TDA 452
DIT 143
HT 2019

Functional Programming 2019
Lab 2

In this lab assignment, you will implement a Haskell program for playing a card game called BlackJack.

- **Lab 2A** consists of the assignments labelled **A0, A1…** (writing out what happens to the size function in Section 3.1, and implementing some of the functions in Section 3.4).
- **Lab 2B** consists of the assignments labelled **B1, B2…** (the remaining functions in Section 3.4 and Section 3.5).

It’s probably a good idea to read the whole document before you start, to familiarize yourself with the overall goals and requirements.

Before you start you will need to download the following two files (which you will not edit):

- Cards.hs
- RunGame.hs

1 Introduction

The game you will implement is a simple variant of the game Black Jack. By doing so you will learn to define recursive functions over recursive data types, and define QuickCheck properties. Since you have not really learned how to do input/output yet, we provide you with a “wrapper” module which takes care of the interactive aspects of the game.

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

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. If the guest goes bust, the bank wins. The bank also wins if both players go bust.

3 Assignments

In summary, your task is the following:

1. Compute `size hand2` (see Section 3.1) by hand, step by step.
2. Implement the Haskell functions and QuickCheck properties listed below.

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

3.1 Recursive types

We provide two Haskell files that you should use, Cards.hs and RunGame.hs. You do not need to understand anything about RunGame, 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:

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

A `Hand` can be either

- `Empty`, i.e. an empty hand, or
- `Add` card hand. Here card is a `Card` and 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:

```haskell
hand2 = Add (Card (Numeric 2) Hearts)
            (Add (Card Jack Spades) Empty)
```

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:

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

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:

A0.

Evaluate `size hand2` by hand, on paper. The result should be `2`, right?

```
size hand2
  = size (Add (Card (Numeric 2) Hearts)
              (Add (Card Jack Spades) Empty))
  = ...
  = 2
```

Now write down this sequence of steps as a definition in your Haskell file (`BlackJack.hs`). This way you can check it easily - each element of your list should have the value 2.

```
sizeSteps :: [Integer]
sizeSteps = [ size hand2
            , size (Add (Card (Numeric 2) Hearts)
                        (Add (Card Jack Spades) Empty))
            -- ... add the remaining steps here
            ,2]
```

The function `size` is included in the Cards module. You can find further examples of 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 RunGame and Cards modules 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 RunGame
import Test.QuickCheck
```

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 RunGame. Download Cards.hs and RunGame.hs and store them in the same directory as BlackJack.hs, but do not modify the files.

You have to implement the following functions:

A1. Implement a function that, given a hand, shows the cards in it in a nice format.

```
display :: Hand -> String
```

For example, display can show the card `Card (Numeric 2) Hearts` as `"2 of Hearts"`, and the card `Card Jack Spades` as `"Jack of Spades"`. If you want to show the card suits as pictures instead, use Unicode characters, `\9829` for Hearts, `\9824` for Spades, `\9830` for Diamonds, and 827 for Clubs. (Note that Windows users may have problems getting unicode to work.)

Hint: Start by writing a function `displayCard :: Card -> String` that shows just a single card. Other useful things will be the append function `(++)` for joining lists (including strings), and the newline character `'\n'`. Try this in ghci: `putStr "Hello\nWorld\n!\n`. When you are done try out your function using `putStr (display hand2)`.

A2. Given a hand, there should be a function that calculates the value of the hand according to the rules given above:

```
value :: Hand -> Integer
```

A hint for writing the value function: defining a few helper functions will be useful, but it can be done in different ways:

- (Option 1) Define a function `initialValue :: Hand -> Integer` that uses 11 for the value of aces, and a function `numberOfAces :: Hand -> Integer` that can be used when computing the final value, if the initial value is over 21.

- (Option 2) Define a function

  ``{.haskell} which takes the value you want to use for the aces (1 or 11) as an extra argument. Call this function to compute the initial value, and then call it again to compute the final value, if the initial value is over 21.

Regardless of which option you choose, it will be useful to have a (local) helper function to compute the values of a given rank:

```
valueRank :: Rank -> Integer
```

A3. Given a hand, is the player bust?

