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?

Shuffling a new deck...
The dealer's first card is: 5S.
Your hand is JH AC (21), what do you do?
huh
Sorry, I didn't understand that.
Your hand is JH AC (21), what do you do?
help
You can "hit" or "stand"
Your hand is JH AC (21), what do you do?
stand
You Win!
Your hand: JH AC (21)
The dealer's hand: 7D 5S KS (22)

Ready?

The dealer's first card is: AD.
Your hand is 3S 8D (11), what do you do?
hit
Your hand is 5C 3S 8D (16), what do you do?
stand
```

```
You Win!
Your hand: 5C 3S 8D (16)
The dealer's hand: AD AS (22)

Ready?

The dealer's first card is: 10H.
Your hand is KD 9C (19), what do you do?
stand
You Win!
Your hand: KD 9C (19)
The dealer's hand: 10D 10H 2H (22)

...

Ready?

The dealer's first card is: 8S.
Your hand is 2C 2S (4), what do you do?
hit
Shuffling a new deck...
Your hand is JS 2C 2S (14), what do you do?
hit
The house wins.
Your hand: QH JS 2C 2S (24)
The dealer's hand: 8S 10S (18)

Ready?
quit
```

1 Dealing Cards (25 points)

The central part of a card game is the deck of cards used to deal cards to the player(s) and dealer. The key point of deck management is that decks should always 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.

A (pseudo-)random number generator can be found in the `random` package on Hackage which provides the `System.Random` module, which provides many types and operations for manipulating and generating various kinds of random values and sequences. Using `newStdGen` and `randoms` from `System.Random`, we can shuffle a `freshDeck` of cards like so:

```haskell
freshDeck :: IO Deck
freshDeck = do
  gen <- newStdGen
  let indexes = randoms gen
  return (shuffle indexes fullDeck)
```

The type of `freshDeck` says it is an `IO` action that will return a `Deck` when completed. `freshDeck`'s code is organized by a `do` which lists 3 separate steps — each of which may `do` `IO` side effects on their own — that need to be performed in sequence from top to bottom every time `freshDeck` is run:

1. First, `freshDeck` will make a new randomness generator "seed" of type `StdGen` using the `newStdGen` operation. `newStdGen` has to perform `IO` to read the current state of the machine to remember what other randomness operations happened before so that a different seed is returned every time it run. This side effect is reflected in its type, `newStdGen :: IO StdGen`. `freshDeck` needs to get access to the actual `StdGen` value it returns, which is done through the `do` bind statement `gen <- newStdGen` that assigns the local variable name `gen` to the result returned this time.

2. Second, `freshDeck` needs a list of random numbers to provide to the `shuffle` function. These random numbers can be generated by the function `randoms` that can generate an (endless) list of random numbers starting from any seed. Note that, as is usual for pseudo-random number generators, the seed provided to `randoms` determines exactly the sequence of "random" numbers it returns. In other words, `randoms 491412` will return the exact same list of random-seeming numbers every time. This makes `randoms` a pure function which does not need to do any `IO` action to figure out its answer, so it has the type `randoms :: StdGen -> [Int]`. Since `randoms gen` has the type `[Int]`, which is not an `IO` type, we can use the syntax `let indexes = randoms gen` to assign its result to the local variable `indexes`.

3. Third, `freshDeck` can generate a new ordering of a `fullDeck` of cards by calling `shuffle indexes fullDeck`, and `return` the shuffled deck as its answer.

Exercise 1.1 (10 points)**.** Implement the function

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

What `draw` does depends on whether or not it is given a `Deck` with any remaining cards in it.

- In the case that `draw` is given a non-empty `Deck`, it 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, `draw` should `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` you got from step 2.

End Exercise 1.1

Hint 1.1. The best way to cover the two cases of what `draw` should do depending on the `Deck` it is given is to *pattern-match* on its argument:

- If its argument matches the non-empty list `(card:deck)`, as in the function call `draw (card:deck)`, then it should just `return (card, deck)`.
- If its argument matches the empty list `[]`, as in the function call `draw []`, then it should `do` the sequence of 3 steps described above to shuffle a new deck and draw from it. *End Hint 1.1*

Exercise 1.2 (10 points)**.** Implement the three functions

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

`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`. **End Exercise 1.2**

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). **End Exercise 1.3**

2 The Dealer's Turn (20 points)

The dealer is forced to play their turn on autopilot following pre-set rules, which can be carried out automatically within the program without having to talk to the user at all. After being dealt their initial hand of two cards, the dealer will draw cards until their hand has a score of 16 or more, at which point they must stop. At the end of this turn, the dealer's hand will either have a score between 16 and 21, which is `OK`, or will have a score of 22 or more, which is `Bust`. The `Status` of a hand — being one of these two options — is represented by this enumeration data type:

