universität innsbruck

Functional Programming	WS 2019/2020	LVA 703024+703025
Exercises 6		Deadline: November 13, 2019, 6am

- Please write all your Haskell functions from this exercise sheet into a single .hs-file and upload it in OLAT.
- You can use a template .hs-file that is provided on the proseminar page.
- The file should compile with ghci.
- Once the file has been uploaded, it cannot be changed or resubmitted!

Exercise 6.1 *Type-Classes* **4 p.**

1. Complete the arithmetic implementation of rational numbers from the lecture (Part 3, slide 35). Which methods are necessary to create a valid `Num` instance and which ones are optional? (See the Haskell documentation: https://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html)

   You should also modify the existing implementation in such a way that the representation is always normalized, i.e., the denominator is always positive and common fractions are cancelled. Here the function for computing the greatest common divisor – `gcd :: Integral a => a -> a -> a` – might be useful. (2 points)

2. Finish the `sqrt` version of the lecture (Part 3, slide 24) by adding suitable type signatures. Test your `Rat` implementation with it by evaluating `sqrt 2 :: Rat`. Note that in the template file, `sqrt` is renamed into `mySqrt` in order to avoid name-conflicts with the Prelude version of `sqrt`. (1 point)

3. The evaluation of `sqrt 2 :: Rat` should fail. Why does it fail? Implement the missing class instances. Test your method again using `sqrt 2 :: Rat`. (1 point)

Exercise 6.2 *Maybe And Tuples* **2 p.**

In the international current exchange system, all currencies will be exchanged through U.S. dollars, if two countries do not have a special treaty. We have a type `data Cash = Currency Integer Double deriving (Show, Eq)` for currency exchange. It should contain an `Integer` for the **total amount of cash** in one currency and its **exchange rate** to U.S. dollar (`Double`). The value of `Currency m rate` in U.S. dollar is:

$$Value_{USD} = m * rate \tag{1}$$

The following exercise is based on generic datatypes for pairs and triples defined as follows: (you can ignore the deriving-statement which is just important for running the tests)

```
data Pair a b = Pair_C a b deriving (Show, Eq)
data Triple a b c = Triple_C a b c deriving (Show, Eq)
```

1. Implement a function `valid_cash :: Cash -> Cash -> Maybe (Triple Cash Integer Double)` to check if an exchange request is valid. The function returns `Nothing` if the transaction is not valid, otherwise it returns (just) a triple in the transaction format containing the **budget cash**`::Cash`, **desired amount of cash**`::Integer` and **the exchange rate**`::Double` in the targeted currency.

   The first argument of `valid_cash` is the **budget cash**`::Cash` for exchanging in the original currency. The second argument is **the desired cash**`::Cash` of the targeted currency after the currency exchange. The transaction is valid under following conditions:

   - The budget cash should have greater value (in U.S. dollars) than the desired cash for exchange.

- All arguments in the constructor `Currency` should be positive
- The desired money should be at least 100 in the targeted currency.

(1 point)

2. Implement a function `exchange :: Maybe (Triple Cash Integer Double) -> Maybe (Pair Cash Cash)` that exchanges for the desired cash. The function takes the output of `valid_cash` as input and returns the **exchanged cash** and the remaining **budget** in their corresponding currencies. You should round **the resulting amount of exchanged cash** to `Integer`.

   Remarks:

   - The prelude function `round :: (RealFrac a, Integral b) => a -> b` might be useful.
   - Patterns can be arbitrarily nested as in `fun_name (Just (Pair_C x y)) = x + y`.

   Test your function with:

```
exchange_currency:: Cash -> Cash -> Maybe (Pair Cash Cash)
exchange_currency a b = exchange (valid_cash a b)
```

   (1 point)

Exercise 6.3 *Type-Checking* 4 p.

For each of the three functions given bellow, determine the result of the type-inference algorithm (slides 27–31 of part 3). Based on this inferred type, determine which of the given type declarations are valid. **You should solve this exercise without GHCi!** You can find the types of all built-in functions that are used in this exercise on slides 19–23 of part 3.

1. `func_1 x y = div x y + x` (1 point)
   - `func_1 :: Integer -> Integer -> Integer`
   - `func_1 :: Float -> Float -> Float`
   - `func_1 :: Integral a => a -> a -> a`
   - `func_1 :: Num a => a -> a -> a`
2. `func_2 x y z = if x then y == z else y /= z` (1 point)
   - `func_2 :: Eq a => a -> a -> a -> a`
   - `func_2 :: Eq a => Bool -> a -> a -> Bool`
   - `func_2 :: Bool -> Bool -> Bool -> Bool`
   - `func_2 :: Ord a => Bool -> a -> a -> Bool`
3. Give a textual description on how the typing algorithm can be extended to guarded equations (in the simple case without `where`). (1 point)
4. (1 point)

```
func_3 x y z
  | x == z = (y, z)
  | y < 0 = (y + z, z)
  | otherwise = (y - z, z)
```

   - `func_3 :: Float -> Float -> Float -> (Float, Float)`
   - `func_3 :: Eq a => a -> a -> a -> (a, a)`
   - `func_3 :: (Eq a, Ord b, Num b) => a -> b -> b -> (b, b)`
   - `func_3 :: (Ord a, Num a) => a -> a -> a -> (a, a)`