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?

The dealer's first card is: 6♢.
Your hand is Q♢ 2♣ (12), what do you do?
huh
I didn't understand that.
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: 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 busted! Q♠ 3♢ 2♣ 7♣ (22) You win!

Ready?
```

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

```haskell
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 `do`ing the following

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

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

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

`draw` should return a pair of (1) the top `Card` of the given `Deck` and (2) the remainder of the `Deck` with the top card removed. In the case where the given `Deck` is empty, `do` the following:

1. print out a message to the user that you are shuffling a new deck,
2. get a `freshDeck` of cards to use, and finally
3. `draw` again from the freshly shuffled `Deck` of cards you got from step 2.

`hitHand` should `do` the following:

1. `draw` one card from the given deck, unpacking the pair of the top card and the remaining `Deck` that is left over afterward, and
2. add it to the given `Hand`, returning a pair of both the new `Hand` (with 1 more card) 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. In particular, `deal` should `do` the following:

1. Hit (by calling `hitHand` above) an empty hand `[]` with the initial `Deck` given to `deal`.
2. Hit again, using the 1-card `Hand` and remaining `Deck` returned by step 1.
3. Return the pair of the 2-card hand and remaining deck from step 2.

NOTE do 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

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

which shows both the `Card`s in the `Hand` followed by its `handValue` in parenthesis. For example, the `Hand` containing the ace of spades and King of diamonds should be `prettyPrint`ed as `"AS KD (21)"` (or as `"A♠ K♢ (21)"` if you are using unicode suits).

2 *Bonus:* User Input (15 extra credit)

In class, you saw both a simplistic and robust method of prompting a user for input and reading the result. The `prompt` function

```
prompt :: String                 -- Question to ask
       -> (String -> Maybe a)    -- Parse the answer
       -> IO a
prompt question parse = do
  putStrLn question
  answer <- getLine
  case parse answer of
    Nothing -> do putStrLn "I didn't understand that."
                  prompt question parse
    Just a  -> return a
```

`do`es the following:

1. prints out the `question` message to the user,
2. reads the `answer` string input by the user, and
3. tries to `parse` the user's `answer`. The two possible results of `parse` are

- `Nothing`, meaning the `parse` was unsuccessful and the user input an incorrect command. In this case, `prompt` informs the user that it couldn't understand that answer, and `prompt`s the user again with the same arguments.
- `Just a`, meaning the `parse` successfully returned the result `a`, which is the final result of `prompt`.

To be more user-friendly, you can implement a version of `prompt` that explains what is expected of the user and allows the user to quit the program.

Bonus Exercise 2.1 (15 extra credit)**.** 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:

```
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

```
  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)

After being given their initial 2-card hand, the player has two possible moves. They can either:

- "hit" which means they ask for another card to be added to their hand, or
- "stand" which means they will keep the hand they have for the remainder of the round.

Both of these moves are represented by this data type:

```
data Move = Hit | Stand
```

Exercise 3.1 (15 points)**.** First, define a function

```
parseMove :: String -> Maybe Move
```

that parses a `String` and tries to return one of the two `Move` values. In the cases of a successful parse, `parseMove` should return `Just`

- `Hit` when given the string `"hit"`, and
- `Stand` when given the string `"stand"`.

Given any other strings, `parseMove` should return `Nothing`.

Next, define a function

```
playerMove :: Hand -> Deck -> Move -> IO (Hand, Deck)
```

which carries out the `Move` given as a parameter:

- Given a `hand :: Hand`, `deck :: Deck`, and `Stand :: Move` as parameters, `playerMove` should return the pair of the given `hand` and `deck` as-is, and `do` nothing else.
- Given a `hand :: Hand`, `deck :: Deck`, and `Hit :: Move` as parameters, `playerMove` should `do`
  1. hit `hand` with `deck`, to get a new `Hand` with 1 more card, and a new remaining `Deck`,
  2. if the `handValue` of the new hand is strictly greater than 21, then return it and the remaining `Deck` from step 1,
  3. otherwise, continue on with the player's turn by running `playerTurn` (that you will define next) with the new hand and remaining deck from step 1.

Finally, implement the player's full turn in the function

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

that asks the player for their next move given their current `Hand` and the `Deck` of cards. `playerTurn` should `do` the following

1. Create a query message that shows the player's current `Hand` (with `prettyPrint`),
2. `prompt` to display the query message from step 1 and parse their result with `parseMove`, and finally
3. perform the `playerMove` (that you already defined above) with the current `Hand` and `Deck` and the `Move` selection returned from step 2.

Note that in some cases, `playerTurn` can call `playerMove`, and `playerMove` can in turn call `playerMove`, and so on. This is a version of recursion known as *mutual recursion*, because `playerMove` and `playerTurn` are both mutually defined in terms of one another.

