Till Mossakowski
Benjamin Junge

3. Excercise sheet

Issued: 2023-10-23
Due: 2023-11-06&2023-11-07

3.1 *Splitting Lists*

Implement the functions

```
splitL :: Int → [a] → [[a]]
mSplitL :: [Int] → [a] → [[a]]
```

without using splitAt from the Prelude.

`splitL i x` splits a list x into two lists at element i s.t. splitL i x = $[[x_0, .., x_i], [x_{i+1}, .., x_{n-1}]]$ if x has n elements, $i \leq n$.

`mSplitL i x` splits a list x at each index given by list i.

The following cases should hold:

```
concat(msplitL ints lst) = lst
length(mSplitL ints lst) = length(ints) + 1
mSplitL [1, 3, 5] "foobar" == [ "fo" , "ob" , "ar" , [] ]
```

3.2 *Sorting*

1. Implement the function

   ```
   sort :: Ord a ⇒ [a] → [a]
   ```

   for sorting lists using a sorting algorithm of your choice.

2. Implement instances Ord and Eq for the ListInt type of exercise sheet 2 such that the following holds:

   ```
   sort([create [1,1,1,1,1], create [4,5,7], create [1,2,3]])
   ⇝ [ListInt [1,2,3] 6, ListInt [4,5,7] 16, ListInt [1,1,1,1,1] 5]
   ```

3.3 *Components and Products*

The Component type is given by **data** `Component a = Component a Natural`. Each component has a description `a` and the amount that is stored within a storage.
The Storage type is given by **type** `Storage a = [Component a]`. A storage never contains two entries with the same description.
The Product type is given by **data** `Product a b = Product a [(b, Int)]` with `a` being the product name and `[(b, Int)]` being a list of components and their amounts needed to create one product of that type.
Two products are equal if their names are equal, likewise two components are equal if their descriptions are equal.

Implement the following control functions for the type Storage:

```
contains :: Eq a ⇒ Storage a → a → Maybe Natural
store :: Eq a ⇒ Storage a → a → Natural → Storage a
remove :: Eq a ⇒ Storage a → a → Natural → Storage
```

contains returns the amount of a component in a storage if it exists.
store ads a component to a storage or ads its amount to an existing entry of the same component.
remove reduces the amount of a given component in a given storage.

Next implement the following functions for producing products from a given storage:

```
isProducible :: (Eq a, Eq b) ⇒ Product a b → Storage a → Bool
produce :: (Eq a, Eq b) ⇒ Product b a → Storage a → Storage a
```

isProducible checks if the storage has enough of the needed components.
produce removes the used components from the storage.

To use the type Natural use ***import*** *Numeric.Natural*

3.4 *Polymorphism Quirks*

1. Evaluate the following expressions with ghci:

   ```
   "" == []
   tail [1] = ""
   tail [1] = []
   ```

   Compare the results and explain possible differences.

2. the read function read :: Read a ⇒ String → a of the typeclass Read is a quasi inverse to the show function of the typeclass show. Evaluate the following expressions with ghci:

   ```
   read "10" == 10
   read "10"
   ```

   Compare the two results and describe the reason for the second expression to produce an error. Describe a way to alter the expression read "10"to stop the error form occurring.

3.5 **Sets*

1. Define the type **data** Set a = Set [a] and implement the following functions to access them:

   ```
   createSet :: Eq a ⇒ [a] → Set a
   union :: Eq a ⇒ Set a → Set a → Set a
   intersection :: Eq a ⇒ Set a → Set a → Set a
   setMinus :: Eq a ⇒ Set a → Set a → Set a
   ```

   createSet turns a list into a Set. Union, intersection and setMinus implement the corresponding standard operations on sets.

2. Implement instances Ord and Eq for Set a such that x < y if $x \subset y$.

   Implement an instance of Show such that:

   ```
   show(createSet [1,2,3,4]) ⇝ {1,2,3,4}
   show(createSet[createSet[1,2],createSet[3,4]]) ⇝ {{1,2},{3,4}}
   ```

   hint: ***instance*** *Show a ⇒ Show Set a* ***where***