Battleships Game

Program Design and Data Structures (1DL201)

Group 4: William Berger, William Brunnsberg, and Jonas Rosengren

March 2021

GitHub Repository:
https://github.com/JoRo-Code/PKDProject/tree/main/final

Contents

1 Introduction

For our project in the course "Program Design and Data Structures", we wanted to utilize what we had learned throughout the course as well as implement new aspects we found interesting. We were also curious regarding how developing a game would work in a functional programming language, specifically Haskell.

After a lot of brainstorming and research we determined that creating the board game "Battleships" from scratch, would be the perfect challenge for this. We would for instance need to construct new data types to store information, create a graphical interface using a library called "Gloss", and design a very simple AI as the opponent.

1.1 Battleships

Battleships is world-wide known board game based on guessing where battleships are located on a grid. A game of Battleships is split into two stages; placing the ships and guessing the coordinates of the opponents ships.

In the first stage, the two players get to place their given fleets of different sizes on their own boards, hidden from the other player. In the *1990 Milton Bradley* version of the game, the board size is 10x10 cells and each player has five ships to place. The ships are the following:

Name	Length
Carrier	5
Battleship	4
Cruiser	3
Submarine	3
Destroyer	2

Once every ship is placed, the game moves on to its second stage.

In the second stage, the players will take turns guessing, also known as shooting, at different coordinates on the opponent's board. When shooting a coordinate, it is revealed whether a ship is placed there or not. The player who manages to hit every cell containing a ship on the opponent's grid first, wins.

2 Summary

The Battleships game we have created allows the user to play a desired number of rounds of Battleships against a very simple AI. The game also keeps track of how many wins the user and the AI have, until the player quits.

Just as the popular board game, our version is split into two stages. The first stage consists of placing down ships on a grid, whereas the second stage consists of each player taking turns guessing where their opponent has placed their ships. The winner is the player who manages to hit every part of the opponent's ships first.

To design this game we have for instance constructed new data types to store information, utilized a library called *Graphics.Gloss* to create a GUI (Graphical User Interface) and implemented randomness to make the experience with our AI more enjoyable.

The GUI allows the user interact with a visual representation of the game. To further enhance the user experience we have also created animations like a moving radar and simple explosions.

3 Use Cases

3.1 Requirements

To run the program, the user must first install the Haskell platform and the following libraries:

- System.Random

- Graphics.Gloss

To perform testing, Test.HUnit is also required.

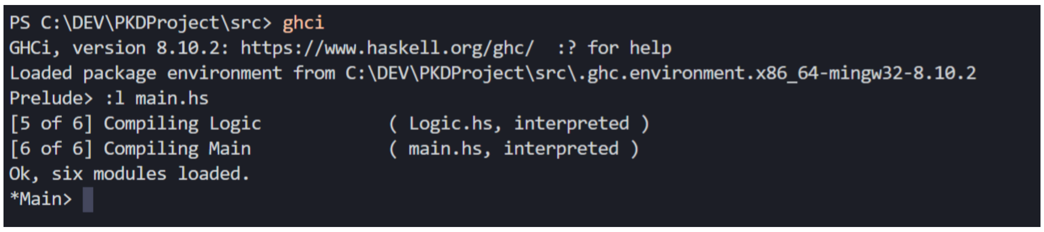
3.2 How To Run the Program

Navigate to the folder called *src* through the terminal. Once there you can run the program in two different ways.

Option 1

1. Type “ghci“ into the terminal
2. Type “:l main.hs“

The gloss window should now start and let you play.



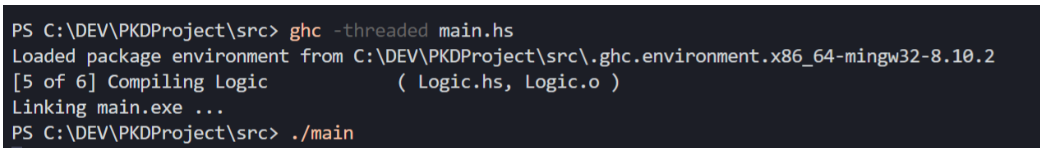
```
PS C:\DEV\PKDProject\src> ghci
GHCi, version 8.10.2: https://www.haskell.org/ghc/  :? for help
Loaded package environment from C:\DEV\PKDProject\src\.ghc.environment.x86_64-mingw32-8.10.2
Prelude> :l main.hs
[5 of 6] Compiling Logic            ( Logic.hs, interpreted )
[6 of 6] Compiling Main             ( main.hs, interpreted )
Ok, six modules loaded.
*Main>
```

Option 2

1. Type “ghc -threaded main.hs“ into the terminal

2. Type "./main"

The gloss window should now start and let you play.

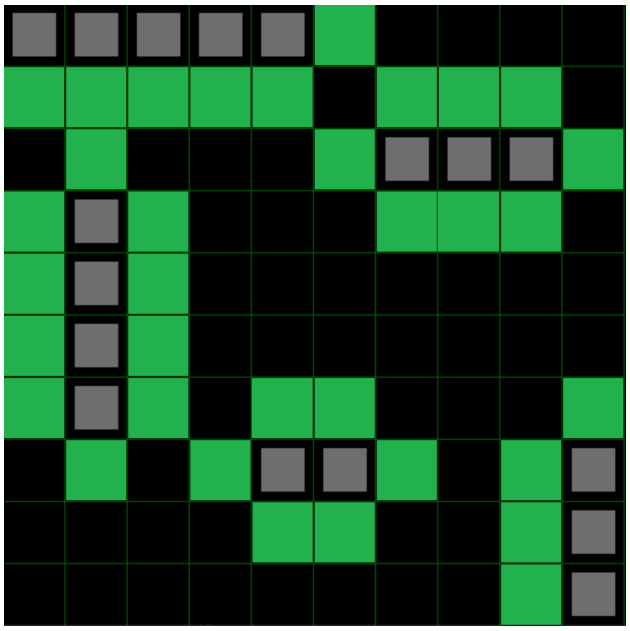


```
PS C:\DEV\PKDProject\src> ghc -threaded main.hs
Loaded package environment from C:\DEV\PKDProject\src\.ghc.environment.x86_64-mingw32-8.10.2
[5 of 6] Compiling Logic            ( Logic.hs, Logic.o )
Linking main.exe ...
PS C:\DEV\PKDProject\src> ./main
```

3.3 Placement Rules

To make the game more enjoyable we decided to implement placement rules for the ships. These rules apply both to the user as well as the AI.

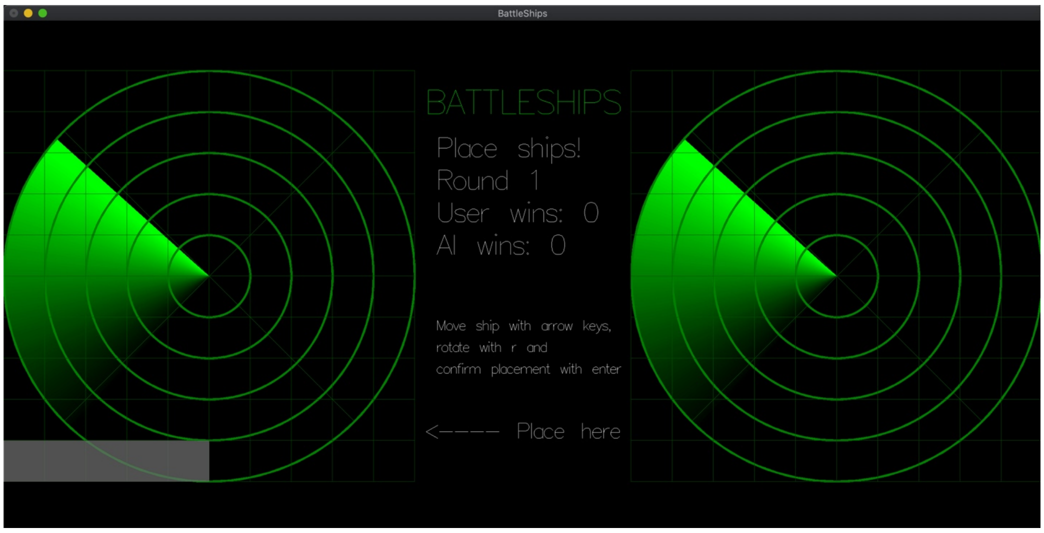
When a ship is placed down it occupies the cells it takes to fit that ship and also all the surrounding cells, meaning that no ships can be placed right next to another. This also means that ships must not overlap. The image below shows all the cells each ship occupies on the board. No other ship can be placed in any of the green (or gray) cells. How we make sure these rules are followed is explained in the logic section.



Furthermore, when a ship's coordinates are mentioned, it is indicating the starting position of the ship. If it is a vertical ship, the starting position is the "top" of the ship, meaning the rest are on the same column but on lower rows. If a ship is horizontally orientated, the starting position is the leftmost part of the ship, meaning the rest are on the same row but on greater columns.

