Effective Functional Programming
Pure Functional Programming
Assignment 1
`Card Games`

Paul Downen

In these exercises, you will learn how to use functional programming to model the basic parts of a blackjack game.

1 Playing Cards (40 points + 20 extra credit)

Blackjack is a game that uses a standard deck of playing cards. Every playing card has one of four *suits*: hearts (♡), diamonds (♢), clubs (♣), or spades (♠). In addition, each card comes in one of the following *ranks*:

- An *ace* (A).
- A numeric card, which has a number between 2 and 10 (inclusive).
- A *face* card, which has the face of one of the three royalty: king (K), queen (Q), or jack (J).

Some example playing cards are the ace of spades (A♠), the five of clubs (5♣), and the jack of hearts (J♡).

Exercise 1.1 (10 points)**.** We can model playing cards using data types in Haskell. For example, a data type that enumerates all the faces of the `Royalty` is defined as

```
data Royalty = Jack | Queen | King
```

Define some data types for representing the rest of the playing cards:

1. Define a data type `Suit` that enumerates the four possible suits (`Hearts`, `Diamonds`, `Clubs`, `Spades`).
2. Using `Suit` and `Royalty`, define a data type `Card` with *three* different constructors for building individual `Card`s, one for each of the above three varieties:
   (a) `Ace` cards have a `Suit`

(b) `Numeric` cards have a specific number (an `Int` between `2` and `10`) as well as a `Suit`, and

(c) `Face` cards have specific `Royalty` (signifying either a `Jack`, `Queen`, or `King`) as well as a `Suit`.

The associated test suite needs to know how to build values of your `Suit` and `Card` data types to check your answers to the following exercises. So let's define some helper functions to do so!

First, define four different top-level constants (`hearts`, `diamonds`, `clubs`, `spades`) that stand for each one of your possible `Suit` values:

```
hearts, diamonds, clubs, spades :: Suit
```

Second, define three functions that can build each type of `Card` value:

```
ace     ::            Suit -> Card
numeric :: Int     -> Suit -> Card
face    :: Royalty -> Suit -> Card
```

End Exercise 1.1

Bonus Exercise 1.2 (5 extra credit)**.** Constructors of data types take arguments of specific types, which limits what contents they can hold. For example, if you say that your `Ace` constructor takes a `Suit` as an argument, then the type checker will prevent it taking anything else: rest assured that the contents of an `Ace Card` will *never* hold a `String` or an `Int` or a `Royal`.

However, sometimes this type-based discipline is not enough to express all the restrictions we have in mind for well-formed values. For example, `Numeric` cards (which take an `Int`) should only ever contain a number value between `2` and `10`, but the type checker will allow `Numeric` to take *any* `Int`. As a result, there may be some `Numeric Card` values that pass the type-checker, but still don't make logical sense in our application.

One solution to this problem is to define a *smart constructor* which adds additional run-time checks or operations to prohibit nonsensical values from being built. For extra credit, make the `numeric` function from Exercise 1.1 into a smart constructor by adding a check to make sure that the given number is one of the possible card values:

- `numeric n suit` should return a normal `Numeric Card` containing `n` and `suit` when `n` is between `2` and `10` (inclusive), otherwise
- `numeric n suit` should raise an error when `n` is smaller than `2` or bigger than `10`.

Hint 1.1. The function

```
error :: String -> a
```

lets you report an error that should never happen at run-time. `error message` will immediately stop the program, and print the `String message` to (hopefully) help explain what went wrong. The generic return type `a` means that `error message` can be used in place of *any* expression, no matter what type of value its supposed to return, because `error message` *never* returns. For example, in the case of `numeric -10 diamonds`, you can use `error message` to stop the program in place of returning an invalid `Card`. *End Hint 1.1*

End Bonus 1.2

Exercise 1.3 (5 points)**.** Derive an `Eq` instance for your `Suit`, `Royalty`, and `Card` data types using the automated deriving mechanism by adding a `deriving` clause to these data type definitions. Likewise, automatically derive an `Enum` instance for `Suit` and `Royalty` by adding the `Enum` to the list of type classes in the `deriving` clause of the `Suit` and `Royalty` data type definitions.

