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 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) A constructor for `Aces`, which contain the `Suit` of the card,

(b) A constructor for `Numeric` cards, which have an `Int` standing in for its number as well as a `Suit`, and

(c) A constructor for `Face` cards, contain a `Royalty` (signifying either a `Jack`, `Queen`, or `King`) as well as a `Suit`.

Exercise 1.2 (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]`.

Exercise 1.3 (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.

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
```

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 Jack  = "J"
  show Queen = "Q"
  show King  = "K"
```

Manually define similar `Show` instances for the `Suit` and `Card` types. `show`ing a `Suit` value should return the following `String` representations:

- `Hearts   = "H"`
- `Diamonds = "D"`
- `Clubs    = "C"`
- `Spades   = "S"`

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

- `Ace suit = "A"` followed by the string representation of the `suit`.

- `Numeric n suit =` the string representation of the `Int n` followed by the string representation of the `suit`.

- `Face royal suit =` the string representation of the `Royalty royal` followed by the string representation of the `suit`.

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

Bonus Exercise 1.5 (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 `"♠"`.

Instead of `show`ing a suit using the ASCII characters H, D, C, S, show the appropriate unicode suit symbol above for each suit value., so that the 10 of spades is `shown` as `"10♠"` and the Queen of diamonds is shown `shown` as `"Q♢"`.

Exercise 1.6 (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 1.2), 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.

2 Blackjack Scoring (20 points)

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-busted 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.

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.

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.

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`?

2.2 *Alternate:* Soft Aces (30 extra credit)

Section 2.2 is enirely 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-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.

Bonus Exercise 2.3 (10 extra credit)**.** 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 (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
  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`.

Bonus Exercise 2.5 (10 extra credit)**.** Define the functions

```
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.

Replace the exercise 2.2 function

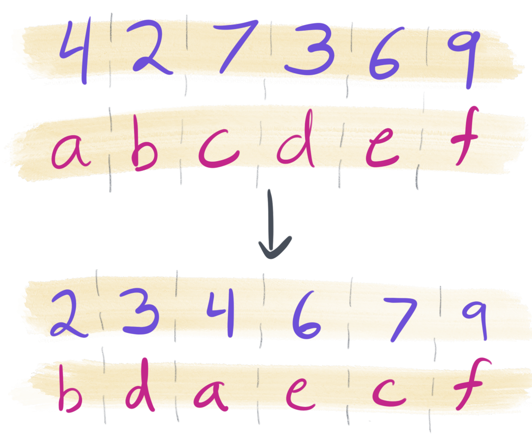
```
handValue :: Hand -> Int
```

with one 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`.

[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.

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 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`.

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.

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

```
indexList   :: [i] -> [a] -> [Indexed i a]
unindexList :: [Indexed i a] -> [a]
```

`indexList` 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, `indexList [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 `indexList` have different lengths, you should drop the extra elements from the end of the longer list.

`unindexList` 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.

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 `indexList` and `unindexList` easier?

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?