Functional Programming Lab 2
A Simple Black Jack Variant*

1 Introduction

In this exercise you will write an implementation of (a simple variant of) the game Black Jack. By doing so you will learn to define recursive functions and QuickCheck properties. Since you have not really learned how to do input/output yet, we provide you with a wrapper which takes care of those things for you.

Later in the course you will learn how to write graphical user interfaces, and if you want to you can then write such an interface for this program. To cater for this possibility, try to write your program in such a way that you can understand it in a month's time.

2 The game

The game you will implement is a simple variant of Black Jack. There are two players, the "guest" and the bank. First the guest plays. She (he) can draw as many cards as she wants, as long as the total value does not exceed 21. When the guest has decided to stop, or gone bust (score over 21), the bank plays. The bank draws cards until its score is 16 or higher, and then it stops.

The value of a hand (the score) is the sum of the values of the cards. The values are as follows:

- Numeric cards have their numeric value, so a nine is worth nine points.
- Jacks, queens and kings are worth ten points each.
- Aces are worth either one point or eleven points. When calculating the value for a hand, *all* aces have the same value; either they are all worth eleven points, or they are all worth one point. Initially the value eleven is used for the aces, but if that leads to a score above 21, then the value one is used instead.

The winner is the player with the highest score that does not exceed 21. If the players end up with the same score, then the bank wins. The bank also wins if both players go bust.

*Author: Koen Lindström Claessen. Thanks to Nils Anders Danielsson, Hampus Ram, Andreas Farre, Sebastian Sylvan and Mary Bergman. Minor modifications to this version by David Sands

3 Your task

Your task is the following:

1. Read this document carefully.
2. Execute *size hand2* (see Section 3.1) by hand, step by step:

   $$\begin{aligned} & size\ hand2 \\ & = size\ (Add\ (Card\ (Numeric\ 2)\ Hearts) \\ & \qquad\qquad (Add\ (Card\ Jack\ Spades)\ Empty)) \\ & = ... \\ & = 2 \end{aligned}$$

   Write down this sequence of equations in the beginning of a Haskell file called `BlackJack.hs`, as a comment.
3. Implement the Haskell functions and QuickCheck properties listed below. Write them in `BlackJack.hs`.
4. While writing the functions (not afterwards), document them.

We will now go through some of the above points in more detail.

3.1 Recursive types

The appendix lists two Haskell files, *Cards* and *Wrapper*. You do not need to understand anything about the wrapper, but you have to understand most of *Cards*. That module contains definitions of data types used to model cards and collections of cards (hands). Read through the file so that you know which types you are supposed to use. (You do not yet need to understand the *Arbitrary* instances in *Cards*.)

One of the types is more difficult than the others:

$$\begin{aligned} & \textbf{data}\ Hand = Empty \mid Add\ Card\ Hand \\ & \qquad\qquad \textbf{deriving}\ (Eq, Show) \end{aligned}$$

A *Hand* can be either

- *Empty*, i.e. an empty hand, or
- *Add card hand*. Here *card* :: *Card* is a card and *hand* :: *Hand* is *another hand*, so this stands for a hand with another card added.

The type is defined in terms of itself; it is recursive. It is easy to create small values of this type, e.g. a hand containing two cards:

$$\begin{aligned} & hand2 = Add\ (Card\ (Numeric\ 2)\ Hearts) \\ & \qquad\qquad (Add\ (Card\ Jack\ Spades)\ Empty) \end{aligned}$$

We start with the empty hand, add the jack of spades, and finally add the two of hearts. It is tiresome to write down all 52 cards in a full deck by hand, though. We can do that more easily by using recursion. That is left as an exercise for you, see below.

We can use recursion both to build something of a recursive type (like a deck of cards) and to take it apart. For instance, say that you want to know the size of a *Hand*. That is easily accomplished using recursion and pattern matching. Note the similarity with recursion over integers:

$$\begin{aligned} &size :: Num\ a \Rightarrow Hand \rightarrow a \\ &size\ Empty &&= 0 \\ &size\ (Add\ card\ hand) &&= 1 + size\ hand \end{aligned}$$

At this stage in the course you might not have fully grasped recursive types; that is after all the reason for doing this assignment. To get a better idea of what happens when a recursive function is evaluated, take some time to work with *size* before you continue. You can for instance evaluate *size hand2* by hand, on paper. The result should be 2, right?

