COM2108 Written Report: Texas Hold’ Em Poker

By Shohail Ismail – ACA23SI

Contents

AI disclaimer

No generative AI tools were used in the preparation of the solution to this work. This refers to both the code and the report parts.

General disclaimer

For the development and testing of the Texas Hold'em Poker Assignment in Haskell, the VSCode editor was chosen. This was due to it being my preferred choice of IDE when developing programs and due to its support for Haskell in the form of extensions, of which I used:

- Haskell (Haskell Language Support) by Haskell
- Haskell Syntax Highlighting by Justus Adam

Design

Top-down approach

The design of this Texas Hold 'Em implementation in Haskell will follow a top-down approach. To do this, the problem will first be decomposed into the core components then into the main functions, which allows for the design to remain modular while being clear.

Core components

Overview

Data structures

- **Card**: Represents every card in a 52-deck using a list of pairs of type [Suit, Rank]:
    - **Suit** – Hearts, Diamonds, Clubs, or Spades
    - **Rank** – 2-10, Jack, Queen, King, Ace

- **Deck:** A list of 52 Cards with one of every suit/rank combination.

- **HandRank:** Represents every hand that can arise in a value-based ranking (weakest 'HighCard' to strongest 'RoyalFlush'). As Haskell assigns a default ordering based on the order in which the constructors are defined in the data type, this will involve starting with the weakest and ending with the strongest hands.

- **PlayerMove:** Represents all possible actions a player can make (i.e., check, fold, call, bet, raise)

- **Player**: Stores all the attributes of a player in a game.
    - name (String) – Player's name
    - holeCards ([Card]) – Player's 2 hole cards
    - chips (Int) – Number of player's chips
    - isDealer (Boolean) – Checks if player is dealer currently
    - isActive (Boolean) – Checks if player is still in game
    - playerType ([PlayerType]) – Player's strategy type (RandomPlayer, PassivePlayer, SmartPlayer, AggressivePlayer).

- **PlayerType:** The strategy type of a certain player, and can be RandomPlayer, PassivePlayer, AggressivePlayer, or SmartPlayer.

- **GameState**: Maintains game-wide information.
    - deck ([Card]) – List of cards remaining in the deck
    - communityCards ([Card]) – 5 shared community cards
    - pot (Int) – Total chips in the pot
    - players ([Player]) – List of players still active in the current game
    - smallBlind (Int) – Small blind bet amount

- bigBlind (Int) – Big blind bet amount
- currentDealer (Int) – Index of current dealer in the list of players. Preferred over type ‘Player’ to simplify game logic and not violate Single Source Of Truth (SSOT) principle.
- currentBets ([Int]) – List of current bets for each player (by index in players list)

Functions

- **Card operations:**
  - **shuffleDeck** – To ensure fairness, the shuffleDeck function uses ‘System.Random’ to generate an unbiased distribution of cards
  - **dealCards** – Distributes a specified number of cards either to players (hole cards) or the table (community cards)

- **Hand evaluation:**
  - **evaluateHand** – Analyses a player’s cards (hole + community cards) and determines the best 5-card hand.
  - **compareHands** – Compares two hands to determine which one ranks higher (also can resolve ties by considering secondary criteria). This will be implemented as a separate function to evaluateHands to prevent evaluateHands from becoming too large and encompassing different functionalities, as that would be complex to test, debug, maintain, and does not adhere to best practices.

- **Betting:**
  - **bettingRound** – Allows for one betting round, where each player makes decisions based on their cards, strategy and chips.
  - **playerAction** – Executes a specific player's action (i.e., an action from PlayerMove).

- **Round mechanics**:
    - **initGame** – Sets up the initial GameState according to rules of current game.
    - **updateGame** – Updates the GameState after each round/action.
    - **gameLoop** – Handles the loop of dealing, running betting rounds, evaluating hands, and determining the winner. Stops the game when a single player remains with chips or when the maximum round bound (100 rounds) is reached.
    - **determineWinner** – Identifies the player(s) with the best hand after all betting rounds are complete, handling ties by splitting the pot.
    - **endRound** – Concludes a game round by awarding the pot to the winner(s), updating chip count, and resetting pot.
    - **rotateDealer** – Rotates the dealer position among players.

- **Player strategies:**
    - **RandomPlayer –** Simulates random decisions made by a player.
    - **PassivePlayer –** Simulates a cautious player who only checks, calls or folds.
    - **AggressivePlayer –** Simulates a bold player who frequently makes bets and raises.
    - **SmartPlayer –** Simulates a player who adjusts their actions to be optimal dependant on hand strength and the current GameState.

Detailed overview

Data structures

Card

```
data Card = Card { suit :: Suit, rank :: Rank }
```

Suit, Rank

```
data Suit = Hearts | Diamonds | Clubs | Spades
```

```
data Rank = Two | Three | Four | Five | Six | Seven |
Eight | Nine | Ten | Jack | Queen | King | Ace
```

Deck

```
type Deck = [Card]
```

HandRank

```
data HandRank = HighCard | OnePair | TwoPair |
ThreeOfAKind | Straight | Flush | FullHouse |
FourOfAKind | StraightFlush | RoyalFlush
```

PlayerMove

```
data PlayerMove = Fold | Check | Call | Bet Int |
Raise Int
```

Player

```
data Player = Player
    { playerType :: PlayerType
    , name :: String
    , holeCards :: [Card]
    , chips :: Int
    , isDealer :: Bool
    , isActive :: Bool
    }
```

PlayerType

```
data PlayerType = RandomPlayer | SmartPlayer |
AggressivePlayer | PassivePlayer
```

GameState

```
data GameState = GameState
    { deck :: Deck
    , communityCards :: [Card]
    , pot ::Int
    , players :: [Player]
    , smallBlind :: Int
    , bigBlind :: Int
    , currentDealer :: Int
    , currentBets :: [Int]
    }
```

Functions

initGame

```
initGame :: Int -> Int -> Int -> [PlayerType] ->
GameState
```

- **Inputs**: Number of players, initial chip count for each player, small and big blind amounts, list of players.
- **Outputs**: The initial GameState containing an empty pot, a shuffled deck, and no community cards

shuffleDeck

```
shuffleDeck :: [Card] -> IO [Card]
```

- **Inputs**: An ordered list of cards ([Card]).
- **Outputs**: A shuffled list of cards using the 'System.Random' library for randomness.

dealCards

`dealCards :: Int -> [Card] -> ([Card], [Card])`

- **Inputs**: Number of cards to deal, the current Deck.
- **Outputs**: A tuple containing dealt cards and the updated Deck.

evaluateHand

`evaluateHand :: [Card] -> HandRank`

- **Inputs**: A list of >= 5 cards.
- **Outputs**: The HandRank (not addressed in 'Data structures' as it is not persistently stored so it is an auxiliary type).

compareHands

`compareHands :: [Card] -> [Card] -> Ordering`

- **Inputs**: Two sets of cards (representing hands).
- **Outputs**: An Ordering (i.e., <, ==, >) indicating which hand is stronger.

determineWinner

`determineWinner :: [Player] -> [Player]`

- **Inputs**: List of players with their evaluated hands.
- **Outputs**: A list of winning player(s).

endRound

`endRound :: GameState -> GameState`

- **Inputs**: Current GameState.
- **Outputs**: Updated GameState.

updateGame

`updateGame :: GameState -> GameState`

- **Inputs**: Current GameState.
- **Outputs**: Updated GameState.

rotateDealer

`rotateDealer :: [Player] -> [Player]`

- **Inputs**: List of players.
- **Outputs**: Updated list of players with the new dealer.

gameLoop

`gameLoop :: GameState -> GameState`

- **Inputs**: Initial GameState.
- **Outputs**: Final GameState after the game is over.

bettingRound

`bettingRound :: GameState -> GameState`

- **Inputs**: Current GameState.
- **Outputs**: Updated GameState.

playerAction

`playerAction :: GameState -> Player -> PlayerMove`

- **Inputs**: Current GameState, the player making the action.
- **Outputs**: The PlayerMove taken by the player.

gameLoop Function

Breakdown

The gameLoop function defines the progression of the current game and encapsulates all the key elements, managing the sequence of events until the game concludes either by a single player holding all chips or when 100 rounds are reached.

1. Initialisation phase
    - Each iteration of the loop represents one round of the game.
    - Cards are dealt with dealCards - originally 2 hole cards (then eventually the community cards).
    - Betting is done using bettingRound, with each player adhering to their pre-assigned strategies.
    - The community cards are revealed, and betting begins again.
    - The winner is evaluated with determineWinner.
    - GameState is updated with updateGame, the pot is redistributed, and variables are reset with endRound.
2. Termination
    - The loop is active until only one player remains with chips or 100 rounds are reached.

Pseudocode

While the pre-defined functions have not been written in pseudocode – as this will be done during the implementation phase – helper functions unique to gameLoop have been defined in pseudocode:

```
-- Loops the game until termination condition reached
function gameLoop(gameState):
    -- Sets up the initial params of game
    gameState = initGame(numPlayers, initialChips,
        smallBlind, playerNames)

    while not gameOver(gameState):
        gameState = runRound(gameState)
    return gameState
```

```
-- Runs one round of poker
function runRound(gameState):
    -- Shuffle the deck
    shuffledDeck = shuffleDeck(gameState.deck)

    -- Deals 2 hole cards to each player
    (holeCards, updatedDeck) = dealCards(2, shuffledDeck)

    -- Updates the deck in current GameState
    gameState.deck = updatedDeck

    -- Assigns cards to players
    gameState.players = assignCards(holeCards, gameState.players)

    -- Pre-flop betting round
    gameState = bettingRound(gameState)

    -- Deals 3 community cards and allows bet (flop)
    gameState = dealAndBet(3, gameState)

    -- Deals 1 more community card and allows bet (turn)
    gameState = dealAndBet(1, gameState)

    -- Deals final community card and allows bet (river)
    gameState = dealAndBet(1, gameState)

    -- Finds winner(s) based on hand strenght
    winners = determineWinner(gameState.players)

    -- Awards pot and resets variables
    gameState = endRound(gameState, winners)

    -- Rotates dealer for next round
    gameState = rotateDealer(gameState)

    return gameState
```

```
-- Assigns cards to players
function assignCards(holeCards, players):
    for each player in players:
        -- Deals 3 card only to active players
        if player.isActive:
            player.holeCards = holeCards.pop(2)

    return players
```

```
-- Helper functon to deal cards and take bets
function dealAndBet(numCards, gameState):
    -- Deal cards and update deck
    (communityCards, updatedDeck) = dealCards(numCards, gameState.deck)
    gameState.deck = updatedDeck

    -- Add to community cards list
    gameState.communityCards += communityCards

    -- Performs a betting round
    gameState = bettingRound(gameState)

    return gameState
```

```
-- Checks if termination conditions have been reached
function gameOver(gameState):
    -- Checks number of active players
    activePlayers = countActivePlayers(gameState.players)

    {- Check if only one player is left or
    the maximum number of rounds has been reached -}
    return activePlayers == 1 or gameState.roundsPlayed >= 100
```

Variable Table

Variable/Input data	Step	Output
numPlayers, initialChips, smallBlind, playerNames	Initial game state setup	GameState (initialised with players, chips, blinds and empty pot).

gameState	Termination check	True (if game over) or False (continue loop)
deck (from gameState)	Shuffle deck	shuffledDeck
shuffledDeck	Deal hole cards	(holeCards, updatedDeck) (hole cards dealt to players, updated deck with dealt cards removed)
updatedDeck	Update deck in game state	GameState with updated deck
holeCards, players (from gameState)	Assign cards	‘players’ list updated with each player's holeCards field populated
gameState (updated with hole cards)	Pre-flop betting	Updated GameState with modified pot, chips, and player states
gameState, deck, communityCards	Deal Flop cards and bet	Updated GameState with 3 cards added to communityCards and updated deck
gameState, deck, communityCards	Deal Turn cards and bet	Updated GameState with 1 card added to communityCards and updated deck
gameState, deck, communityCards	Deal River cards and bet	Updated GameState with 1 card added to communityCards and updated deck
players, communityCards (from gameState)	Determine winner(s)	List of winners
gameState, winners	End round	Updated GameState with the pot resetted, chips redistributed, and cleared betting data
gameState	Rotate the dealer	Updated GameState with reassigned dealer and incremented currentDealer