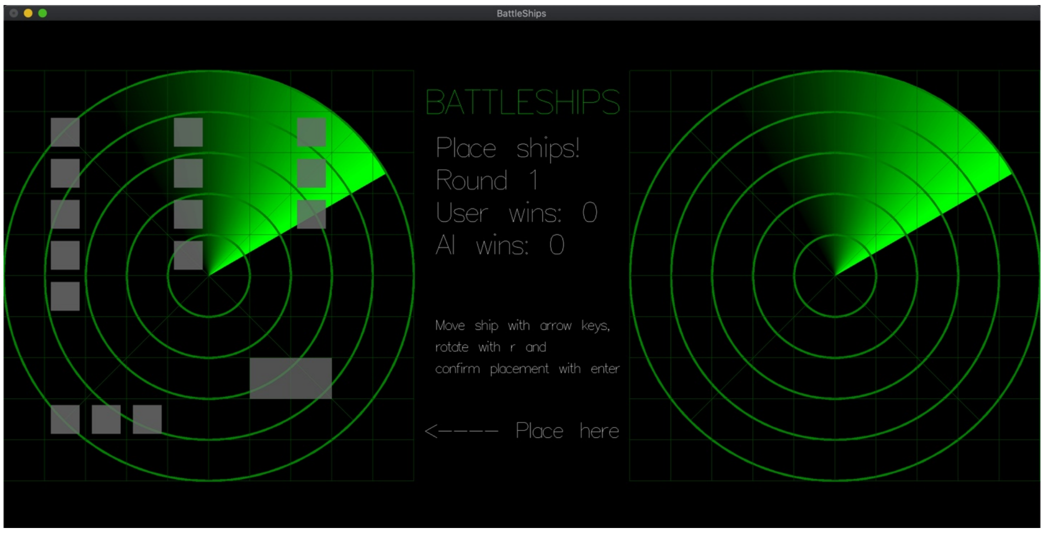
3.4 How To Play

When the program starts, a new window appears. In this window there are two boards with animated radars as backgrounds and a part in the middle that shows score and provides information on how to play, which is visible in the following figure:



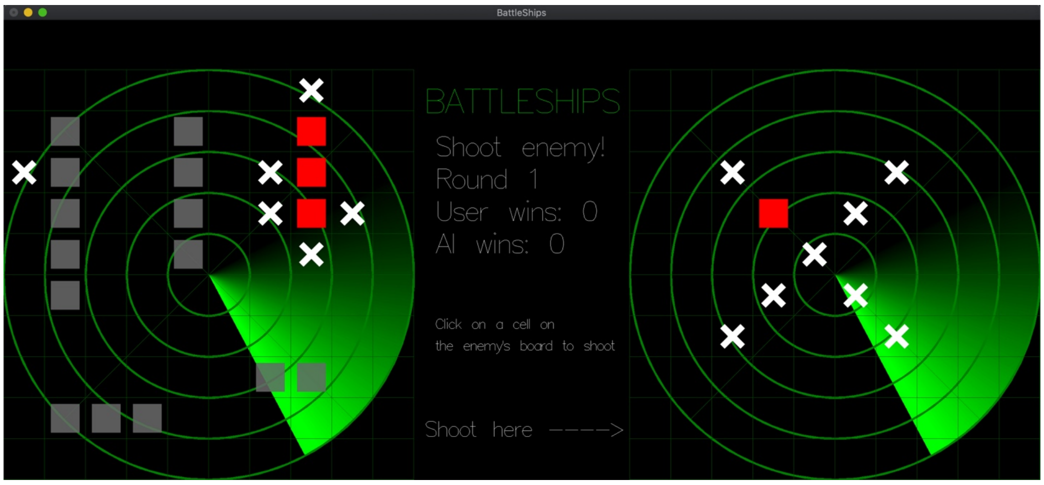
The board to the left is the user's board, and the board to the right is the AI's board. In other words, the user will place their ships on the board to the left and make guesses on the board to the right.

Once the program is started, the first stage begins. The user needs to place down the given ships on their board according to the placement rules. To move around the ships, the arrow-keys are used. It is also possible to rotate the ships by pressing *R* on the keyboard. When the user is satisfied with a certain position, they need to press the enter-key to confirm the placement. The first stage ends when all the given ships are placed.

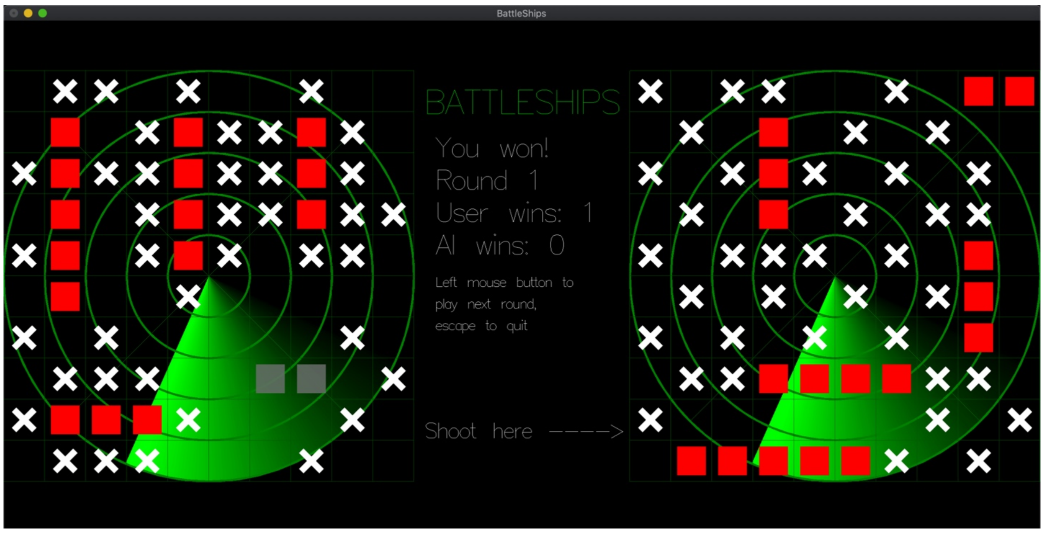


The figure above shows an example of what it looks like when the user has placed down four ships and is currently placing down the fifth ship. A placed ship is differentiated from a ship currently being placed, by being split into multiple squares, representing the length of the ship.

For the second stage, the user needs to use the mouse to left-click on the board to the right to make guesses. As mentioned, the rightmost board is the AI's board, where ships are randomly placed at the beginning of each round. When the user clicks on a cell, a cross or a square will appear depending on what that particular square contains. A white cross symbolizes a miss, whereas a red square means the user has successfully managed to hit a part of one of the AI's ships. After each click, the AI will also make a guess on the user's board.



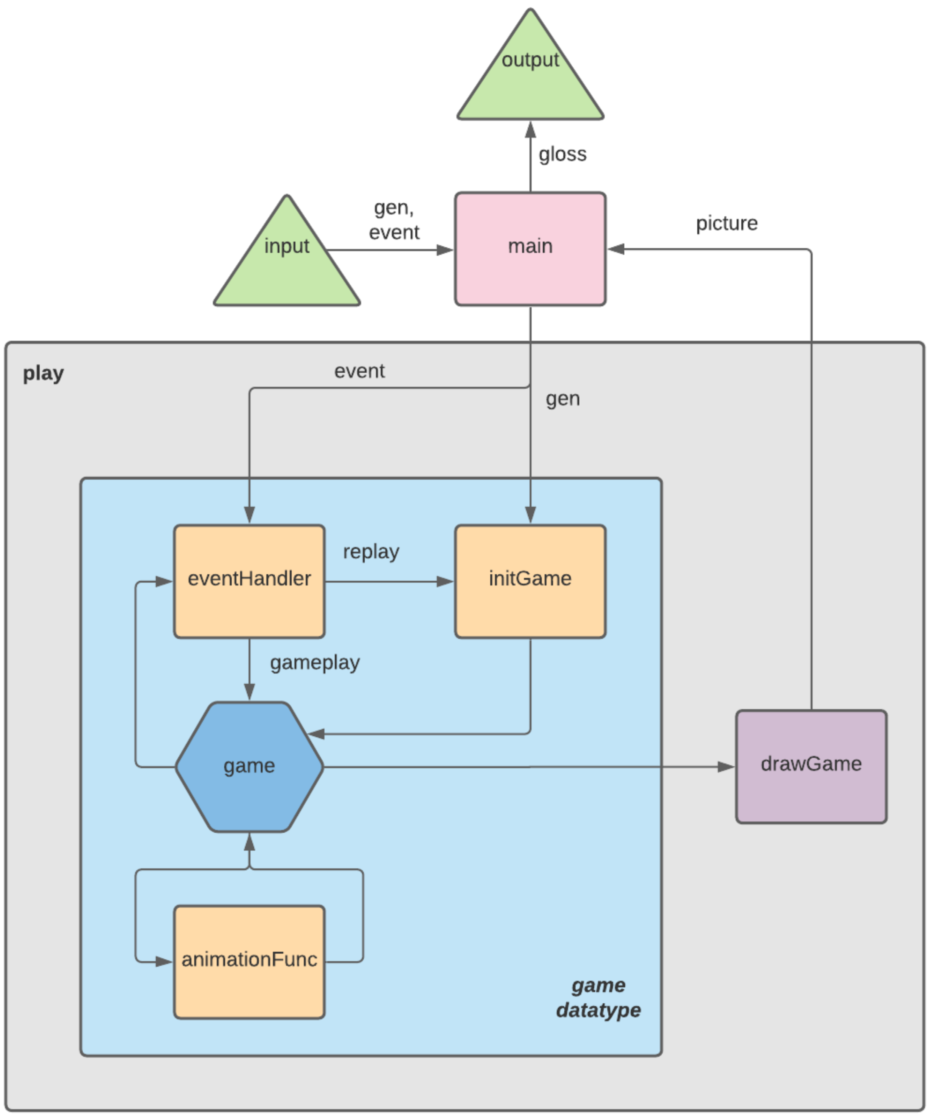
After making some guesses, the board could look as the figure above. In this example, the AI has managed to sink one of the user's ships, and is currently in the lead. The winner is determined by who succeeds in hitting every part of every ship first. When the round is over, a text with the result will appear in the middle of the screen. The scoreboard will also update. In the image below, it shows the example-round when it is over, and the user has won. In the middle it now says *You won!*, and the scoreboard is updated with one win to the user.



To start another round, the user has to click on the screen. It is possible to quit the game at any time by pressing *Esc* on the keyboard.

4 Program Documentation

4.1 Control Flow



The program begins with **main**, which handles inputs and outputs. Together with *game* (the datatype), **play** transforms these inputs into a picture which **main** visualizes with Gloss.

The main idea is that *game* is updated by the yellow functions in the flowchart while **drawGame** converts *game* into a picture. *game* is initialized with **initGame**, using gen to create a randomized gameAIBoard. **eventHandler** updates *game* according to events passed from main. To replay, **eventHandler** calls **initGame** with a newGen to reinitialize *game* when a player has won. **animationFunc** updates *game* with time, in particular the radar and explosion. To transform *game* into a visual representation, **drawGame** converts *game* into a picture which is passed back to **main**.