The function *size* is included in the *Cards* module. You can find more recursive functions in the lecture notes.

3.2 Properties

To help you we have included a couple of QuickCheck properties below. Your functions must satisfy these properties. If they do not, then you know that something is wrong. Testing helps you find bugs that you might otherwise have missed. Note that you also have to write some properties yourself, as indicated below.

The purpose of writing properties is three-fold:

1. They serve as a specification before you write your functions.
2. During the implementation phase they help you with debugging.
3. When you are finished they serve as mathematically precise documentation for your program.

So, to take maximum advantage of the properties, write them before you write the corresponding functions, or at the same time.

3.3 Documentation

The code has to be documented. See the *Wrapper* and *Cards* modules (below) to get an idea about what kind of documentation is expected of you. Try to follow these guidelines:

- Focus on what the function does and how one can use it, but not on how the function is implemented. Of course, if a function is complicated then the implementation also has to be documented, but you are not supposed to write complicated functions in this assignment.
- Try to keep the documentation as short as possible, without sacrificing clarity. Long comments make the code harder to read.

Similar arguments apply to documentation as for properties; write the documentation when you write your functions, not afterwards.

3.4 Functions

Write all code in a fresh file called `BlackJack.hs`. To make everything work, add the following lines in the top of the file:

module *BlackJack* **where**

import *Cards*
import *Wrapper*

This tells the Haskell system that the module is called *BlackJack*, and that you want to use the functions and data types defined in *Cards* and *Wrapper*. Download `Cards.hs` and `Wrapper.hs` and store them in the same directory as `BlackJack.hs`, but do not modify the files.

You have to implement the following functions.

- You need to define a function that returns an empty hand:

  $empty :: Hand$

- Given a hand, there should be a function that calculates the value of the hand according to the rules given above:

  $value :: Hand \rightarrow Integer$

  A hint for writing the value function: Start with writing a function `valueRank :: Rank -> Integer`, that calculates the value of a Rank (think about what to do for an Ace!), and perhaps also a function `valueCard :: Card -> Integer`, that calculates the value of a Card. Furthermore, it is a good idea to define and use a function `numberOfAces :: Hand -> Integer`, that calculates the number of aces in a given hand. Write these three functions *before* you start defining the function *value*.

- Given a hand, is the player bust?

  $gameOver :: Hand \rightarrow Bool$

- Given one hand for the guest and one for the bank (in that order), which player has won?

  $winner :: Hand \rightarrow Hand \rightarrow Player$

  Here *Player* is a new data type, defined in the *Wrapper* module:

  data *Player* = *Guest* | *Bank*
  deriving (*Show*, *Eq*)

- Given two hands, $<\!\!+$ puts the first one on top of the second one:

  $(<\!\!+) :: Hand \rightarrow Hand \rightarrow Hand$

(Note that a function name with only symbols indicates an infix operator. It is used just like + or −, with the operator between its arguments: $h_1 <\!\!+ h_2$.)

This function must satisfy the following QuickCheck properties. The function should be associative:

$$prop\_onTopOf\_assoc :: Hand \to Hand \to Hand \to Bool$$
$$prop\_onTopOf\_assoc\ p_1\ p_2\ p_3 = p_1 <\!\!+ (p_2 <\!\!+ p_3) == (p_1 <\!\!+ p_2) <\!\!+ p_3$$

Furthermore the size of the combined hand should be the sum of the sizes of the two individual hands:

$$prop\_size\_onTopOf :: Hand \to Hand \to Bool$$

The implementation of this property is not given here, you have to write it yourselves.

- You also need to define a function that returns a full deck of cards:

  $$fullDeck :: Hand$$

  You could do this by listing all 52 cards, like we did with two cards above. However, that is very tedious. Instead, do it like this: Write a function which given a suit returns a hand consisting of all the cards in that suit. Then combine the 13-card hands for the four different suits into one hand using $<\!\!+$.

- Given a deck and a hand, draw one card from the deck and put on the hand. Return both the deck and the hand (in that order).

  $$draw :: Hand \to Hand \to (Hand, Hand)$$

  If the deck is empty, report an error using *error*:

  ```
  error "draw: The deck is empty."
  ```

  By changing the type of *draw* one could get around this rather ugly solution. We will get to that later in the course. Maybe you can think of a way already now?

