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)

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 numbered 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)**.** Define some data types for representing playing cards. First, define a data type `Suit` that enumerates the four possible suits, and a data type `Royalty` that enumerates the three possible faces of royalty. Using `Suit` and `Royalty`, define a data type `Card` that represents each of the above three varieties (aces, numbers, and faces) of playing cards. In the case of a numbered card, the number can be represented as an `Int`.

Exercise 1.2 (5 points)**.** Both a deck of cards and a hand of cards can be represented as a list of cards. Define type synonyms `Deck` and `Hand` for lists of cards.

Exercise 1.3 (5 points)**.** Derive an `Eq` instance for `Suit`, `Royalty`, and `Card` using the automated `deriving` mechanism. Likewise, derive an `Enum` instance for `Suit` and `Royalty`.

Exercise 1.4 (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
```

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

Manually define `Show` instances for `Suit`, `Royalty` and `Card`. `show`ing a suit and royalty value should return the following string representations:

- heart: ♡
- diamond: ♢
- club: ♣
- spade: ♠
- king: K
- queen: Q
- jack: J

`show`ing a card should display a string signifying its rank immediately followed by the string signifying its suit. The string representations of ranks should be:

- ace: A
- number n: the string representation of the `Int` n
- face r: the string representation of the `Royalty` r

For example, a 10 of spades should be `show`n as `"10♠"` and a queen of diamonds should be `show`n as `"Q♢"`.

Hint 1.1. 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 `"♠"`.

Exercise 1.5 (10 points)**.** Define the lists `suits::[Suit]`, `faces::[Royalty]`, and `numbers::[Int]` containing all suit, face, and valid numeric card values. 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 face. The order of `fullDeck` does not matter.

2 Blackjack Scoring (20 points)

You only need to complete one set of exercises from section 2.1 OR section 2.2. Earning 20 points by successfully completing all of section 2.1 counts as a 100% grade for this section. Alternatively, earning all 35 points possible in section 2.2 will result in 15 extra credit points in addition to the 100% grade for this section. When in doubt, it is better to 100% complete section 2.1 than to half-complete the exercises in section 2.2. At the end, the goal is to have a functioning implementation of `handValue` using either the simplified or full scoring rules.

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" is an automatic loss. Otherwise, when comparing two non-busted, 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
- any face card: 10
- a numeric card n: the same n as its number

A card's suit does not affect its score. Define a function

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

for calculating the numeric score value of a `Card` according to the above simplified rules.

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.

2.2 *Alternate:* Soft Aces (35 points)

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-busted 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 busted score 22.

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

```
scoreValue :: Score -> Int
```

that calculates the total numeric value (including both the soft and hard parts) of the `Score`.

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.

Bonus Exercise 2.4 (5 extra credit)**.** Also define a function

```
scoreValues :: Score -> [Int]
```

that calculates *all* possible numeric values with each soft portion counted or ignored. For example, the singular `scoreValue` will calculate a single total score assuming that every soft ace is counted as 11 points, whereas the plural `scoreValues` calculates several totals considering the case when each soft ace is counted as 1 or 11.

Hint 2.1. Consider a hand of 2 aces. Its score consists of a hard portion of 2 (each ace must count for at least 1 point) and two soft 10s (one for each ace). There are two different ways of thinking about the two-ace hand, depending on whether or not the different aces are considered distinct:

- With indistinct aces, there are three scores:
    1. 2 when both aces count as 1,
    2. 12 when one ace counts as 1 and the other as 11, and
    3. 22 when both aces count as 11.
- With distinct aces, there are four scores:
    1. 2 when the first and second aces both count as 1,
    2. 12 when the first ace counts as 1 and the second counts as 11,
    3. 12 when the first ace counts as 11 and the second counts as 1, and
    4. 22 with the first and second aces both count as 11.

Either counting scheme above is acceptable for this exercise.

Exercise 2.5 (10 points)**.** 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 Semnigroup [a] where
  xs <> ys = xs ++ ys

instance Monoid [a] where
  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`. 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`.

Exercise 2.6 (10 points)**.** Define the functions

[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.

```
cardScore :: Card -> Score
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.

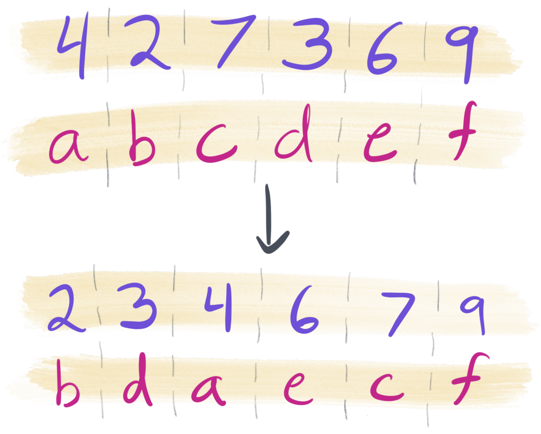
Define a function

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

that calculates the value of a hand by first calculating its (potentially soft) `Score`, improving that `Score` to avoid a bust when possible, and then calculating the numeric value of the improved `Score`.

3 Deck Shuffling (30 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 pairs together an item `a` with an index `i`. Derive a `Show` type class instance for this data type.

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"). 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.

Exercise 3.3 (15 points)**.** Define a shuffling function

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

that permutes the given list `[a]` according to the "random" list of indexes `[Int]`. The `shuffle` function should

1. combine together each `Int` and `a`, pairwise, from the two input lists to produce a list of `Indexed Int a`,
2. sort the list of `Indexed Int a` (according to the `Ord (Indexed Int a)` instance ordering) from step 1, and
3. return a list of `a`s obtained from the items of the list from step 2.

For example,

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

Hint 3.1. The list sorting function

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

can be found in the standard Data.List module included with GHC.