Functional Programming SuSe19
Exercise Sheet 2 (due Wednesday, 08.05.2019)

Prof. Dr. J. Giesl M. Hark

Notes:

- To solve the programming exercises you should use the Glasgow Haskell Compiler **GHC**, available for free at `https://www.haskell.org/ghc/`. You can use the command "`ghci`" to start an interactive interpreter shell.
- Please solve these exercises in **groups of four**!
- The solutions must be handed in **directly before (very latest: at the beginning of)** the exercise course on Wednesday, 08.05.2019, 14:30, in lecture hall **AH I**. Alternatively you can drop your solutions into a box which is located right next to Prof. Giesl's office (until 30 minutes before the exercise course starts). Also, please **print** the source code of your solutions for the programming exercises.
- In addition, please upload the source code of your solutions for the programming exercises in a single **ZIP**-archive via `RWTHmoodle` before the exercise course on Wednesday, 08.05.2019, 14:30. Please name your archive `Sheet_i_Mat1_Mat2_Mat3_Mat4.zip`, where `i` is the number of the sheet and `Mat_1`...`Mat_4` are the immatriculation numbers of the group members. It is sufficient if **one** of the group members uploads your solution. Files, which are not accepted by **GHCi**, will not be marked.
- Please write the **names** and **immatriculation numbers** of all students on your solution. Also please staple the individual sheets!

In all exercises, if not stated otherwise, also give the type declarations of the functions to implement. Moreover, you can always use functions from previous parts of the exercise, even if you did not manage to solve them. You can also write auxiliary functions.

Programming Exercise 1 (Data Types): (1.5 + 2 + 1.5 + 3 + 2 = 10 points)

In this exercise we consider priority queues storing arbitrary data:

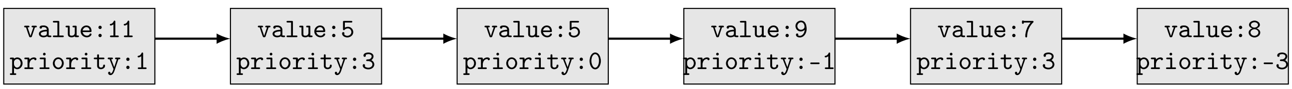

Figure 1: Priority queue storing values of type `Int`.

a) Give a definition for a data type `PriorityQueue` for priority queues storing data of arbitrary type. The type `PriorityQueue` should have the two constructors: `Push`, for inserting an element with a priority of type `Int` into the priority queue, and `EmptyQueue`, for the empty queue. Furthermore, define a variable `p` of type `PriorityQueue Int` reflecting the queue as shown in Fig. 1.

b) Write a function `isWaiting` that gets an element of type `a` and a queue of type `PriorityQueue a` and returns `True` if and only if the element is stored in the queue. For example, `isWaiting 5 p == True` and `isWaiting 6 p == False`. The function should be applicable for as many types `a` as possible.

Hints:

- Remember that the predefined type class `Eq` contains all types providing the equality operator `==`.

c) Write a function `fromList` that given a list of type `[(a,Int)]` computes a priority queue of type `PriorityQueue a` such that for every element `(x,n)` of the list the resulting queue contains the element `x` with priority `n`. For example, `fromList [(11,1), (5,3), (5,0), (9,-1), (7,3), (8,-3)]` should yield an expression of type `PriorityQueue Int` as in Fig. 1.

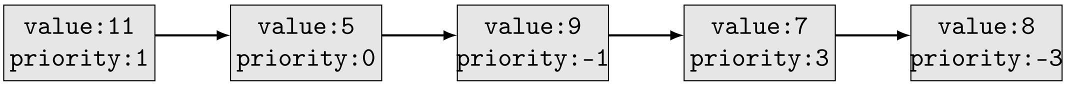

Figure 2: Reduced priority queue from Fig. 1.

d) Write a function `pop` that given a nonempty priority queue `queue` returns a pair `(x,q)` where the first entry `x` is the value with highest priority in `queue` and the second entry `q` is `queue` where the element with value `x` and the highest priority is deleted. For example `pop p` should yield a pair `(5,q)`, where `q::PriorityQueue Int` corresponds the queue given in Fig. 2.

Hints:

- If `x` occurs several times with the highest priority in `queue`, then only one of the corresponding elements should be deleted in `q`.
- If `queue` contains several values with the highest priority, then your implementation should choose one of them. So `pop p` could also yield `(7,q')`, where `q'` results from `p` by deleting the element with value `7` and priority `3`.
- You may need `minBound :: Int`, the smallest `Int`, and `max :: Int -> Int -> Int`.

e) Write a function `toList` that given a `PriorityQueue` returns a list containing the values from the queue sorted in decreasing priority. For example, `toList p == [5,7,11,5,9,8]`.

