# Assignment 4

In this assignment we will write a program to play Blackjack. You can familiarize yourself with the rules at this page[1]. Here are the rules in brief (we will ignore some more elaborate play options and stick to a "normal" play, and keep to one player):

- The game is played with a normal deck of playing cards.
- Each card has a value: Aces count for 1 or 11 (more on that later), face cards count for 10, while all other cards count for their number.
- The goal is to reach a score as close to 21 as possible, but not above.
- The player can choose whether to count any ace as 1 or 11, depending on what works better for them. The dealer's moves are forced as we will describe shortly. The effect is that aces end up counting as 11 if this would not cause you to go above 21, and as 1 otherwise.
- A combination of an ace and a face card (a total of 21 in two cards) is called a *natural* or *blackjack*.
- We will call a hand with score over 21 (with aces counting as 1) *busted*.
- The play goes as follows:
  - The dealer deals two cards to the player and two cards to themselves. Only one of the dealer's cards is face up.
  - If the player and the dealer both have naturals, then it's a tie.
  - Otherwise, if the player has a natural, then they win.
  - Otherwise, if the dealer has a natural, then the player loses.
  - Otherwise, the player if offered the chance to draw another card (*Hit*) or to stop (*Stand*). This process ends when the player stands or when they are busted.
  - If the player is busted then they lose..
  - The dealer then reveals their other card. If they have a score of 17 or above then they stop. Otherwise they draw from the deck until their score is 17 or above or until they are busted.
  - If the dealer is busted (but the player is not), then the player wins.
  - Otherwise, if the player's score is *larger* than the dealer's then the player wins, otherwise the player loses.

In this lab we will implement a system for playing Blackjack. It will allow us to test a predetermined strategy as well as play an interative game.

Ways to interact with the system:

[1] https://bicyclecards.com/how-to-play/blackjack/

- Compile as normal via ghc assignment4.
- You can run the automated tests via ./assignment4 tests after you compile.
- You can combine compilation and test run in one command via:
  ```
  ghc assignment4 && ./assignment4 tests
  ```
- Most of the IO actions cannot be automatically tested. You should instead just try them out in the interactive terminal and observe what they do.
- Once you have written the appropriate code, you can see an automated play against a hard-coded strategy by doing ./assignment4 auto.
- Once you have written the appropriate code, you can run a simulation of multiple automated runs and see the total results via ./assignment4 sim xxxx where xxxx is the number of simulations to run (1000 is a good minimum, don't try more than 10000 unless you're willing to wait for a while).
- Once you have written the appropriate code, you can play the game interactively via ./assignment4 play.

We start with some type definitions:

```
data Suit = Hearts | Diamonds | Clubs | Spades     deriving (Eq, Show)
data Value = Ace | Num Int | Jack | Queen | King  deriving (Eq, Ord, Show)
type Card = (Suit, Value)
type Deck = [Card]
type Hand = [Card]
data Play = Hit | Stand                   deriving (Eq, Show)
data Result = Win | Loss | Tie            deriving (Eq, Show)
type Strategy = Hand -> Card -> Play
```

These definitions should be fairly explanatory, except perhaps for the last one. A "strategy" is basically the instructions for how the player is going to play: If they have the provided hand and the dealer has the provided card facing up, then they will take the appropriate action.

As an example, we consider the dealer's strategy: If their hand value is less than 17 then they choose Hit, otherwise they choose Stand. Its code looks like this:

```
dealer :: Strategy
dealer hand _ | count hand < 17  = Hit
              | otherwise        = Stand
```

1. We start with some simple methods to print cards: Write the methods showCard and showDeck that turn a Card and a Deck respectively into a string. Note that there is already a meaningful method show for suits, so you can do show suit to get a string from the suit. You may find it helpful to write a showValue function to turn a Value into a string. Note that showDeck prints the cards in reverse order (as this will work better for us later on).

   You MUST implement showDeck in a "point-free" way: Write it as a composition of three functions: reverse, map, and intercalate, the last two being provided their first parameter. You can use either of the two composition styles, . and >.>.

2. Next, implement a hasAces function that given a hand returns whether it has any aces in it. You MUST implement this function in a point free way, by doing a partial application to the any function, providing it with a suitable test function (perhaps called isAce).

