Haskell Exercises

1 Sum of two numbers

write in the module **`Assignment01`** a function

```
      sumOfTwoInts :: Int -> Int -> Int
```

This function should calculate the sum of two integers.

Run the tests under **`Tests01.hs`** to check your solution.

2 Sum of squares

write in the module **`Assignment02`** a function

```
      sumOfSquares :: [Int] -> Int
```

This function should return the sum of the squares of the integers in the given list.

Run the tests under **`Tests02.hs`** to check your solution.

Keywords: recursion

3 Arbitrary sums

In this function we generalize the previous function. Let us suppose that we need also **`sumOfCubes`** and **`sumOfDoubles`**.

In order to avoid writing the same code twice or three times we want now to build the sum of the "processed" lists.

write in the module **`Assignment03`** a function

```
      mappedSum :: (Int -> Int) -> [Int] -> Int
```

This function should take as first parameter a function that will be applied to each element of the given list.

Run the tests under **`Tests03.hs`** to check your solution.

Keywords: recursion, higher order functions

4 Find the n^th element in the list

write in the module **`NthElement`** a function

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

This function should return the element at the given position of the given list. Notice that the first element has the index 0.
You can assume that the list has always enough elements, so you can forget about error handling.
Do not use the haskell function “!!”.

Run the tests under **`Tests04.hs`** to check your solution.

Keywords: recursion

5 Reverse a list

write in the module **`ListReversion`** a function

```
reverse' :: [a] -> [a]
```

This function should reverse the given list. For example the list ["Hello, "Brave", "New", "World"] should be reversed to ["World", "New", "Brave", "Hello"].
Do not use the list concatenation function ++, “reverse” or “init”.
Define instead using “where” or “let” a tail-recursive inner helper function. You should understand what tail-recursion (or end-recursion) is.

Run the tests under **`Tests05.hs`** to check your solution.

Keywords: recursion, tail-recursion

6 Compare dates

Let us assume that a date is defined by a tuple of three integers with the format “(year, month, day)” such as (2013, 1, 25). We neglect here the check of date validation.

write in the module **`CompareDates`** a function

```
isOlder :: (Int, Int, Int) -> (Int, Int, Int) -> Bool
```

This function should compare the two given dates and return true if and only if the first date is older than the second one.

Run the tests under **`Tests06.hs`** to check your solution.

Keywords: pattern matching

7 Number of dates for a given month

write in the module **`CompareDates`** a function

```
numberInMonth :: [(Int, Int, Int)] -> Int -> Int
```

This function should return for a given list of dates and a month the number of dates with that month.

Run the tests under **`Tests07.hs`** to check your solution.

Keywords: pattern matching, recursion

8 Number of dates for the given months

write in the module **`CompareDates`** a function

```
numberInMonths :: [(Int, Int, Int)] -> [Int] -> Int
```

This function should return for a given list of dates and a list of months the number of dates with one of those months.

Run the tests under **`Tests08.hs`** to check your solution.

Keywords: recursion

9 Dates with the given month

write in the module **`CompareDates:`**

- a type alias **`Date`** for a triple of integers.
- a function **`datesInMonth :: [Date] -> Int -> [Date]`**

The function should return for a given list of dates and a month the list of dates with the given month.

Run the tests under **`Tests09.hs`** to check your solution.

Keywords: datatypes, recursion, pattern matching

10 Oldest date

write in the module **`CompareDates`** a function

```
oldestDate :: [(Int, Int, Int)] -> Maybe (Int, Int, Int)
```

This function should return for a given list of dates the oldest date wrapped in a **`Just`**. If the given list is empty then the function should return **`Nothing`**.

Run the tests under **`Tests10.hs`** to check your solution.

Keywords: Maybe, recursion

11 All Strings except ...

write in the module **`AllExcept`** a function

```
    allExcept :: [Char] -> [[Char]] -> Maybe [[Char]]
```

that for a given string and list of strings ***`lst`*** returns either **`Nothing`** if **`lst`** does not contains the given string, or **`Just lst'`** where **`lst'`** is identic to lst except that the first occurrence of the given string is removed.
Examples:

```
allExcept "Hans" ["Fritz", "Fred"] = Nothing
allExcept "Hans" ["Fritz", "Hans", "Fred", "Hans"] = ["Fritz", "Fred", "Hans"]
```

Run the tests under **`Tests11.hs`** to check your solution.
Find out if tail-recursion is useful here or not.
Keywords: recursion, Maybe

12 Game of Cards – Color and Rank

write in the module **`Cards`** two functions

```
    cardColor :: Card -> Color
    cardValue :: Card -> Int
```

that return for a given card its color and its rank respectively. The value of King, Queen and Jack is 10. The value of Ace is 11. The values of the other cards are their numbers.