Note: The flow chart is simplified. The actual changes to *game* isn't specified due to a lot of changes for each edge.

4.2 Data Types

To store information and make the code easier to understand, we have constructed new data types. To make sense of the code, we will begin with explaining the new data types.

SquareState

data SquareState = Checked | NotChecked

Every square in the grid has a SquareState; *Checked* or *NotChecked.* Checked means that the square has been shot at, whereas NotChecked means the opposite. With this data type we can easily keep track of which squares can be shot at, since it should not be possible to shoot at an already Checked square.

Cell

data Cell = Empty SquareState | Ship SquareState

This data type provides information regarding if a square contains a *Ship* or if it is *Empty.* It also includes the SquareState which tells whether a cell has been shot or not. Thus a cell represents the four different possibilites of a cell:

Empty NotChecked || Empty Checked || Ship NotChecked || Ship Checked

Player

data Player = User | AI

The Player data type consists of two value constructors, *User* and *AI*. This is used to keep track of whose turn it is to make a move and for keeping score.

GameStage

data GameStage = Placing Player | Shooting Player

GameStage determines which of the two stages the game currently is in. The first stage is *Placing*, where the players choose the locations for the ships. The second stage is *Shooting*, where the players take turns shooting at squares. The data type also includes the current player, meaning there are four different possibilities of the GameStage:

Placing User || Placing AI || Shooting Player || Shooting AI

Direction

data Direction = Horizontal | Vertical

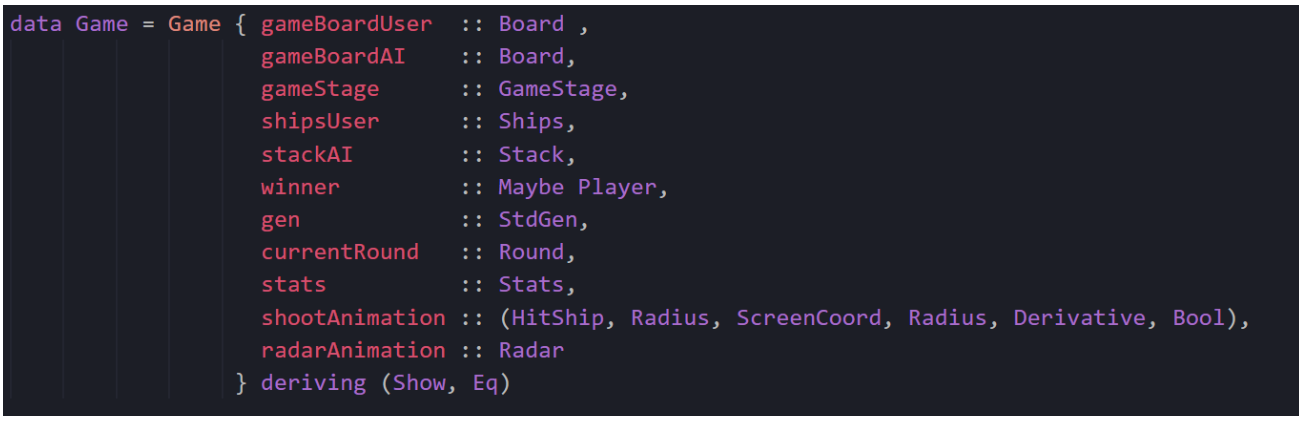
When placing the ships we need to know whether they are *Vertical* or *Horizontal*. With this information we are able to determine if the ship has a valid placement or if it is out of bounds.

Game

This is the most crucial data type in the project. The instance created using this data type is what is being manipulated through input from user. It holds all information about the current game state.

- ***gameBoardUser*** - the game board the user places the ships on, AI shoots on this board.
- ***gameBoardAI*** - the game board the AI places the ships on, user shoots on this board.
- ***gameStage*** - the current stage of the game.
- ***shipsUser*** - all ships the user have left to place before the game switches to the shooting stage of the game.
- ***stackAI*** - the current stack for the AI.

- ***winner*** - if the game has a winner or not.
- ***gen*** - the last random seed used.
- ***currentRound*** - the current round.
- ***stats*** - how many wins each player has.
- ***shootAnimation*** - holds information on how and whether a shoot animation should be displayed.
- ***radarAnimation*** - the different radiuses of the radar, and the current angle of the rotating arc.



```haskell
data Game = Game { gameBoardUser  :: Board ,
                   gameBoardAI    :: Board,
                   gameStage      :: GameStage,
                   shipsUser      :: Ships,
                   stackAI        :: Stack,
                   winner         :: Maybe Player,
                   gen            :: StdGen,
                   currentRound   :: Round,
                   stats          :: Stats,
                   shootAnimation :: (HitShip, Radius, ScreenCoord, Radius, Derivative, Bool),
                   radarAnimation :: Radar
                 } deriving (Show, Eq)
```

4.3 Type Synonyms

To further improve the readability of the code, we have constructed type synonyms for the types. It could be beneficial for understanding the code if we explain some of them.

Row, Col & CellCoord

type Col = Int

type Row = Int

type CellCoord = (Col, Row)

As mentioned, the boards consists of grids. Every square in the grid has a column (col) and a row which together in a tuple forms CellCoord. This combined with the Cell-data type is what is used to place, shoot and modify the board.

Board & BoardSize

type Board = Array (Col, Row) Cell

type BoardSize = Int

The boards are two-dimensional arrays whose sizes are determined by a global BoardSize called *n*. This size is changeable in the code and makes the game customizable for the user's preference.

ShootList & Stack

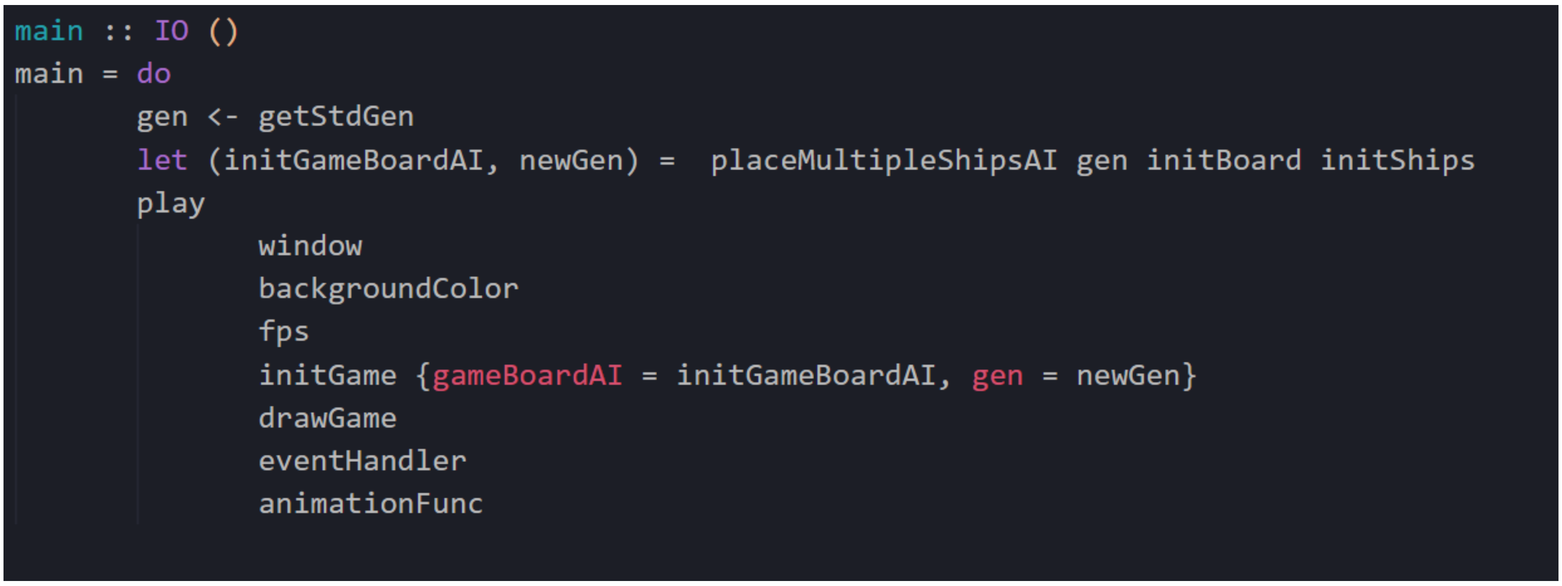
type ShootList = [(CellCoord,Cell)]

type Stack = [(CellCoord,Cell)]

ShootList and Stack are used for the AI and have the same types, yet are used in different ways. The ShootList can be considered as a prioritised rest of all cells that have not yet been shot at, while the Stack is where the AI will pick cells to shoot from. More on this in the AI section.

4.4 Main

The main module is what combines the other modules into a working program with graphics that allows input from the user via the keyboard and mouse. The **main** function is what lets the user play a game in a window. It utilizes the **play** function inside of the **Graphics.Gloss.Interface.Pure.Game** module. In the arguments that play is called with we can find **initGame**, **drawGame**, **eventHandler** and **animationFunc**. In the same order they were mentioned, they are responsible for: initializing the game state, converting the current game state to a picture, handle the input events from the user, and perform animations on the screen. A more in-depth explanation on how they work can be found in the sections below.