Updated gameState	Recursive call for the next round	The process repeats until gameOver(gameState) returns True
gameState (game finished)	Terminate loop and return final GameState	Final GameState with the winner and final game data

SmartPlayer Function

Breakdown

The SmartPlayer strategy type makes decisions optimally based on the player's hand strength (likelihood of winning based on current cards), the ratio of a bet against the potential reward, and the game context (number of active players and chips). Given the inputs of GameState (i.e., the community cards, current bets, and the pot) and Player (the player's hole cards and remaining chips), this will mean 3 main calculations can be implemented and aggregated for a final decision:

- **calculateHandStrength**
    - Evaluates the rank of the player's current hand by combining the player's hole cards with the community cards and computing a normalised score between 0 and 1, with 1 being the best*. This will likely require many helper functions to generate combinations and evaluate the strength of the hand. Additionally, the strength will be based on the real-life probabilities of gaining such a hand.
- **calculateChipPotRatio**
    - Determines whether the pot odds are favourable to continue playing by calculating the ratio between the cost to call against the potential reward of winning the pot. A smaller ratio indicates the pot is large relative to the bet, therefore making it more favourable to continue playing.
- **calculateAverageChips**
    - Calculates the average number of chips held by players in the game, which helps dictate the ratio of the player's chips against the average. If the player's chips are above the average, an

aggressive playstyle is affordable, whereas a more cautious approach would be adopted at below-average chip levels.

***** One point to note is that each HandRank has a numerical value assigned to it manually even though HandRank is an ordered data type and so can use functions such as fromEnum (which can convert an ordered data type into numerical values). However, as pseudocode must be language-neutral, values have been assigned manually in lieu of using this function.

Pseudocode

This pseudocode maps the functions necessary to implement SmartPlayer.

```
function smartStrategy(gameState, player):
    -- Ensures fold by default if player is not active
    if not player.isActive:
        return "Fold"

    -- Calculates hand strength
    handStrength = calculateHandStrength(player.holeCards +
        gameState.communityCards)

    -- Calculates ratio of a player's chips to pot winnings
    chipPotRatio = calculateChipPotRatio(gameState, player)

     -- Decision if player can check
    if max(gameState.currentBets) == 0:
        return "Check"

    -- Decisions for strong hand (irrespective of pot ratio)
    if handStrength >= 0.8
        return strongHandPlay(gameState, player)
    -- Decisions for mediocre hand with low chip:pot ratio
    else if handStrength >= 0.5 and chipPotRatio <= 0.3:
        return "Call"
    -- Weak card strength or high chip:pot ratio
    else:
        return "Fold"


function calculateHandStrength(cards):
    -- Evaluate best 5-card combination
    bestHand = evaluateHand(cards)

    -- Assign a numerical value to hand based on strength
    handRank = rankHand(bestHand)

    return handRank
```

```
-- Assigns numerical value to hands from 0 - 1
function rankHand(hand):
    if isRoyalFlush(hand):
        return 1.0
    else if isStraightFlush(hand):
        return 0.95
    else if isFourOfAKind(hand):
        return 0.85
    else if isFullHouse(hand):
        return 0.75
    else if isFlush(hand):
        return 0.65
    else if isStraight(hand):
        return 0.55
    else if isThreeOfAKind(hand):
        return 0.45
    else if isTwoPair(hand):
        return 0.35
    else if isOnePair(hand):
        return 0.25
    {- Hand with no combinations has highest card's rank
    normalised by dividing it by 13 (maximum rank) -}
    else:
        return highCard(hand) / 13


-- Checks if hand is a straight flush with an ace
function isRoyalFlush(hand):
    ranks = map(card => card.rank, hand)
    return "Ace" in ranks and isStraightFlush(hand)
```

```
function isStraightFlush(hand):
    return isFlush(hand) and isStraight(hand)


function isFourOfAKind(hand):
    ranks = map(card => card.rank, hand)
    return any(countOccurrencesOfRank(ranks, rank) == 4 for rank in ranks)


function isFullHouse(hand):
    ranks = map(card => card.rank, hand)
    return any(countOccurrencesOfRank(ranks, rank) == 3 for rank in ranks)
           any(countOccurrencesOfRank(ranks, rank) == 2 for rank in ranks)


function isFlush(hand):
    suits = map(card => card.suit, hand)
    return all(suit == suits[0] for suit in suits)


function isStraight(hand):
    ranks = sort(map(card => card.rank, hand))
    return all(ranks[i] == ranks[0] + i for i in range(5))


function isThreeOfAKind(hand):
    ranks = map(card => card.rank, hand)
    return any(countOccurrencesOfRank(ranks, rank) == 3 for rank in ranks)


function isTwoPair(hand):
    ranks = map(card => card.rank, hand)
    pairs = filter(rank => countOccurrencesOfRank(ranks, rank) == 2, ranks
    return len(set(pairs)) == 2
```

```
function isOnePair(hand):
    ranks = map(card => card.rank, hand)
    return any(countOccurrencesOfRan(ranks, rank) == 2 for rank in ranks)


function highCard(hand):
    ranks = map(card => card.rank, hand)
    return max(ranks)


-- Counts occurrences of rank in hand
function countOccurrencesOfRank(hand, rank):
    return len(filter(x => x == rank, hand))


function calculateChipPotRatio(gameState, player):
    -- Finds how much player must contribute to stay in game
    betToCall = max(gameState.currentBets) - player.currentBet

    -- Calculates potential payout if win occurs
    potentialReward = gameState.pot + betToCall

    -- Returns the ratio of the above calculations
    return betToCall / potentialReward


-- Dictates next moves upon holding strong cards
function strongHandPlay(gameState, player):
    averageChips = calculateAverageChips(gameState.players)

    {- If player has significantly more chips than the average,
    aggressively raise, otherwise call -}
    if player.chips > 2 * averageChips:
        return "Raise " + (gameState.currentBets[player.index] * 2)
    else:
        return "Call"
```

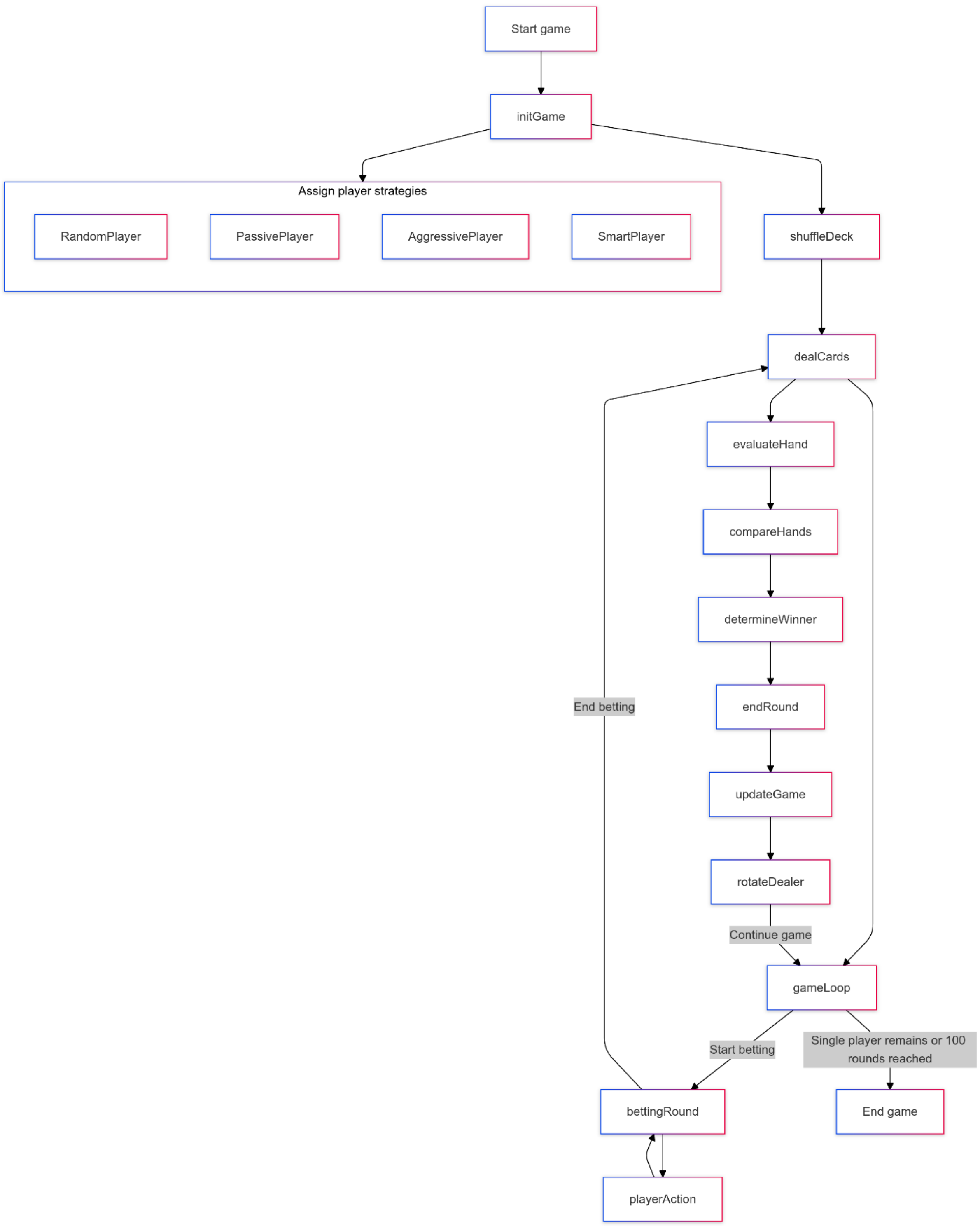
```

-- Calculates average chips held by active players
function calculateAverageChips(players):
    totalChips = 0
    activePlayerCount = 0
    for each player in players:
        if player.isActive:
            totalChips += player.chips
            activePlayerCount += 1
    return totalChips / activeCount
```

Variable Table

Variable/Input data	Step	Output
Player.isActive	Check if player is active	Implicit ‘Fold’ if inactive, else continue
GameState, Player	Initiate smart strategy	‘Fold’, ‘Check’, ‘Call’, or ‘Raise’ based on player's state and hand strength
Player.holeCards, GameState.communityCards	Combines player and community cards for evaluation	Set of cards used to calculate hand strength
cards	Calculate hand strength using calculateHandStrength()	Numerical value (0–1) indicating the strength of the player's strongest hand
hand	Rank hand using rankHand()	Numerical hand rank assignment (e.g., 1.0 for Royal Flush, 0.95 for Straight Flush, etc.)
hand	Check for each hand type	Boolean (True if current hand matches hand type)
GameState, Player	Calculate chips-to-pot ratio using calculateChipPotRatio()	Ratio of betToCall against potentialReward
handStrength >= 0.8	Strong change decision	Call strongHandPlay() for aggressive bets and playstyle
handStrength >= 0.5 && chipPotRatio <= 0.3	Mediocre chance decision	Decision to ‘Call’
handStrength < 0.5	Weak chance decision	Decision to ‘Fold’

GameState, Player	Aggressive play strategy using strongHandPlay()	‘Raise’ or ‘Call’ based on chip count relative to the average
GameState.players	calculateAverageChips()	Average chips held by all currently active players
hand, rank	Count number of times a certain rank appears in a hand (useful for hand strength evaluation functions)	Number of times a rank appears in a given hand

Flowchart



This flowchart is not definitive by any means as it shows only the main functions which, apart from the ones specified in the assignment criteria, are subject to change. It serves only as a template for future development.

