Effective Functional Programming
User Interaction
Assignment 2
`Blackjack`

Paul Downen

In the previous assignment, you modeled the core functionality of a playing card game, namely blackjack. Now, you will build on that core by writing an interface for playing blackjack as a command-line program, using your types and functions defined in assignment 1 as a library. In the end, you will have written an executable program from start to finish that interacts with the user to play the card game. The following is an example input/output interaction:

```
Welcome to blackjack!

Ready?
help
Press Enter to continue
Ready?

The dealer's first card is: 6♢.
Your hand is Q♢ 2♣ (12), what do you do?
huh
I didn't understand that.
You can either "hit" or "stand".
Your hand is Q♢ 2♣ (12), what do you do?
hit
Your hand is 2♡ Q♢ 2♣ (14), what do you do?
hit
You busted! 10♠ 2♡ Q♢ 2♣ (24) The house wins.

Ready?

The dealer's first card is: 2♠.
Your hand is 5♣ 5♢ (10), what do you do?
hit
Your hand is 10♣ 5♣ 5♢ (20), what do you do?
stand
```

```
The dealer reveals the hand: J♣ 2♠ 7♡ (19)
You win!

Ready?

The dealer's first card is: A♢.
Your hand is A♠ J♠ (21), what do you do?
stand
The dealer reveals the hand: A♢ K♣ (21)
Tie; nobody wins.

...

Ready?

The dealer's first card is: J♡.
Your hand is K♢ 3♣ (13), what do you do?
hit
Your hand is A♡ K♢ 3♣ (14), what do you do?
hit
Your hand is 7♠ A♡ K♢ 3♣ (21), what do you do?
stand
The dealer reveals the hand: 9♣ J♡ 2♢ (21)
Tie; nobody wins.

Ready?

The dealer's first card is: 9♠.
Your hand is 9♢ 4♣ (13), what do you do?
hit
You busted! 9♡ 9♢ 4♣ (22) The house wins.

Ready?

Shuffling a new deck...
The dealer's first card is: 2♣.
Your hand is Q♡ Q♢ (20), what do you do?
stand
The dealer is busted! Q♠ A♠ 3♢ 2♣ 6♣ (22) You win!

Ready?
quit
Bye.
```

1 Dealing Cards (30 points)

The central part of a card game is the deck of cards, which is used to deal cards to the player and the dealer. The key point of deck management is that decks should be randomized by shuffling before they are used (so that the order is unpredictable), and that cards are drawn one-at-a-time from the top of a deck until it runs out, at which point a new deck is shuffled and used. Since you already wrote a `shuffle` function in assignment 1 which uses a list of indexes to decide the order of the shuffle, the challenge is to generate "enough" random numbers to shuffle the deck.

Random number generators aren't included in the Haskell standard library packaged with GHC, but they can be found in the `random` package on Hackage which provides the `System.Random` module. In `System.Random`, there is the generic function

```
randoms :: (Random a, RandomGen g) => g -> [a]
```

which consumes a random generator state of type `g` to produce an infinite list of random values `[a]`. The only restriction on the generic types are the type class constraints `RandomGen g` and `Random a`. The `System.Random` module provides a standard generator type `StdGen` with an accompanying `RandomGen` instance along with the IO operation

```
newStdGen :: IO StdGen
```

that uses the state of the system to create a fresh new random number generator. The `System.Random` module also provides a `Random` instance for the `Int` type. So by specializing the generic types in `randoms` like so:

```
randoms :: StdGen -> [Int]
```

this function can be used to generate a random list of `Int`s, which will be more than enough for the purpose of shuffling a deck of cards.

Exercise 1.1 (10 points)**.** Using the `newStdGen` and `randoms` from `System.Random`, implement the IO operation

```
freshDeck :: IO Deck
```

which returns a fresh new deck of playing cards by

1. creating a new `StdGen` random number generator,
2. using that new `StdGen` value to generate an infinite list of random `Int`s,
3. return the result of `shuffle`ing a `fullDeck` of cards using the list of random numbers to determine the order.

Exercise 1.2 (15 points)**.** Implement the three functions

```haskell
draw    :: Deck -> IO (Card, Deck)
hitHand :: Hand -> Deck -> IO (Hand, Deck)
deal    :: Deck -> IO (Hand, Deck)
```

`draw` should return the top `Card` of the given `Deck`, along with the remainder of the `Deck` with the top card removed. In the case where the given `Deck` is empty, print out a message that you are shuffling a new deck, get a `freshDeck`, and then `draw` from that `Deck`.

`hitHand` should `draw` one card from the given deck and add it to the given hand, returning both the larger hand and the remaining deck.

`deal` should `draw` two cards from the given deck and return a `Hand` consisting of those two cards as well as the remaining deck. This is equivalent to hitting an empty hand twice. NOTE that you should not re-use the same `Deck` for the two draws, but use the `Deck` returned from the first `draw` to perform the second `draw`.

Exercise 1.3 (5 points)**.** Implement a pretty printing function for nicely displaying `Hand`s

```haskell
prettyPrint :: Hand -> String
```

which shows both the `Card`s in the `Hand` followed by its `handValue` in parenthesis.

2 User Input (15 points)

In class, you saw both a simplistic and robust method of prompting a user for input and reading the result. Now, you will implement a more user-friendly version of `prompt` that explains what is expected of the user and allows the user to quit the program.