```
data Status = Bust | OK
```

Exercise 2.1 (5 points)**.** Implement the function

```
checkHand :: Hand -> Status
```

which determines whether the given hand is `OK` because it has a score of 21 or less, or `Bust` because it has a score of 22 or more.

Your decision on the status of a hand can use either the simplified scoring rules calculated by `handValue` or the full scoring rules calculated by `betterHandValue` from assignment 1, if you finished the extra credit. Using either scoring rules correctly will earn the same points in this assignment. You only need to consistently use one or the other throughout these exercises. **End Exercise 2.1**

The game of blackjack has several moving parts, which all have to be kept track of at each step. To help keep everything organized, we'll be using this data type for the state of a `Game` which holds all of the information available on the `Table` shared by the player and dealer:

```
data Game a = Table {
  player :: Hand,
  dealer :: Hand,
  deck   :: Deck,
  ask    :: a
  }
```

This data type definition uses *record syntax*, which provides a few helpful (but optional) shorthands for managing a structure like `Table` with many parts. In addition to creating the constructor named

```
Table :: Hand -> Hand -> Deck -> a -> Game a
```

for building new `Game a` states, it also creates field accessor functions

```
player :: Game a -> Hand
dealer :: Game a -> Hand
deck   :: Game a -> Deck
ask    :: Game a -> a
```

which take a `Game a` and extract one of the parts: the `Hand` held by either the `player` or `dealer`, the current `deck` of cards, and finally an `ask` field that we'll be using later to talk to the user. This record data type definition is equivalent to the following code:

```
data Game a = Table Hand Hand Deck a

player (Table p _ _ _) = p
dealer (Table _ c _ _) = c
deck   (Table _ _ d _) = d
ask    (Table _ _ _ a) = a
```

In addition, record syntax gives us a more convenient way to build `Game a` values by using the names of the fields, rather than positions. For example, to make an empty game from scratch, we could apply `Table` to four arguments as in `Table [] [] [] ()`, or we could name each field that is being initialized as in `Table{player = [], dealer = [], deck = [], ask = ()}`. By using names, the order doesn't matter. We can also create slight modifications of `Game a` in a similar way. If `t` is already a `Game a` value, then `t{deck = newDeck}` is the same as `t` in all fields except for `deck` which contains `newDeck` instead. As another example, `t{dealer = ace spaces : dealer t}` is the same as `t` except that the `dealer`'s `Hand` now starts with the ace of spades, and then continues as the original `dealer` hand from `t`.

As an example of how to use this record-builder syntax in practice, here is a function which will `hit` the `dealer`'s hand by taking the current `Game` state of the `table`, and return a new `Game` state where only the `dealer` and `deck` fields have been changed:

```
hitDealer :: Game a -> IO (Game a)
hitDealer table = do
  (dealer', deck') <- hitHand (dealer table) (deck table)
  return (table{dealer = dealer', deck = deck'})
```

Exercise 2.2 (15 points)**.** Implement the function

```
dealerTurn :: Game a -> IO (Status, Game a)
```

which performs dealer's autopilot turn given their current `Hand` of cards in the `Game` state, and returns the `Status` of their final hand along with the updated `Game` state.

At each step, the next action of `dealerTurn table` depends entirely on the value (calculated from `handValue` *or* `betterHandValue`) of the `dealer`'s current `Hand` on the `table`:

- If the value of the dealer's `Hand` (in `dealer table`) is less than or equal to 16 (according to the same scoring rules used in Exercise 2.1), then `dealerTurn table` should `do` the following:
  1. hit the dealer's current hand on the table (`hitDealer table`), then

  2. run `dealerTurn` again with the updated table from step 1.

- Otherwise, immediately return the `Status` of the dealer's `Hand` along with the current `table`, unchanged. **End Exercise 2.2**

3 Talking to the User (30 points+25 extra credit)

In class, you saw methods of prompting a user for input and reading the result. The simplest one,