Implementation and testing

The implementation of the code will be done based heavily on the pseudocode and functionality defined above. Additional helper functions that have not been accounted for may also need to be added. Pre-planning for these functions is difficult and not entirely necessary, and so they have not been expanded in detail above. The order of implementation will be based on the flowchart, which in itself follows a logical flow of 'function x' being necessary to implement 'function x+1'.

Step 1

Data types

Data types are confirmed to be functional through successful compilation with no errors. They are implemented in the same order they have been written in the above sections (i.e., Suit, Rank, Card, Deck, PlayerMove, PlayerType, Player, GameState). To allow for easier debugging, all data types are deriving Show for the testing phase, among other typeclasses such as Eq (for checking equality), Num (for arithmetic operations), and Ord (for ordering).

initDeck

Before implementing initGame, a helper function was initiated called initDeck to initialise the deck with the standard 52 cards and allow it to be tested in isolation (this also allowed for a learning experience in testing with QuickCheck). To allow for easier testing (i.e., not having to copy and paste all data types/functions that need to be tested into Testing.hs), I will place the code inside a module 'TexasHoldEm' so the necessary data can be easily imported to 'Testing' file.

C: > Users > shoha > Testing.hs > ...

```haskell
import Test.QuickCheck ( quickCheck ) | import Test.QuickCheck ( quickCheck )
import Test.QuickCheck
import Data.List (nub)
import TexasHoldEm (Suit, Rank, Card, Deck, initDeck)

-- checks is deck has 52 cards
initDeck52 :: Bool
initDeck52 = length initDeck == 52

-- checks only one of every card type
initDeckUnique :: Bool
initDeckUnique = length (nub initDeck) == 52

main :: IO ()
main = do
  quickCheck initDeck52
  quickCheck initDeckUnique
```

```
ghci> :load Testing
[1 of 3] Compiling TexasHoldEm      ( TexasHoldEm.hs, interpreted )
[2 of 3] Compiling Main             ( Testing.hs, interpreted )
Ok, two modules loaded.
ghci> main
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
ghci>
```

The function is confirmed to work as expected.

shuffleDeck

The next function to implement before initGame can be implemented is shuffleDeck. The random aspect of the shuffling has been done with randomRIO as opposed to StdGen due to its simplicity of implementation while being random enough for the scope of this project. Additionally, there is no need for reproducible shuffles, which is one of the main advantages of StdGen over randomRIO. The code was tested during implementation to check that every card is shuffled (although it acknowledged that – owing to the nature of randomRIO – there is the possibility of cards remaining stationary even after shuffling).

```
ghci> :l TexasHoldEm.hs
[1 of 1] Compiling TexasHoldEm      ( TexasHoldEm.hs, interpreted )
Ok, one module loaded.
ghci> let deck = initDeck
ghci> shuffedDeck <- shuffleDeck deck
Chosen card: Card {suit = Spades, rank = Ten}
Chosen card: Card {suit = Hearts, rank = Two}
Chosen card: Card {suit = Clubs, rank = King}
Chosen card: Card {suit = Hearts, rank = Nine}
Chosen card: Card {suit = Diamonds, rank = Jack}
Chosen card: Card {suit = Diamonds, rank = Five}
Chosen card: Card {suit = Diamonds, rank = Two}
Chosen card: Card {suit = Diamonds, rank = Six}
Chosen card: Card {suit = Hearts, rank = Three}
Chosen card: Card {suit = Clubs, rank = Jack}
Chosen card: Card {suit = Diamonds, rank = Eight}
```

This output extended for 52 lines and showcased every card being chosen once for shuffling. This means that this function is working as expected.

dealCards

This function was implemented as it is both the last in Step 1 of the assignment criteria and the last function before initGame can be defined. After creating it, it was tested with shuffleDeck to confirm the functionality of both. This test returned a list of 5 dealt cards and 47 remaining cards successfully, and was tested twice to ensure randomness, therefore indicating that dealCards' functionality is working.

C: > Users > shoha > Testing.hs > Haskell > main

```haskell
import Test.QuickCheck
import Data.List (nub)
import TexasHoldEm (Suit, Rank, Card, Deck, initDeck, shuffl

main :: IO ()
main = do
  -- Create and shuffle the deck
  let fullDeck = initDeck
  shuffledDeck <- shuffleDeck fullDeck

  -- Deal 5 cards
  let (dealtCards, updatedDeck) = dealCards 5 shuffledDeck
  putStrLn "D:"
  print dealtCards
  putStrLn "R:"
  print updatedDeck
```

```
ghci> main
D:
[Card {suit = Hearts, rank = Five},Card {suit = Clubs, rank = Nine},Card {suit = Diamonds, rank
 = Four},Card {suit = Hearts, rank = Two},Card {suit = Spades, rank = Nine}]
R:
[Card {suit = Spades, rank = Jack},Card {suit = Hearts, rank = Three},Card {suit = Diamonds, ra
nk = Nine},Card {suit = Diamonds, rank = Ace},Card {suit = Clubs, rank = Eight},Card {suit = Di
amonds, rank = Three},Card {suit = Spades, rank = Ace},Card {suit = Diamonds, rank = Five},Card
 {suit = Spades, rank = King},Card {suit = Clubs, rank = Four},Card {suit = Diamonds, rank = Tw
o},Card {suit = Spades, rank = Four},Card {suit = Spades, rank = Seven},Card {suit = Hearts, ra
nk = Four},Card {suit = Hearts, rank = Ten},Card {suit = Clubs, rank = Six},Card {suit = Clubs,
 rank = Jack},Card {suit = Spades, rank = Two},Card {suit = Clubs, rank = Seven},Card {suit = H
earts, rank = Seven},Card {suit = Clubs, rank = Ace},Card {suit = Hearts, rank = Six},Card {sui
```

```
ghci> main
D:
[Card {suit = Spades, rank = Three},Card {suit = Spades, rank = Two},Card {suit = Hearts, rank
= Two},Card {suit = Clubs, rank = King},Card {suit = Hearts, rank = Three}]
R:
[Card {suit = Clubs, rank = Six},Card {suit = Spades, rank = Seven},Card {suit = Hearts, rank =
 Ace},Card {suit = Clubs, rank = Two},Card {suit = Diamonds, rank = Three},Card {suit = Clubs,
rank = Nine},Card {suit = Clubs, rank = Jack},Card {suit = Spades, rank = Eight},Card {suit = D
iamonds, rank = Two},Card {suit = Hearts, rank = Ten},Card {suit = Spades, rank = Ace},Card {su
it = Hearts, rank = Four},Card {suit = Hearts, rank = Jack},Card {suit = Diamonds, rank = Seven
```

Step 2

initGame

initGame will be implemented to allow for a game to be started, which will then allow for evaluateHand and determineWinner to be implemented.

To allow for easier debugging, all data types are deriving Show for the testing phase.

As initGame also instantiates players, a rudimentary RandomPlayer has been developed in order for initGame to be tested effectively. This has been done using randomRIO and unsafePerformIO to allow for output within a pure function without unnecessarily propagating IO throughout the codebase. However, as the other player types do not require IO within their logic and all player types draw on the same Strategy data type, it is not sensible to include this within randomStrategy, therefore randomRIO will be removed upon a full implementation of RandomPlayer's randomStrategy.

Additionally, the GameState's deck will be temporarily set to empty to avoid unnecessary information cluttering the test terminal.

```haskell
  -- Creates the first GameState
  let initialGameState = GameState
        { deck = [] -- for testing only (replaced with shuffledDeck)
        , communityCards = []
        , pot = 0
        , players = instantiatedPlayers
        , smallBlind = smallBlind
        , bigBlind = smallBlind * 2
        , currentDealer = 0
        , currentBets = replicate numPlayers 0
        }

  return initialGameState

-- for testing purposes, unsafePerformIO used for simplicity/testing
randomStrategy :: GameState -> Player -> PlayerMove
randomStrategy _ _ =
  -- (0 - 4 range as that is the range for PlayerMove constructors
  let randomIndex = unsafePerformIO $ randomRIO (0 :: Int, 4 :: Int)
  in case randomIndex of
      0 -> Fold
      1 -> Check
      2 -> Call
      3 -> Bet 100
      4 -> Raise 101
```

In order to then map randomStrategy to a PlayerType, an intermediary function makeMove is created, which maps each strategy type to its corresponding PlayerType (although PassivePlayer, AggressivePlayer and SmartPlayer have been commented out for now). initGame will now be tested using a test harness to initialise a game state and generate random moves for each player using a function called randMove, with the results returned with mapM_ as opposed to mapM as just the prints of the moves are needed, and not the result of calling them.

```haskell
testPlayerTypes :: [PlayerType]
testPlayerTypes = [RandomPlayer, RandomPlayer, RandomPlayer]

main :: IO ()
main = do
  gameState <- initGame 3 1000 10 testPlayerTypes

  -- Generates 2 random moves for each player to test RandomPlayer's RNG
  mapM_ (randMove gameState) (players gameState)
  mapM_ (randMove gameState) (players gameState)

randMove :: GameState -> Player -> IO ()
randMove gameState player = do
  let move = randomStrategy gameState player
  putStrLn $ name player ++ " = " ++ show move
```

```
ghci> main
GameState {deck = [], communityCards = [], pot = 0, players = [Player {playerType = RandomPlaye
r, name = "1", holeCards = [], chips = 1000, isDealer = False, isActive = True},Player {playerT
ype = RandomPlayer, name = "2", holeCards = [], chips = 1000, isDealer = False, isActive = True
},Player {playerType = RandomPlayer, name = "3", holeCards = [], chips = 1000, isDealer = False
, isActive = True}], smallBlind = 100, bigBlind = 200, currentDealer = 0, currentBets = [0,0,0]
}
1 = Fold
2 = Call
3 = Raise 100
1 = Bet 100
2 = Bet 100
3 = Call
1 = Fold
2 = Bet 100
3 = Fold
```

Although the amounts in the output are static, this is due to the static nature of the current GameState (empty pot, no current bets, and no community cards) making the moves appear predictable even though they will – in a functional game – be pseudo-random.

randomStrategy

As stated previously, the 'impure' randomness of RandomRIO within randomStrategy has been replaced with mkStdGen, which allows for a fixed seed to be passed for randomness, therefore making it more appropriate for testing and debugging, as well as aligning with the original type signature for PlayerType. However, as mkStdGen produces the same random outputs for a given seed, this seed must be modified each time a player makes a move. The way it will do this is by taking advantage of the fact that there are 52! permutations for a deck of cards to be shuffled, meaning there is a ~0% chance of 2 shuffles to have the same first x cards where x<25. This is useful for the initial round. The seed passed to mkStdGen to generate a random number is complex, consisting of a sum of the current pot, current bets, length of community cards, and the numerical value of the suit and rank of the first 15 cards in the current deck.

As the game has not been properly initialised with values and players, testing this function yielded the same results as when testing initGame, therefore this is considered to be completed, with further testing will be delayed until a proper round of the game can be tested. It is, however, presumed functional due to displaying random moves (just with predictable amounts).

evaluateHand

To create evaluateHand, the hand ranking functions from the above pseudocode for SmartPlayer was taken and implemented, along with 2 helper functions (with one being countOccurencesOfRank which appeared in the aforementioned pseudocode). Although these helper functions are not strictly necessary and simply modularise repeating parts of code, this is done to adhere to best practices, specifically the DRY (Don't Repeat Yourself) principle in software development. The rankings are also arranged in descending order of points, as some hands are subsets of others and so must be checked first (i.e., Royal Flush is a subset of Flush).

