

# Functional Programming

## Excercise Sheet 2

Emilie Hastrup-Kiil (379455), Julian Schlacht (402403),  
Niklas Gruhn (389343), Maximilian Loose (402472)

### 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 v1 n1) == (Cons v2 n2) = (v1 == v2) && (n1 == n2)
    _ == _                    = False
```

b)

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

c)

```
instance Mono Integer where
    binOp a b = a*b
    one = 1

instance Eq a => Mono (List a) where
    binOp x Nil = x
    binOp Nil x = x
    binOp (Cons a1 b1) (Cons a2 b2) = Cons a1 (binOp b1 (Cons a2 b2))
    one = Nil
```

d)

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

## Excercise 3

a)

```
neq :: Eq a => a -> [a] -> [a]
neq x xs = x : filter (/=x) xs

removeDuplicates :: Eq a => [a] -> [a]
removeDuplicates xs = foldr neq xs xs
```

b)

```
differentDigits :: Int -> Int
differentDigits x = foldr (+) 0 (map (\x -> 1) (removeDuplicates (show x)))
```

**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
```