Functional Programming
Excercise Sheet 2

Emilie Hastrup-Kiil (379455), Julian Schacht (402403),
Niklas Gruhn (389343), Maximilian Loose (402372)

Excercise 1

a)

```
data PriorityQueue a = Push a Int (PriorityQueue a) | EmptyQueue
   deriving Show

p :: PriorityQueue Int
p = Push 11 1 (Push 5 3 (Push 5 0 (Push 9 (-1) (Push 7 3 (Push 8 (-3)
   EmptyQueue)))))
```

b)

```
isWaiting :: Eq a => a -> PriorityQueue a -> Bool
isWaiting _ EmptyQueue               = False
isWaiting x (Push v _ n) | x == v    = True
                         | otherwise = isWaiting x n
```

c)

```
fromList :: [(a,Int)] -> PriorityQueue a
fromList []           = EmptyQueue
fromList ((x,p):xs)   = Push x p (fromList xs)
```

d)

```
-- auxiliary functions
delete :: PriorityQueue a -> Int -> PriorityQueue a
delete EmptyQueue _   = EmptyQueue
delete (Push v p n) x = if x==p then n else (Push v p (delete n x))
findElement :: PriorityQueue a -> Int -> a
findElement (Push v p n) x = if x==p then v else findElement n x

highestPriority :: PriorityQueue a -> Int
highestPriority EmptyQueue   = minBound
highestPriority (Push v p n) = max p (highestPriority n)

--main function
pop :: PriorityQueue a -> (a,PriorityQueue a)
pop x = (findElement x h, delete x h)
    where h = highestPriority x
```

e)

```
toList :: PriorityQueue a -> [a]
toList EmptyQueue = []
toList q = x : toList y
   where (x,y) = pop q
```

Excercise 2

a)

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

instance Eq a => Eq (List a) where
  (==) Nil Nil = True
  (==) (Cons x xs) (Cons y ys)
    | x /= y = False
    | otherwise = xs == ys
  (==) _ _ = False
```

b)

```
class Eq a => Mono a where
  binOp :: a -> a -> a
  one :: a
  pow :: Word -> a -> a
  pow 0 _ = one
  pow n x = binOp x (pow (n-1) x)
```

c)

```
instance Mono Integer where
  binOp x y = x * y
  one = 1

instance Eq a => Mono (List a) where
  binOp (Cons x xs) ys = Cons x (binOp xs ys)
  binOp Nil ys = ys
  one = Nil
```

d)

```
multiply :: Mono a => [(Word, a)] -> a
multiply [] = one
multiply ((n,x):xs) = binOp (pow n x) (multiply xs)
```

Excercise 3

a)

```
removeDuplicates :: Eq a => [a] -> [a]
removeDuplicates xs = foldr dropAll xs xs
  where
    dropAll x ys = x : filter (/=x) ys
```

b)

```
differentDigits :: Int -> Int
differentDigits number = foldr count 0 uniqueDigits
  where
    uniqueDigits = removeDuplicates (show number)
    count _ n = n+1
```

Excercise 4

a)

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

q :: Polynomial Int
q = Coeff 4 3 (Coeff 2 1 (Coeff 5 0 Null))

foldPoly :: (a -> Int -> b -> b) -> b -> Polynomial a -> b
foldPoly f d Null = d
foldPoly f d (Coeff a b c) = f a b (foldPoly f d c)
```

b)

```
degree :: Polynomial Int -> Int
degree x = foldPoly (\ c n m -> if n > m then n else m) minBound x
```