  To return two values a and b in a pair, use the syntax (a, b). You can also pattern match on pairs:

  $$first :: (a, b) \to a$$
  $$first\ (x, y) = x$$

- Given a deck, play for the bank according to the rules above (starting with an empty hand), and return the bank's final hand:

  $$playBank :: Hand \to Hand$$

  To write this function you will probably need to introduce a help function that takes two hands as input, the deck and the bank's hand. To draw a card from the deck you can use **where** in the following way:

```
playBank' deck bankHand ...
                                ...
  where (deck', bankHand') = draw deck bankHand
```

If you have not seen **where** before, read about it in the book.

- Given a *StdGen* and a hand of cards, shuffle the cards and return the shuffled hand:

  ```
  shuffle :: StdGen → Hand → Hand
  ```

  A *StdGen* is a random number generator. Import the *System.Random* library:

  ```
  import System.Random
  ```

  Now, if g is a random number generator, then *randomR* (lo, hi) g is a pair (x, g'), where x is a number between *lo* and *hi* (inclusive), and g' is a new random number generator. Note that to get several random numbers you have to use *different* random number generators; if you used g this time, then you have to use g' (or some other generator) the next time. If you were to reuse g then you would get the same result again.

  As an example, the following function takes a random number generator as input and uses it to calculate two random integers between 0 and 10, inclusive:

  ```
  twoRandomIntegers :: StdGen → (Integer, Integer)
  twoRandomIntegers g = (n1, n2)
    where (n1, g1) = randomR (0, 10) g
          (n2, g2) = randomR (0, 10) g1
  ```

  Note that if we had used g in the last line as well, then n_1 and n_2 would be equal, so instead we use the new random number generator g_1 returned by *randomR*.

  By the way, you can construct a value of type *StdGen* by using *mkStdGen*:: *Int* → *StdGen*.

  So, now that we know how to handle random numbers, how can we shuffle a deck of cards? If you want a (small) challenge, do not read the next three paragraphs.

  One way to shuffle the cards would be to pick an arbitrary card from the deck and put it on top of the deck, and then repeat that many times. However, how many times should one repeat? If one repeats 52 times, then the probability that the last card is never picked is about 36%. This means that the last card is often the same, which of course is not good.

  A better idea is to pick an arbitrary card and put it in a new deck, then pick another card and put it on top of the new deck, and so on. Then we know that we have a perfectly shuffled deck in 52 steps (given that the random number generator is perfect, which it is not).

  Note that for both approaches we need a function that removes the n-th card from a deck.

The function *shuffle* has to satisfy some properties. First, if a card is in a deck before it has been shuffled, then it should be in the deck afterwards as well, and vice versa:

```
prop_shuffle_sameCards :: StdGen → Card → Hand → Bool
prop_shuffle_sameCards g c h =
  c `belongsTo` h == c `belongsTo` shuffle g h
```

For this we need the helper function *belongsTo*, which returns *True* iff the card is in the hand.

```
belongsTo :: Card → Hand → Bool
c `belongsTo` Empty      = False
c `belongsTo` (Add c' h) = c == c' || c `belongsTo` h
```

(By using ` we can turn a function into a binary operator.)

The above property does not guarantee that the size of the deck is preserved by *shuffle*; all cards could be duplicated, for instance. You have to write a property which states that the size is preserved:

```
prop_size_shuffle :: StdGen → Hand → Bool
```

3.5 Interface

You have barely touched upon input/output in the lectures, so we have to provide you with a wrapper (the module *Wrapper*) that takes care of those things. All you have to do is to write the functions above, package them together (as explained below), and then call the wrapper with the package as an argument.

To "package up" these functions, write the following code:

```
implementation = Interface
  { iEmpty    = empty
  , iFullDeck = fullDeck
  , iValue    = value
  , iGameOver = gameOver
  , iWinner   = winner
  , iDraw     = draw
  , iPlayBank = playBank
  , iShuffle  = shuffle
  }
```

To run the program, define

```
main :: IO ()
main = runGame implementation
```

in your source file, load the file, and run *main*.