```
simplePrompt :: String -> IO String
simplePrompt question = do
  putStrLn question
  answer <- getLine
  return answer
```

just asks the user a given `question` and returns their `answer`, both represented as plain, unstructured, unsanitary `String`s.

To effectively respond to the user, we need to add some pre-processing that checks their answers to either put them in a more predictable form or do an action directly. Conventionally, you would have to put together *all* your pre-processing into a big switch-loop that keeps prompting for input until it gets a real answer. But we can do something better! The trick is to consider a prompt action — which takes a `String` to show the user and does some `IO` to return their answer of type `a` — as a regular value of type `Prompt a`:

```
type Prompt a = String -> IO a
```

The simple version of prompting above counts as a `Prompt` that returns a `String`, `simplePrompt :: Prompt String`. We can now build a collection of modifiers which change the way we interact with the user, letting us handle each step of pre-processing separately in a way that can be later combined into the fully-featured `Prompt` action.

For example, we might want to add a special case so that when the user enters `quit`, the program exits. This functionality can be added onto a prompt action (which returns a plain `String`) like so:

```
untilQuit :: Prompt String -> Prompt String
untilQuit prompt = \question -> do
  quitOrAnswer <- prompt question
  case quitOrAnswer of
    "quit" -> exitSuccess
    answer -> return answer
```

The `untilQuit` function takes an existing `Prompt String` value, and returns a new `Prompt String`. When this new `Prompt String` is given a `question` to present the user, `untilQuit prompt` will `do` the following:

1. Ask the given `question` via the original `prompt` action and remember the user's response—which might be any string—in the variable `quitOrAnswer`.

2. Check which `case` the response `quitOrAnswer` matches:

   - If the response is exactly the string `"quit"`, then the program immediately exits using the `exitSuccess` operation from `System.Exit`.
   - If the response is any other `answer` not matching `"quit"`, then that `answer` is returned.

3.1 Interpreting Answers (30 points)

Exercise 3.1 (10 points)**.** To be more user-friendly, the program should explain what it expects to see when the user asks for help. Implement the function

```
helpAdvice :: Prompt String -> String -> Prompt String
```

which takes an existing `Prompt String` and adds additional advice to tell the user when they input `"help"`. `helpAdvice prompt advice` should return a new `Prompt String` that, when given a `question`, will `do` the following:

1. Ask the given `question` via the original `prompt` action and remember the response.

2. Check which of the following `case`s the response from step 1 matches:

   - If it is exactly the string `"help"`, then print out the given `advice`, and then try to prompt the user again with the same helpful advice and question (`helpAdvice prompt advice question`).
   - If it is any other string, then return it without doing anything else.

End Exercise 3.1

Hint 3.1. Notice how the overall behavior for `helpAdvice prompt advice` is very similar to `untilQuit prompt` above. To get started, you can look at the code in `untilQuit` and modify the special string it looks for, and what to do when that string is found. *End Hint 3.1*

Exercise 3.2 (10 points)**.** Sometimes, running a prompt action once might fail to produce any answer, because the user entered some nonsense that couldn't be understood. In these situations, we need to re-run the same prompt action over and over until the user finally enters something sensible that we can use.

Implement the function

```
retryPrompt :: Prompt (Maybe a) -> Prompt a
```

that repeatedly re-runs the given `Prompt (Maybe a)` action until it finally returns `Just` a real answer, that can be returned. `retryPrompt prompt` should return a new `Prompt a` that, when given a `question`, will `do` the following:

1. Ask the given `question` via the original `prompt` action and remember the response.

2. Check which of the following `case`s the response from step 1 matches:

   - If it is `Just answer`, for any `answer` of type `a`, then `return` the plain `answer` as-is.
   - If it is `Nothing`, then print out the message telling the user they weren't understood (`"Sorry, I didn't understand that."`), and retry the same thing again (`retryPrompt prompt question`).

End Exercise 3.2

Hint 3.2. Notice how `retryPrompt` is similar to both `helpAdvice prompt advice` and `untilQuit prompt` above. Now, the difference is that instead of looking for a special string, we are looking for either `Just answer` (in order to `return answer`) versus `Nothing` (which triggers the re-run similar to `helpAdvice`'s response to `"help"`). *End Hint 3.2*

Exercise 3.3 (10 points)**.** Sometimes we need to change the answer typed by the user into some other form, like converting a string of digits into a number.

Implement the function