Programming Exercise 2 (Type Classes): (1 + 2.5 + 1.5 + 2 = 7 points)

a) Consider the type `List a` from the lecture that is defined as follows:

```
data List a = Nil | Cons a (List a) deriving Show
```

Declare `List a` as an instance of the type class `Eq` whenever `a` is an instance of `Eq`. Implement the method `(==)` such that it computes equality between lists entrywise.

b) Give a declaration for a type class `Mono` for monoids as a subclass of `Eq` with the following methods:

- `binOp :: a -> a -> a` (for "**bin**ary **op**eration").
- `one :: a` (the neutral element of the monoid).
- `pow :: Word -> a -> a` (for "**pow**er)"). `Word` is the predefined type for nonnegative integers. Here, for an object `x` of type `a` and a number `n :: Word` the expression `pow n x` should stand for `binOp x (binOp x ...(binOp x one))` with `n` occurences of `binOp`. So `pow 0 x == one`.

The class declaration should contain a default implementation for `pow`. In contrast, the functions `binOp` and `one` have to be implemented in the instances of the type class `Mono`.

c) Declare the built-in type `Integer`[1] and `List a` from a) as instances of the type class `Mono` for as many types `a` as possible. For `Integer` the binary operation should be `(*)` and the neutral element is `1`. For `List a` the binary operation is concatenation of the lists where the empty list is the neutral element.

For example `binOp (-3) 2 = -6` and `binOp (Cons 'a' Nil) (Cons 'b' (Cons 'c' Nil)) = Cons 'a' (Cons 'b' (Cons 'c' Nil))`.

d) For any monoid `a`, implement a function `multiply` that given a list of type `[(Word,a)]` multiplies the powers of the pairs from the list, i.e., `multiply [(`x_1`,`y_1`),(`x_2`,`y_2`),...,(`x_n`,`y_n`)] == binOp (pow `x_1` `y_1`) (binOp (pow `x_2` `y_2`) (binOp ...(binOp (pow `x_n` `y_n`) one)))`. The empty product `multiply []` is defined to be the neutral element of the monoid `a`.

For example `multiply [(3,Cons 'a' Nil),(1,Cons 'b' Nil),(2,Cons 'c' (Cons 'd' Nil))] == Cons 'a' (Cons 'a' (Cons 'a' (Cons 'b' (Cons 'c' (Cons 'd' (Cons 'c' (Cons 'd' Nil)))))))`.

[1] `Integer` is a type for arbitrary large intger numbers, whereas `Int` represents integers of fixed range (implementation defined, at least 30 bit).

Programming Exercise 3 (Using Higher-Order Functions): (2 + 2 = 4 points)

In this exercise you may **not** use any predefined functions except `map`, `foldr`, `filter`, `+`, constructors and comparisons. Also, you may **not** use explicit recursion.

a) Implement a function `removeDuplicates :: Eq a => [a] -> [a]` that given a list of an instance of `Eq` computes a list that contains exactly the same elements as the input list but only with a single occurrence. For example `removeDuplicates [1,1,2,1,-1,1,1,2,3,1] == [1,2,-1,3]`.

Hints:

- The order of the resulting list is irrelevant.

b) Implement a function `differentDigits :: Int -> Int` that counts the number of different digits occurring in the decimal representation of an integer. For example, `differentDigits 08052019 == 6` and `differentDigits 111231112111 == 3`.

Hints:

- The function `show :: Int -> String` converts an integer into its decimal string representation. You are allowed to use it in this subexercise.

Programming Exercise 4 (Defining Higher-Order Functions): (2 + 2 = 4 points)

Consider the following data type which represents univariate polynomials with arbitrary coefficients:

```
data Polynomial a = Coeff a Int (Polynomial a)| Null deriving Show
```

With `Null` we denote the zero polynomial und `Coeff c n p` represents the polynomial $c \cdot x^n + p$.
For example, the polynomial $4 \cdot x^3 + 2 \cdot x + 5$ with integer coefficients is represented by the term `q` with `q = Coeff 4 3 (Coeff 2 1 (Coeff 5 0 Null))` and type `q :: Polynomial Int`.

a) Write a function `foldPoly :: (a -> Int -> b -> b) -> b -> Polynomial a -> b` that behaves similar to the `foldr` function on lists, i.e., `foldPoly f e p` replaces every occurrence of the constructor `Coeff` by `f` and every occurrence of `Null` by `e` in `p`. For example, `foldPoly (\c n m -> c *3^n + m) 0 q` evaluates to `4 * 3^3 + 2 * 3 + 5 == 119`.

b) Write a function `degree :: Polynomial Int -> Int` that computes the degree of a polynomial with integer coefficients. We define `degree Null == minBound`, where `minBound :: Int` is the smallest integer on your system, as a placeholder for $-\infty$. The degree of a nonzero polynomial is the maximal power of x occurring with a nonzero coefficient. For example the degree of $4 \cdot x^3 + 2 \cdot x + 5$ is 3. For simplicity, we assume that if a polynomial contains the expression `...Coeff c n ...` then `n` does not occur as the second argument of `Coeff` again. You may **not** use any predefined functions except comparisons. You may **not** use explicit recursion. You can use `foldPoly` from the previous exercise.