Exercise 2.1 (15 points)**.** To interact with the user, your program needs to prompt them for input. The logic of prompting the user for input, parsing that input into some other type, handling improper inputs by repeated prompting, and reacting to proper inputs can be abstracted out into a function:

```haskell
prompt :: String                -- query message
       -> String                -- help message
       -> (String -> Maybe a)   -- parse input to get an 'a' value
       -> (a -> IO b)           -- reaction to the 'a' input
       -> IO b                  -- result after successful input
```

the IO action

```haskell
  prompt query help parse act :: IO b
```

should perform the following steps:

1. Print out the `query` message `String`.

2. Read in a line from the user (referred to as `input` in the following).
3. Depending on `input` from step 2, do one of the following actions:
   - If `input` is `"quit"`, exit the program.
   - If `input` is `"help"`, print out the `help` message and go to step 1.
   - If `input` is neither `"quit"` nor `"help"`, and `parse input` is `Nothing`, then print out a message stating that the input was not understood, print out the `help` message, and go back to step 1.
   - If `input` is neither `"quit"` nor `"help"`, and `parse input` is `Just x`, then return the result of `act x`.

Implement the `prompt` function as described above.

Hint 2.1. When you run the `exitSuccess :: IO a` action from the `System.Exit` module, the program will immediately exit.

3 The Player's and Dealer's Turns (35 points)

Exercise 3.1 (15 points). Use `prompt` to implement the function

```
playerTurn :: Hand -> Deck -> IO (Hand, Deck)
```

that queries the player for their next move, given their current `Hand` and the `Deck` of cards. The query message should show the player's current hand and ask what they want to do next. The help message should say that the player can either hit or stand. The parsing function should convert the input string to a `Move`. Finally, the action to be performed depends on the `Move` selected by the user:

1. On a `Hit`, hit the player's current hand. If the value of the new hand is larger than 21, then return immediately with the player's busted hand and the remaining deck. Otherwise, query the player again for their next move with their new hand and remaining deck.
2. On a `Stand`, return the the player's current hand and the deck.

In contrast with the player, the dealer in blackjack is on complete autopilot. Like the player, the dealer is dealt an initial hand of two cards. When it comes to the dealer's turn, the dealer will draw cards until their hand value is larger than 16 or they bust; whichever comes first.

Exercise 3.2 (15 points). Implement the function

```
dealerTurn :: Hand -> Deck -> IO (Hand, Deck)
```

which performs dealer's autopilot turn given their current `Hand` of cards. If the value of the dealer's hand is strictly less than 17, then hit the dealer's hand and go back to the top of the dealer's turn with the new hand and remaining deck. Otherwise, immediately return the dealer's current hand and deck.

4 Putting It All Together (25 points)

Now that the logic for the player's and dealer's turns have been implemented, you can now put together the main game loop for playing blackjack.

Exercise 4.1 (20 points)**.** Implement the primary game loop function

```
gameLoop :: Deck -> IO a
```

The game loop should

1. Ask if the player is ready, and wait for them to press enter via `prompt`.
2. Deal the dealer their initial hand of two cards; one of them is visible and revealed to the player and the other one is hidden and kept secret.
3. Deal the player their initial hand of two cards and then do the `playerTurn` to get their final hand. If the player busted (*i.e.,* the value of their hand is strictly greater than 21), inform them that they lose and go to the top of the game loop.
4. Do the `dealerTurn` to get the dealer's final hand. If the dealer busted, inform the player that they win and go to the top of the game loop.
5. Finally, then compare the player's and dealer's hands. A blackjack (*i.e.,* a two-card hand with the value 21) is the best hand, and otherwise, the hand with the largest value is the best. For example, a hand containing an ace and a face card is a blackjack, and beats a hand consisting of a 9, 8, and 4 which has a total value of 21. The best hands wins, and if both hands are equal (that is, two blackjacks or two non-blackjack hands of equal value) the result is a tie and nobody wins. Print out the result of the round and go back to the top of the game loop.

Exercise 4.2 (5 points)**.** Implement the main entry point to the program

```
main :: IO ()
```

which should print a welcome message, shuffle up a fresh deck, and then begin the `gameLoop`.

5 *Bonus:* Betting (10 extra credit)

Bonus Exercise 5.1 (5 extra credit)**.** Blackjack is typically played as a means of gambling. Extend your game with a betting mechanism. The player should start with an initial wallet of money. Then, at the start of a round before any cards are dealt, ask the player how much money they would like to bet for that round which leaves their wallet and goes into a betting pool. If the player wins the round, they should get back twice the amount that they bet. If the player loses, then they lose their bet. For example, if the player starts with $100, bets $10 and wins the round, they end up with $110 in their wallet afterward. Otherwise, if they start with $100, bet $10 and lose the round, they end up with $90.

Hint 5.1. The `read :: String -> Int` function can convert a `String` to an `Int`. But beware! If `read :: String -> Int` is given a non-numeric string, like `"123abc"`, it will raise an exception. So before `read`ing a `String`, to get an `Int`, take care to check that the string valid, *i.e.,* a string of only numeric characters.

Bonus Exercise 5.2 (5 extra credit)**.** Another standard move that a player can make in blackjack is to *surrender*. When a player surrenders, they choose to lose that round, but get back half of the money they bet. For example, if a player starts with $100 in their wallet, bets $10, and then surrenders, they would end up with $95. Add `Surrender` as another `Move` a player can make, and extend the `playerTurn` logic to handle the case where the player chooses to surrender.