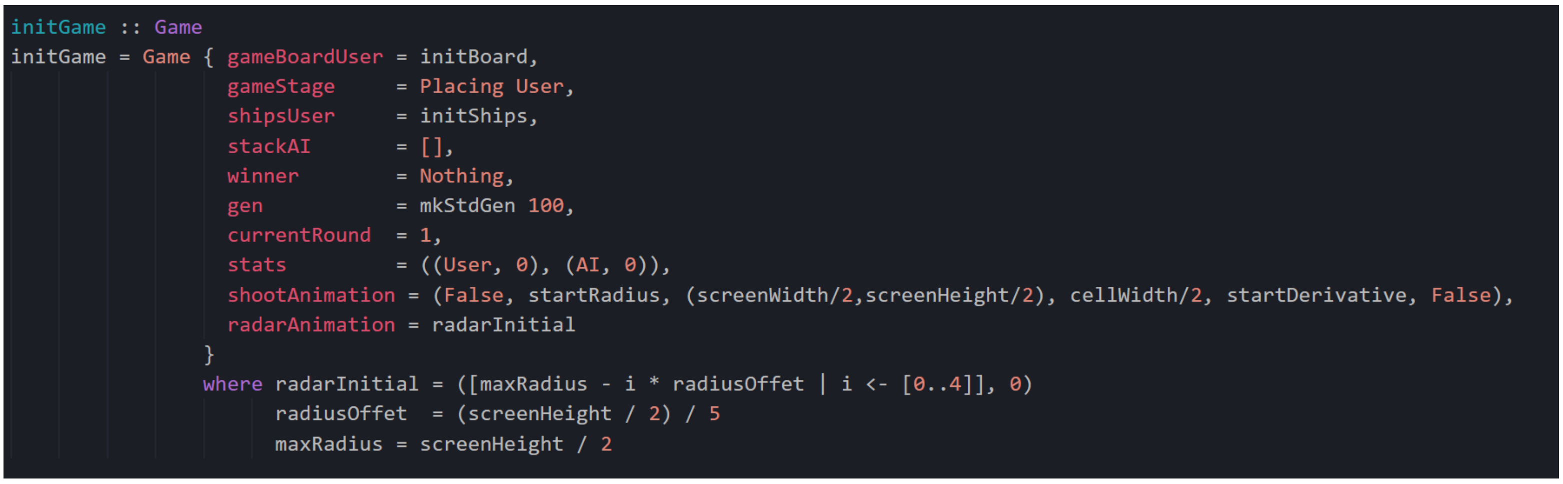


```haskell
main :: IO ()
main = do
        gen <- getStdGen
        let (initGameBoardAI, newGen) =  placeMultipleShipsAI gen initBoard initShips
        play
                window
                backgroundColor
                fps
                initGame {gameBoardAI = initGameBoardAI, gen = newGen}
                drawGame
                eventHandler
                animationFunc
```

4.5 Game

initGame

InitGame doesn't take any arguments, thus it is not a function, but rather just a value. It represent the initial game state. In short what it does: Assigns the starting values to the fields inside the game state data type, either via a direct value or a function that evaluates to a value. The gameBoardAI field inside of the data type is not assigned a value here. This is done in the main module, where it adds a randomly generated board. More on that in the AI section.

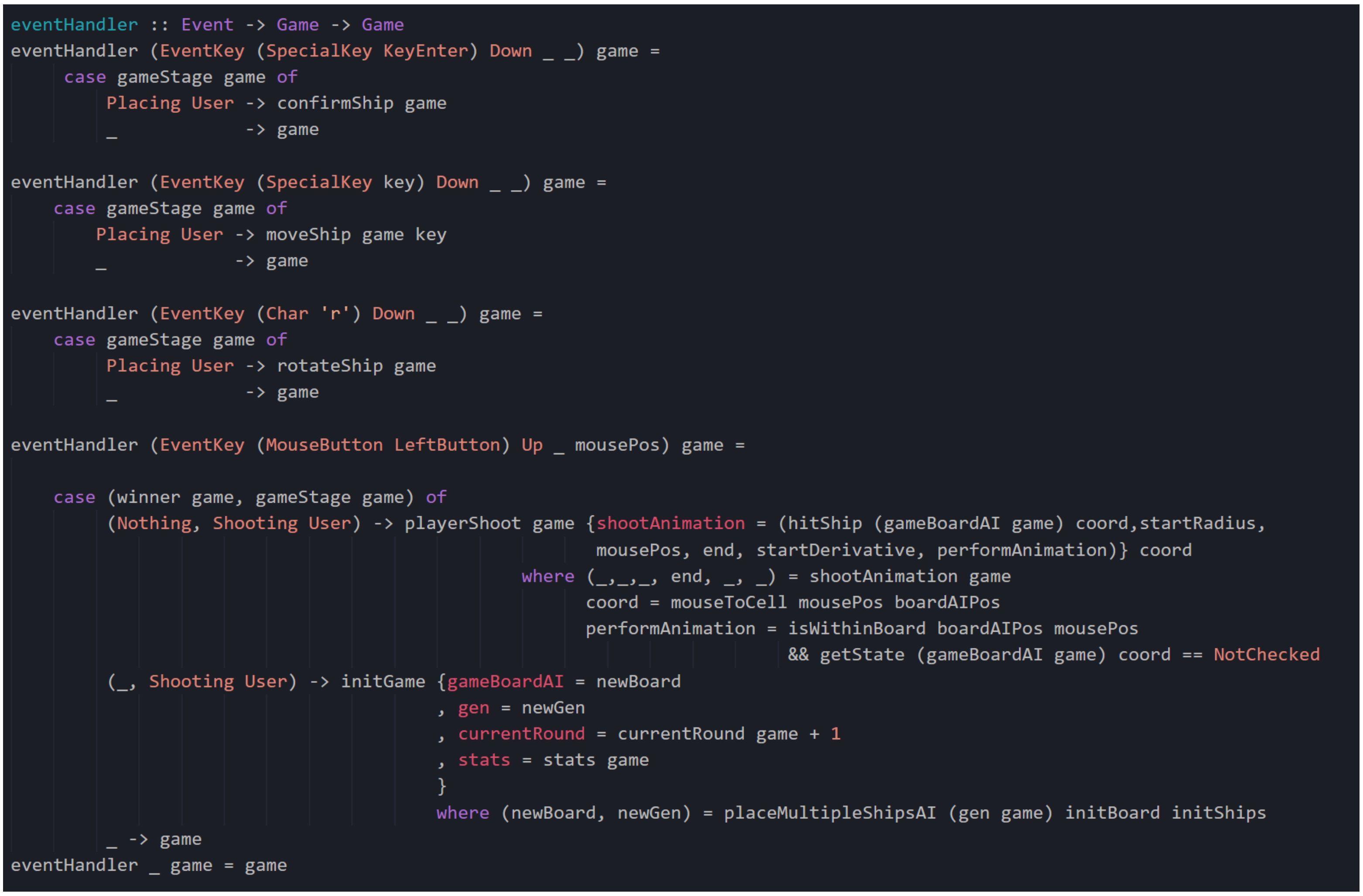


```haskell
initGame :: Game
initGame = Game { gameBoardUser = initBoard,
                  gameStage     = Placing User,
                  shipsUser     = initShips,
                  stackAI       = [],
                  winner        = Nothing,
                  gen           = mkStdGen 100,
                  currentRound  = 1,
                  stats         = ((User, 0), (AI, 0)),
                  shootAnimation = (False, startRadius, (screenWidth/2,screenHeight/2), cellWidth/2, startDerivative, False),
                  radarAnimation = radarInitial
                }
                where radarInitial = ([maxRadius - i * radiusOffet | i <- [0..4]], 0)
                      radiusOffet  = (screenHeight / 2) / 5
                      maxRadius = screenHeight / 2
```

4.6 Logic

The logic module is the largest module in the program. Its role is to get input from the user and manipulate the game state data type depending on the current game stage and the input from the user.

4.6.1 EventHandler



```haskell
eventHandler :: Event -> Game -> Game
eventHandler (EventKey (SpecialKey KeyEnter) Down _ _) game =
     case gameStage game of
         Placing User -> confirmShip game
         _            -> game

eventHandler (EventKey (SpecialKey key) Down _ _) game =
    case gameStage game of
        Placing User -> moveShip game key
        _            -> game

eventHandler (EventKey (Char 'r') Down _ _) game =
    case gameStage game of
         Placing User -> rotateShip game
         _            -> game

eventHandler (EventKey (MouseButton LeftButton) Up _ mousePos) game =

    case (winner game, gameStage game) of
         (Nothing, Shooting User) -> playerShoot game {shootAnimation = (hitShip (gameBoardAI game) coord,startRadius,
                                                       mousePos, end, startDerivative, performAnimation)} coord
                                                where (_,_,_, end, _, _) = shootAnimation game
                                                      coord = mouseToCell mousePos boardAIPos
                                                      performAnimation = isWithinBoard boardAIPos mousePos
                                                                         && getState (gameBoardAI game) coord == NotChecked
         (_, Shooting User) -> initGame {gameBoardAI = newBoard
                                        , gen = newGen
                                        , currentRound = currentRound game + 1
                                        , stats = stats game
                                        }
                                        where (newBoard, newGen) = placeMultipleShipsAI (gen game) initBoard initShips
         _ -> game
eventHandler _ game = game
```

The eventHandler function handles individual input events, and manipulates the game argument in different ways depending on the input event and the current game stage that can be found in the current game state. It consist of four main pattern matchings on different types of events. Below is information regarding what the function will achieve on each of these inputs.

SpecialKey KeyEnter

This event is triggered when the enter key is pressed. It checks the current gameStage of the game state argument. If the game is in the *Placing User* stage it will call the function **confirmShip** that handles the placement of the ship that is currently being moved on the screen by the user. If the game is in any other stage it leaves the game state unchanged.

SpecialKey key

This event is triggered when any SpecialKey is pressed (except the enter key). If the game is in the *Placing user* stage it will call the function **moveShip** with the game state as well as the key that was pressed. This allows the moveShip function to work out what direction the ship should move towards. If the game is in any other stage it leaves the game state unchanged.

Char 'r'

This event is triggered when the key *R* is pressed. If the game is in the *Placing user* stage it will call the function **rotateShip**. If the game is in any other stage it leaves the game state unchanged.

MouseButton LeftButton

This pattern is split into two major cases. If there currently is no winner, and the gameStage is *Shooting User* it will call the function **playerShoot**, with an updated game state. The shootAnimation inside of the game state is changed, and this controls if and what type of animation should be displayed when the user clicked.