This was then tested by passing in cards that fulfil the criteria to see if they are correctly flagged. As Testing.hs contained mostly generic code, only one of the tests has been shown as an example and for brevity, with the test for a high card also shown as its test is unique to the others. This test revealed an error with the isFlush function.

```haskell
testRF :: Bool ···
testSF :: Bool ···
test4OAK :: Bool ···
testFH :: Bool ···
testF :: Bool ···
testS :: Bool ···
test3OAK :: Bool ···
test2p :: Bool ···
test1p :: Bool
test1p =
  evaluateHand
    [Card Hearts Two, Card Diamonds Two, Card Hearts Three,
    Card Hearts Four, Card Hearts Five]
    == OnePair
testHC :: Bool
testHC =
  let hand = [Card Spades Ace, Card Spades King, Card Spades Jack,
              Card Spades Ten, Card Hearts Four]
  in evaluateHand hand == HighCard && highestCard hand == Card Spades Ace

main :: IO ()
main = do
  putStrLn "Royal Flush: "
  quickCheck testRF
  putStrLn "Straight Flush: "
  quickCheck testSF
  putStrLn "4 Of A Kind: "
  quickCheck test4OAK
  putStrLn "Full House: "
  quickCheck testFH
  putStrLn "Flush: "
  quickCheck testF
```

```
ghci> main
Royal Flush:
+++ OK, passed 100 tests.
Straight Flush:
+++ OK, passed 100 tests.
4 Of A Kind:
+++ OK, passed 100 tests.
Full House:
+++ OK, passed 100 tests.
Flush:
*** Failed! Falsified (after 1 test):
Straight:
+++ OK, passed 100 tests.
3 Of A Kind:
+++ OK, passed 100 tests.
Two pair:
+++ OK, passed 100 tests.
One pair:
+++ OK, passed 100 tests.
High Card:
+++ OK, passed 100 tests.
```

This was presumed to be a simple logic issue, but, when fixed, still returned the error result. Upon further inspection, it was found that the hand used for Flush was also a Straight Flush, leading to it being identified as such before isFlush is reached. This was verified then fixed by passing in a hand that is a Flush but not a Straight Flush, which then worked as expected.

```haskell

testRF :: Bool ···
testSF :: Bool ···
test4OAK :: Bool ···
testFH :: Bool ···
testF :: IO ()
testF = do
  let hand =evaluateHand [Card Hearts Two, Card Hearts Three, Card Hearts
            Card Hearts Five, Card Hearts Six]
  print hand
testS :: Bool ···
test3OAK :: Bool ···
test2p :: Bool ···
test1p :: Bool ···
testHC :: Bool ···

main :: IO ()
```

```
ghci> main
Royal Flush:
+++ OK, passed 100 tests.
Straight Flush:
+++ OK, passed 100 tests.
4 Of A Kind:
+++ OK, passed 100 tests.
Full House:
+++ OK, passed 100 tests.
Flush (Actualy is):
StraightFlush
Straight:
+++ OK, passed 100 tests.
3 Of A Kind:
+++ OK, passed 100 tests.
Two pair:
+++ OK, passed 100 tests.
One pair:
+++ OK, passed 100 tests.
High Card:
+++ OK, passed 100 tests.
```

```haskell
testRF :: Bool ...

testSF :: Bool ...

test4OAK :: Bool ...

testFH :: Bool ...

testF :: Bool
testF = evaluateHand
    [ Card Hearts Two,
      Card Diamonds Three,
      Card Hearts Four,
      Card Spades Five,
      Card Clubs Six
    ]
    == Straight

testS :: Bool ...
```

```
ghci> main
Royal Flush:
+++ OK, passed 100 tests.
Straight Flush:
+++ OK, passed 100 tests.
4 Of A Kind:
+++ OK, passed 100 tests.
Full House:
+++ OK, passed 100 tests.
Flush:
+++ OK, passed 100 tests.
Straight:
+++ OK, passed 100 tests.
3 Of A Kind:
+++ OK, passed 100 tests.
Two pair:
+++ OK, passed 100 tests.
One pair:
+++ OK, passed 100 tests.
High Card:
+++ OK, passed 100 tests.
ghci>
```

For the second round of testing, evaluateHand will be tested by passing in cards that do not fulfil the criteria to see if they are correctly flagged as such. This uncovered an error in isFullHouse.

```haskell
testRF :: Bool…
testSF :: Bool…
test4OAK :: Bool…
testFH :: Bool…
testF :: Bool…
testS :: Bool…
test3OAK :: Bool…
test2p :: Bool…
test1p :: Bool
test1p =
  evaluateHand
    [ Card Hearts Two,
      Card Diamonds Seven,
      Card Spades Three,
      Card Hearts Four,
      Card Hearts Five
    ]
    == OnePair
testHC :: Bool
testHC =
  let hand =
        [ Card Spades Ace,
          Card Spades King,
          Card Spades Jack,
          Card Spades Ten,
          Card Hearts Four
        ]
   in evaluateHand hand == HighCard && highestCard hand == Card Spades Two

main :: IO ()
main = do
```

```
ghci> main
Royal Flush:
*** Failed! Falsified (after 1 test):
Straight Flush:
*** Failed! Falsified (after 1 test):
4 Of A Kind:
*** Failed! Falsified (after 1 test):
Full House:
+++ OK, passed 100 tests.
Flush:
*** Failed! Falsified (after 1 test):
Straight:
*** Failed! Falsified (after 1 test):
3 Of A Kind:
*** Failed! Falsified (after 1 test):
2 pair:
*** Failed! Falsified (after 1 test):
1pair:
*** Failed! Falsified (after 1 test):
High Card:
*** Failed! Falsified (after 1 test):
ghci>
```

After reading through the code, this error seems to arise as isFullHouse checks if the hand passes isThreeOfAKind and isOnePair, however it does not check these independently, and so a Three Of A Kind hand will contain a One Pair within it. Therefore, the code was changed to check each of these independently using the countOccurencesOfRank helper function, which then resulted in the correct output.

```
ghci> main
Royal Flush:
*** Failed! Falsified (after 1 test):
Straight Flush:
*** Failed! Falsified (after 1 test):
4 Of A Kind:
*** Failed! Falsified (after 1 test):
Full House:
*** Failed! Falsified (after 1 test):
Flush:
*** Failed! Falsified (after 1 test):
Straight:
*** Failed! Falsified (after 1 test):
3 Of A Kind:
*** Failed! Falsified (after 1 test):
2 pair:
*** Failed! Falsified (after 1 test):
1pair:
*** Failed! Falsified (after 1 test):
High Card:
*** Failed! Falsified (after 1 test):
```

With this, evaluateHand is confirmed to be working as expected.

compareHands

While the hand-comparison functionality of this function will be simple to implement, the tie-breaker scenarios will likely require a number of additional functions owing to the different types and resolutions of ties in Poker.

Initially, compareHands is implemented and tested with non-tie scenarios to ensure functionality.

```haskell
testCompHands :: IO ()
testCompHands = do
  let
    -- Flush v Straight (no tie)
    noTieFlush :: [Card]
    noTieFlush = [Card Hearts Ace, Card Hearts King, Card Hearts Ten, Car
    noTieStraight :: [Card]
    noTieStraight = [Card Spades Six, Card Diamonds Seven, Card Clubs Eig
    resultNoTie = compareHands noTieFlush noTieStraight

    -- Flush v Flush (tie)
    tieFlush1 :: [Card]
    tieFlush1 = [Card Hearts Ace, Card Hearts King, Card Hearts Ten, Card
    tieFlush2 :: [Card]
    tieFlush2 = [Card Hearts Three, Card Hearts Jack, Card Hearts Queen,
    resultTie = compareHands tieFlush1 tieFlush2

  putStrLn $ "Expected GT: " ++ show resultNoTie
  putStrLn $ "Expected EQ: " ++ show resultTie
```

```
ghci> testCompHands
Expected GT: GT
Expected EQ: EQ
```

As mentioned previously, there are different types of ties in Poker, and certain ways of breaking them specific to Texas Hold'em. For most hands, simply the highest-ranked card in the comparison breaks the tie, but certain hands require groups of cards to be compared, as opposed to singular cards. First the simple case (highestCardTie) will be implemented and tested before the other cases.

```haskell
testFlushTie :: String
testFlushTie =
  let hand1 = [Card Hearts Ace, Card Hearts King, Card Hearts Jack, Card H
      hand2 = [Card Diamonds Ace, Card Diamonds Queen, Card Diamonds Ten,
      result = compareHands hand1 hand2
   in case result of
        -- hand 1's King beats hand 2's Queen despite both being Flush
        GT -> "passed: hand 1 wins"
        _ -> "failed"

testFlushHC :: String
testFlushHC =
  let hand1 = [Card Spades Ace, Card Diamonds King, Card Clubs Queen, Card
      hand2 = [Card Hearts Seven, Card Hearts Six, Card Hearts Five, Card
      result = compareHands hand1 hand2
   in case result of
        -- Despite hand 1 having higher card, Flush > High Card
        LT -> "passed: hand 2  wins"
        _ -> "failed"

testHCTie :: String
testHCTie =
  let hand1 = [Card Spades King, Card Diamonds Jack, Card Clubs Nine, Card
      hand2 = [Card Clubs King, Card Hearts Queen, Card Spades Ten, Card D
      result = compareHands hand1 hand2
   in case result of
        -- High Card:  both have King, hand 2's Queen beats hand 1's Jack
        LT -> "passed: hand 2 wins"
        _ -> "failed"
```

```
Ok, two modules loaded.
ghci> testFlushTie
"passed: hand 1 wins"
ghci> testFlushHC
"passed: hand 2  wins"
ghci> testHCTie
"passed: hand 2 wins"
```

The first edge-case tie-breaker scenario is a One Pair tie, in which both players have the same hand of One Pair, and so the rank of these must be compared to clarify the winner; this can be done by incorporating the previously-defined function countOccurencesOfRank, thereby simplifying the code. If the ranks are the same, their three remaining cards (kickers) must be compared for the highest value, the functionality for which is already covered by highestCardTie.

```haskell
testDiffPair :: String
testDiffPair =
  let hand1 = [Card Hearts King, Card Diamonds King, Card Spades Ten,
               Card Clubs Seven, Card Diamonds Three]
      hand2 = [Card Hearts Queen, Card Diamonds Queen, Card Spades Ten,
               Card Clubs Seven, Card Diamonds Three]
      result = compareHands hand1 hand2
  in case result of
       -- hand 1's pair of Kings beats hand 2's pair of Queens
       -- all other cards are the same to ensure functionality
       GT -> "passed: hand 1 wins"
       _  -> "failed"

testSamePairKicker :: String
testSamePairKicker =
  let hand1 = [Card Hearts Queen, Card Diamonds Queen, Card Spades Ace,
               Card Clubs Nine, Card Diamonds Two]
      hand2 = [Card Spades Queen, Card Clubs Queen, Card Spades Ten,
               Card Clubs Nine, Card Diamonds Two]
      result = compareHands hand1 hand2
  in case result of
       -- hand 1 kicker (Ace) beats hand 2's kicker (10)
       GT -> "passed: hand 1 wins"
       _  -> "failed"
```

```
ghci> testDiffPair
"passed: hand 1 wins"
ghci> testSamePairKicker
"passed: hand 1 wins"
```