```
gameOver :: Hand -> Bool
```

A4. Given one hand for the guest and one for the bank (in that order), which player has won?

```
winner :: Hand -> Hand -> Player
```

Here `Player` is a new data type, defined in the RunGame module:

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

B1. Given two hands, `<+` puts the first one on top of the second one:

```
(<+) :: Hand -> Hand -> 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: `h1 <+ h2`.) This function must satisfy the following QuickCheck properties. The function should be associative:

```
prop_onTopOf_assoc :: Hand -> Hand -> Hand -> Bool
prop_onTopOf_assoc p1 p2 p3 =
    p1<+(p2<+p3) == (p1<+p2)<+p3
```

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

```
prop_size_onTopOf :: Hand -> Hand -> Bool
```

The implementation of this property is not given here, you have to write it yourselves. Don't forget to actually test these properties with `quickCheck`!

B2. 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, and not an acceptable solution. One way is to work with lists of cards and use list comprehensions, and convert the result to a Hand. Another way is to 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 `<+`. Try out your solution using `putStr (display fullDeck)`

B3. 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 -> Hand -> (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, like in this example:

```
first :: (a, b) -> a
first (x,y) = x
```

B4. 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 -> Hand
```

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

```
playBankHelper deck hand ... ...
  where (smallerDeck,biggerHand) = draw deck hand
```

B5. This is the hard question! It involves the use of a new type, `StdGen`, a *random number generator* which is explained below (but note that random number generation is not what makes this a harder assignment).

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

```
shuffleDeck :: StdGen -> Hand -> Hand
```

A `StdGen` is, as mentioned, 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` gives us 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 (and that would not be so random!). As an example, the function `dieRoll` below takes a random number generator and produces two integers in the range 1 to 6 (like rolling two dice):

```
dieRoll :: StdGen -> (Integer,Integer)
dieRoll g = (n1, n2)
  where (n1, g1) = randomR (1,6) g
        (n2, _ ) = randomR (1,6) g1
```

Note that if we had used `g` in the last line as well, then `n1` and `n2` would be equal, so instead we use the new random number generator `g1` 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` shuffleDeck 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 an infix 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

B6. You have barely touched upon input/output in the lectures, so we provide the module RunGame to 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 `runGame` function with the package as an argument. To "package up" these functions, write the following code:

```
implementation = Interface
  { iFullDeck = fullDeck
  , iValue    = value
  , iDisplay  = display
  , iGameOver = gameOver
  , iWinner   = winner
  , iDraw     = draw
  , iPlayBank = playBank
  , iShuffle  = shuffleDeck
  }
```

To run the program, define

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

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

Submission

Submit your solutions in a single file called `BlackJack.hs`. Please do not submit the given modules `Cards.hs` and `RunGame.hs`.

For each part (A1…, B1…), use Haskell comments to mark the part of the file that contains the answer to that part. For answers in natural language, use English; write your answers also in Haskell comments.

You must submit using the Fire system, groups of 2. This will be strictly enforced by the fire system, so exceptions to allow groups of one can only be made if they are agreed in advance. Groups of three are not allowed.

Before you submit your code, Clean It Up!

- Keep lines below 80 characters; don't use tab characters (they cause other problems for Haskell).
- Give type declarations for (at least) the functions you are asked to write (you may omit them for simple quickCheck properties).
- Remove junk (unused or commented-out code, unnecessary comments).
- Simplify overcomplicated functions, and avoid cut-and-paste coding.

Feel free to use the hlint program to help with many of these issues and other haskell style issues.

We *hope* to add some Automatic checks in Fire that will run when you submit your answers, e.g. running hlint and perhaps testing some of your functions with QuickCheck. The purpose of this is to give you some quick feedback and to help speed up the grading process. (If the feedback you get is not helpful, you can safely ignore it.)

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 RunGame 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.

Last modified: 2019-11-04 by Dave. 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 and Thomas Hallgren.