3. Next, write a value function that given a card produces its "numerical value": Aces will count as 1 for now, face cards count for 10, while other cards count for their values.

   Also write a normalValue function that given a hand returns the total value of that hand by adding the values of the individual cards. You MUST implement this in a point-free way, by combining the sum function with a suitable application of map.

   Also write a function count that given a hand computes the correct "count" of points for the hand: It starts off with the result of normalValue and possibly adjusts it by 10 if there were any aces and adding 10 won't get the score above 21.

4. Next, we implement some utility methods for hands.

   Write functions isTwentyOne and isBusted which given a hand return whether the count is exactly 21 (respectively over 21). You MUST implement these functions in a point-free way, by composing the count method together with an operator section for the operator == or > as needed.

   Now implement a function isNatural to detect if a hand is a "natural", namely has count of 21 and consists of exactly two cards.

5. Next we implement strategies. You are being provided a dealer strategy, which implements the dealer's play. You are asked to implement your own strategy for a player, called simplePlayer. An example strategy could be "If the dealer card is an Ace of a face card, then hit if our count is less than 18 and stand otherwise, but if the dealer card is a number card then hit if our count is less than 17 and stand otherwise." You are free to implement whatever strategy you like. (You will have to write your own tests for that of course)

6. Next we start working towards dealing and actually playing the game. Start by implementing two functions, deal and draw. deal is given a deck and returns a pair of a hand containing the first two cards from the deck and a deck containing the remaining cards. draw draws the single card from the top and returns a pair of the card and the remaining deck. These are both fairly simple functions. You do NOT need to worry about running out of cards in your deck (so your matching doesn't need to deal with an empty deck case for example). Use a simple pattern match for these, and do NOT rely on any built-in list functions.

7. Now we need to write some of our main functions. Write a function dealPlayer which takes as input four things: A strategy, a hand (the player's hand), a card (the player's visible card) and a deck. It is supposed to effectively play the strategy, and return a pair of the resulting player's hand and the remaining deck.

   - The function should ask the provided strategy with the provided hand and dealer card, and find out the play.

- If the play is a Stand then it can return the correct state.
- If the play is a Hit then it must draw a card from the deck, add it to the hand, then recursively continue.

You will likely need to use a `case ... of` construct, make sure to read up on it.

8. Next we need a function `playRound`. This function takes in a strategy and a deck, and it effectively "plays" one round:

   - It deals cards to the player and the dealer.
   - It uses `dealPlayer` to carry out the player's play.
   - It uses `dealPLayer` to carry out the dealer's play.
   - It returns a triple of the player's hand, the dealer's hand, and the remaining deck.

   Your function needs to be very careful to keep track of the updated deck as it does each of the above steps. Each next step needs to use the updated deck returned from the earlier step.

9. Next we need a function `determineResult` which takes as input two hands (player's and dealer's) and returns the result of comparing those two hands, as a value of type `Result`. Specifically:

   - If they are both "natural" then it's a tie (`Tie`).
   - Otherwise if the player has a natural then they win (`Win`).
   - Otherwise if the dealer has a natural then the player loses (`Loss`).
   - Otherwise if the player is busted then they lose.
   - Otherwise if the dealer is busted then the player wins.
   - Otherwise if the player's count is larger than the dealer's then they win, otherwise they lose.

   This will likely be a simple function with many guard statements.

10. In order to be able to actually play the game, we need the ability to first create a full deck and then to shuffle the deck. This is difficult to do in a functional programming setting, as it fundamentally requires that the output of a computation be varied. Make sure you have understood the section on random numbers[2] first before attempting this part.

    This part consists of writing five functions that collectively perform a shuffle. Refer to the notes above for details on these functions: `getRs`, `pluck`, `shuffle`, `shuffleGen`, `shuffleIO`.

11. Once you have the general shuffling functions in place, it is time to tie them into our game. Our goal is to produce a shuffled deck. This will occur in two steps:

[2] ../notes/random_numbers.html

- Write a value fullDeck which contains the full deck of 52 cards. You can do this easily with a list comprehension, together with two helper functions allValues and allSuits.
- Write an action shuffledDeck which simply uses shuffleIO on the fullDeck.