The `Enum` type class informs Haskell how to enumerate through the values of a type, as used in a list enumeration like `[1..10]`. **End Exercise 1.3**

Exercise 1.4 (5 points)**.** A hand of cards can be represented as a list of `Card`s. We can give a more informative name to this type by defining this `type` alias

```
type Hand = [Card]
```

Define another type synonym named `Deck` that is also an aliases for a list of `Card`s. **End Exercise 1.4**

Exercise 1.5 (10 points)**.** The `Show` type class informs Haskell how to render the values of a type as `String`s using the associated function

```
show :: Show a => a -> String
```

We can `show` the values of a particular type any way we want by manually define a custom `instance` of the `Show` type class. For example, here is a declaration of `Show Royalty` with the following `instance` definition:

```
instance Show Royalty where
  -- show :: Royalty -> String
  show Jack  = "J"
  show Queen = "Q"
  show King  = "K"
```

Manually define similar `Show` instances for the `Suit` and `Card` types. First, define a function

```
asciiSuit :: Suit -> String
```

that displays a single letter for each `Suit`:

- `asciiSuit Hearts` should be `"H"`
- `asciiSuit Diamonds` should be `"D"`

- `asciiSuit Clubs` should be `"C"`
- `asciiSuit Spades` should be `"S"`

We can then tell Haskell to use this function whenever a `Suit` is printed by:

```
instance Show Suit where
  -- show :: Suit -> String
  show = asciiSuit
```

Second, define an instance for `Show Card` which defines how to convert each card into a `String` displaying its rank and suit. The string representations of the three different ranks should be:

- `Ace suit = "A"` immediately followed by the `shown suit`.
- `Numeric n suit =` the string representation of the `Int n` immediately followed by the `shown suit`.
- `Face royal suit =` the string representation of the `Royalty royal` immediately followed by the `shown suit`.

For example, a 10 of spades should be `shown` as `"10S"` and a Queen of diamonds should be `shown` as `"QD"`. **End Exercise 1.5**

Bonus Exercise 1.6 (5 extra credit)**.** The unicode codes for characters corresponding to the card suits are (in hexadecimal):

- ♡: 2661
- ♢: 2662
- ♣: 2663
- ♠: 2660

Hexadecimal unicode codes can be used in Haskell strings by escaping them with \x$n$, where n is the code. For example, the string `"\x2660"` corresponds to `"♠"`.

For extra credit, define a function

```
unicodeSuit :: Suit -> String
```

that returns the appropriate unicode suit symbol above for each suit value.

Feel free to change the `Show Suit` instance to use `unicodeSuit` instead of `asciiSuit` so that the 10 of spades is `shown` as `"10♠"` and the Queen of diamonds is shown `shown` as `"Q♢"`. **End Bonus 1.6**

Exercise 1.7 (10 points)**.** Sometimes, it's handy to have a complete list of all the parts of the playing cards for use in other definitions. Since we already know how to `Enum`erate the faces of `Royalty` (from Exercise 1.3), we can list them all like so:

```
royals :: [Royalty]
royals = [Jack ..]
```

The `royals` uses the enumeration `Jack ..` to all three faces, starting with `Jack` (the first constructor defined in the `Royalty` enumeration), continuing with `Queen` (the next constructor of `Royalty`) and ending with `King` (the final constructor.

Define similar lists

```
suits   :: [Suit]
numbers :: [Int]
```

containing all the `Suit`s, and all the valid `Numeric` card values between `2` and `10`. Use these lists to create a full deck

```
fullDeck :: Deck
```

containing *all* playing cards: an `Ace` for every `Suit`, a `Numeric` card for every `Suit` and number between `2` and `10`, and a `Face` card for every `Suit` and `Royalty`. The order of `fullDeck` does not matter. **End Exercise 1.7**

Bonus Exercise 1.8 (10 extra credit)**.** Sometimes, a dealer will play with multiple decks mixed together, to make it more difficult to count cards. Define a function that returns the combination of multiple decks mixed together (in any order)

```
multiDeck :: Int -> Deck
```

so that `multiDeck 0` is empty, `multiDeck 1` is `fullDeck`, `multiDeck 2` is a combination of two `fullDeck`s, and so on. **End Bonus 1.8**

2 Blackjack Scoring (20 points + 40 extra credit)

2.1 *Regular:* Simple Scoring (20 points)

In the game of blackjack, each player's hand is given a numeric score and the goal is draw cards and achieve the highest score without going over 21. A hand with a score over 21 is called a "bust", and is an automatic loss. Otherwise, when comparing two non-bust hands, the hand with the higher score wins.

Exercise 2.1 (10 points)**.** Simplifying the rules of blackjack, each card can be assigned the following numeric score value based on its rank:

- `Ace` = 11
- a `Numeric` card n = the same n as its number
- any `Face` card = 10

A card's suit does not affect its score. For example, both the 7 of spades (7♠) and 7 of diamonds (7♢) have a score of 7. The Queen of hearts (Q♡) and Queen of clubs (Q♣) have the same score 10, as does the Jack of spades (J♠).

