2 turn!]"
  if player1 then drawBoard board2 else drawBoard board1
  -- Get the x,y coordinates to attack
  putStrLn "\nEnter x coordianate: "
  xcord <- getLine
  putStrLn "Enter y coordinate: "
  ycord <- getLine
  -- Curry the attack function
  let turn = attack ((read $ ycord :: Int), (read $ xcord :: Int))
  -- Switch next player
  let nextPlayer = not player1
  -- if it's player1's turn, attack board 2
  -- otherwise, attack board 1
  if player1
    then game board1 (turn board2) nextPlayer
    else game (turn board1) board2 nextPlayer

-- The entrypoint, make it printable
main = do
  -- Initial greeting message
  putStrLn "Let the games begin!\n"
  putStrLn "There is a 4-cell ship somewere,\ntype coordinates and shoot"
  -- Add ships for player1 and player2
  let ships1 = [(1, 1), (1, 2), (1, 3), (1, 4)]
  let ships2 = [(0, 1), (0, 2), (0, 3), (0, 4)]
  -- Put the ships on a 5x5 board
  let board1 = placeShips ships1 (makeBoard 5)
  let board2 = placeShips ships2 (makeBoard 5)
  -- Start the game by making player1 attack player2
  game board1 board2 True
```

**Screenshots**

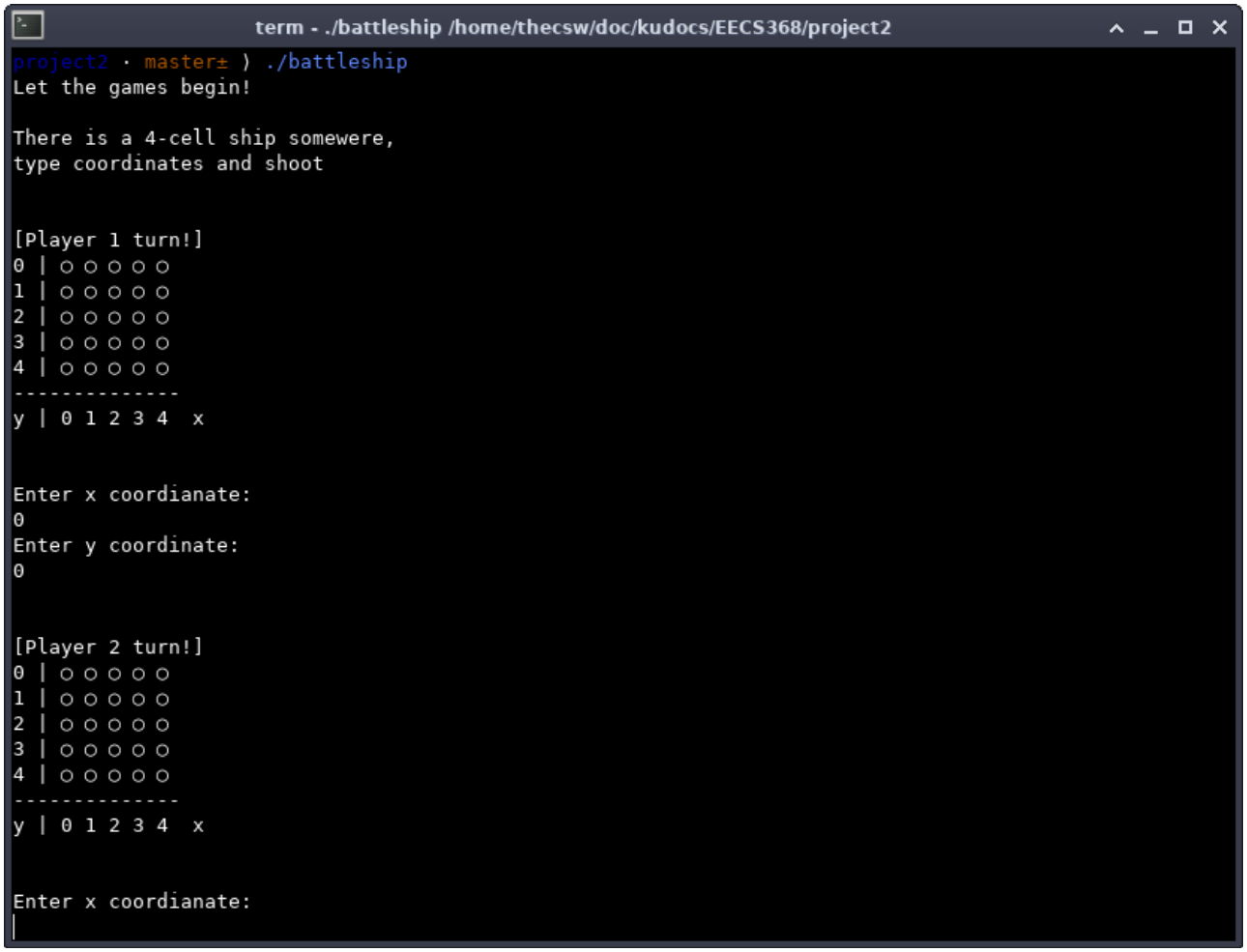


Figure 1: Example 1

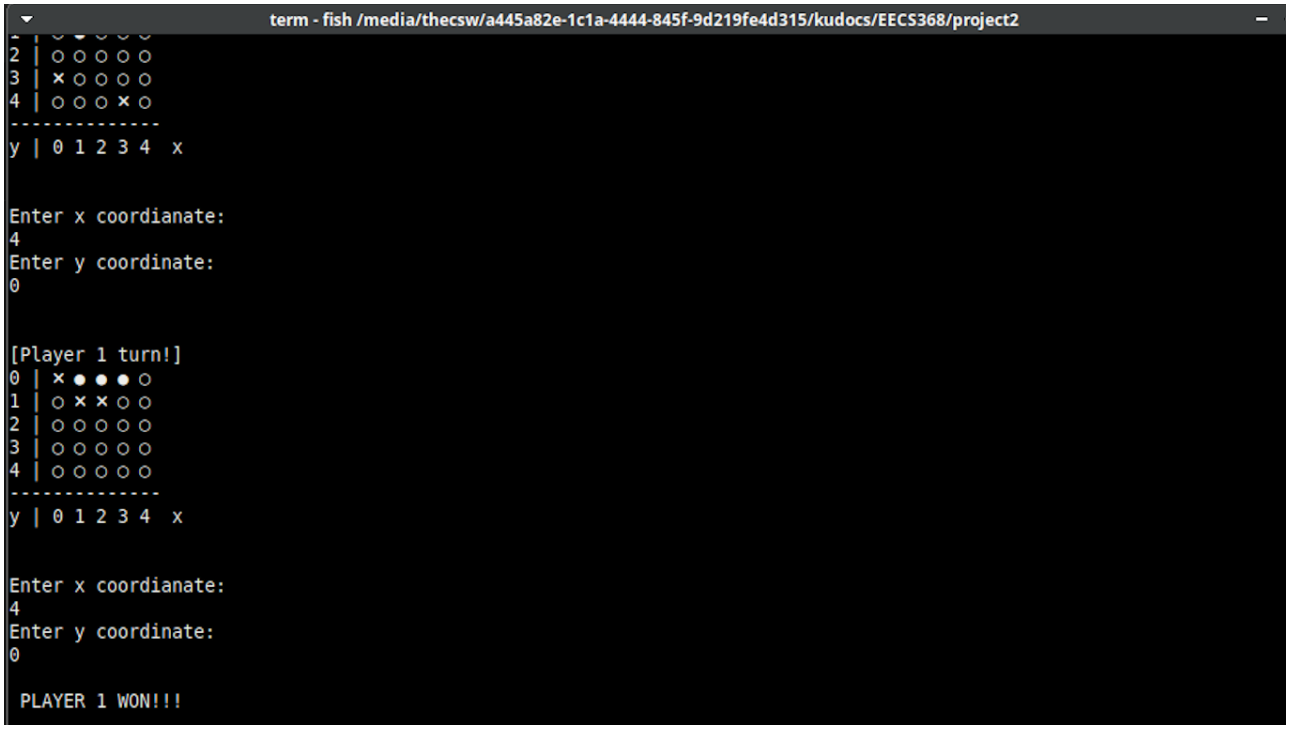


Figure 2: Example 2

**Full output**

