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
  ```

- 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:

```haskell
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:

```haskell
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. Start with a function isNatural to detect if a hand is a "natural", namely has count of 21 and consists of exactly two cards.

   Next, write a function isBusted which given a hand returns whether the count is over 21. You MUST implement this function in a point-free way, by composing the count method together with an operator section for the operator >.

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.

Start code:

```haskell
module Main where

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

```haskell
(>.>) :: (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

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

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

simplePlayer :: Strategy
dealer pHand dCard = Stand
```