```
changeAnswerBy :: Prompt a -> (a -> b) -> Prompt b
```

that modifies the answer of a given `Prompt a` by some function `a -> b`. Calling `changeAnswerBy prompt change` should return a new `Prompt b` that, when given a `question`, will `do` the following:

1. Ask the given `question` via the original `prompt` action and remember the response (of type `a`).

2. `return` a result of type `b` using the `change` function on `answer`.

End Exercise 3.3

Hint 3.3. Inside of `changeAnswerBy prompt change`, you can use `change` just like any other function for converting `a`s into `b`s, so that `return (change answer)` will return a result of type `b` assuming that `answer` has type `a`. *End Hint 3.3*

Hint 3.4. The generic types (`a` and `b`) are your friend! As long as you run `prompt question` *exactly* one time and fill in all `undefined`s, there is only one possible thing that `changeAnswerBy prompt change question` can do which passes the type checker. This also happens to conveniently be the right answer to the exercise. *End Hint 3.4*

3.2 *Bonus:* Sanitization (25 extra credit)

Bonus Exercise 3.4 (5 extra credit)**.** Sometimes users type responses with small variations that can be easily corrected, like capitalization. Instead of an expected response like `"stand"`, the user might type `"Stand"` or `"STAND"` or `"sTaNd"`. But if we ignore capitalization, these are all the same.

Implement the function

```
makeAllLowercase :: String -> String
```

which transforms a string by replacing every uppercase letter with its lowercase equivalent. **End Bonus 3.4**

Hint 3.5*.* The `toLower :: Char -> Char` function from the `Data.Char` module takes a single character and will either

- return the corresponding lowercase letter when given `'A'` to `'Z'`, or
- return the same character given if it is not an uppercase letter. *End Hint 3.5*

Bonus Exercise 3.5 (15 extra credit)**.** Another innocent variation in user input is extra stray space characters before or after the real response. Instead of `"stand"`, the user might accidentally type a leading space `" stand"` or trailing spaces like `"stand    "` which should be ignored.

Implement the function

```
trimLeadingSpaces :: String -> String
```

which removes all consecutive space characters at the very *beginning* of a `String` and leave the rest alone. For example, `trimLeadingSpaces "    abc"` should be trimmed to `"    abc"`, and `trimLeadingSpaces "abc   "` should be the same `"abc  "`.

Implement the functions

```
seekNonSpace       :: String -> Maybe (String, Char, String)
trimTrailingSpaces :: String -> String
```

`trimTrailingSpaces` should remove all consecutive space characters at the very *end* of a `String`, and leave the rest alone. For example, `trimTrailingSpaces "abc    "` should be `"abc"`, while `trimTrailingSpaces "   abc"` should be the same `"   abc"`.

To help with implementing `trimTrailingSpaces`, the `seekNonSpace` helper function should look for the first *non-space* character in a string.

- In the case where `str` has *any* non-space character in it, `seekNonSpace str` should return `Just (spaces, non, rest)` where `non` is the first non-space character in `str`, `spaces` is the string of 0 or more spaces found before `non`, and `rest` is the remaining string left after `non`.
- In the case where `str` has *only* space characters in it, `seekNonSpace str` should return `Nothing`.

For example, since the empty string `""` doesn't contain a non-space character, `seekNonSpace ""` should return `Nothing`. Additionally, given a non-space character `'c'`, a sequence of (for example) 3 spaces `"␣␣␣"`, and *any* string `str`, calling `seekNonSpace("␣␣␣"++['c']++str)` should return `Just("␣␣␣",c,str)` where the first component is the same 3 spaces. **End Bonus 3.5**

Bonus Exercise 3.6 (5 extra credit)**.** Implement the function

```
sanitizeAnswer :: Prompt String -> Prompt String
```

which takes a `Prompt String` and a returns a new `Prompt String` which sanitizes the user's answer by returning a `String` where

- all uppercase letters are converted to lowercase, and
- all leading and trailing space characters have been removed. **End Bonus 3.6**

Hint 3.6*.* Can you re-use `changeAnswerBy` to automatically apply the string-processing functions in Bonus Exercises 3.4 and 3.5 to the answer of the prompt? *End Hint 3.6*

4 The Player's Turn (25 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 4.1 (5 points)**.** 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`.

This function is used to build a `Prompt Move` which interprets the user's choice of action:

```
promptMove :: Prompt Move
promptMove = untilQuit simplePrompt
             `helpAdvice` "You can \"hit\" or \"stand\""
             `parseAnswerAs` parseMove
```

End Exercise 4.1

Exercise 4.2 (10 points)**.** First, define a function

```
hitPlayer :: Game a -> IO (Game a)
```

that will hit the player's hand by drawing a card from the `deck` in the given `Game a` and moving it to the `player` hand.

Next, define a function

```
playerMove :: Move -> Game a -> IO (Maybe (Game a))
```

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

- Given `Stand` as the first parameter, `playerMove` should `return Nothing`.
- Given `Hit` and `table` as parameters, `playerMove` should
  1. `hitPlayer` on the current `table`,
  2. `return Just` the updated game state from step 1. **End Exercise 4.2**

Hint 4.1. For comparison, look at the definition of `hitDealer` in section 2. *End Hint 4.1*

Hint 4.2. 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. *End Hint 4.2*

To actually talk to the user to get their decision, we can use the `ask` field of an appropriate `\Game` state. This `askPlayer` function will show the player their current hand to ask what they want to do, and then return their decision as a `Move` value:

```
askPlayer :: Game (Prompt Move) -> IO Move
askPlayer table = do
  let query = "Your hand is "
              ++ prettyPrint (player table)
              ++ ", what do you do?"
  ask table query
```

`askPlayer table` avoids all the details of interpreting the user's response because the `table`'s type `Game (Prompt Move)` forces the `ask` field to be a `Prompt Move`. That means, every time we feed `ask table` a question, we always get some `Move` in response which must be either `Hit` or `Stand`. The details on what happens if the user types something else must already be handled internally inside `ask table`.

Exercise 4.3 (10 points)**.** Finally, implement the player's full turn in the function

```
playerTurn :: Game (Prompt Move) -> IO (Status, Game (Prompt Move))
```

that asks the player for their next move given the current game state, which includes the current `player` hand. `playerTurn table` should `do` the following:

1. Ask the player on the table for their next move (`askPlayer table`).
2. perform the `playerMove` (that you already defined above) with their `Move` selection from step 1 the current `table`.

3. Check whether anything changed from step 2:

    - In the `case` where step 2 returns `Nothing`, then return the `Status` of the `player`'s `Hand` (via Exercise 2.1) and the originally given `table`.
    - In the `case` where step 2 returns `Just` a new `Game` state, then check the `Status` of `player`'s new `Hand`:
        - If the `player`'s new `Hand` is `OK`, then continue the `playerTurn` again with the new `Game` state from step 2.
        - If the `player`'s new `Hand` is `Bust`, then `return` the `Bust` status with the updated `Game` state. **End Exercise 4.3**

5 Putting It All Together (25 points + 10 extra credit)

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 5.1 (10 points)**.** Implement a function

```
dealHands :: Game a -> IO (Game a)
```

that deals an initial 2-card `Hand` to both the `player` and `dealer` from the top of the `deck`. **End Exercise 5.1**

Hint 5.1. Remember to maintain good `Deck` management, to make sure cards aren't accidentally duplicated or dropped from the top of the `deck`! *End Hint 5.1*

Exercise 5.2 (10 points)**.** Define the function `compareHands`

```
data Result = Lose | Tie | Win

compareHands :: Game a -> Result
```

which returns the `Result` of a round based on the `player` and `dealer` hand (from the point of view of the `player`). So `compareHands` returns a `Win` if the `player` has the winning `Hand`, `Lose` if the `dealer` has the winning `Hand`, and a `Tie` otherwise.

Note that a `Bust` hand (according to `checkHand`) cannot ever `Win`, so you must take this into account before comparing the values of the two hands. In the case that someone has gone `Bust`, `compareHands` should return:

- `Win` if the `dealer` is `Bust`,
- `Lose` if the `player` is `Bust`,
- and `Tie` if they are both `Bust`.

Otherwise, in the case that both `Hand`s are `OK`, `compareHands` should return

- `Win` if the `player`'s `Hand` has a higher value,
- `Lose` if the `dealer`'s `Hand` has a higher value, or
- `Tie` if both `Hand`s have equal values,

using the same hand scoring method as `compareHands`. **End Exercise 5.2**

Exercise 5.3 (5 points)**.** Now, you can implement a full round of blackjack in the function

