universität innsbruck

Functional Programming	WS 2021	LVA 703025
Exercise Sheet 12, 10 points		Deadline: Wednesday, January 26, 2022, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_12.tgz` provided on the proseminar page.
- Your .hs-file(s) should be compilable with ghci and be uploaded in OLAT.

Exercise 1 *Cyclic Lists* **5 p.**

We say that a number n is *special* if and only if it satisfies one of the following two conditions:

- $n = 1$, or
- there is some special number m such that $n = 3m$ or $n = 7m$ or $n = 11m$

The aim of this exercise is to compute the infinite list of all special numbers in ascending order.

1. Write a function `merge` that merges two lists into one. `merge xs ys` should fulfill the following conditions:
   - All elements in `merge xs ys` are also elements from `xs` or `ys`.
   - If `xs` and `ys` are in ascending order and contain no duplicates, then `merge xs ys` is in ascending order and contains no duplicates.

   Example: `merge [1,18,200] [19,150,200,300] = [1,18,19,150,200,300]` (1 point)

2. Define the infinite list `sNumbers` that computes the infinite list of special numbers in ascending order without duplicates as a cyclic list.

   Hint: Use the function `merge` and functions like `map (3*)`. Also have a look at the definition of `fibs` on slide 7 of lecture 12.

   Example: `take 10 sNumbers = [1,3,7,9,11,21,27,33,49,63]` (2 points)

3. Convince yourself that the computation of special numbers is not that easy and also not that efficient without infinite lists: implement a function `sNum :: Int -> Integer` where `sNum i` computes the `i`-th special number, i.e., `sNum i == sNumbers !! i`, where the implementation of `sNum` must not use lists, and compare the execution times of `sNum 200` and `sNumbers !! 200`.

   Hint: Try to define a predicate that tests whether a number is special; a special number has a prime factorization of a very specific shape. (2 points)

Solution 1

```
merge (x:xs) (y:ys)
  | x == y = x : merge xs ys
  | x < y  = x : merge xs (y:ys)
  | otherwise  = y : merge (x:xs) ys
merge [] ys = ys
merge xs [] = xs

sNumbers = 1 : merge (merge l3 l7) l11
  where l3 = map (3*) sNumbers
        l7 = map (7*) sNumbers
        l11 = map (11*) sNumbers

deleteMultiple m x
  | x `mod` m == 0 = deleteMultiple m (x `div` m)
  | otherwise = x

isSNumber = (1 ==) . deleteMultiple 11 . deleteMultiple 7 . deleteMultiple 3

sNum :: Int -> Integer
sNum = go 1 where
  go x n
    | isSNumber x = if n == 0 then x else go (x + 1) (n - 1)
    | otherwise = go (x + 1) n
```

A number is special iff its prime factorization only contains the numbers 3, 7 and 11.
The sequence of integers defined in `sNumbers` is a variant of the *Hamming numbers* which are all numbers whose prime factorization only contains the numbers 2, 3 and 5.[1]
The computation time of `sNum 200 = 6417873` requires around 22 seconds, whereas `sNumbers !! 200` is done immediately. The main problem in the computation of `sNum n` is that all intermediate numbers between `1` and `sNum n` are explicitly tested, whether they are special or not. By contrast, non-special numbers are never created in the definition of `sNumbers`.

Exercise 2 *Partitions* **5 p.**

In this exercise, you will develop an abstract datatype to represent *partitions*. Given a set A with n elements, a partition is a set of disjoint sets whose union is A. In other words: a partition distributes the n elements of A into different groups, where each element is a member of exactly one group.
In Haskell, we want to have a type `Partition a` that represents a partition of some (finite) set of elements of type A. It must support at least the following operations:

Discrete partition Given a set A, return the *discrete partition* over that set, i.e. the partition in which every element is its own group. (e.g. the discrete partition of $\{1, 2, 3\}$ is $\{\{1\}, \{2\}, \{3\}\}$.

Relatedness Given a partition and two elements, determine whether two elements are *related* (i.e. they are in the same group).

Representative Given a partition and one element that is in it, return a canonical representative of that element's group. "Canonical" means that if two elements are related, they must be assigned the same representative.

Joining Given a partition and two elements, merge the two elements' groups into one single group (see Figure 3).

To make your life easier, you may assume an `Ord` instance on the value type `a`.

1. Create an abstract datatype for partitions. If your type has any invariants, document them. For example, if you were to write an abstract datatype for natural numbers, a reasonable representation would be `data Nat = Nat Integer` with the invariant that the integer be non-negative.

[1] See https://en.wikipedia.org/wiki/Regular_number and https://rosettacode.org/wiki/Hamming_numbers

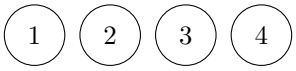

Figure 1: The discrete partition of the set $\{1, 2, 3, 4\}$.