```
 (master *$*)$ ./battleship
Let the games begin!

There is a 4-cell ship somewere,
type coordinates and shoot


[Player 1 turn!]
0 | ○ ○ ○ ○ ○
1 | ○ ○ ○ ○ ○
2 | ○ ○ ○ ○ ○
3 | ○ ○ ○ ○ ○
4 | ○ ○ ○ ○ ○
--------------
y | 0 1 2 3 4  x


Enter x coordianate:
2
```

```
Enter y coordinate:
0


[Player 2 turn!]
0 | ○ ○ ○ ○ ○
1 | ○ ○ ○ ○ ○
2 | ○ ○ ○ ○ ○
3 | ○ ○ ○ ○ ○
4 | ○ ○ ○ ○ ○
--------------
y | 0 1 2 3 4  x


Enter x coordianate:
0
Enter y coordinate:
0


[Player 1 turn!]
0 | ○ ○ ● ○ ○
1 | ○ ○ ○ ○ ○
2 | ○ ○ ○ ○ ○
3 | ○ ○ ○ ○ ○
4 | ○ ○ ○ ○ ○
--------------
y | 0 1 2 3 4  x


Enter x coordianate:
1
Enter y coordinate:
0


[Player 2 turn!]
0 | ✕○ ○ ○ ○
1 | ○ ○ ○ ○ ○
2 | ○ ○ ○ ○ ○
3 | ○ ○ ○ ○ ○
```

```
4 | ○ ○ ○ ○ ○
--------------
y | 0 1 2 3 4  x


Enter x coordianate:
3
Enter y coordinate:
4


[Player 1 turn!]
0 | ○ ● ● ○ ○
1 | ○ ○ ○ ○ ○
2 | ○ ○ ○ ○ ○
3 | ○ ○ ○ ○ ○
4 | ○ ○ ○ ○ ○
--------------
y | 0 1 2 3 4  x


Enter x coordianate:
1
Enter y coordinate:
1


[Player 2 turn!]
0 | ✕○ ○ ○ ○
1 | ○ ○ ○ ○ ○
2 | ○ ○ ○ ○ ○
3 | ○ ○ ○ ○ ○
4 | ○ ○ ○ ✕○
--------------
y | 0 1 2 3 4  x


Enter x coordianate:
0
Enter y coordinate:
3
```

```
[Player 1 turn!]
0 | ○ ● ● ○ ○
1 | ○ ✕○ ○ ○
2 | ○ ○ ○ ○ ○
3 | ○ ○ ○ ○ ○
4 | ○ ○ ○ ○ ○
--------------
y | 0 1 2 3 4  x


Enter x coordianate:
3
Enter y coordinate:
0


[Player 2 turn!]
0 | ✕○ ○ ○ ○
1 | ○ ○ ○ ○ ○
2 | ○ ○ ○ ○ ○
3 | ✕○ ○ ○ ○
4 | ○ ○ ○ ✕○
--------------
y | 0 1 2 3 4  x


Enter x coordianate:
3
Enter y coordinate:
0


[Player 1 turn!]
0 | ○ ● ● ● ○
1 | ○ ✕○ ○ ○
2 | ○ ○ ○ ○ ○
3 | ○ ○ ○ ○ ○
4 | ○ ○ ○ ○ ○
--------------
```

```
y | 0 1 2 3 4  x


Enter x coordianate:
2
Enter y coordinate:
1


[Player 2 turn!]
0 | ✕○ ○ ✕○
1 | ○ ○ ○ ○ ○
2 | ○ ○ ○ ○ ○
3 | ✕○ ○ ○ ○
4 | ○ ○ ○ ✕○
--------------
y | 0 1 2 3 4  x


Enter x coordianate:
0
Enter y coordinate:
0


[Player 1 turn!]
0 | ○ ● ● ● ○
1 | ○ ✕✕○ ○
2 | ○ ○ ○ ○ ○
3 | ○ ○ ○ ○ ○
4 | ○ ○ ○ ○ ○
--------------
y | 0 1 2 3 4  x


Enter x coordianate:
0
Enter y coordinate:
0
```

```
[Player 2 turn!]
0 | ✕○ ○ ✕○
1 | ○ ○ ○ ○ ○
2 | ○ ○ ○ ○ ○
3 | ✕○ ○ ○ ○
4 | ○ ○ ○ ✕○
--------------
y | 0 1 2 3 4  x


Enter x coordianate:
1
Enter y coordinate:
1


[Player 1 turn!]
0 | ✕● ● ● ○
1 | ○ ✕✕○ ○
2 | ○ ○ ○ ○ ○
3 | ○ ○ ○ ○ ○
4 | ○ ○ ○ ○ ○
--------------
y | 0 1 2 3 4  x


Enter x coordianate:
1
Enter y coordinate:
1


[Player 2 turn!]
0 | ✕○ ○ ✕○
1 | ○ ● ○ ○ ○
2 | ○ ○ ○ ○ ○
3 | ✕○ ○ ○ ○
4 | ○ ○ ○ ✕○
--------------
y | 0 1 2 3 4  x
```

```
Enter x coordianate:
4
Enter y coordinate:
0


[Player 1 turn!]
0 | ✕● ● ● ○
1 | ○ ✕✕○ ○
2 | ○ ○ ○ ○ ○
3 | ○ ○ ○ ○ ○
4 | ○ ○ ○ ○ ○
--------------
y | 0 1 2 3 4  x


Enter x coordianate:
4
Enter y coordinate:
0

 PLAYER 1 WON!!!
```

Author: Sagindyk Urazayev

Email: University of Kansas, Department of Electrical Engineering and Computer Science, Lawrence, KS 66045 (thecsw@ku.edu)

Created: 2020-05-07 Thu 21:21

Validate