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)

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`.

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)