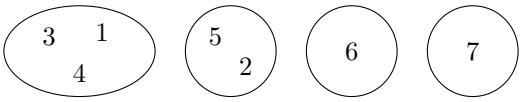

Figure 2: The partition $\{\{1, 3, 4\},\ \{2, 5\},\ \{6\}, \{7\}\}$, which partitions the set $\{1, 2, 3, 4, 5, 6, 7\}$.

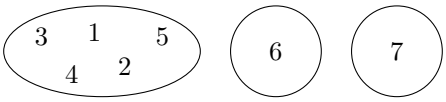

Figure 3: The partition from Figure 2 after elements 4 and 5 have been joined.

Also provide an instance of `Eq` and a function `toLists :: Ord a => Partition a -> [[a]]` that returns a list of all the groups (the order is irrelevant, but no duplicate elements are allowed).

Be careful to ensure that your implementation of `==` actually returns `True` for any two partitions that are equal in the mathematical sense.

The code template already provides a function `pretty :: Ord a => Partition a -> String` that allows you to print your partitions in mathematical notation. (2 points)

2. Implement the functions

```
discrete :: Ord a => [a] -> Partition a
representative :: Ord a => a -> Partition a -> a
related :: Ord a => a -> a -> Partition a -> Bool
join :: Ord a => a -> a -> Partition a -> Partition a
```

For the `discrete` operation, you may assume that the input list contains no duplicate elements. If any of the other functions are given an element that is not in the partition, the result may be whatever you want (including a crash). (2 points)

Example:

```
testPartition = join 1 3 $ join 2 5 $ join 1 4 $ discrete [1..7]

toLists $ discrete [1..4] = [[1],[2],[3],[4]]
toLists testPartition == [[1,3,4],[2,5],[6],[7]]
related 1 3 testPartition == True
related 1 5 testPartition == False
representative 3 testPartition == 1   -- or 3 or 4
toLists $ join 4 5 testPartition == [[1,2,3,4,5],[6],[7]]
```

Note: The order of the lists returned by `toLists` is up to you and may be different from the one in the above examples.

3. As an application of your abstract datatype, the code template contains an implementation of *Kruskal's algorithm* for computing the minimal spanning forest of a weighted undirected graph. This algorithm works like this:

   - Start with the discrete partition of the graph's vertices
   - Traverse the edges of the graph by order of ascending weights
   - For every edge, check if the two vertices are related (i.e. in the same group of the partition). If not, add the edge to the result and join the two vertices in the partition.

   In other words, the algorithm uses your `Partition` datatype to keep track of which of the vertices are connected already. Every group in the partition corresponds to a connected component of the forest that has been built up so far.

   Use the tests in the code template to check that the algorithm works correctly with your abstract datatype. (1 point)

Solution 2

```haskell
-- Invariant: all the lists are sorted and disjoint.
newtype Partition a = Partition [[a]]
  deriving Show

-- Two partitions are equal iff they contain exactly the same groups
-- Note that in our representation, the groups need not be in the same order,
-- so we sort them before comparing them.
instance Ord a => Eq (Partition a) where
  Partition xss == Partition yss = sort xss == sort yss

toLists :: Ord a => Partition a -> [[a]]
toLists (Partition xss) = xss

pretty :: (Show a, Ord a) => Partition a -> String
pretty = braces . intercalate ", " . map (braces . intercalate ", " . map show) . sort .
map sort . toLists
  where braces s = "{" ++ s ++ "}"


-- The discrete partition in which no two elements are related.
discrete :: Ord a => [a] -> Partition a
discrete xs = Partition [[x] | x <- xs]

-- Returns the group of an element
lookup :: Ord a => a -> Partition a -> [a]
lookup x (Partition xss) = go xss
  where go [] = []
        go (ys : yss) = if x `elem` ys then ys else go yss

-- Returns the canonical representative for the given element's group. We simply choose the minimum,
-- which is possible since we have an "Ord" instance. Because we always keep the groups sorted, the
-- minimum is simply the first element.
representative :: Ord a => a -> Partition a -> a
representative x part = head (lookup x part)

-- Determines whether two elements are related
related :: Ord a => a -> a -> Partition a -> Bool
related x y part = representative x part == representative y part

-- makes two elements related, i.e. joins their groupes
join :: Ord a => a -> a -> Partition a -> Partition a
join x y part@(Partition xss) =
  if y `elem` groupX then part else Partition $ sort (groupX ++ groupY) : xss'
  where xss' = filter (\xs -> not (x `elem` xs) && not (y `elem` xs)) xss
        groupX = lookup x part
        groupY = lookup y part
```