The second edge case occurs where both players have a Two Pair; in this scenario, the first pair is compared, followed by the second pair (if needed), and finally the kicker (if needed). Before implementing, it is apparent that onePairTie and twoPairTie will share a lot of the same functionality, and while the original idea was to incorporate these into a single function, planning ahead for the code shows that this functionality will also be shared in a large part by tie-breakers for Three Of A Kind and Four Of A Kind. Therefore, the functionality of onePairTie will be moved into a helper function for modularity, after which onePairTie and twoPairTie will be refactored and created respectively.

```haskell
-- PREVIOUS IMPLEMENTATION
{-
onePairTie :: [Card] -> [Card] -> Ordering
onePairTie hand1 hand2 =
  -- Gets number of each rank in each players' hands
  let rankOccurences1 = zip [Two .. Ace] (countOccurrencesOfRank hand1)
      rankOccurences2 = zip [Two .. Ace] (countOccurrencesOfRank hand2)

      -- Finds which rank has a count of 2 in each hand
      rankPair1 = fst $ head $ filter (\(_, count) -> count == 2) rankOccurences1
      rankPair2 = fst $ head $ filter (\(_, count) -> count == 2) rankOccurences2

      -- Sorts kicker in descending order for for comparison
      kickers1 = sortBy (comparing Down) [rank c |
                                c <- hand1, rank c /= rankPair1]
      kickers2 = sortBy (comparing Down) [rank c |
                                c <- hand2, rank c /= rankPair2]
   in -- Compares rank pairs, or kickers if rank pairs are equal
      if rankPair1 /= rankPair2
        then compare rankPair1 rankPair2
        else compare kickers1 kickers2
-}
```

```haskell
-- Determines stronger hand given 2, accounting for ties
compareHands :: [Card] -> [Card] -> Ordering
compareHands hand1 hand2 =
  let rank1 = evaluateHand hand1
      rank2 = evaluateHand hand2
   in if rank1 /= rank2
        then compare rank1 rank2 -- Compare in case of no tie
        else case rank1 of -- Compare in case of tie
          OnePair -> compareRankClusters hand1 hand2 1 -- Tied with One Pair
          TwoPair -> compareRankClusters hand1 hand2 2 -- Tied with Two Pair
          _ -> highestCardTie hand1 hand2 -- Tied with any other hand

-- Tie breaker for simple tie where only high cards are compared
highestCardTie :: [Card] -> [Card] -> Ordering
highestCardTie hand1 hand2 =
  let highestRankIn1 = sortBy (comparing Down) $ map rank hand1
      highestRankIn2 = sortBy (comparing Down) $ map rank hand2
   in compare highestRankIn1 highestRankIn2

-- CURRENT IMPLEMENTATION
compareRankClusters :: [Card] -> [Card] -> Int -> Ordering
compareRankClusters hand1 hand2 clusterSize =
     -- Gets number of each rank in each players' hands
  let rankOccurences1 = zip [Two .. Ace] (countOccurrencesOfRank hand1)
      rankOccurences2 = zip [Two .. Ace] (countOccurrencesOfRank hand2)

      -- Finds which rank has a count of clusterSize in each hand
      rankGroup1 = sortBy (comparing Down) [r |
                  (r, count) <- rankOccurences1, count == clusterSize]
      rankGroup2 = sortBy (comparing Down) [r |
                  (r, count) <- rankOccurences2, count == clusterSize]

      -- Sorts kicker in descending order for for comparison
      kickers1 = sortBy (comparing Down) [rank c |
                        c <- hand1, rank c `notElem` rankGroup1]
      kickers2 = sortBy (comparing Down) [rank c |
                        c <- hand2, rank c `notElem` rankGroup2]
   in -- Compares rank pairs, or kickers if rank pairs are equal
      if rankGroup1 /= rankGroup2
        then compare rankGroup1 rankGroup2
        else compare kickers1 kickers2
```

This can now be tested by testing Two Pair tie-breaker scenarios. One thing that should be mentioned is that, until now, the situation of both players having the same hand (either by getting the same hole cards or the River community cards forming their strongest hand) has not been addressed. This is due to the fact that it will be defined in the next function, determineWinner, during which the relevant function(s) can be modified to include an output for if the Ordering returns EQ (i.e., both hands are equal). For this reason, tests for this situation have not been included previously and will not be until the implementation of determineWinner.

```haskell
-- Test 1: Hand 1 wins due to higher first pair (Kings > Queens)
testTwoPairTie1 :: String
testTwoPairTie1 =
  let hand1 = [Card Hearts King, Card Diamonds King, Card Clubs Queen, Card Spades Queen, C
      hand2 = [Card Clubs Queen, Card Clubs Queen, Card Diamonds Four, Card Spades Four, Ca
      result = compareHands hand1 hand2
   in case result of
        -- hand 1's first pair (Kings) beats hand 2's first pair (Queens)
        GT -> "passed: hand 1 wins"
        _ -> "failed"

-- Test 2: Hand 2 wins due to higher second pair (Jacks > Tens)
testTwoPairTie2 :: String
testTwoPairTie2 =
  let hand1 = [Card Hearts King, Card Diamonds King, Card Clubs Eight, Card Spades Eight, C
      hand2 = [Card Hearts King, Card Diamonds King, Card Clubs Ace, Card Spades Ace, Card
      result = compareHands hand1 hand2
   in case result of
        -- hand 2 second pair (Ace) beats hand 2's second pair (Eight)
        LT -> "passed: hand 2 wins"
        _ -> "failed"

testSamePairKicker :: String
testSamePairKicker =
  let hand1 = [Card Hearts Queen, Card Diamonds Queen, Card Spades Ace,
               Card Clubs Ace, Card Diamonds Three]
      hand2 = [Card Spades Queen, Card Clubs Queen, Card Clubs Ace,
               Card Diamonds Ace, Card Diamonds Two]
      result = compareHands hand1 hand2
   in case result of
        -- hand 1's kicker (Three) beats hand 2's kicker (Two)
        GT -> "passed: hand 1 wins"
        _ -> "failed"
```

```
ghci> testTwoPairTie1
"passed: hand 1 wins"
ghci> testTwoPairTie2
"passed: hand 2 wins"
ghci> testSamePairKicker
"passed: hand 1 wins"
```

Following on from this, the tie-breakers for Three Of A Kind and Four Of A Kind can be implemented in tandem due to the similar functionality. Full House will also be implemented, however this will require a separate function owing to the nature of Full House hands, in which there is a cluster of 3 cards followed by a cluster of 2 cards. To resolve this, the helper function fullHouseTie will be able to call compareRankClusters twice with different values for clusterSize dependant on if the trip is equal in a given hand. These functions are then tested.

```haskell
threeOfAKindtest :: String
threeOfAKindtest =
  let hand1 = [Card Hearts King, Card Diamonds King, Card Clubs King, Card Spades Ten, Card Hearts Nine]
      hand2 = [Card Hearts Queen, Card Diamonds Queen, Card Clubs Queen, Card Spades Ten, Card Hearts Nine]
      result = compareHands hand1 hand2
   in -- hand 1 wins: trip of Kings > tripp of Queens
      case result of
        GT -> "passed: hand 1 wins"
        _ -> "failed"

fourOfAKindtest :: String
fourOfAKindtest =
  let hand1 = [Card Hearts Five, Card Diamonds Five, Card Clubs Five, Card Spades Five, Card Hearts Nine]
      hand2 = [Card Hearts Nine, Card Diamonds Nine, Card Clubs Nine, Card Spades Nine, Card Hearts Ten]
      result = compareHands hand1 hand2
   in -- hand 2 wins: quad of Nine > quad Five
      case result of
        LT -> "passed: hand 2 wins"
        _ -> "failed"

fullHouseTripTest :: String
fullHouseTripTest =
  let hand1 = [Card Hearts King, Card Diamonds King, Card Clubs King, Card Spades Queen, Card Hearts Queen]
      hand2 = [Card Hearts Queen, Card Diamonds Queen, Card Clubs Queen, Card Spades Jack, Card Hearts Jack]
      result = compareHands hand1 hand2
   in -- hand 1 wins: trip of Kings > trip of Queens
      case result of
        GT -> "passed: hand 1 wins"
        _ -> "failed"

fullHousePairTest :: String
fullHousePairTest =
  let hand1 = [Card Hearts Queen, Card Diamonds Queen, Card Clubs Queen, Card Spades King, Card Hearts King]
      hand2 = [Card Hearts Queen, Card Diamonds Queen, Card Clubs Queen, Card Spades Ace, Card Hearts Ace]
      result = compareHands hand1 hand2
   in -- hand 2 wins: pair of Aces > pair of Kings
      case result of
        LT -> "passed: hand 2 wins"
        _ -> "failed"
```

```
ghci> main
3 OAK: passed: hand 1 wins
4 OAK: passed: hand 2 wins
Full House high triple: passed: hand 1 wins
Full House high pair: passed: hand 2 wins
```

determineWinner

Before determining the winner of a round, a helper function must be created which assesses every single combination of a given player's hole cards and the community cards to find their best hand to compare to the others' best hands. As there are only 21 combinations, this will not be a particularly costly operation, therefore all combinations can be assessed with a brute force method such as ‘subsequences’ as it both simplifies the code and fits the scope of complexity of the project. After implementing, to ensure correctness, both functions will be tested once separately and then once together.

One point to note is that, while the type signature of determineWinner was to be ‘determineWinner :: [Player] -> [Player]’ as specified in the documentation, it has been changed to ‘determineWinner :: GameState -> [Player]’. This is because, whereas in the original documentation the plan was to implement the functionality of assessPlayerHand within evaluateHand, it is now going to be implemented in determineWinner. This is done for a variety of reasons:

- To keep evaluateHand modular and adhering to a single concept.
- To allow determineWinner to be tested on a fresh GameState without explicitly evaluating hands first.
- To make it easier for coding and maintenance as the full GameState has all internal details (such as holeCards and communityCards) already fully encapsulated.

