universität innsbruck

Functional Programming	WS 2023/2024	LVA 703025
Exercise Sheet 6, 10 points		Deadline: Tuesday, November 21, 2023, 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 .hs file in OLAT.
- 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-4.18.0.0/docs/Prelude.html#t:Num` for a detailed description of this type class. (2 points)

Solution 1

```haskell
data Rat = Rat Integer Integer

normaliseRat :: Rat -> Rat
normaliseRat (Rat n d) = normaliseSign (n `div` g) (d `div` g) where
  g = gcd n d
  normaliseSign n d
    | d < 0 = Rat (negate n) (negate d)
    | otherwise = Rat n d

instance Eq Rat where
  Rat n1 d1 == Rat n2 d2 = n1 * d2 == n2 * d1

instance Ord Rat where
  r1 <= r2 = let
    Rat n1 d1 = normaliseRat r1
    Rat n2 d2 = normaliseRat r2
    in n1 * d2 <= n2 * d1

instance Show Rat where
  show r = pretty (normaliseRat r) where
    pretty (Rat n d)
      | d == 1 = show n
      | otherwise = show n ++ "/" ++ show d

createRat n d = normaliseRat (Rat n d)

instance Num Rat where
  Rat n1 d1 + Rat n2 d2 = createRat (n1 * d2 + n2 * d1) (d1 * d2)
  Rat n1 d1 * Rat n2 d2 = createRat (n1 * n2) (d1 * d2)
  negate (Rat n d) = Rat (negate n) d
  fromInteger i = Rat i 1
  abs (Rat n d) = Rat (abs n) (abs d)
  signum r
     | r < 0 = -1
     | r > 0 = 1
     | otherwise = 0
```

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.

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

1. One instance of a monoid can be obtained by using addition on numbers as the binary operation. Moreover, lists form a monoid where the binary operation is the append function. In both instances, the choice of the neutral element can be deduced from the monoid laws.
   Define the described `MonoidC` instances for `Double`, `Integer` and `[a]`. (1 point)

2. We consider extended tally lists to represent numbers, where a plus sign indicates +1, and a minus sign indicates -1. For instance, `"++"`, `"+++-"` and `"+--+++"` represent the number 2, whereas `"---"` and `"--+--"` represent the number -3.

Define a function to normalise such tally lists, such that after normalisation the tally lists do not contain any sublist of the form `"+-"` or `"-+"`. For instance, all tally lists that represent the number 2 should normalise to `"++"`. (1 point)

3. We define a datatype `Tally` with constructor `PM :: String -> Tally` to represent extended tally lists. Make `Tally` an instance of `Eq`, `Show`, and `MonoidC` (with tally list addition as the binary operation). In the latter case, make sure that the result of an addition is normalised. Moreover, `PM xs == PM ys` should return `True` whenever `xs` and `ys` represent the same number. 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 [[1,3,7]]` and `combine [1,3,7]` differ, and
   - explain the difference between the result of `combine [PM "++--+", PM "+---+", PM "---+"]` and that of `combine ["++--+", "+---+", "---+"]`. (1 point)

Solution 2

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

instance MonoidC Integer where
  binop = (+)
  neutral = 0

instance MonoidC Double where
  binop = (+)
  neutral = 0

instance MonoidC [a] where
  binop = (++)
  neutral = []


data Tally = PM String

normalise :: String -> String
normalise "" = ""
normalise (x : xs) = case normalise xs of
  y : ys -> if x == y then x : y : ys else ys

instance Eq Tally where
  PM xs == PM ys = normalise xs == normalise ys

instance Show Tally where
  show (PM xs) = show xs

instance MonoidC Tally where
  binop (PM xs) (PM ys) = PM (normalise (xs ++ ys))
  neutral = PM []

combine :: MonoidC a => [a] -> a
combine [] = neutral
combine (x : xs) = x `binop` combine xs

{-
A) combine :: [Integer] -> Integer is a function that computes the sum of a list of integers
B) combine :: [String] -> String, or more generally combine :: [[a]] -> [a]
  is a function that concatenates a list of lists into a single list
C) because of B), combine [[1,3,7]] is just [1,3,7], and combine [1,3,7] = 11 because of A)
D) combine [PM "++--+", PM "+---+", PM "---+"] is the addition of three tally lists, which results
   in PM "--", whereas combine ["++--+", "+---+", "---+"] is just string concatenation, i.e.,
   the result is "++--++---+---+".
-}
```