Define a function

```
cardValue :: Card -> Int
```

for calculating the numeric score value of a `Card` according to the above rules. **End Exercise 2.1**

Hint 2.1. The `cardValue` function can pattern match on the shape of the `Card` parameter it is given as an input, inspecting the constructor that was used to build it. Since there are three different constructors of `Card` values, you can define `cardValue` in three different lines: one for `Ace`s, one for `Numeric` cards, and one for `Face` cards. *End Hint 2.1*

Exercise 2.2 (10 points)**.** Define a function

```
handValue :: Hand -> Int
```

for calculating the total score of a `Hand` by summing up the value of each card in the hand. For example, the value of an empty hand is `0`, the value of a hand with exactly one card `c` is `cardValue c`, the value of a hand with two cards `c` and `d` is `cardValue c + cardValue d` and so on. **End Exercise 2.2**

Hint 2.2. Remember, `handValue` is given a list as an argument, and you can always pattern-match on a list argument to answer the two main cases of the function:

1. What happens if `handValue` is given an empty list `[]`?
2. What happens if `handValue` is given a non-empty list `(c:h)` containing a `Card c` and the rest of the `Hand h`?

Alternatively, consider these functions from the standard library:

```
map :: (a -> b) -> [a] -> [b]
sum :: [Int] -> Int
```

Can you somehow put these together with your answer in Exercise 2.1 to calculate the score of an entire `Hand`? *End Hint 2.2*

2.2 *Bonus:* Soft Aces (40 extra credit)

Section 2.2 is entirely optional, and contains an opportunity to earn some extra credit by going above and beyond the expectations for this assignment. Consider these bonus exercises only after successfully completing the rest of the assignment.

In the full rules of blackjack, some scores are "soft," meaning that they can be lowered to avoid a bust. In particular, an ace is valued at *either* 11 or 1, depending on which results in the better, non-bust score. A score which cannot be lowered any more is "hard." For example, the hand A♠ 4♡ has the soft score of 15 by valuing the ace as 11. Drawing an additional card to get the hand 7♣ A♠ 4♡ has the hard score of 12 by valuing the ace as 1, since valuing the ace as 11 would lead to the bust score 22.

Bonus Exercise 2.3 (10 extra credit)**.** Define a `Score` data type that keeps separate track of a score's soft parts (contributed by soft aces, which can be removed) and hard parts (which are mandatory and cannot be removed).

Define the conversion functions

```
cardScore  :: Card  -> Score
scoreValue :: Score -> Int
```

`cardScore` calculates the `Score` for a single `Card` and `scoreValue` calculates the total numeric value (including both the soft and hard parts) of a `Score`. Composing these two functions should always do the same thing as `cardValue` from Exercise 2.1:

```
scoreValue (cardScore card) == cardValue card
```

Define the two checking functions

```
isSoft, isHard :: Score -> Bool
```

`isSoft` takes a `Score` and returns `True` when given `Score` has any soft portion to it. `isHard score` is the opposite (`isHard score == not (isSoft score)` and vice versa), and returns `True` when `score` is completely hard. For example, the `Score` from any ace card is always soft

```
isSoft (cardScore (Ace suit)) == True
```

In contrast, the `Score` from any face or numeric card is always hard

```
isHard (cardScore (Face royal suit)) == True
isHard (cardScore (Numeric n suit))  == True
```

End Bonus 2.3

Bonus Exercise 2.4 (10 extra credit)**.** The Haskell type classes `Semigroup` and `Monoid` defined as

```
class Semigroup a where
  (<>) :: a -> a -> a

class Semigroup a => Monoid a where
  mempty  :: a

  mconcat :: [a] -> a
  mconcat []     = mempty
  mconcat (x:xs) = x <> mconcat xs
```

describe an interface for types with values can be summed together. The binary operator `x <> y` represents combining two values of the type `a`, and `mempty` represents the neutral value that does not change the total. The additional derived method `mconcat` shows how to sum up any list of values via the `Monoid`

interface.[1] Since `mconcat` has a default definition given inside the `Monoid` class, you do not need to define it in a particular instance; leaving out definition of `mconcat` will result in using the default definition given above.

An example instance of `Monoid` is lists, where the neutral element `mempty` is the empty list and the binary operator `(<>)` is list append as follows:

```
instance Semigroup [a] where
  -- (<>) :: [a] -> [a] -> [a]
  xs <> ys = xs ++ ys

instance Monoid [a] where
  -- mempty :: [a]
  mempty = []
```

Other examples of `Semigroup` and `Monoid` instances are numeric sums (where `mempty` is 0 and `(<>)` is addition) and products (where `mempty` is 1 and `(<>)` is multiplication). Many more instances can be found in the `Data.Semigroup` and `Data.Monoid` module from the standard library.