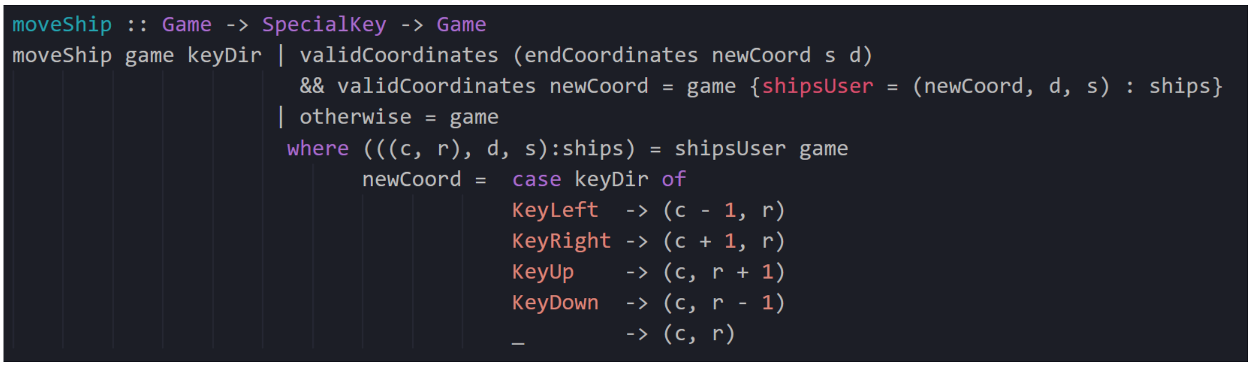
If there is a winner and gameStage is *Shooting User* it matches the last case and calls **initGame**, and the new game state is the initial game state but with some values overridden. Since returning initGame would reset the game state to its default state, some values have to be updated. For example the round increase with one, since a new round is about to start. **PlaceMultipleShipsAI** generates a new board with AI placed ships and binds the value to the gameBoardAI field in the game state. When this has been done the new game state is returned.

4.6.2 Placing

In the last section three main functions for ship placement was introduced, **confirmShip, moveShip** and **rotateShip**. These functions utilizes different help function. Any function names in bold mentioned inside of the moveShip, rotateShip and confirmShip descriptions has its own explanation with a picture of its code below. Functions in italic text are explained in words as they are mentioned.

moveShip

Updates the current placing ship's starting cell coordinates according to key input

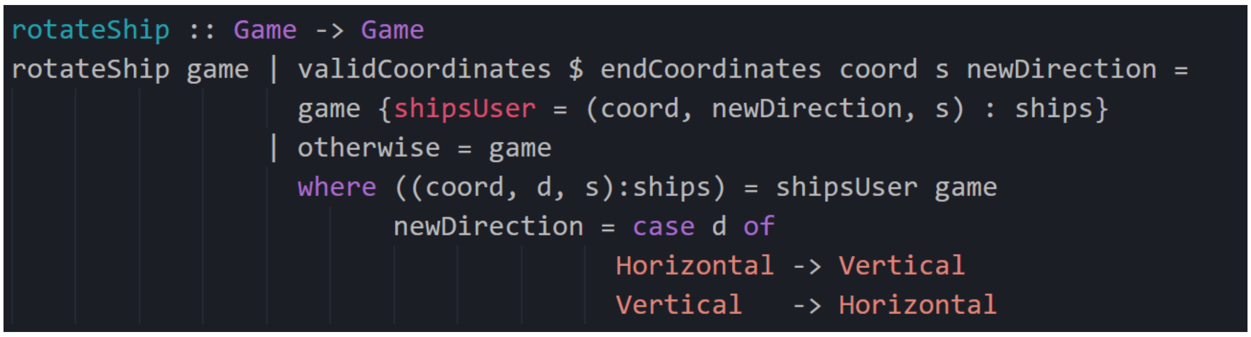


```
moveShip :: Game -> SpecialKey -> Game
moveShip game keyDir | validCoordinates (endCoordinates newCoord s d)
                       && validCoordinates newCoord = game {shipsUser = (newCoord, d, s) : ships}
                     | otherwise = game
                      where (((c, r), d, s):ships) = shipsUser game
                            newCoord =  case keyDir of
                                        KeyLeft  -> (c - 1, r)
                                        KeyRight -> (c + 1, r)
                                        KeyUp    -> (c, r + 1)
                                        KeyDown  -> (c, r - 1)
                                        _        -> (c, r)
```

This function takes two arguments, *Game*, representing the current game state, and *SpecialKey*, the key the user pressed. ShipsUser is a list of ships inside of the game state data type. Extracting the very first element in this list is the ship currently being shown on the screen. A ship is represented by ((c,r), d, s) where (c,r) is the starting coordinates of the ship, d is what direction the ship has, and s is the total size the ship occupies. Applying a case to the keyDir argument allows the function to calculate the new starting coordinates of the ship, depending on what key the user clicked. It then checks if these new coordinates are valid using the **validCoordinates** and **endCoordinates** functions. If they are valid the extracted ship is assigned the new coordinates, and prepended to the list of ships (shipsUser) in the game state. If any of the coordinates are invalid, no changes are made to the game state.

rotateShip

Changes the current placing ship's direction to the opposite

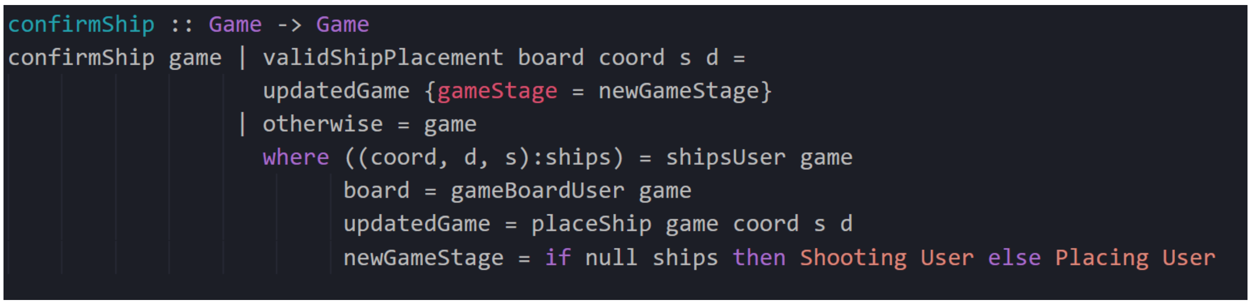


```haskell
rotateShip :: Game -> Game
rotateShip game | validCoordinates $ endCoordinates coord s newDirection =
                  game {shipsUser = (coord, newDirection, s) : ships}
                | otherwise = game
                  where ((coord, d, s):ships) = shipsUser game
                        newDirection = case d of
                                     Horizontal -> Vertical
                                     Vertical   -> Horizontal
```

This function takes a single argument, *Game*, representing the current game state. Similar to the moveShip function it extracts the first ship from shipsUser. It checks if the end coordinates of the same ship but with opposite direction would be valid using the **validCoordinates** and **endCoordinates** functions. This means that a ship that would have any part outside of the board if it was rotated counts as invalid and thus cannot be rotated. If it is valid the extracted ship's direction is updated to its opposite and prepended to the list of ships (shipsUser). Otherwise the game state is left unchanged and returned.

confirmShip

Puts current placing ship on board and changes gameStage if all ships have been placed



```haskell
confirmShip :: Game -> Game
confirmShip game | validShipPlacement board coord s d =
                   updatedGame {gameStage = newGameStage}
                 | otherwise = game
                   where ((coord, d, s):ships) = shipsUser game
                         board = gameBoardUser game
                         updatedGame = placeShip game coord s d
                         newGameStage = if null ships then Shooting User else Placing User
```

This function takes a single argument, *Game*, representing the current game state. It extracts the current placing ship from shipsUser and checks if the placing of that ship would be valid using the function **validShipPlacement**.

If yes, it returns the game state where the ship have been placed, utilizing the **placeShip** function. If that was the last ship in shipsUser the gameStage is also updated, to Shooting User, meaning that now the user can shoot on the AI's board. If the placement is invalid it returns the game state unchanged.

placeShip

Places a ship with certain starting coordinates, size and direction, on the gameBoardUser inside of the given game state

```haskell
placeShip :: Game -> CellCoord -> ShipSize -> Direction -> Game
placeShip game _ 0 _= game
placeShip game coord s d | validShipPlacement board coord s d =
                           game {gameBoardUser = placeShipAux board coord s d,
                                 shipsUser     = tail $ shipsUser game}
                         | otherwise = game
                        where board = gameBoardUser game
```

This function takes the current game state, starting coordinates, size and direction of a ship. It checks if the ship placement is valid using the **validShipPlacement** function. If it is valid it updates gameBoardUser inside of the game state data type. It places the ship on the user board using the **placeShipAux** function. After doing so it updates the list of ships inside of *shipsUser*. Thus the user now has one less ship to place until all ships have been placed. If the ship placement is invalid it returns the game state unchanged.

placeShipAux

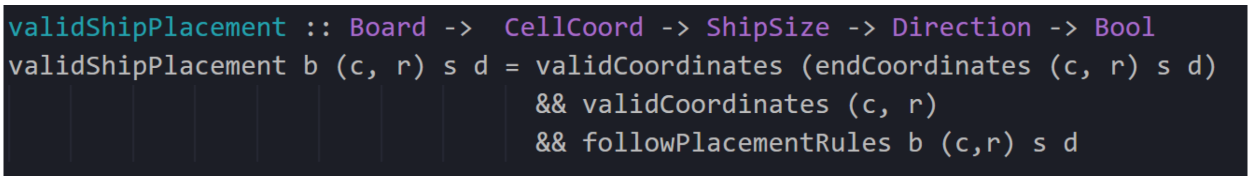
Places a ship with certain starting coordinates, size and direction, on the given board

```haskell
placeShipAux :: Board -> CellCoord -> ShipSize -> Direction -> Board
placeShipAux b _ 0 _= b
placeShipAux b (c, r) s Vertical = placeShipAux (b // [((c, r), Ship NotChecked)]) (c, r - 1) (s - 1) Vertical
placeShipAux b (c, r) s Horizontal = placeShipAux (b // [((c, r), Ship NotChecked)]) (c + 1, r) (s - 1) Horizontal
```