```
playRound :: Game (Prompt Move) -> IO (Result, Game (Prompt Move))
```

that takes the current state of the `Game` and returns the `Result` of the round (`Win`, `Lose`, `Tie`) along with the updated game state. `playRound` should `do` the following:

1. Run the `playerTurn` on the initial `Game` state to get the `Status` of the `player`'s final `Hand` and an updated `Game` state.
2. Run the `dealerTurn` on the `Game` state from step 2 to get the `Status` of the `dealer`'s final `Hand` and another updated `Game` state.
3. `return` the `Result` of the round from `compareHands` of the final `Game` state from step 3, along with that `Game` state.

End Exercise 5.3

The main loop of the game is given to you in the function

```
gameLoop :: Game (Prompt Move) -> IO a
```

that takes the current `Game` state, and then

1. asks if the player is ready, and waits for them to press enter,
2. deals both the `dealer` and `player` 2 cards,
3. reveals the dealer's first card to the user,
4. plays a round of blackjack (`playRound`),
5. prints out a message revealing the final `Hand`s along with the `Result` of the round,
6. and finally, restarts the `gameLoop` with the updated `Game` state.

This `gameLoop` is used by the `main` entry point to the program, which prints a nice welcome message, and then starts the loop with an empty `Table`.

Bonus Exercise 5.4 (5 extra credit)**.** It can happen that both the player and the dealer would go bust in the same round. Standard rules for blackjack always have the player go first, and the dealer second. To give an (unfair) advantage to the dealer, the round is stopped once the player goes bust, immediately giving the dealer the win even if they would also go bust n the same round.

Implement this dealer advantage by updating your `playRound` function from Exercise 5.3 to check the `Status` returned by both `playerTurn` and `dealerTurn`:

1. If `playerTurn` returns `Bust`, then print a message telling the user that `"You are bust!"`. Then immediately return a `Lose` and the state of the `Game` right at the end of the `playerTurn`, skipping the `dealerTurn`.
2. Otherwise, `playerTurn` returns an `OK Hand`. In this case, continue on to the `dealerTurn` as usual.
   (a) If `dealerTurn` returns `Bust`, then print a similar message telling the user that `"The dealer is bust."` and return `Win` and the state of the `Game` at the end of the `dealerTurn`.
   (b) Otherwise, both have an `OK Hand`. In this case, return the `Result` of comparing the two `Hands` as usual.

End Bonus 5.4

Bonus Exercise 5.5 (5 extra credit)**.** The victor proclaimed by `compareHands` in Exercise 5.3 is a simplification of the full game rules. The real game has the notion of a natural *blackjack*, which is hand of exactly 2 cards with a total value of 21. A natural 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.

Because everyone starts with a 2-card hand, the only way to get a natural blackjack is if those initial 2 cards add up to 21. So the check for natural blackjacks only need to happen at the very beginning of a round, before anyone has drawn any cards. If neither player is dealt a natural blackjack, they can never draw into one. If either player starts a round with a natural blackjack, then it will beat anything the other player could get, so the player' and dealer's turns are skipped since it is pointless to play the round out.

To check for natural blackjacks, first implement the functions

```
isBlackjack    :: Hand -> Bool
blackjackCheck :: Game a -> Maybe Result
```

`isBlackjack` determines if a single `Hand` counts as a natural blackjack. `blackjackCheck` looks at the `Game` state to see if the `Result` of the round can already be determined from the opening hands. `blackjackCheck` returns

- `Just Win` if only the `player` has a natural blackjack,

- `Just Lose` if only the `dealer` has a natural blackjack,
- `Just Tie` in the unlikely event that *both* opening hands are natural blackjacks, and
- `Nothing` if neither are natural blackjacks.

Next, add one extra step to the start of `playRound` that does a `blackjackCheck` on the starting `Game` state. If it returns `Nothing`, the round proceeds as before. But if it returns `Just` a result, then `playRound` should announce a `"Blackjack!"` and return that result. **End Bonus 5.5**