For the first round of tests, the entire output of assessPlayerHand will be displayed to ensure that all combinations are being displayed, correctly categorised as the appropriate rank, and have the highest one chosen to be the best rank. A common GameState will be created for ease of documentation and brevity.

```haskell
-- Shared GameState for testing purposes
sharedTestGS :: GameState
sharedTestGS =
  GameState
    { deck = [], -- deck isn't set as it will be defined explicitly
      communityCards =
        [ Card Spades Ace,
          Card Clubs Four,
          Card Spades Two,
          Card Diamonds Five,
          Card Hearts Ten
        ],
      pot = 0,
      players =
        [ Player RandomPlayer "Fadhila" [Card Clubs Ace, Card Diamonds Ace] 100 True True,
          Player RandomPlayer "Faidha" [Card Spades Three, Card Clubs Six] 100 False True,
          Player RandomPlayer "Shohail" [Card Diamonds Seven, Card Spades Jack] 100 False True
        ],
      smallBlind = 10,
      bigBlind = 20,
      currentDealer = 0,
      currentBets = [0, 0, 0]
    }

-- Assesses Fadhila's Three Of A Kind
test3OAKAssessment :: Bool
test3OAKAssessment =
  let hole = holeCards (head (players sharedTestGS))
      community = communityCards sharedTestGS
      handRank = assessPlayerHand hole community
   in handRank == ThreeOfAKind
```

```haskell
-- Assesses Fiadha's Straight and prints out the working of assessPlayerHand
testStraightAssessment :: Bool
testStraightAssessment =

  -- Get all combinations of 5-card hands
  let hole = holeCards (players sharedTestGS !! 1)
      community = communityCards sharedTestGS
      all5CardCombos = filter ((== 5) . length) (subsequences (hole ++ community))
      combosWithRank = map (\combo -> (combo, evaluateHand combo)) all5CardCombos

      -- Display the ones that are recognised poker hands (ignoring High Cards)
      nonHighCards = filter (\(_, rank) -> rank /= HighCard) combosWithRank
      formattedOutput = unlines $ zipWith formatCombos [1..] combosWithRank
  in trace formattedOutput (assessPlayerHand hole community == Straight)
  where
    -- Format each combination with an index and highlight meaningful hands
    formatCombos :: Int -> ([Card], HandRank) -> String
    formatCombos idx (combo, rank) =
      let format = if rank /= HighCard then "\n*" else "\n"
      in format ++ show idx ++ ". " ++ show combo ++ " == " ++ show rank

-- Assesses Shohail's High Card
tesHCAssesment :: Bool
tesHCAssesment =
  let hole = holeCards (players sharedTestGS !! 2)
      community = communityCards sharedTestGS
      handRank = assessPlayerHand hole community
   in handRank == HighCard

-- Determines winner out of the trio
testWinner :: Bool
testWinner =
  let winners = determineWinner sharedTestGS
   in map name winners == ["Fadhila"] -- Faidha wins with a Six-High Straight

main :: IO ()
main = do
  putStrLn $ "Fadhila = " ++ show test3OAKAssessment
  putStrLn $ "Faidha = " ++ show testStraightAssessment
  putStrLn $ "Shohail = " ++ show tesHCAssesment
  putStrLn $ "Winner is Fadhila = " ++ show testWinner
```

```
ghci> main
Fadhila = True
Faidha =
1. [Card {suit = Spades, rank = Three},Card {suit = Clubs, rank = Six},Card {suit = Spades, rank = Ace},Card {sui
t = Clubs, rank = Four},Card {suit = Spades, rank = Two}] == HighCard

2. [Card {suit = Spades, rank = Three},Card {suit = Clubs, rank = Six},Card {suit = Spades, rank = Ace},Card {sui
t = Clubs, rank = Four},Card {suit = Diamonds, rank = Five}] == HighCard

3. [Card {suit = Spades, rank = Three},Card {suit = Clubs, rank = Six},Card {suit = Spades, rank = Ace},Card {sui
t = Spades, rank = Two},Card {suit = Diamonds, rank = Five}] == HighCard

*4. [Card {suit = Spades, rank = Three},Card {suit = Clubs, rank = Six},Card {suit = Clubs, rank = Four},Card {su
it = Spades, rank = Two},Card {suit = Diamonds, rank = Five}] == Straight

5. [Card {suit = Spades, rank = Three},Card {suit = Spades, rank = Ace},Card {suit = Clubs, rank = Four},Card {su
it = Spades, rank = Two},Card {suit = Diamonds, rank = Five}] == HighCard

6. [Card {suit = Clubs, rank = Six},Card {suit = Spades, rank = Ace},Card {suit = Clubs, rank = Four},Card {suit
= Spades, rank = Two},Card {suit = Diamonds, rank = Five}] == HighCard

7. [Card {suit = Spades, rank = Three},Card {suit = Clubs, rank = Six},Card {suit = Spades, rank = Ace},Card {sui
t = Clubs, rank = Four},Card {suit = Hearts, rank = Ten}] == HighCard

8. [Card {suit = Spades, rank = Three},Card {suit = Clubs, rank = Six},Card {suit = Spades, rank = Ace},Card {sui
t = Spades, rank = Two},Card {suit = Hearts, rank = Ten}] == HighCard

9. [Card {suit = Spades, rank = Three},Card {suit = Clubs, rank = Six},Card {suit = Clubs, rank = Four},Card {sui
t = Spades, rank = Two},Card {suit = Hearts, rank = Ten}] == HighCard

10. [Card {suit = Spades, rank = Three},Card {suit = Spades, rank = Ace},Card {suit = Clubs, rank = Four},Card {s
uit = Spades, rank = Two},Card {suit = Hearts, rank = Ten}] == HighCard

11. [Card {suit = Clubs, rank = Six},Card {suit = Spades, rank = Ace},Card {suit = Clubs, rank = Four},Card {suit
 = Spades, rank = Two},Card {suit = Hearts, rank = Ten}] == HighCard

12. [Card {suit = Spades, rank = Three},Card {suit = Clubs, rank = Six},Card {suit = Spades, rank = Ace},Card {su
it = Diamonds, rank = Five},Card {suit = Hearts, rank = Ten}] == HighCard

13. [Card {suit = Spades, rank = Three},Card {suit = Clubs, rank = Six},Card {suit = Clubs, rank = Four},Card {su
it = Diamonds, rank = Five},Card {suit = Hearts, rank = Ten}] == HighCard

14. [Card {suit = Spades, rank = Three},Card {suit = Spades, rank = Ace},Card {suit = Clubs, rank = Four},Card {s
uit = Diamonds, rank = Five},Card {suit = Hearts, rank = Ten}] == HighCard

15. [Card {suit = Clubs, rank = Six},Card {suit = Spades, rank = Ace},Card {suit = Clubs, rank = Four},Card {suit
 = Diamonds, rank = Five},Card {suit = Hearts, rank = Ten}] == HighCard
```

```
16. [Card {suit = Spades, rank = Three},Card {suit = Clubs, rank = Six},Card {suit = Spades, rank = Two},Card {su
it = Diamonds, rank = Five},Card {suit = Hearts, rank = Ten}] == HighCard

17. [Card {suit = Spades, rank = Three},Card {suit = Spades, rank = Ace},Card {suit = Spades, rank = Two},Card {s
uit = Diamonds, rank = Five},Card {suit = Hearts, rank = Ten}] == HighCard

18. [Card {suit = Clubs, rank = Six},Card {suit = Spades, rank = Ace},Card {suit = Spades, rank = Two},Card {suit
 = Diamonds, rank = Five},Card {suit = Hearts, rank = Ten}] == HighCard

19. [Card {suit = Spades, rank = Three},Card {suit = Clubs, rank = Four},Card {suit = Spades, rank = Two},Card {s
uit = Diamonds, rank = Five},Card {suit = Hearts, rank = Ten}] == HighCard

20. [Card {suit = Clubs, rank = Six},Card {suit = Clubs, rank = Four},Card {suit = Spades, rank = Two},Card {suit
 = Diamonds, rank = Five},Card {suit = Hearts, rank = Ten}] == HighCard
18. [Card {suit = Clubs, rank = Six},Card {suit = Spades, rank = Ace},Card {suit = Spades, rank = Two},Card {suit
 = Diamonds, rank = Five},Card {suit = Hearts, rank = Ten}] == HighCard

19. [Card {suit = Spades, rank = Three},Card {suit = Clubs, rank = Four},Card {suit = Spades, rank = Two},Card {s
uit = Diamonds, rank = Five},Card {suit = Hearts, rank = Ten}] == HighCard

20. [Card {suit = Clubs, rank = Six},Card {suit = Clubs, rank = Four},Card {suit = Spades, rank = Two},Card {suit
 = Diamonds, rank = Five},Card {suit = Hearts, rank = Ten}] == HighCard

21. [Card {suit = Spades, rank = Ace},Card {suit = Clubs, rank = Four},Card {suit = Spades, rank = Two},Card {sui
t = Diamonds, rank = Five},Card {suit = Hearts, rank = Ten}] == HighCard

True
Shohail = True
Winner is Fadhila = True
```

As the test is confirmed to be working as expected, assessPlayerHand can now be tested within determineWinner for its ability to navigate tie scenarios.

While testing this feature, a random pre-initialised GameState brought to light an issue in evaluateHand previously unconsidered, which is that an Ace can appear in both a high and low Straight, meaning it must wrap around properly within the isStraight function. Due to this being unconsidered, when this situation arose during testing, there was an error.

```haskell
-- Shared GameState for tie scenrio
sharedTestGS :: GameState
sharedTestGS =
  GameState
    { deck = [], -- deck isn't set as it will be defined explicitly
      communityCards =
        [ Card Spades Queen,
          Card Clubs Four,
          Card Spades Jack,
          Card Diamonds King,
          Card Hearts Ten
        ],
      pot = 0,
      players =
        [ Player RandomPlayer "Shohail" [Card Clubs Ace, Card Diamonds Ace] 100 True True,
          Player RandomPlayer "Nefeli" [Card Spades Ace, Card Hearts Ace] 100 False True,
          Player RandomPlayer "Mariam" [Card Diamonds Two, Card Spades Three] 100 False True
        ],
      smallBlind = 10,
      bigBlind = 20,
      currentDealer = 0,
      currentBets = [0, 0, 0]
    }
```

```
ghci> main
Shohail = *** Exception: succ{Rank}: tried to take `succ' of last tag in enumeration
CallStack (from HasCallStack):
  error, called at .\TexasHoldEm.hs:10:122 in main:TexasHoldEm
```

The problem lied in the isStraight function attempting to use ‘succ x’ on Ace where there was no successor to it in the defined Rank data type. To avoid this, fromEnum is used to convert each rank into a data type, thereby allowing for consecutive sequences to be checked without failing at Ace. Also, to ensure an Ace-low Straight is recognised, a separate clause has been written to account for this.

```haskell
-- PREVIOUS IMPLEMENTATION
isStraight :: [Card] -> Bool
isStraight cards =
  let ranks = map rank cards
      sortedRanks = sort ranks
      checkForSeqence [] = True
      checkForSeqence [_] = True
      checkForSeqence (x : y : rest) = succ x == y && checkForSeqence (y : rest)
   in checkForSeqence sortedRanks
```

```haskell
-- NEW IMPLEMENTATION
isStraight :: [Card] -> Bool
isStraight cards =
  let ranks = map rank cards
      sortedRanks = sort ranks

      -- Checks for consecutive sequence of ranks (default case)
      checkForSequence :: [Rank] -> Bool
      checkForSequence [] = True
      checkForSequence [_] = True
      checkForSequence (x : y : rest) =
        (fromEnum y == fromEnum x + 1) && checkForSequence (y : rest)

      -- Accounts for Ace-low Straight (edge case)
      aceLow = [Ace, Two, Three, Four, Five]
   in (sortedRanks == aceLow) || checkForSequence sortedRanks

```

With this, determineWinner can be tested to see if it can handle tie scenarios.

```haskell
-- Shared GameState for tie scenrio
sharedTestGS :: GameState
sharedTestGS =
  GameState
    { deck = [], -- deck isn't set as it will be defined explicitly
      communityCards =
        [ Card Spades Queen,
          Card Clubs Four,
          Card Spades Jack,
          Card Diamonds King,
          Card Hearts Ten
        ],
      pot = 0,
      players =
        [ Player RandomPlayer "Shohail" [Card Clubs Ace, Card Diamonds Ace] 100 True True,
          Player RandomPlayer "Nefeli" [Card Spades Ace, Card Hearts Ace] 100 False True,
          Player RandomPlayer "Mariam" [Card Diamonds Two, Card Spades Three] 100 False True
        ],
      smallBlind = 10,
      bigBlind = 20,
      currentDealer = 0,
      currentBets = [0, 0, 0]
    }

-- Assesses Shohail's Straight
testShohailStraight :: Bool
testShohailStraight =
  let handRank = assessPlayerHand (holeCards (head (players sharedTestGS))) (communityCards sharedTestGS)
   in handRank == Straight

-- Assesses Nefeli's Straight
testNefeliStraight :: Bool
testNefeliStraight =
  let handRank = assessPlayerHand (holeCards (players sharedTestGS !! 1)) (communityCards sharedTestGS)
   in handRank == Straight

```

```haskell
-- Assesses Maria's High Card
testMariamHC :: Bool
testMariamHC =
  let handRank = assessPlayerHand (holeCards (players sharedTestGS !! 2)) (communityCards sharedTestGS)
   in handRank == HighCard

-- Determines the winners out of the trio
testWinners :: Bool
testWinners =
  let winners = determineWinner sharedTestGS
   in map name winners == ["Shohail", "Nefeli"]

-- Main function for output
main :: IO ()
main = do
  putStrLn $ "Shohail = " ++ show testShohailStraight
  putStrLn $ "Nefeli = " ++ show testNefeliStraight
  putStrLn $ "Mariam = " ++ show testMariamHC
  putStrLn $ "Winners = " ++ show testWinners
```

```
ghci> main
Shohail = True
Nefeli = True
Mariam = True
Winners = True
```

As the function is working as expected, we can now move on to Step 3.