This is a recursive function that takes a board, starting coordinates, size and direction. It works as a helper function for **placeShip** and places a single ship. It has three different pattern matchings, where the two last ones are the inductive cases. It calls itself recursively with the board where it has changed the cell at the coordinates (c,r) to *Ship NotChecked*. There it also has changed the coordinates depending on the direction and the size of the ship is decremented by 1. It does so until it reaches the base case, when the size of the ship is 0. Then, the whole ship is placed on the board, and it is returned.

validShipPlacement

Check if ship placement is valid

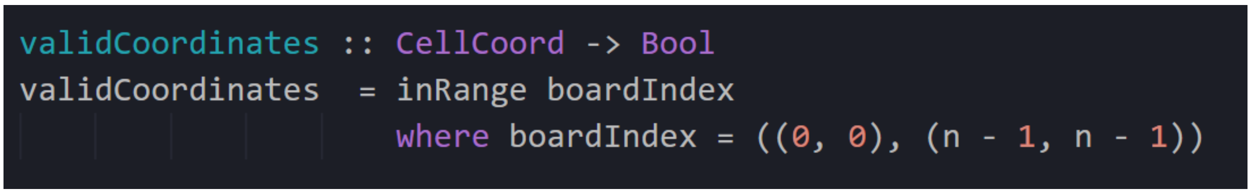


```haskell
validShipPlacement :: Board ->  CellCoord -> ShipSize -> Direction -> Bool
validShipPlacement b (c, r) s d = validCoordinates (endCoordinates (c, r) s d)
                                  && validCoordinates (c, r)
                                  && followPlacementRules b (c,r) s d
```

This function takes a board, the starting coordinates of a ship, the size of the ship and its direction, and determines if it is a valid placement. Both the starting coordinates and the end coordinates are checked for validity using a function called *validCoordinates*. This is to make sure the whole ship stays within the board. Next up, a function called *followPlacementRules* is called, and this function checks that none of the surrounding cells contain a ship. This is to make sure all ship placements follow the placement rules presented in section 3.2. If either of these function calls returns False it results in an invalid placing, and thus does not allow the player to place that ship.

validCoordinates

Checks if coordinates are within the range of a board

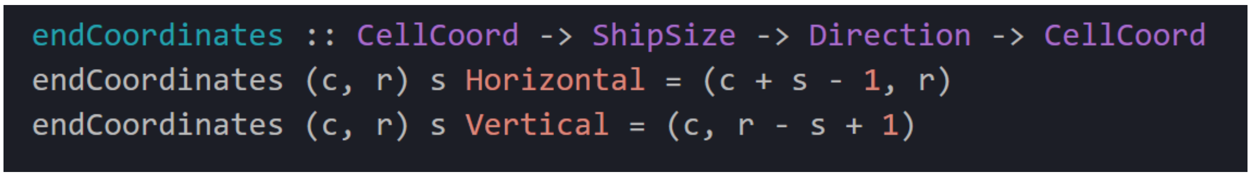


```haskell
validCoordinates :: CellCoord -> Bool
validCoordinates  = inRange boardIndex
                    where boardIndex = ((0, 0), (n - 1, n - 1))
```

This function takes a single cell coordinate as the argument, and checks if it's inside of the board. This function utilizes the global n value (default value of n is 10, meaning a 10x10 playing board).

endCoordinates

Calculates the end position of a ship



```haskell
endCoordinates :: CellCoord -> ShipSize -> Direction -> CellCoord
endCoordinates (c, r) s Horizontal = (c + s - 1, r)
endCoordinates (c, r) s Vertical = (c, r - s + 1)
```

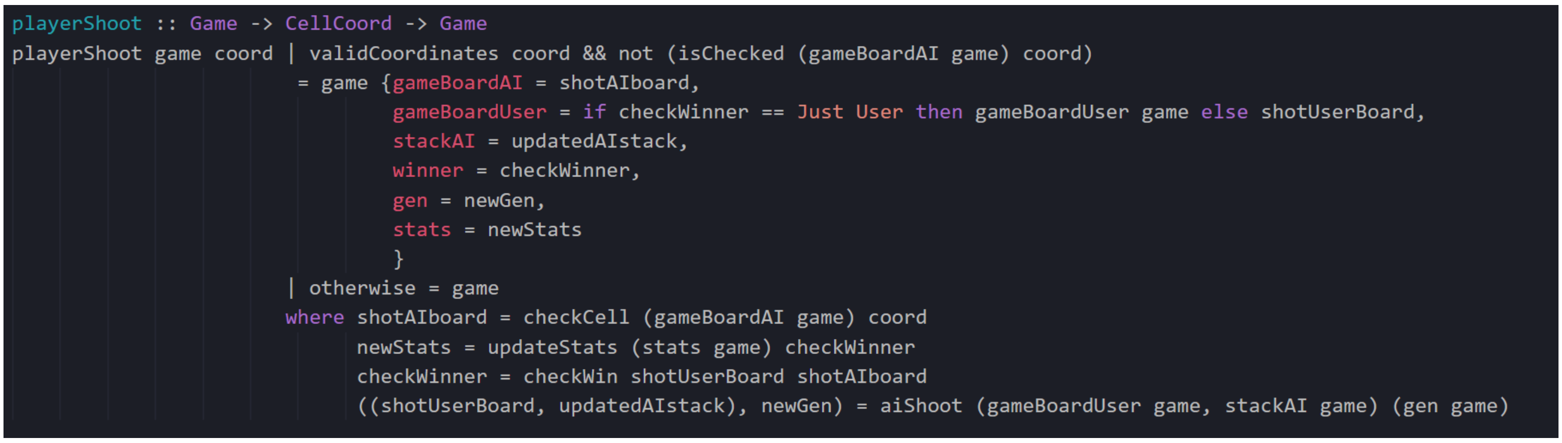
This function takes the starting coordinates of a ship, the size of the ship, and its direction. From this it will calculate the end coordinates of that ship and return it.

4.6.3 Shooting

In the eventHandler a function called *playerShoot* was introduced. It will be explained below.

playerShoot

Shoots the coordinates for user. Calls AI to shoot. Updates game accordingly.



```haskell
playerShoot :: Game -> CellCoord -> Game
playerShoot game coord | validCoordinates coord && not (isChecked (gameBoardAI game) coord)
                        = game {gameBoardAI = shotAIboard,
                                gameBoardUser = if checkWinner == Just User then gameBoardUser game else shotUserBoard,
                                stackAI = updatedAIstack,
                                winner = checkWinner,
                                gen = newGen,
                                stats = newStats
                                }
                       | otherwise = game
                       where shotAIboard = checkCell (gameBoardAI game) coord
                             newStats = updateStats (stats game) checkWinner
                             checkWinner = checkWin shotUserBoard shotAIboard
                             ((shotUserBoard, updatedAIstack), newGen) = aiShoot (gameBoardUser game, stackAI game) (gen game)
```

The function takes only two arguments, the current game state and the cell coordinates of the cell the user wants to shoot. In the eventHandler, these cell coordinates are created from a function called *mouseToCell* that converts the coordinates of the mouse to cell coordinates. If the coordinates are valid and the cell at those coordinates is not checked it will return an updated game state. The updated game state contains the new board that have checked the given coordinates, and if there still is no winner at that point, the AI will shoot on the user's board. More on that in the AI section. It also updates the winner, and the stats if someone won, using the *checkWin* and *updateStats* functions. If the coordinates are invalid and/or the cell is already checked the function returns the game state unchanged.

4.7 Graphics

The graphics are made by the rendering and animation modules. The rendering module deals with the visualization of the game while the animation module updates the game for each time step.

The **drawGame** function turns the game into a picture which then can be showed by the gloss-GUI.

Rendered objects by **drawGame**:

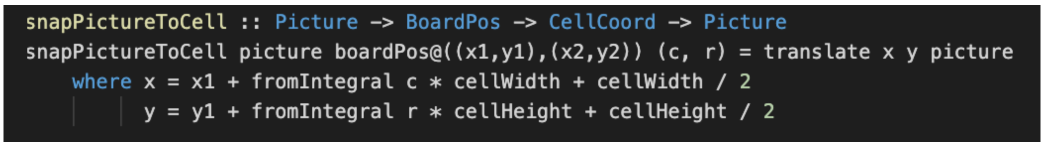
1. gameboards
    (a) grids
    (b) cells
        i. empty
        ii. miss
        iii. checked ship
        iv. not checked ship
2. moving ship
3. user info
    (a) current stage
    (b) score and round
    (c) instructions
4. animation
    (a) radar
    (b) explosion

For all objects above **drawGame**:

- assigns picture
- assigns position on screen
- applies coloring to the picture

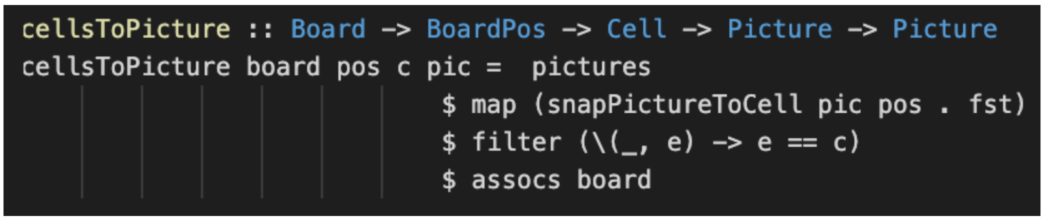
The drawGame-picture is built up step by step by different functions and pictures. For instance, the board cells are combined into a single picture like this:

1. the cell contents and positions are extracted from the board.
2. depending on cell content each cell gets an assigned cell-picture (**crossPicture**, **shipPicture** or none).
3. **snapPictureToCell** - cell-picture is translated to cell location



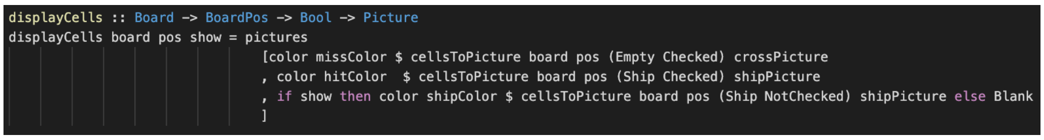
```
snapPictureToCell :: Picture -> BoardPos -> CellCoord -> Picture
snapPictureToCell picture boardPos@((x1,y1),(x2,y2)) (c, r) = translate x y picture
    where x = x1 + fromIntegral c * cellWidth + cellWidth / 2
          y = y1 + fromIntegral r * cellHeight + cellHeight / 2
```

4. **cellsToPicture** - all cells with the same content get combined into a single picture of positioned cell-pictures from previous step.



