Till Mossakowski
Benjamin Junge

2. Excercise sheet

Issued: 2023-10-17
Due: 2023-10-23 & 2023-10-24

2.1 *Modelling the basics of a casino*

(1) The casino uses jetons[1] in the colours red(1€), green(5€), blue(10€), silver(50€) and gold(100€) as their currency. Upon entering the casino one can buy jetons for money(in €). Upon leaving the casino the jetons can be traded for the corresponding amount of money(in €)

Define an enumeration type Jeton and a function that returns a jetons value:

```
value :: Jeton → Int
```

(2) A collection of jetons jc is either empty or a pair of (jeton jc). Based on this define a recursive type Jetons and implement the functions count and payoff which return the amount of jetons in and the value of a jeton collection.

```
count :: Jetons → Int
payoff :: Jetons → Int
```

(3) Implement a function for buying jetons which returns a collection of jetons with a payoff value equal to the money value given. In addition this collection should be as small as possible. (e.g. the smallest collection for 16€ would be a collection with 1 blue, 1 green and 1 red jeton)

```
buy :: Int → Jetons
```

The following test cases should hold:

```
count (buy 16) ⇝ 3
payoff (buy 16) ⇝ 16
count (buy 187) ⇝ 8
payoff (buy 187) ⇝ 187
```

In general the following should hold: if `s` $\geq$ `0` then `payoff (buy s)` $=$ `s`.

[1] coin-like tokens used in casinos, similar to poker chips

2.2 *Text alignments*

Implement the flush left and flush right alignment for text as follows: Define a datatype as a selection of alignment types.

```
data Format = Flushleft | Flushright
```

Define the function

```
align :: Format → Int → String → String
```

which aligns a text according to the following rules:

1. words aren't split apart.
2. the amount of symbols per line shouldn't exceed the symbol limit given by the Int. Only exception to this are words that exceed that limit on their own
3. a line should be filled as much as possible without exceeding the symbol limit.

Example:
`putStrLn $ align Flushleft 5 "ab␣cd␣edfghij␣klm"`

produces:
```
ab cd
edfghij
klm
```

2.3 *Safe calculations*

A cheap way of handling potentially undefined values or invalid arguments is using the predefined type `Maybe a` instead of type `a`. Implement the following two functions

```
safeDiv :: Maybe Double → Maybe Double → Maybe Double
safeRoot :: Maybe Double → Maybe Double
```

which implement safe versions of `(/)` and `sqrt` that utilize the `Maybe Double` type to catch invalid argument values. Explain the advantage of using `Maybe Double` for the argument type instead of just `Double`.

2.4 *The simplest enumeration type*

The lecture has defined `Bool` as the simplest enumeration type, but is this actually true? Is there a simpler enumeration type and if so, where was it used in the lecture slides before and is it useful or obsolete?

2.5 *Hiding values*

Suppose you want to write an application that uses large lists of integers and that needs to look at (and use) the sums of all values in a list (called value of the list) with high frequency. Instead of calculating the size every single time one could define a type to store the value of a list once it has been instantiated and alter it when modified. Given the definition of the type ListInt as `data ListInt = ListInt [Int] Int deriving (Show)` implement the following functions on this type:

```
create :: [Int] → ListInt
getList :: ListInt → [Int]
getSum :: ListInt → Int
add :: Int → ListInt → ListInt
liconcat :: ListInt → ListInt → ListInt
```

`create` acts as a encapsulated constructor that constructs the type with correct values. `add` prepends or appends (depending on what you like better) an integer to an existing ListInt while updating its sum value. `liconcat` concatenates lists of two ListInts and updates the sum value accordingly. `getList` and `getSum` are simple getter functions.

2.6 *Mixed lists

Haskell lists are defined as **data** `[a] = [] | a:[a]` which forces all elements of a list to be of equal type. In contrast a list in Python can contain any combination of types. Define a type that is a toned down version of this kind of list called mixed list (ML) such that its elements can either be of type String, Int or Bool. Aditionally define the following functions on Mixed lists.

```
prepend :: ML → ML → ML
mconcat :: ML → ML → ML
countComponents :: ML → String
extractBool :: ML → [Bool]
extractInt :: ML → [Int]
extractString :: ML → [String]
```

where prepend prepends a single element mixed list to an existing one, mconcat concatenates two mixed lists and the extractX functions return all values of type X of a mixed list as a normal list. countComponents counts the numbers of Ints, Strings and Bools in an ML and prints the information as a formatted string. (e.g: "Strings: 1, Integers: 48, Bools: 5") Test your functions accordingly.