The play card **`Card`** is defined as follows:

```
type Card = (Suit, Rank)
data Suit = Clubs | Diamonds | Hearts | Spades deriving (Show, Eq)
data Rank = Jack | Queen | King | Ace | Num Int deriving (Show, Eq)
data Color = Red | Black deriving (Show, Eq)
```

Run the tests under **`Tests12.hs`** to check your solution.
Keywords: datatypes, pattern matching

13 Game of Cards – Remove Card from List

write in the module **`Cards`** a function

```
    removeCard :: [Card] -> Card -> Either IllegalMove [Card]
```

that removes the given card from the given list, if it exists, and returns the result wrapped in **`Right`**. If the card does not exist in the list, then the function should return **`Left IllegalMove`**.

`IllegalMove` is defined as follows:

```
data IllegalMove = IllegalMove deriving (Show, Eq)
```

Run the tests under **`Tests13.hs`** to check your solution.
Keywords: datatypes, Either

14 Game of Cards – All have the same Color?

write in the module **`Cards`** a function

```
    allSameColor :: [Card] -> Bool
```

that returns **`True`** if all cards in the given list have the same color.

Run the tests under **`Tests14.hs`** to check your solution.
Keywords: datatypes, pattern matching, recursion

15 Game of Cards – Sum of Card values

write in the module **`Cards`** a function

```
    sumCards :: [Card] -> Int
```

that returns the sum of the values of the cards in the given list.

Run the tests under **`Tests15.hs`** to check your solution.
Use a tail-recursive inner helper function.
Keywords: tail-recursion, recursion

16 Game of Cards – Score Evaluation

write in the module **`Cards`** a function

```
    score :: [Card] -> Int -> Int
```

that calculates the score for a given list of cards and a given goal value. The score is calculated as follows (similar to the game Blackjack):

Temporary Score: If the sum of the card values is greater than the given goal value, then the temporary score is three times the difference between the goal value and the sum of the card values. If the sum is smaller or equal to the goal value then the temporary score is the simple difference between them.

Final Score: .The final score is the same as the temporary score except if all the cards have the same color. In that case the final score is the (rounded) half of the temporary score.

The function **`score`** should return the final score.

Run the tests under **`Tests16.hs`** to check your solution.

17 Game of Cards – Simulation of the game flow

write in the module **`Cards`** a function

```
    officiate :: [Card] -> [Move] -> Int -> Either IllegalMove Int
```

This function should simulate the game flow.
`Move` is defined as follows:

```
    data Move = Discard Card | Draw deriving (Show, Eq)
```

The function gets a list of cards (the initial stack), a list of moves and a goal value (for calculation the score). In the good case the function should return the final score of the game after evaluating the given moves. Otherwise **`IllegalMove`**.

Use a recursive inner helper function, thattakes some parameters that represents the current game state.

The game goes as follows:

- At the beginning the player has no cards in the hand.
- The game is over at the latest when no more moves are available. Then the score should be evaluated.
- The player can discard a card that he has in the hands. The discarded card does NOT get back in the stack.
- If the player tries to discard a card that he does not have, then the function should return **`Left IllegalMove`**.
- When the player tries to draw a card and the stack is empty, then the game is over. The final score should be evaluated.
- If the user draws a card (and the stack is not empty), then he gets the next card from the stack in his hands. If the sum of the values of his cards is greater then the goal value, the the game is over and the final score should be evaluated and returned.

Run the tests under **`Tests17.hs`** to check your solution.
Keywords: pattern matching, recursion

18 Only Capitals

write in the module **`StringsHOF`** a function

```
      onlyCapitals :: [[Char]] -> [[Char]]
```

that returns for the given list of non empty strings a list with only the strings that begin with capital letters. The order should be preserved.

Run the tests under **`Tests18.hs`** to check your solution.
Do not use recursions.
Keywords: higher order functions

19 Find the longest string

write in the module **`StringsHOF`** two functions

```
      longestString1 :: [[Char]] -> [Char]
      longestString2 :: [[Char]] -> [Char]
```

that for a given list with the strings returns the string with the biggest length. If there are many string with biggest length, the should **`longestString1`** return the first one and **`longestString2`** the last one.

Run the tests under **`Tests19.hs`** to check your solution.
Do not use recursions. Write a helper function that should be used by both functions and that take a

function as parameter that compares two string lengths.
The Helper function should have the following signature:
```
longestStringHelper :: (Int -> Int -> Bool) -> [[Char]] -> [Char]
```

Keywords: higher order functions, fold

20 <u>Longest capitalized string</u>

write in the module **`StringsHOF`** a function
```
     longestCapitalized :: [[Char]] -> [Char]
```
that for a given list of non empty strings returns the longest capitalized string. If the list is empty then the function should return the empty string.
If many capitalized strings have the biggest length, the the function should return the first one.

Keywords: function composition