4 Possible extensions

The following is completely optional, but if you want to do more, there are many possibilities:

- Now the bank draws all its cards after the guest has finished. It would be more fun if the guest could see the bank's cards while playing.
- The rules are not really proper Black Jack rules.
- There are many other card games, many of which may be more fun than this one.
- You probably have some ideas yourself.

Most of the ideas above require that you program input/output (I/O) yourself. You have seen how to do simple I/O in the lectures. Use the *Wrapper* module as a starting point.

Note that doing something extra will not directly affect your grade, but can of course be beneficial in the long run. Take care to do the compulsory part above before attempting something more advanced, though.

A The *Wrapper* module

```
module Wrapper where

import Data.Char
import System.Random
import Cards

-- The interface to the students' implementation.

data Interface = Interface
  { iEmpty    :: Hand
  , iFullDeck :: Hand
  , iValue    :: Hand -> Integer
  , iGameOver :: Hand -> Bool
  , iWinner   :: Hand -> Hand -> Player
  , iDraw     :: Hand -> Hand -> (Hand, Hand)
  , iPlayBank :: Hand -> Hand
  , iShuffle  :: StdGen -> Hand -> Hand
  }

-- A type of players.

data Player = Guest | Bank
              deriving (Show, Eq)

-- Runs a game given an implementation of the interface.

runGame :: Interface -> IO ()
runGame i = do
  putStrLn "Welcome to the game."
  g <- newStdGen
  gameLoop i (iShuffle i g (iFullDeck i)) (iEmpty i)
```

```
-- Play until the guest player is bust or chooses to stop.

gameLoop :: Interface -> Hand -> Hand -> IO ()
gameLoop i deck guest = do
  putStrLn ("Your current score: " ++ show (iValue i guest))
  if iGameOver i guest then do
    finish i deck guest
   else do
    putStrLn "Draw another card? [y]"
    yn <- getLine
    if null yn || not (map toLower yn == "n") then do
      let (deck', guest') = iDraw i deck guest
      gameLoop i deck' guest'
     else
      finish i deck guest

-- Display the bank's final score and the winner.

finish :: Interface -> Hand -> Hand -> IO ()
finish i deck guest = do
  putStrLn ("The bank's final score: " ++ show (iValue i bank))
  putStrLn ("Winner: " ++ show (iWinner i guest bank))
  where
  bank = iPlayBank i deck
```

B The *Cards* module

```
{-# OPTIONS -fno-warn-missing-methods #-}
module Cards where

import Test.QuickCheck
import Random

-- A card has a rank and belongs to a suit.

data Card = Card { rank :: Rank, suit :: Suit }
            deriving (Eq, Show)

instance Arbitrary Card where
  arbitrary = do
    suit <- arbitrary
    rank <- arbitrary
    return (Card rank suit)

-- All the different suits.

data Suit = Hearts | Spades | Diamonds | Clubs
            deriving (Eq, Show)
```

```
instance Arbitrary Suit where
  arbitrary = oneof [ return Hearts, return Spades
                    , return Diamonds, return Clubs ]

-- A rank is either a numeric card, a face card, or an ace. The
-- numeric cards range from two to ten.

data Rank = Numeric Integer | Jack | Queen | King | Ace
            deriving (Eq, Show)

instance Arbitrary Rank where
  arbitrary = frequency [ (1, return Jack)
                        , (1, return Queen)
                        , (1, return King)
                        , (1, return Ace)
                        , (9, do n <- choose (2, 10)
                                 return (Numeric n))
                        ]

-- A hand of cards. This data type can also be used to represent a
-- deck of cards.

data Hand = Empty | Add Card Hand
            deriving (Eq, Show)

-- This instance on average yields larger hands than the one given in
-- the lecture.

instance Arbitrary Hand where
  arbitrary = frequency [  (1,  return Empty)
                        ,  (10, do card  <-  arbitrary
                                   hand  <-  arbitrary
                                   return (Add card hand))
                        ]

-- The size of a hand.

size :: Num a => Hand -> a
size Empty            = 0
size (Add card hand)  = 1 + size hand

-- We also need to be able to generate random number generators. (This
-- does not really belong in this file, but is placed here to reduce
-- the number of files needed.)

instance Arbitrary StdGen where
  arbitrary = do n <- arbitrary
                 return (mkStdGen n)
```