Define a `Semigroup` and `Monoid` instance for `Score` by implementing `(<>)` and `mempty` so they can be automatically summed together with `mconcat`. In your definition, the `scoreValue` of `mempty` should be `0`, and the `scoreValue` of two combined scores `x <> y` should be `scoreValue x + scoreValue y`.

As a guide, your `Semigroup Score` and `Monoid Score` instances should obey the following equalities

```
mempty <> x   == x
x <> mempty   == x
(x <> y) <> z == x <> (y <> z)
```

for any `Score`s `x`, `y`, and `z`. In addition, softness should be the dominant trait when scores are combined. That is to say:

- `mempty` should be a hard score,
- `x <> y` should be hard whenever *both of* `x` and `y` are hard, and
- `x <> y` should be soft whenever *one of* `x` or `y` are soft.

End Bonus 2.4

Bonus Exercise 2.5 (5 extra credit)**.** Define the function

```
handScore :: Hand -> Score
```

for calculating the `Score` of an individual `Card` and the total `Score` of a `Hand`. Note that the score of an ace should include both a soft part (contributing a value of 10) and a hard part (contributing a value of 1), whereas all other cards only have a hard score value.

End Bonus 2.5

[1]The standard library definition of `Monoid` also gives a derived method `mappend` for backwards compatibility, which is just another name for the `(<>)` operator by default.

Bonus Exercise 2.6 (5 extra credit)**.** We need to be able to compare two scores, to determine the winner of a game. In the normal case, when both players have a numeric score of 21 or less, we can just compare these two numbers to see who wins. However, a bust score whose value is over 21 counts as an automatic loss, so it should be considered "less" than a score like 18, even though the numeric value is technically larger.

First, define a function

```
isBust :: Score -> Bool
```

which checks if a score's value is 22 or greater.

Next, define `Eq` and `Ord` instances for `Score`, taking care to account properly for any bust score. Your definition for `compare` should say that:

- any two bust scores are equal (meaning they are tied),
- a bust score is less than (meaning it loses to) any other non-bust score,
- any two non-bust scores are compared on their numeric value.

Your definition for `(==)` should return `True` exactly when `compare` says that the two scores have an `EQ Ordering`.

Hint 2.3. See Exercise 3.2 for more information about how the `Ord` and how to implement its associated `compare` function. *End Hint 2.3*

Hint 2.4. Even though `Ord a` requires `Eq a`, you may define `compare` from `Ord` first and then refer to it inside the definition of `(==)` from `Eq`. *End Hint 2.4*

End Bonus 2.6

Bonus Exercise 2.7 (10 extra credit)**.** Define a function

```
improveScore :: Score -> Score
```

that "improves" a `Score` by lowering/eliminating soft scores (from soft aces) that lead to a bust. `improveScore` should choose among all the possibilities with or without the available soft scores and return the "best" score whose value is closest to 21 without going over. This best `Score` is either the one with the highest value (according to `scoreValue`) that is less than or equal to 21 if possible, or otherwise the one with the lowest possible score greater than 21.

Calculating the "best" improved score can be a tricky operation to get right. To help you out, here are some properties of how `improveScore` should behave:

- A score of 21 or under and a completely hard score cannot be improved.
- A score that `isSoft` and `isBust` with a value of 22 or more can be improved by lowering the value by a multiple of 10. Doing so will either make the score better (by making it no longer a bust), or will completely harden it (to the lowest possible option over 21).
- Improvement will never change a score more than once. In other words, it is *idempotent*: `improveScore (improveScore x) == improveScore x`.

- If adding an 11-valued ace to a score would result in a value of 21 or less, then the soft 11 is added. Otherwise, adding an ace to a score already valued at 11 or more only adds a hard 1 instead.
- In more general terms, combining any two already-improved scores — one hard and one soft – and improving the sum will either do nothing, or reduce it by 10. Given *any* two already-improved scores, improving their combination will only ever reduce the total value by 0, 10, or 20.