12. Next up is a method playShuffled. It takes in a strategy and returns an action that uses playRound to play that strategy, then returns the pair of hands of the player and the dealer at the end of the round.

13. Next up you should implement an autoPlay action. It uses playShuffled to play the simplePlayer strategy, then prints out the hands of the player and the dealer using the provided showLabeledHand function. It ends up with printing the result on a new line, using determineResult. The result should be three lines.

    After you implement this function, you can now execute ./assignment4 auto and have an automatic game played.

14. Next you will help implement a function called playMany. It is used by the provided function simulate. It takes as input an integer n, then plays n rounds and tallies the results. playMany is provided for you, but you will need to implement the two helper methods it uses.

    One of these methods is playShuffledResult. It takes a strategy and produces an IO Result. You MUST implement this as a composition of two functions: One is playShuffled, and the other is the result of applying fmap to a suitable function. fmap is a function like map but it works with IO instead: If f :: a -> b then fmap f :: IO a -> IO b.

    The other method is called countStats. It takes as input a list of results, [Result], and retuns a pair of integers (Int, Int) holding the number of wins and losses that showed up (ignoring ties). You MUST implement this as foldr f (0, 0), where you must implement the function f to update the current count with the current value. For example if the current value is a Win and we have so far 5 wins and 3 losses, we should now have 6 wins and 3 losses. There are some tests for this function.

    Once you have implemented these methods, you should now be able to execute ./assignment4 sim 500 to run 500 simulations and print the results. You can see this way how good your simplePlayer algorithm is, and try to make it better.

15. In this last part, you will implement the manual player interaction. It is split up into a series of functions that you need to implement. Each will be a do sequence of actions.

    The top level function is manualPlay, with type IO (). It should do the following:

    - Print an empty line
    - Get a shuffled deck using the shuffledDeck action and storing its result
    - Draw the player's card and the dealer's card, and remember the updated deck through each step

- Call the playerInteract function which takes as input the player's hand, the dealer's hand and the remaining deck.
- End by recursively calling manualPlay, to allow the player to start a new round.

The next function to write is playerInteract, with type Hand –> Hand –> Deck –> IO (). It should do the following steps:

- Show the player's hand (using showLabeledHand) as well as the dealer's top card only (using showLabeledHand again).
- Use readPlayerChoice (you'll implement it next) to read a player's choice an store its result in a variable.
- Use a case ... of construct on that choice to consider two cases:
  - If the choice was Hit, then you must draw a card from the deck to add to the player's hand, then recursively call playerInteract with the updated player's hand, the dealer's hand and the updated deck.
  - If the choice was Stand, then you use the dealPlayer function to play the dealer's hand, resulting in an updated dealer's hand and deck, and then call the endGame function with the player's and dealer's hands. You will implement the endGame function in a moment.

Now implement the endGame function. It is given the two hands for the player an the dealer, then uses showLabeledHand to print them both, then print on a line a message for the result, which is computed via determineResult.

Lastly, implement the readPlayerChoice action, which does the following:

- Prompts the user to type "Hit" or "Stand".
- Then you use the action hFlush stdout. This makes sure your message is printed out even without a newline.
- Use getLine to read the user's input.
- Use a case ... of construct to test the user input:
  - If the input is the string "Hit" or the string "Stand", then return the corresponding Play values Hit or Stand.
  - Otherwise, recursively call readPlayerChoice to let the user type again.

If you have implement these correctly, you'll be able to play the game using ./assignment4 play.