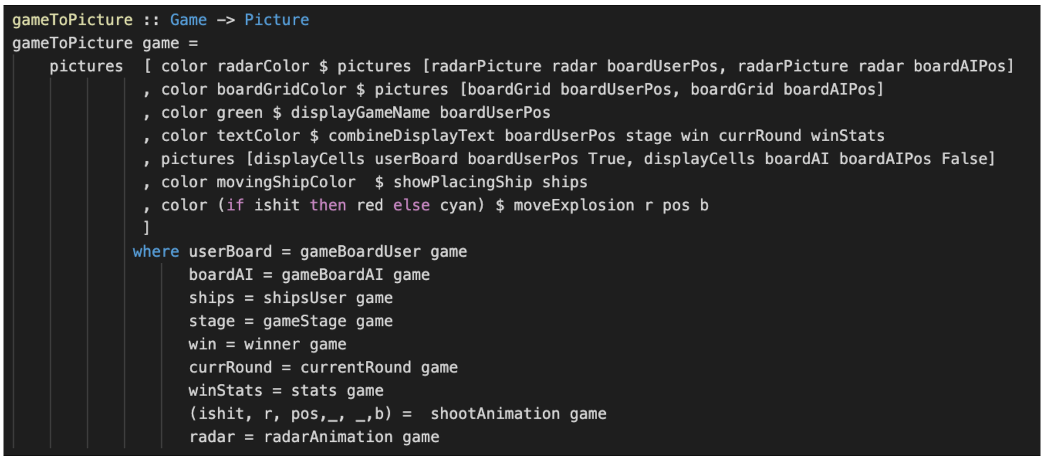
```
cellsToPicture :: Board -> BoardPos -> Cell -> Picture -> Picture
cellsToPicture board pos c pic =  pictures
                          $ map (snapPictureToCell pic pos . fst)
                          $ filter (\(_, e) -> e == c)
                          $ assocs board
```

5. **displayCells** - applying color and combining all cells of different types into a single picture



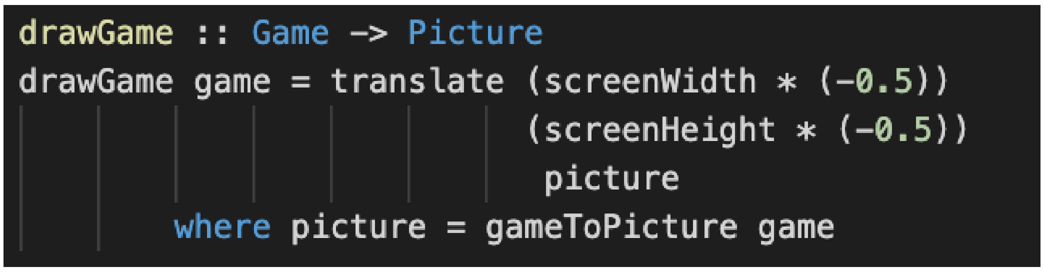
```
displayCells :: Board -> BoardPos -> Bool -> Picture
displayCells board pos show = pictures
                                [color missColor $ cellsToPicture board pos (Empty Checked) crossPicture
                                , color hitColor  $ cellsToPicture board pos (Ship Checked) shipPicture
                                , if show then color shipColor $ cellsToPicture board pos (Ship NotChecked) shipPicture else Blank
                                ]
```

The board cells are then combined with the other objects' pictures into a single picture with **gameToPicture**.



```haskell
gameToPicture :: Game -> Picture
gameToPicture game =
    pictures  [ color radarColor $ pictures [radarPicture radar boardUserPos, radarPicture radar boardAIPos]
              , color boardGridColor $ pictures [boardGrid boardUserPos, boardGrid boardAIPos]
              , color green $ displayGameName boardUserPos
              , color textColor $ combineDisplayText boardUserPos stage win currRound winStats
              , pictures [displayCells userBoard boardUserPos True, displayCells boardAI boardAIPos False]
              , color movingShipColor  $ showPlacingShip ships
              , color (if ishit then red else cyan) $ moveExplosion r pos b
              ]
             where userBoard = gameBoardUser game
                   boardAI = gameBoardAI game
                   ships = shipsUser game
                   stage = gameStage game
                   win = winner game
                   currRound = currentRound game
                   winStats = stats game
                   (ishit, r, pos,_, _,b) =  shootAnimation game
                   radar = radarAnimation game
```

To adjust for the starting position of (0,0) in the middle of the screen by gloss, the final picture gets translated into the lower bottom left corner by **drawGame**. The reason for this is that we used Tsoding's rendering functions (see Appendix) as a base for the rendering.



```haskell
drawGame :: Game -> Picture
drawGame game = translate (screenWidth * (-0.5))
                          (screenHeight * (-0.5))
                           picture
        where picture = gameToPicture game
```

animationFunc

The animation of the radar and the explosions are fairly simple. For each time step the angle of the radar arc increases, which results in a spinning motion. The graphics are still taken care of in the rendering module. **animationFunc** only updates the game which then gets fed into **drawGame**.

The same is done for the explosions. The radius of the explosion increases with time, but slower and slower. To make the explosion finish at a specific radius, the showing parameter of the explosion is changed to False if this boundary has been reached. Since the explosion is triggered by user input, the logic module tells what color should be rendered.

4.8 AI

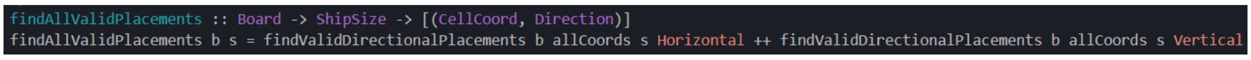
The AI we have created uses different algorithms to determine where it will place its ships and shoot. Since the placing-part and shooting-part completely differs from each other, we will explain them separately.

4.8.1 Placing

The placing-algorithm begins with calculating every valid placement of a ship on a board. It then creates a list of those placements for both directions, vertical and horizontal, and randomly picks one to update the board with. The same thing will then be done with the next ship on the new updated board, and this repeats until every given ship is placed. In result, the placements will be of random coordinates in random directions.

findAllValidPlacements

Finds all valid placements of a ship on board

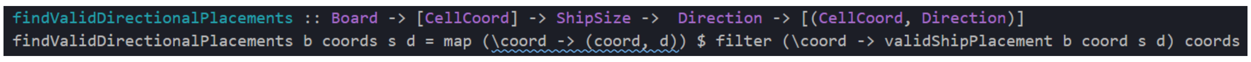


```
findAllValidPlacements :: Board -> ShipSize -> [(CellCoord, Direction)]
findAllValidPlacements b s = findValidDirectionalPlacements b allCoords s Horizontal ++ findValidDirectionalPlacements b allCoords s Vertical
```

To find all the valid placements of a ship, the function *findAllValidPlacements* is used. It takes the current board and the size of the ship. Then, it creates a list of every valid Horizontal placement and concatenates it with a list of every valid vertical placement for all coordinates, with the help of *findValidDirectionalPlacements*.

findValidDirectionalPlacements

Finds all possible positions of a ship in a certain direction

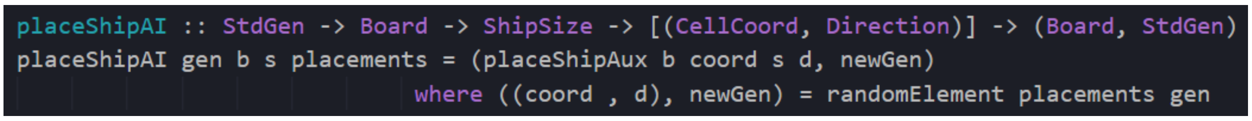


```
findValidDirectionalPlacements :: Board -> [CellCoord] -> ShipSize ->  Direction -> [(CellCoord, Direction)]
findValidDirectionalPlacements b coords s d = map (\coord -> (coord, d)) $ filter (\coord -> validShipPlacement b coord s d) coords
```

This function takes the current Board, a list of the starting coordinates for the ship, the size of the ship and a Direction. It then creates a list of tuples with each starting coordinates and the direction and filters it to the elements with valid placements.

placeShipAI

Updates board with a random placement of ship

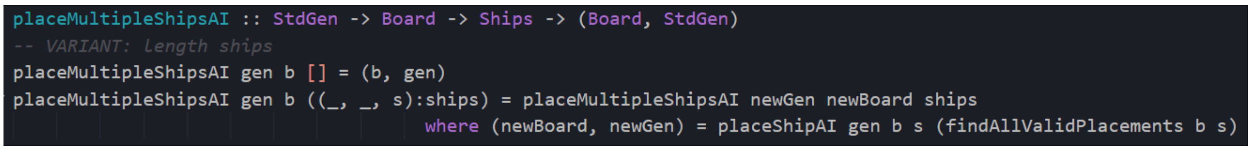


```haskell
placeShipAI :: StdGen -> Board -> ShipSize -> [(CellCoord, Direction)] -> (Board, StdGen)
placeShipAI gen b s placements = (placeShipAux b coord s d, newGen)
                                where ((coord , d), newGen) = randomElement placements gen
```

When the list of the valid placements have been made, it is time to place down the ships with *placeShipsAI*. A random valid placement is then picked with *randomElement* and placed with the help of *placeShipAux*, explained in the Logic section. Briefly explained, *randomElement* is a function that takes a list of some type and a random seed and picks a random element of the list with the help of the random seed.

placeMultipleShipsAI

Places ships on random places on the board



```haskell
placeMultipleShipsAI :: StdGen -> Board -> Ships -> (Board, StdGen)
-- VARIANT: length ships
placeMultipleShipsAI gen b [] = (b, gen)
placeMultipleShipsAI gen b ((_, _, s):ships) = placeMultipleShipsAI newGen newBoard ships
                                              where (newBoard, newGen) = placeShipAI gen b s (findAllValidPlacements b s)
```

To place every given ship, *placeMultipleShipsAI* is used. It takes a random seed, the current updated board, a list of given ships and places every ship in the list on the board with the help of *placeShipsAI*. It terminates when every ship in the list has been placed. Once every ship has been placed, the game moves on to its next stage; shooting.