Define a function

```
betterhandValue :: Hand -> Int
```

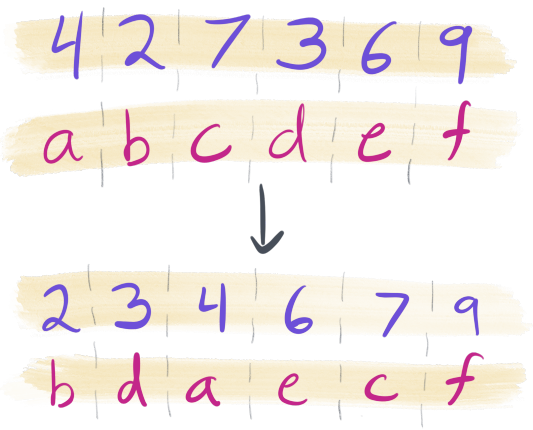
which calculates a (potentially) better value for a hand by

1. calculating the `Score` of the hand,
2. improving that `Score` once, and
3. returning the numeric value of the improved `Score`.

End Bonus 2.7

3 Deck Shuffling (40 points)

Of course, before playing a game, the deck of cards should be shuffled. One straightforward way to shuffle a deck is to sort the deck based on a random ordering. For example, consider the following illustration wherein two lists are rearranged pairwise by sorting:



At first, the purple list of numbers is in some "random" order, whereas the red list of letters is in order. After sorting the two lists pairwise, the purple numbers are put into ascending order, which forces the red letters into a "random" order.

Exercise 3.1 (5 points)**.** Define a polymorphic data type `Indexed i a` which a single constructor that pairs together an item `a` with an index `i`. Derive a `Show` type class instance for this data type using `deriving`. **End Exercise 3.1**

Exercise 3.2 (10 points)**.** The `Ord a` type class specifies how values of a type `a` can be ordered relative to one another. `Ord a` includes many ordering operations (<, >, *etc.*) that are all derivable from the `compare` function

```
compare :: Ord a => a -> a -> Ordering
```

The `Ordering` type is an enumeration of the values `LT` (for "less than"), `EQ` (for "equal"), and `GT` (for "greater than") defined in the standard library as:

```
data Ordering = LT | EQ | GT
```

To define an instance of `Ord`, only the `compare` function needs to be implemented.

Manually define an `Ord (Indexed i a)` which depends on `Ord i` by implementing the `compare` function for `Indexed i a`. Only the index part (of type `i`) of the `Indexed i a` value should be considered for the purposes of `compare`ison, and the item part (of type `a`) should be completely ignored. Additionally, define an `Eq (Indexed i a)` instance by implementing the `(==)` function in a way that similarly only compares indexes for equality and ignores the item.

End Exercise 3.2

Exercise 3.3 (10 points)**.** Define the two functions:

```
addIndexes    :: [i] -> [a] -> [Indexed i a]
removeIndexes :: [Indexed i a] -> [a]
```

`addIndexes` should return a list of type `[Indexed i a]` by combining (pairwise) the elements of the given list of indices (of type `[i]`) and list of values (of type `[a]`). For example, `addIndexes [3,2,1] "abc"` should return a list where `'a'` is indexed at `3`, `'b'` is indexed at `2`, and `'c'` is indexed at `1`. Note, if the two lists passed to `addIndexes` have different lengths, you should drop the extra elements from the end of the longer list.

`removeIndexes` should return a list of type `[a]` by extracting only the values of type `a` from the given list of indexed values (of type `[Indexed i a]`). The indices of type `i` from the input should be discarded. **End Exercise 3.3**

Hint 3.1. These two functions which are automatically imported by default:

```
zip :: [a] -> [b] -> [(a, b)]
map :: (a -> b) -> [a] -> [b]
```

The `zip` function pairwise combines two lists, and returns the list of pairs. The `map` function transforms a list of `a`s to a list of `b`s by applying the given function (`a -> b`) to every element of the given list (`[a]`).

Can you think of a way to somehow use these functions to make writing `addIndexes` and `removeIndexes` easier? *End Hint 3.1*

Exercise 3.4 (15 points)**.** The above shuffling process can be broken down in these three steps:

1. combine together each `Int` and `a`, pairwise, from the two input lists to produce a list of type `[Indexed Int a]`,
2. sort the list of type `[Indexed Int a]` (according to the definition of ordering given by `Ord (Indexed Int a)` from Exercise 3.2) from step 1, and
3. return a list of `a`s obtained from the items of the list from step 2.

Now, define a shuffling function

```
shuffle :: [Int] -> [a] -> [a]
```

that permutes the given list `[a]` according to the "random" list of indexes `[Int]`. For example,

```
shuffle [4, 2, 7, 3, 6, 9] "abcdef" == "bdaecf"
```

Hint 3.2. The list sorting function

```
sort :: Ord a => [a] -> [a]
```

can be found in the standard Data.List module included with GHC. Can you combine `sort` with your answers to Exercise 3.3 to define the `shuffle` function?

End Hint 3.2

End Exercise 3.4