## Start code

```
module Main where

import Test.HUnit
import System.Random
import Data.List (intercalate)
```

```haskell
import Control.Monad (replicateM)
import System.Environment (getArgs)
import System.IO (hFlush, stdout)

data Suit = Hearts | Diamonds | Clubs | Spades    deriving (Eq, Show)
data Value = Ace | Num Int | Jack | Queen | King  deriving (Eq, Ord, Show)
type Card = (Suit, Value)
type Deck = [Card]
type Hand = [Card]
data Play = Hit | Stand                 deriving (Eq, Show)
data Result = Win | Loss | Tie          deriving (Eq, Show)
type Strategy = Hand -> Card -> Play

infixl 9 >.>
(>.>) :: (a -> b) -> (b -> c) -> (a -> c)
f >.> g = \x -> g (f x)

showCard :: Card -> String
showCard (s, v) = ""

showHand :: Hand -> String
showHand = \s -> "stub.␣replace␣everything␣after␣the␣=."

hasAces :: Hand -> Bool
hasAces = \_ -> False

value :: Card -> Int
value _ = -1

normalValue :: Hand -> Int
normalValue = \h -> -1

count :: Hand -> Int
count = \h -> -1

isNatural :: Hand -> Bool
isNatural h = False

isTwentyOne :: Hand -> Bool
isTwentyOne = \_ -> False

isBusted :: Hand -> Bool
isBusted = \_ -> False

dealer :: Strategy
dealer hand _ | count hand < 17  = Hit
              | otherwise        = Stand

simplePlayer :: Strategy
simplePlayer pHand dCard = Stand

deal :: Deck -> (Hand, Deck)
deal d = ([], d)                -- dummy implementation

draw :: Deck -> (Card, Deck)
draw _ = ((Spades, Ace), [])    -- dummy value
```

```haskell
dealPlayer :: Strategy -> Hand -> Card -> Deck -> (Hand, Deck)
dealPlayer strat hand card deck = (hand, deck)

playRound :: Strategy -> Deck -> (Hand, Hand, Deck)
playRound strat deck = ([], [], deck)

determineResult :: Hand -> Hand -> Result
determineResult pHand dHand = Tie

getRs :: RandomGen g => Int -> g -> ([Int], g)
getRs n gen = ([], gen)

pluck :: Int -> [a] -> (a, [a])
pluck n (x:xs)  = (x, xs)

shuffle :: [Int] -> [b] -> [b]
shuffle rs xs = xs

shuffleGen :: RandomGen g => [a] -> g -> ([a], g)
shuffleGen xs gen = (xs, gen)

shuffleIO :: [a] -> IO [a]
shuffleIO d = return d

shuffledDeck :: IO Deck
shuffledDeck = return []

playShuffled :: Strategy -> IO (Hand, Hand)
playShuffled strat = return ([], [])

showLabeledHand :: String -> Hand -> String
showLabeledHand label hand = label ++ ":␣" ++ show (reverse hand)

playShuffledResult :: Strategy -> IO Result
playShuffledResult = \strat -> return Tie

autoPlay :: IO ()
autoPlay = return ()

countStats :: [Result] -> (Int, Int)
countStats = foldr f (0, 0)
    where f _ _ = (0, 0)

playMany :: Int -> IO (Int, Int)
playMany n = fmap countStats $ replicateM n (playShuffledResult simplePlayer)

simulate :: Int -> IO ()
simulate n = do
  (wins, losses) <- playMany n
  putStrLn $ "Wins:␣" ++ show wins ++ "␣Losses:␣" ++ show losses
  putStrLn $ "Win␣ratio:␣" ++ show (fromIntegral(wins) / fromIntegral(wins + losses))

readPlayerChoice :: IO Play
readPlayerChoice = return Stand
```

```
playerInteract :: Hand -> Hand -> Deck -> IO ()
playerInteract pHand dHand deck = return ()

endGame :: Hand -> Hand -> IO ()
endGame pHand dHand = return ()

manualPlay :: IO ()
manualPlay = return ()

naturalHand = [(Clubs, Ace), (Hearts, Jack)]
seventeen   = [(Clubs, Num 8), (Hearts, Num 5), (Hearts, Num 4)]
sixteen     = [(Clubs, Num 7), (Hearts, Num 5), (Hearts, Num 4)]
aceAsOne    = [(Clubs, Num 10), (Clubs, Ace), (Hearts, Jack)]
bustedHand  = [(Clubs, Num 8), (Clubs, Num 5), (Hearts, Jack)]
sampleDeck = [
    (Spades,Num 3), (Clubs,Num 8), (Diamonds,Num 7),
    (Spades,Num 7), (Clubs,Num 4), (Hearts,Num 10),
    (Diamonds,King), (Diamonds,Num 9), (Diamonds,Num 4),
    (Clubs,Num 9)]
drawsOnce hand _ | length hand > 2     = Stand
