universität innsbruck

Functional Programming	WS 2024/2025	LVA 703025
Exercise Sheet 6, 10 points		Deadline: Tuesday, November 19, 2024, 8pm

- Mark your completed exercises in the OLAT course of the PS.
- You can use a template .hs file that is provided on the proseminar page.
- Upload your modified `Template_06.hs` file in OLAT.
- Do not change the first lines of `Template_06.hs`, in particular do not add any `import` instructions.
- Your .hs file must be compilable with ghci.

Exercise 1 *Rational Numbers* **5 p.**

Implement rational numbers in Haskell. Here, rational numbers are represented by two integers, the numerator and the denominator. For instance the rational number $\frac{-3}{5}$ can be represented as `Rat (-3) 5` when using the following data type definition:

```
data Rat = Rat Integer Integer
```

1. Implement a normalisation function `normaliseRat :: Rat -> Rat` for rational numbers so that all of `Rat 2 4`, `Rat (-1) (-2)` and `Rat 1 2` are transformed into the same internal representation. Furthermore, implement a function `createRat :: Integer -> Integer -> Rat` that, given two `Integer`s, returns a normalised `Rat`. (1 point)

   Hint: the Prelude contains a function `gcd` to compute the greatest common divisor of two integers.

2. Make `Rat` an instance of `Eq` and `Ord`. Of course, `Rat 2 4 == Rat 1 2` should evaluate to `True`. (1 point)

3. Make `Rat` an instance of `Show`. Make sure that `show r1 == show r2` whenever `r1 == r2` for two rational numbers `r1` and `r2`. In particular, `show (Rat 1 2) == show (Rat 2 4)` should evaluate to `True`. Moreover, integers should be represented without the `"/"` symbol. (1 point)

   Examples:
   ```
   show (Rat (-4) (-1)) == "4"
   show (Rat (-3) 2) == "-3/2"
   show (Rat 3 (-2)) == "-3/2"
   ```

4. Make `Rat` an instance of `Num`. See https://hackage.haskell.org/package/base-4.20.0.1/docs//Prelude.html#t:Num for a detailed description of this type class. (2 points)

Exercise 2 *Monoids* **5 p.**

A monoid is an algebraic structure that consists of an associative binary operation $\circ$ and a neutral element e where the following laws are satisfied for all x, y, z:

- $x \circ (y \circ z) = (x \circ y) \circ z$
- $x \circ e = e \circ x = x$

We model monoids in Haskell in the following class.

```
class MonoidC a where
  binop :: a -> a -> a
  neutral :: a
```

1. Consider the following instances of monoids:
   - Numbers with multiplication as the binary operation.
   - Boolean values with logical AND as the binary operation.
   - Lists, where the binary operation is concatenation.

   In all three instances, the choice of the neutral element can be deduced from the monoid laws. Define the described `MonoidC` instances for `Integer`, `Bool` and `[a]`. (1 point)

2. We consider simple voting sequences to keep track of votes on our social media posts, where the letter U indicates an upvote (+1), and the letter D indicates a downvote (-1). For instance, `"UU"`, `"UUUD"` and `"UDDUUU"` represent a total vote of +2, whereas `"DDD"` and `"DDUDD"` represent a total vote of -3.

   Define a function `normaliseVote` that simplifies any voting sequence so that it does not contain any adjacent canceling pairs (`"UD"` or `"DU"`). All sequences that represent the same total vote should normalize to the same form. For instance, all sequences that represent a total vote of +2 should normalise to `"UU"`. (1 point)

3. We define a datatype `VoteSeq` with constructor `VS :: String -> VoteSeq` to represent these voting sequences. Make `VoteSeq` an instance of `Eq`, `Show`, and `MonoidC` (with total vote addition as the binary operation). Ensure that the result of any addition is normalized. Moreover, `VS xs == VS ys` should return `True` whenever `xs` and `ys` represent the same total vote. The `show` function should just return the internal string representation. (1 point)

4. Define a function `combine` that takes a list of elements $x_1, \ldots, x_n$ in a monoid, and computes the combination of these elements, i.e., $x_1 \circ \ldots \circ x_n \circ e$. The function definition should include the (most general) type of `combine`. (1 point)

5. After completing the previous tasks,
   - describe informally what the function `combine :: [Integer] -> Integer` computes,
   - describe informally what the function `combine :: [String] -> String` computes,
   - describe why `combine [[4,2,0]]` and `combine [4,2,0]` differ, and
   - explain the difference between the result of `combine [VS "UUDDU", VS "UDDDU", VS "DDDU"]` and that of `combine ["UUDDU", "UDDDU", "DDDU"]`. (1 point)