4.8.2 Shooting

For the shooting, the AI mainly utilizes three algorithms. These algorithms are called randomized guessing, parity and hunt. It is worth mentioning that this AI is not very "smart", which is the biggest shortcoming for this game. See section 5 for more information regarding the shortcomings.

The randomized guessing is exactly what it sounds like. With the help of the library *System.Random*, our AI shoots at random cells on the board. To know which cells to shoot at, a ShootList is created with a random order. The element of random makes for a better experience when playing against the AI. However, this algorithm alone makes for a very bad AI that guesses on unnecessarily many cells.

To further improve the AI we therefore also implemented a so called parity-algorithm. To help understand how parity works, we can use chess as a parable. A chess board consists of both black and white squares, where every other square is black and respectively white. If we imagine the grid of Battleships to look the same way, the parity algorithm basically only guesses on one of the colors. This results in much more effective guesses, since it only shoots at every other square. That is, until it hits a ship.

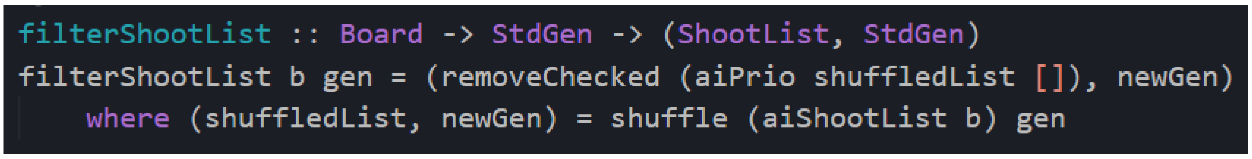
When the AI guesses on a correct cell, the hunt-algorithm begins. This method takes every cohesive cell to a correctly guessed cell and adds it to a stack. The stack is what the AI uses to guess on cells. In other words, the AI will guess on all cohesive cells to a Ship.

In conclusion, the AI shoots randomly among every other square until it hits a ship, whereas it shoots every cohesive square to the ship. With this algorithm in mind, we can now explain the code behind it.

The main function to the shooting part of the AI is *aiShoot*. This praticular function calls on all of the other AI shooting functions. To understand what it does, some of the other functions has to be explained more thoroughly.

filterShootList

Creates a ShootList with random order of all cells with state NotChecked from board and sorts by priority

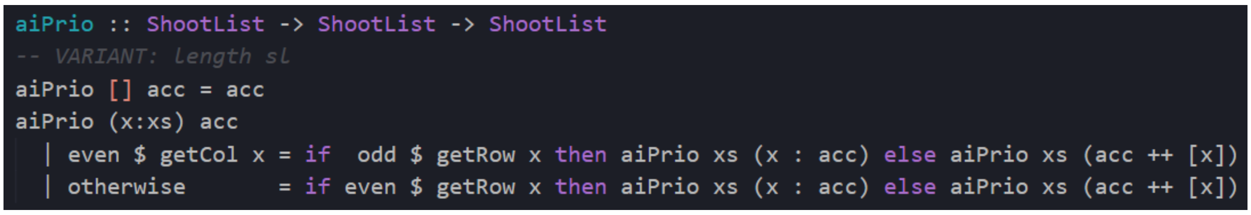


```haskell
filterShootList :: Board -> StdGen -> (ShootList, StdGen)
filterShootList b gen = (removeChecked (aiPrio shuffledList []), newGen)
    where (shuffledList, newGen) = shuffle (aiShootList b) gen
```

As mentioned, the AI uses a random ShootList to shoot at cells. The ShootList is finalized with the function *filterShootList*, which takes the current board and a random seed. This function creates a ShootList with *aiShootList*, randomizes it with *shuffle* and combines the necessary adjustments to it with *removeChecked* and *aiPrio*. The *removeChecked*-function removes already checked cells from the list by filtering out cells with state Checked, and *aiPrio* sorts it by priority.

aiPrio

Sorts a ShootList so AI prioritises cells that are not next to each other.

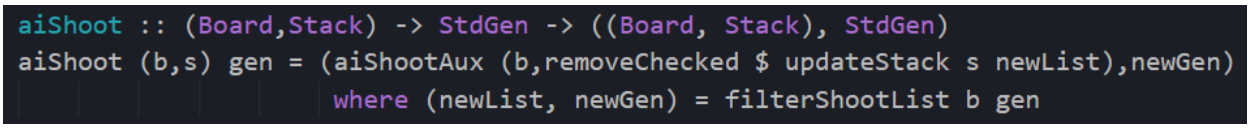


```haskell
aiPrio :: ShootList -> ShootList -> ShootList
-- VARIANT: length sl
aiPrio [] acc = acc
aiPrio (x:xs) acc
  | even $ getCol x = if  odd $ getRow x then aiPrio xs (x : acc) else aiPrio xs (acc ++ [x])
  | otherwise       = if even $ getRow x then aiPrio xs (x : acc) else aiPrio xs (acc ++ [x])
```

This function sorts a ShootList to prioritize every other element first, but still preserves a random order among its prioritized elements, with the help of an accumulator. This is basically what makes the parity-algorithm mentioned earlier work. If a cell has an even column number and an odd row number or if it has an odd column number and an even row number it will be prepended to the accumulator, meaning it is prioritized. Otherwise it will be appended, meaning it is not prioritized. The function terminates once every element on the ShootList has been moved to the accumulator. The final result is that the AI will make random guesses among every other square.

aiShoot

Checks the first cell in the Stack, if stack is empty it checks the first cell in current ShootList

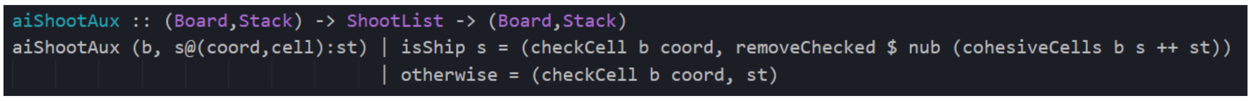


```haskell
aiShoot :: (Board,Stack) -> StdGen -> ((Board, Stack), StdGen)
aiShoot (b,s) gen = (aiShootAux (b,removeChecked $ updateStack s newList),newGen)
                     where (newList, newGen) = filterShootList b gen
```

The main function for the shooting part of the AI, *aiShoot*, takes the current board, a stack of the next cells it will shoot at and a random seed. It then checks the first cell in the stack and updates the board. If the stack is empty, it will be updated with the first cell in the ShootList created with *filter-ShootList*. This is done with the help of the function *updateStack*. It handles the different cases, a hit or a miss, with its helper function *aiShootAux*.

aiShootAux

Checks the first cell in the Stack



```haskell
aiShootAux :: (Board,Stack) -> ShootList -> (Board,Stack)
aiShootAux (b, s@(coord,cell):st) | isShip s = (checkCell b coord, removeChecked $ nub (cohesiveCells b s ++ st))
                                  | otherwise = (checkCell b coord, st)
```

This function takes the current board, a stack with at least one element and the filtered and sorted ShootList. If the first cell in the stack is a ship, the hunt-algorithm will begin. With the help of *cohesiveCells*, all cohesive cells will be prepended to the stack, and the first cell in the stack which was a ship, will be checked. The board will also be updated. If the first cell is not a ship, it will only be checked and removed from the stack. This means that the stack eventually can become empty, but in *aiShoot* it makes sure that an empty stack never will be used as an argument in *aiShootAux*, so that is not a problem.

5 Shortcomings

Improved AI

When the AI already was created, we decided to implement a new game rule which prevents placements next to an already placed ship. However, our AI was originally created with the intention of checking all cells next to a ship to find ships placed next to each other. This resulted in the AI now searching unnecessarily many cells around a ship instead of moving on to the random guessing when finding an entire ship. After many attempts of adapting the AI to this new game rule, we were not able to successfully modify it completely.

We got as far as the AI having massive improvements for ships placed vertically, but it was no improvement for horizontally placed ships. Since we did not want the AI to only have an advantage for vertical ships, we decided to discard the new improvements. The reasoning to why we did not improve the AI for horizontal ships as well, is because it got quite complex with all special cases and therefore it was extremely time consuming to make the improvements. Since time was something we not had a lot of at this point, we determined it was better to focus on other aspects of the project.

Using Stack package

The AI uses a stack as its structure for making its next guess. Unfortunately, this is not a stack from the package *Data.Stack*, but rather a stack-type we ourselves constructed. The reason for this is because we had a lot of problems when trying to install and implement new packages and libraries to the project. When we finally had gotten one particular library to start working, another library which already worked perfectly stopped working. This was a reoccurring issue which was persistent throughout the project, and resulted in us working on our own simplified stack implementation.

Multiplayer

Before we decided to create a Battleships game from scratch, we wanted to create a program which worked for multiple users simultaneously. When brainstorming the possibilities of making games, we thought of multiplayer to be a very interesting aspect to implement. However, upon hours of research and some attempts on making a server in Haskell, we realised it was too difficult for us since we had no prior knowledge within networking.

6 Appendix

Inspiration & Credit

Credit to Tsoding.
https://github.com/tsoding/profun/blob/master/functional/src/Rendering.hs

We have used the following functions by Tsoding:

- xCell
- snapPictureToCell
- cellsOfBoard
- boardGrid