                 | otherwise           = Hit
testGen  = mkStdGen 1
testGen2 = mkStdGen 2

tests = TestList [
    showCard (Clubs, Ace)             ~?= "Ace of Clubs",
    showCard (Spades, Num 4)          ~?= "4 of Spades",
    showCard (Diamonds, Jack)         ~?= "Jack of Diamonds",
    showCard (Hearts, Queen)          ~?= "Queen of Hearts",
    showHand naturalHand              ~?= "Jack of Hearts, Ace of Clubs",
    hasAces naturalHand               ~?= True,
    hasAces seventeen                 ~?= False,
    hasAces []                        ~?= False,
    value (Clubs, Ace)                ~?= 1,
    value (Clubs, Jack)               ~?= 10,
    value (Clubs, Num 5)              ~?= 5,
    normalValue naturalHand           ~?= 11,
    normalValue seventeen             ~?= 17,
    normalValue aceAsOne              ~?= 21,
    count naturalHand                 ~?= 21,
    count seventeen                   ~?= 17,
    count aceAsOne                    ~?= 21,
    count bustedHand                  ~?= 23,
    isTwentyOne bustedHand            ~?= False,
    isTwentyOne naturalHand           ~?= True,
    isTwentyOne seventeen             ~?= False,
    isTwentyOne aceAsOne              ~?= True,
    isBusted bustedHand               ~?= True,
    isBusted naturalHand              ~?= False,
    isNatural naturalHand             ~?= True,
    isNatural seventeen               ~?= False,
    isNatural aceAsOne                ~?= False,
    dealer seventeen (Clubs, Ace)     ~?= Stand,
    dealer sixteen   (Clubs, Ace)     ~?= Hit,
    deal sampleDeck                   ~?= ([sampleDeck!!0, sampleDeck!!1],
                                            drop 2 sampleDeck),
    draw sampleDeck                     ~?= (sampleDeck!!0, tail sampleDeck),
    playRound drawsOnce sampleDeck      ~?= (
                                            [sampleDeck!!4, sampleDeck!!0, sampleDeck!!1],
                                            [sampleDeck!!5, sampleDeck!!2, sampleDeck!!3],
                                            drop 6 sampleDeck),
    determineResult
      naturalHand naturalHand           ~?= Tie,
    determineResult
      naturalHand aceAsOne              ~?= Win,
    determineResult
      aceAsOne naturalHand              ~?= Loss,
    determineResult
      bustedHand seventeen              ~?= Loss,
    determineResult
      seventeen bustedHand              ~?= Win,
    determineResult
      bustedHand bustedHand             ~?= Loss,
    determineResult
      seventeen seventeen               ~?= Loss,
    determineResult
      sixteen seventeen                 ~?= Loss,
    determineResult
      seventeen sixteen                 ~?= Win,
    fst (getRs 4 testGen)               ~?= [3, 1, 1, 0],
    fst (getRs 4 testGen2)              ~?= [1, 0, 0, 0],
    pluck 3 "ABCD"                       ~?= ('D', "ABC"),
    pluck 2 "ABCD"                       ~?= ('C', "ABD"),
    pluck 1 "ABCD"                       ~?= ('B', "ACD"),
    pluck 0 "ABCD"                       ~?= ('A', "BCD"),
    shuffle [3,1,1,0] "ABCD"             ~?= "DBCA",
    fst(shuffleGen "ABCD" testGen)       ~?= "DBCA",
    fst(shuffleGen "ABCD" testGen2)      ~?= "BACD",
    countStats []                        ~?= (0, 0),
    countStats [Win, Win, Loss]          ~?= (2, 1),
    countStats [Loss, Tie, Loss]         ~?= (0, 2)
  ]

main :: IO ()
main = do
   args <- getArgs
   case args of
      ("tests" : _) -> do runTestTT tests
                          return ()
      ("auto" : _) -> autoPlay
      ("play" : _) -> manualPlay
      ("sim": nums : _) -> simulate $ read nums
      _   -> putStrLn "Allowed␣options:␣tests,␣auto,␣play,␣sim␣<nTimes>"
```