universität innsbruck

Functional Programming	WS 2023/2024	LVA 703025
Exercise Sheet 12, 10 points		Deadline: Tuesday, January 23, 2024, 8pm

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_12.tgz` provided on the proseminar page.
- Upload your solutions in OLAT.
- Your `*.hs` files must be compilable with `ghc`.

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 in `xs` or `ys`.
   - All elements in `xs` or `ys` are also elements in `merge xs 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 of `sNum 200 = 6417873` requires around 22 seconds, whereas `sNumbers !! 200` is done almost immediately. The main problem in the computation of `sNum n` is that **all** intermediate numbers between `1` and `sNum n` are explicitly tested, regardless of whether they are special or not, whereas non-special numbers are never created in the definition of `sNumbers`.

Exercise 2 *Sets* **5 p.**

In this exercise, we consider an abstract datatype to represent *sets* with the following (minimalistic) interface:

```
insert :: Eq a => a -> Set a -> Set a -- insertion of a single element
empty :: Set a                        -- the empty set
delete :: Eq a => a -> Set a -> Set a -- deletion of an element from a set
member :: Eq a => a -> Set a -> Bool  -- testing whether an element is in a set
foldSet :: (a -> b -> b) -> b -> Set a -> b
```

Note that for deletion, it is not required that the deleted element is in the set, similar to the mathematical definition of a set where $\{1,2,3\} \setminus \{4\} = \{1,2,3\}$ and does not give rise to an error.
Folding over a set should satisfy the property `foldSet f e` $\{x_1, \ldots, x_n\}$ `= f` x_1 `(f` x_2 $\ldots$`(f` x_n `e)` $\ldots$`)`.
For example, if `s` represents the set $\{1,2,3\}$, then `foldSet f e s` may evaluate to `f 1 (f 2 (f 3 e))` or `f 3 (f 1 (f 2 e))` or even `f 1 (f 2 (f 3 (f 2 e)))`, since $\{1,2,3\} = \{3,1,2\} = \{1,2,3,2\}$.

1. We have provided an initial implementation of sets in the module `ListSet` in `template_12.tgz`. Write a separate module `SetMore` that imports `ListSet` and provides the following additional operations on sets: (3 points)

   ```
   union :: Eq a => Set a -> Set a -> Set a          -- a) 1 point
   intersection :: Eq a => Set a -> Set a -> Set a   -- b) 1 point
   isEmpty :: Set a -> Bool                          -- c) 1 point
   ```

   You may *not* modify `ListSet`. You can find a test application in module `Main`.

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

2. Provide a better implementation of the abstract set interface than `ListSet`, e.g., one that is based on lists without duplicates or sorted lists. You may change `Eq a` into `Ord a` if desired. Also, provide an `Eq` instance for your set implementation.

   Replace the import of `ListSet` by your new module in `SetMore` and in the test application in `Main`, and analyse the performance difference between the two versions. (2 points)

Solution 2

```haskell
-- SetMore
module SetMore where

import ListSet

union :: (Eq a) => Set a -> Set a -> Set a
union a b = foldSet insert a b

intersection :: (Eq a) => Set a -> Set a -> Set a
intersection a b = foldSet (\x s -> if x `member` a then insert x s else s) empty b

isEmpty :: Set a -> Bool
isEmpty = foldSet (\_ _ -> False) True


-- BetterSet

module BetterSet (Set, insert, delete, foldSet, empty, member) where

newtype Set a = Set [a] deriving (Show)

-- invariant: lists contain no duplicates

insert :: (Eq a) => a -> Set a -> Set a
insert x (Set xs)
  | x `elem` xs = Set xs
  | otherwise = Set (x : xs)

delete :: (Eq a) => a -> Set a -> Set a
delete x (Set xs) = case span (/= x) xs of
  (ys, _ : zs) -> Set (ys ++ zs)
  _ -> Set xs

foldSet :: (a -> b -> b) -> b -> Set a -> b
foldSet f e (Set xs) = foldr f e xs

member :: (Eq a) => a -> Set a -> Bool
member x (Set xs) = x `elem` xs

empty :: Set a
empty = Set []

instance (Eq a) => Eq (Set a) where
  (Set xs) == (Set ys) = length xs == length ys && all (`elem` ys) xs
```

The new set implementation is slower when it comes to inserting new elements into the set. However, the other operations become faster, since the representation does not contain duplicates anymore.