Hint 3.1. Like all other data types, you can pattern-match on the `Move` argument given to `playerMove` give two separate actions of what to do for the two different `Move` choices.

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. The dealer's next action depends entirely on the value (calculated from `handValue`) of the dealer's current `Hand`:

- If the value of the dealer's hand is less than or equal to 16, then `dealerTurn` should `do` the following:
  1. hit the dealer's current `Hand` with the current `Deck`, then
  2. run `dealerTurn` again with the new `Hand` and remaining `Deck` returned from step 1.
- 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)**.** The main loop of the game will be the function

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

that takes the current value of the shared `Deck` as an argument, and doesn't return (because the game can potentially go on forever. In order to prepare to write `gameLoop`, you will need to helper functions.

First, define the function

```
bustCheck :: String -> String -> Hand -> Deck -> IO ()
```

that checks if the given `Hand` is busted (meaning `handValue` returns a value of 22 or greater for that `Hand`). When a player or dealer gets a busted hand, they automatically loose, immediately ending the round and restarting the main game loop from the top. The two `String` arguments passed to `bustCheck` tell it who that `Hand` belongs to, and the winner if the `Hand` is busted.

The action of `bustCheck who win hand deck` depends on the value of the given `hand`:

- If the `handValue` of `hand` is equal to 22 or greater, then `bustedCheck` should `do` the following:
    1. Print out a message saying that `who` busted, along with the `prettyPrinted` `hand` and proclaming the `win`.
    2. Begin a new `gameLoop` with the given `deck`.
- Otherwise, the `handValue` is 21 or less, and `bustedCheck` should just immediately `return`.

Next, define the following function `compareHands`

```
data Result = Lose | Tie | Win

compareHands :: Hand -> Hand -> Result
```

which returns the result of two `Hand`s. Since we will already be ruling out busted hands as soon as they happen, we can assume that both `Hand`s given to `compareHands` has a value of 21 or less, so the winner is determined by whoever has the highest value, according to `handValue`.

`compareHands playerHand dealerHand` should return

- `Win` if the first argument `playerHand` has a higher value,
- `Lose` if the second argument `dealerHand` has a higher value, or
- `Tie` if both arguments have equal values.

Now you can implement the primary game loop function

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

The game loop should `do` the following

1. Ask if the player is ready, and wait for them to press enter via `prompt`. Since we are just waiting for the player to enter anything, and will be ignoring any `String` they input at this time, you can use the "parsing" function that `Just` returns any given `String` as-is.

2. Using the initial `Deck` given to `gameLoop`, `deal` to the dealer to get their initial hand of 2 cards and the remaining deck.
3. Reveal the first card of the dealer's 2-card `Hand` to the player by printing a message; the other card is hidden and kept secret.
4. Using the `Deck` returned by step 2, `deal` the player their initial hand of 2 cards and the remaining deck.
5. Run `playerTurn` using the player's hand and deck returned from step 4 to get their final hand and a new remaining deck.
6. Perform the `bustCheck` on the player's hand and deck returned from step 5; if so, say that the `"player"` busted, and that `"The house wins."`
7. Run `dealerTurn` using the `Hand` returned by step 2 and the *most recent* `Deck` returned by step 5 to get the dealer's final hand.
8. Perform another `bustCheck`, this time on the dealer's hand and deck from the previous step 7; if so say that the `"dealer"` busted, and that `"You win!"`
9. Using the player's final hand from step 5 and dealer's final hand from step 7, check who won by `compareHands`. Chech which `case` the comparison returns, and depending on the result, print out if the player won, the dealer won, or if there is a tie.
10. Finally go back to the top of the game loop.

Bonus Exercise 4.2 (5 extra credit)**.** The victor proclaimed by `compareHands` in exercise 4.1 is a simplification of the full game rules. The real game has the notion of the *blackjack*, which is hand of exactly 2 cards with a total value of 21. A blackjack is only possible with one ace card combined with any other 10-valued card (a King, Queen, Jack, or numeric 10 card). Blackjack hands are considered more valuable than any other 21-card hand. For example, the hand containing exactly A♢ K♣ will beat the hand containing 9♣ J♡ 2♢, even though they both have the same score of 21.

Implement the full game rules by taking blackjack (that is, 2-card 21-valued hands) into account in your `comapreHands` function. This should check for the extra cases where both hands have a value of 21, but one is made up of only 2 cards while the other has 3 or more cards, so that the 2-card hand wins.

Exercise 4.3 (5 points)**.** Implement the main entry point to the program

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

which should `do` the following

1. print a nice welcome message,
2. shuffle up a `freshDeck` to start with, and then
3. 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.