Step 3

bettingRound

This function will ensure that betting occurs for each street per round. Due to its repetitive nature, it will be implemented as a recursive function, with the base case being that all players have taken an action in the current betting round. As the requirements for each move have already been written out in randomStrategy (i.e., can only bet/check if nobody has betted yet, can only call after bet, etc.)

As a Player is needed to test this and only RandomPlayer is available, RandomPlayer's randomisation will be temporarily deactivated and replaced with determined outputs to allow for more efficient debugging of bettingRound; a print statement has also been issued after each move for this purpose. Fold, Check and Bet were tested (with one example shown below) and found to be successful and only able to be taken according to the rules of Poker.

```haskell
commonGS :: GameState
commonGS =
  GameState
    { deck = [],
      communityCards = [],
      pot = 0,
      players =
        [ Player RandomPlayer "p1" [] 100 True True,
          Player RandomPlayer "p2" [] 100 False True,
          Player RandomPlayer "p3" [] 100 False True
        ],
      smallBlind = 5,
      bigBlind = 10,
      currentDealer = 0,
      currentBets = [0, 0, 0]
    }

-- Only p1 should be able to make a bet, with the others Folding.
testBet :: IO ()
testBet = do
  print commonGS -- premove
  putStrLn "\nBet\n"
  updatedGS <- bettingRound commonGS
  print updatedGS -- after making Bet

main :: IO ()
main = do
  testBet
```

```
ghci> main
GameState {deck = [], communityCards = [], pot = 0, players = [Player {playerType = RandomPlayer, name = "
p1", holeCards = [], chips = 100, isDealer = True, isActive = True},Player {playerType = RandomPlayer, nam
e = "p2", holeCards = [], chips = 100, isDealer = False, isActive = True},Player {playerType = RandomPlaye
r, name = "p3", holeCards = [], chips = 100, isDealer = False, isActive = True}], smallBlind = 5, bigBlind
 = 10, currentDealer = 0, currentBets = [0,0,0]}

Bet

Testing: Player p1 makes a move: Bet 5
Testing: Player p2 makes a move: Fold
Testing: Player p3 makes a move: Fold
GameState {deck = [], communityCards = [], pot = 5, players = [Player {playerType = RandomPlayer, name = "
p1", holeCards = [], chips = 95, isDealer = True, isActive = True},Player {playerType = RandomPlayer, name
 = "p2", holeCards = [], chips = 100, isDealer = False, isActive = False},Player {playerType = RandomPlaye
r, name = "p3", holeCards = [], chips = 100, isDealer = False, isActive = False}], smallBlind = 5, bigBlin
d = 10, currentDealer = 0, currentBets = [0,0,0]}
```

However, when testing Check and Raise (with one example shown below), many errors presented. The first being that the pot remained at 0 even though all players called the current bet.

```haskell
commonGS :: GameState
commonGS =
  GameState
    { deck = [],
      communityCards = [],
      pot = 0,
      players =
        [ Player RandomPlayer "p1" [] 100 True True,
          Player RandomPlayer "p2" [] 50 False True,
          Player RandomPlayer "p3" [] 100 False True
        ],
      smallBlind = 5,
      bigBlind = 10,
      currentDealer = 0,
      currentBets = [0, 20, 20]
    }

testCall :: IO ()
testCall = do
  print commonGS
  updatedGameState <- bettingRound commonGS
  print updatedGameState
```

```
GameState {deck = [], communityCards = [], pot = 0, players = [Player {playerType = RandomPlayer, name = "
p1", holeCards = [], chips = 100, isDealer = True, isActive = True},Player {playerType = RandomPlayer, nam
e = "p2", holeCards = [], chips = 50, isDealer = False, isActive = True},Player {playerType = RandomPlayer
, name = "p3", holeCards = [], chips = 100, isDealer = False, isActive = True}], smallBlind = 5, bigBlind
= 10, currentDealer = 0, currentBets = [0,20,20]}
Testing: Player p1 makes a move: Call
Testing: Player p2 makes a move: Call
Testing: Player p3 makes a move: Call
GameState {deck = [], communityCards = [], pot = 0, players = [Player {playerType = RandomPlayer, name = "
p1", holeCards = [], chips = 80, isDealer = True, isActive = True},Player {playerType = RandomPlayer, name
 = "p2", holeCards = [], chips = 30, isDealer = False, isActive = True},Player {playerType = RandomPlayer,
 name = "p3", holeCards = [], chips = 80, isDealer = False, isActive = True}], smallBlind = 5, bigBlind =
10, currentDealer = 0, currentBets = [0,0,0]}
```

After analysing, it is shown that this is because the total amount called is not added to the pot, however the player’s chips are being correctly reduced. Therefore, the error does not lie in the logic for chip deduction but rather in using the correct amount to update the pot. Specifically, the amount being added to the pot is using only currentMinBet instead of also accounting for the difference between currentMinBet and the player’s existing bet. After rectifying this mistake, the functionality of bettingRound was tested again.

```haskell
-- OLD IMPLEMENTATION
-- Matches current minimum betting (or max bet made so far)
Call -> do
  let updatedChips = chips player - minBet
      updatedBets = updatePlayerBet (currentBets gs) playerIndex minBet
      updatedPot = pot gs + minBet
      updatedPlayers = updatePlayerChips (players gs) playerIndex updatedChips
  return gs {players = updatedPlayers, currentBets = updatedBets, pot = updatedPot}
--
Raise amount -> do
  let raiseAmount = minBet + amount
      updatedChips = chips player - raiseAmount
      updatedBets = updatePlayerBet (currentBets gs) playerIndex (playerBet + raiseAmount)
      updatedPot = pot gs + raiseAmount
      updatedPlayers = updatePlayerChips (players gs) playerIndex updatedChips
  return gs {players = updatedPlayers, currentBets = updatedBets, pot = updatedPot}

```

```haskell
-- CURRENT IMPL.
Call -> do
  let updatedChips = chips player - (minBet - playerBet)
      updatedBets = updatePlayerBet (currentBets gs) playerIndex minBet
      updatedPot = pot gs + (minBet - playerBet)
      updatedPlayers = updatePlayerChips (players gs) playerIndex updatedChips
  return gs {players = updatedPlayers, currentBets = updatedBets, pot = updatedPot}
-- Raises above current min. bet and sets new min.
Raise amount -> do
  let raiseAmount = (minBet - playerBet) + amount
      updatedChips = chips player - raiseAmount
      updatedBets = updatePlayerBet (currentBets gs) playerIndex (playerBet + raiseAmount)
      updatedPot = pot gs + raiseAmount
      updatedPlayers = updatePlayerChips (players gs) playerIndex updatedChips
  return gs {players = updatedPlayers, currentBets = updatedBets, pot = updatedPot}

```

```
ghci> testCall
GameState {deck = [], communityCards = [], pot = 0, players = [Player {playerType = RandomPlayer, name = "
p1", holeCards = [], chips = 100, isDealer = True, isActive = True},Player {playerType = RandomPlayer, nam
e = "p2", holeCards = [], chips = 50, isDealer = False, isActive = True},Player {playerType = RandomPlayer
, name = "p3", holeCards = [], chips = 100, isDealer = False, isActive = True}], smallBlind = 5, bigBlind
= 10, currentDealer = 0, currentBets = [0,20,20]}
Testing: Player p1 makes a move: Call
Testing: Player p2 makes a move: Raise 5
Testing: Player p3 makes a move: Call
GameState {deck = [], communityCards = [], pot = 50, players = [Player {playerType = RandomPlayer, name =
"p1", holeCards = [], chips = 80, isDealer = True, isActive = True},Player {playerType = RandomPlayer, nam
e = "p2", holeCards = [], chips = 25, isDealer = False, isActive = True},Player {playerType = RandomPlayer
, name = "p3", holeCards = [], chips = 95, isDealer = False, isActive = True}], smallBlind = 5, bigBlind =
 10, currentDealer = 0, currentBets = [0,0,0]}
```

As the return now shows the pot being incremented correctly, accounting for a randomised raise, and resetting the current bets at the end of the round, the Call move is confirmed to be working as expected.

However, another error can now be seen which is that all the players have not met the conditions for the betting round to end; all players must have the same final bet through a series of calls and raises (or lack thereof through checking) or must all fold. This points to a logical flaw in runBetting where players are not visited again once a subsequent player raises.

To fix this, the base case will be changed to a flag that will track if all players have made valid moves so it can end the round, with flags being set to False whenever a Raise is made so that the players can be recursively revisited again for their moves. After this was implemented, when testing the round, it was apparent that the randomStrategy originally thought to be random was returning a disproportionate number of Folds compared to any other moves.

```
ghci> testRaise
GameState {deck = [], communityCards = [], pot = 20, players = [Player {playerType = RandomPlayer, name =
"p1", holeCards = [], chips = 100, isDealer = True, isActive = True},Player {playerType = RandomPlayer, na
me = "p2", holeCards = [], chips = 80, isDealer = False, isActive = True},Player {playerType = RandomPlaye
r, name = "p3", holeCards = [], chips = 30, isDealer = False, isActive = True}], smallBlind = 5, bigBlind
= 10, currentDealer = 0, currentBets = [10,20,0]}

Raise

Test: p1: Call
Test: p2: Fold
Test: p3: Fold
GameState {deck = [], communityCards = [], pot = 30, players = [Player {playerType = RandomPlayer, name =
"p1", holeCards = [], chips = 90, isDealer = True, isActive = True},Player {playerType = RandomPlayer, nam
e = "p2", holeCards = [], chips = 80, isDealer = False, isActive = False},Player {playerType = RandomPlaye
r, name = "p3", holeCards = [], chips = 30, isDealer = False, isActive = False}], smallBlind = 5, bigBlind
 = 10, currentDealer = 0, currentBets = [0,0,0]}
```

```
ghci> testRaise
GameState {deck = [], communityCards = [], pot = 20, players = [Player {playerType = RandomPlayer, name =
"p1", holeCards = [], chips = 100, isDealer = True, isActive = True},Player {playerType = RandomPlayer, na
me = "p2", holeCards = [], chips = 80, isDealer = False, isActive = True},Player {playerType = RandomPlaye
r, name = "p3", holeCards = [], chips = 30, isDealer = False, isActive = True}], smallBlind = 5, bigBlind
= 10, currentDealer = 0, currentBets = [0,20,20]}

Raise

Test: p1: Fold
Test: p2: Fold
Test: p3: Fold
GameState {deck = [], communityCards = [], pot = 20, players = [Player {playerType = RandomPlayer, name =
"p1", holeCards = [], chips = 100, isDealer = True, isActive = False},Player {playerType = RandomPlayer, n
ame = "p2", holeCards = [], chips = 80, isDealer = False, isActive = False},Player {playerType = RandomPla
yer, name = "p3", holeCards = [], chips = 30, isDealer = False, isActive = False}], smallBlind = 5, bigBli
nd = 10, currentDealer = 0, currentBets = [0,0,0]}
```

After much modification of the conditions to generate a number for stdMkGen, the decision was made to instead implement randomRIO. Although this was initially not going to be done due to the use of unsafePerformIO, this issue was resolved by changing the type signature of randomStrategy and associated functions such as makeMove. While this may pose problems when implementing other player strategies, this is the only way to resolve the issue of randomness, and so it has been solved this way.

After this change was implemented, randomStrategy's outputs were tested using QuickCheck, specifically a property test was designed to evaluate the correctness of randomStrategy under varying game states to ensure that all generated moves were valid. To achieve this, Arbitrary instances were defined for all relevant data types, thereby enabling QuickCheck to generate and execute 100 random test cases automatically.

```haskell
import (implicit) Prelude ( zip, ($), Eq((==)), Monad(return), Num((+), (*), ... (17 items)) | import (implicit) Prelude ( zip, ($), Eq((==)), Monad(return), Num
import Control.Monad
import Test.QuickCheck ( choose, elements, vectorOf, (==>), forAll, ... (5 items)) | import Test.QuickCheck ( choose, elements, vectorOf, (==>), forAll, ... (
import Test.QuickCheck
import Test.QuickCheck.Monadic ( assert, monadicIO, run ) | import Test.QuickCheck.Monadic ( assert, monadicIO, run )
import Test.QuickCheck.Monadic
import TexasHoldEm ( Rank(Ace, Two), Suit(..), Card(Card), PlayerMove(Raise, Fold, ... (13 items)) | import TexasHoldEm ( randomStrategy, Card(Card), G
import TexasHoldEm

prop_randomStrategy :: GameState -> Property
prop_randomStrategy gameState =
  not (null (players gameState))
    ==> forAll (elements (players gameState)) -- Makes sure >= 1 players in game, else test skipped
    $ \player ->
      -- Selects a RandomPlayer from the game
      playerType player
        == RandomPlayer
          ==> monadicIO
        $ do
          move <- run $ randomStrategy gameState player -- Run the IO function
          let playerChips = chips player
              playerIndex = head [i | (i, p) <- zip [0 ..] (players gameState), name p == name player]
              playerBet = currentBets gameState !! playerIndex
              minBet = maximum (currentBets gameState)
          -- Checks if each move made is valid against predefined conditions
          assert $ case move of
            Fold -> playerChips < minBet || minBet == 0
            Check -> minBet == 0
            Bet amount -> minBet == 0 && amount >= smallBlind gameState && amount <= playerChips
            Call -> playerChips >= minBet - playerBet
            Raise amount -> playerChips >= (minBet - playerBet) + amount

-- Arbitrary instances for random test data
instance Arbitrary Suit where
  arbitrary :: Gen Suit | arbitrary :: Gen Suit
  arbitrary = elements [Hearts, Diamonds, Clubs, Spades]

instance Arbitrary Rank where
  arbitrary :: Gen Rank | arbitrary :: Gen Rank
  arbitrary = elements [Two .. Ace]

-- Generaets card with random Rank and Suit
instance Arbitrary Card where
  arbitrary :: Gen Card | arbitrary :: Gen Card
  arbitrary = Card <$> arbitrary <*> arbitrary
```

```haskell

-- Generates random Player with varying chips
instance Arbitrary Player where
  arbitrary :: Gen Player | arbitrary :: Gen Player
  arbitrary = do
    name <- arbitrary
    holeCards <- vectorOf 2 arbitrary
    chips <- choose (0, 1000) -- Generates random chip count within range
    return $ Player RandomPlayer name holeCards chips False True

-- Generates random GameState with 4 players, varying small blinds, and varying current bets
instance Arbitrary GameState where
  arbitrary :: Gen GameState | arbitrary :: Gen GameState
  arbitrary = do
    deck <- vectorOf 52 arbitrary
    communityCards <- vectorOf 5 arbitrary
    pot <- arbitrary
    smallBlind <- choose (1, 10)
    let bigBlind = smallBlind * 2
    players <- vectorOf 4 arbitrary
    currentDealer <- choose (0, length players - 1)
    currentBets <- vectorOf (length players) (choose (0, 100))
    return $ GameState deck communityCards pot players smallBlind bigBlind currentDealer currentBets

main :: IO ()
main = do
  let args = stdArgs {maxSuccess = 100} -- Runs 100 tests
  quickCheckWith args prop_randomStrategy
```

```
ghci> :l Testing.hs
[1 of 3] Compiling TexasHoldEm      ( TexasHoldEm.hs, interpreted )
[2 of 3] Compiling Main             ( Testing.hs, interpreted )
Ok, two modules loaded.
ghci> main
+++ OK, passed 100 tests.
```

As the test has passed and bettingRound is fully implemented, gameLoop can now be implemented.

gameLoop

DNF

Step 4

PassivePlayer

This strategy is fairly straightforward (as much of the code for this feature has already been implemented in randomStrategy) and describes a player who only checks, calls and fold, never betting or raising. Since the makeMove function was wrapped in the IO monad for randomStrategy but passiveStrategy is deterministic, we can directly return the result in makeMove; this will also need to be done for the other player strategies. This function was briefly tested to ensure functionality, and as it is a simple function there are no edge cases or more robust testing methods needed. AggressivePlayer's strategy can now be implemented.

AggressivePlayer

The implementation of AggressivePlayer is once again very simple and comparable to PassivePlayer in many ways, with the caveat of always betting and raising when possible, calling and folding if forced, and never checking. This was fairly quick to implement and test briefly, and as it is a simple function, no robust testing methods are needed at this point nor have any edge cases been revealed in design planning.

SmartPlayer

DNF

Step 5

HumanPlayer

The additional feature to be added was one that allows for a human player to participate in the game through the command line. This was chosen as it fits the IO function already implemented by makeMove and allows for robust testing. To implement HumanPlayer, the humanStrategy function was created, allowing the player to interact through commands such as fold, check, call, bet <amount>, and raise <amount>. To read in this input, getLine from the Text.Read library for its additional ability to parse and validate the commands (i.e., Text.Read.readMaybe was used to safely parse numbers in the input for bet/raise without causing runtime errors).

One possible bug that has been considered is users typing the lowercase version of commands, which has been accounted for through the use of Data.Char.toLower to ensure all input is case-insensitive. This approach was chosen over direct read calls because readMaybe provides a more controlled way of handling potentially malformed inputs, and instead of crashing the program, gracefully returns an error message and re-prompts the player in a loop until a satisfactory answer is given.

Finally, to ensure a friendly user experience, relevant game information such as the player's chips, the pot size, the community cards, etc. will be displayed before prompting for input.

When testing this function, all actions gave the expected result, with the round ending when Fold was called, and bettingRound being called 5 times before termination when using non-fold moves. Thereby, all streets of preflop, flop, turn, river, and showdown are available to be played before the round ends. Therefore, although gameLoop was not able to be created and there are considerations to be made regarding the UX, HumanPlayer performs the functionality as described in the specification. For brevity, the end and start GameState outputs have been omitted.

FOLD SCENARIO:

```
ghci> main


Welcome to the Curry-no Royale, may the purest of luck be with you! Your details are as follows:


You are player 1
Your chips: 1000
The current community cards: []
Your hole cards: []
The pot is currently at 0 thousand dollars.
Minimum bet to call: 0
(Please enter F (fold), Ch (check), C (call), B <amount> (Bet <amount>), R <amount> (Raise <amount>)
 What is your move:
F


The round is over - look below to see how you did!
```

NON-FOLD SCENARIO:

```
ghci> main


Welcome to the Curry-no Royale, may the purest of luck be with you! Your details are as follows:


You are player 1
Your chips: 1000
The current community cards: []
Your hole cards: []
The pot is currently at 0 thousand dollars.
Minimum bet to call: 0
(Please enter F (fold), Ch (check), C (call), B <amount> (Bet <amount>), R <amount> (Raise <amount>)
 What is your move:
C

You are player 1
Your chips: 1000
The current community cards: []
Your hole cards: []
The pot is currently at 230 thousand dollars.
Minimum bet to call: 210
(Please enter F (fold), Ch (check), C (call), B <amount> (Bet <amount>), R <amount> (Raise <amount>)
 What is your move:
C 10
Invalid move - please try again!

You are player 1
Your chips: 1000
The current community cards: []
Your hole cards: []
The pot is currently at 230 thousand dollars.
Minimum bet to call: 210
(Please enter F (fold), Ch (check), C (call), B <amount> (Bet <amount>), R <amount> (Raise <amount>)
 What is your move:
B 10
Invalid move - please try again!

You are player 1
Your chips: 1000
The current community cards: []
Your hole cards: []
The pot is currently at 230 thousand dollars.
Minimum bet to call: 210
(Please enter F (fold), Ch (check), C (call), B <amount> (Bet <amount>), R <amount> (Raise <amount>)
 What is your move:
B
Invalid move - please try again!
```

```
You are player 1
Your chips: 1000
The current community cards: []
Your hole cards: []
The pot is currently at 230 thousand dollars.
Minimum bet to call: 210
(Please enter F (fold), Ch (check), C (call), B <amount> (Bet <amount>), R <amount> (Raise <amount>)
 What is your move:
R
Invalid move - please try again!

You are player 1
Your chips: 1000
The current community cards: []
Your hole cards: []
The pot is currently at 230 thousand dollars.
Minimum bet to call: 210
(Please enter F (fold), Ch (check), C (call), B <amount> (Bet <amount>), R <amount> (Raise <amount>)
 What is your move:
R 10

You are player 1
Your chips: 780
The current community cards: []
Your hole cards: []
The pot is currently at 1104 thousand dollars.
Minimum bet to call: 180
(Please enter F (fold), Ch (check), C (call), B <amount> (Bet <amount>), R <amount> (Raise <amount>)
 What is your move:
Ch
Invalid move - please try again!

You are player 1
Your chips: 780
The current community cards: []
Your hole cards: []
The pot is currently at 1104 thousand dollars.
Minimum bet to call: 180
(Please enter F (fold), Ch (check), C (call), B <amount> (Bet <amount>), R <amount> (Raise <amount>)
 What is your move:
Ch 10
Invalid move - please try again!
```

```
You are player 1
Your chips: 780
The current community cards: []
Your hole cards: []
The pot is currently at 1104 thousand dollars.
Minimum bet to call: 180
(Please enter F (fold), Ch (check), C (call), B <amount> (Bet <amount>), R <amount> (Raise <amount>)
 What is your move:
R 100

You are player 1
Your chips: 500
The current community cards: []
Your hole cards: []
The pot is currently at 2150 thousand dollars.
Minimum bet to call: 50
(Please enter F (fold), Ch (check), C (call), B <amount> (Bet <amount>), R <amount> (Raise <amount>)
 What is your move:
C


The round is over - look below to see how you did!

ghci>
```

Critical reflection

Before starting the project, I expected the implementation of a Texas Hold'Em game to be straightforward, especially given Haskell's coding efficiency. However, as I began to work, I saw significant difficulties in integrating the recursive game logic and ensuring that each function worked cohesively. One example, in particular, was creating randomStrategy for RandomPlayer, wherein I mistakenly used stdMkGen instead of randomRIO for the former's purity, despite the fact it was not sufficient for the project and there were 'impure' uses of IO in many other places throughout the project, thus making it both necessary and justified to use randomRIO from the start. This highlighted to me the importance of pre-planning code thoroughly and anticipating its future uses and functionality before implementation.

This issue was made apparent to me when implementing bettingRound, and this is where I realised that my focus on making the code modular had distracted me from considering it in the scope of the wider project. Reflecting on this, I now feel more aware of how planning code can enhance its robustness, clarity, and ease of implementation – especially in functional programming environments.

Another point of learning that I have gotten from this assignment is through the preparation of documentation where I learned the value of proper time-management and planning. Namely, the evaluation of what went well, what areas show problems, and how planning code in advance can enhance not only coding ability but also robust implementation of projects. Moving forward, I will manage my time more effectively to ensure I can implement all features of a project and give priority to the core functionality, as opposed to supplementary features.

In the future, I will apply the knowledge I have acquired by balancing systematic testing with a comprehensive design phase, in addition to structuring the code into 'steps' similar to this assignment, thereby providing myself with much-needed reflection and reallocation of tasks for my code. However, if I was faced with a similar challenge (rewriting/refactoring existing code to conform to new code), I would learn from my lessons and adopt a test-driven approach to maintain functionality while keeping the work efficient.

In summary, I believe that this project has taught me many things not only related to coding but also in relation to myself. Particularly that problem-solving in programming is a process requiring patience, planning, iteration, and improvement – similar to the process of navigating life and all its related experiences.