III. Funktionale Programmierung

- 1. Prinzipien der funktionalen Programmierung
- 2. Deklarationen
- 3. Ausdrücke
- 4. Muster (Patterns)
- 5. Typen und Datenstrukturen
- **6. Funktionale Programmiertechniken:** Funktionen höherer Ordnung

Funktionen höherer Ordnung: `comp`

```
comp :: (b -> c) -> (a -> b) -> (a -> c)

comp f g = \x -> f (g x)
```

Argument vom Typ: **`(b -> c)`**

Ergebnis vom Typ: **`(a -> b) -> (a -> c)`**

Funktionen höherer Ordnung: curry

```haskell
plus :: (Int, Int) -> Int
plus (x, y) = x + y
```

uncurry ↑ ↓ curry

```haskell
plus :: Int -> Int -> Int
plus x y = x + y
```

```haskell
curry :: ((a,b) -> c) -> a -> b -> c
curry f = g
          where g x y = f (x,y)
```

```haskell
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry g = f
            where f (x,y) = g x y
```

Funktionen höherer Ordnung: `map`

```
suclist :: [Int] -> [Int]
suclist []        = []
suclist (x:xs)    = suc x : suclist xs
```

```
sqrtlist :: [Float] -> [Float]
sqrtlist []       = []
sqrtlist (x:xs)   = sqrt x : sqrtlist xs
```

suclist $[\, x_1, \ldots, x_n \,] = [\, \text{suc } x_1, \ldots, \text{suc } x_n \,]$

sqrtlist $[\, x_1, \ldots, x_n \,] = [\, \text{sqrt } x_1, \ldots, \text{sqrt } x_n \,]$

map g $[\, x_1, \ldots, x_n \,] = [\, g\ x_1, \ldots, g\ x_n \,]$

```
f :: [a] -> [b]
f []      = []
f (x:xs)  = g x : f xs
```

```
map :: (a -> b) -> [a] -> [b]
map g []      = []
map g (x:xs)  = g x : map g xs
```

Funktionen höherer Ordnung: `map`

```
suclist :: [Int] -> [Int]
suclist = map suc
```

```
sqrtlist :: [Float] -> [Float]
sqrtlist = map sqrt
```

suclist [x_1 ,...,x_n] = [suc x_1 ,..., suc x_n]
sqrtlist [x_1 ,...,x_n] = [sqrt x_1 ,..., sqrt x_n]
map g   [x_1 ,...,x_n] = [g x_1 ,..., g x_n]

```
f :: [a] -> [b]
f []     = []
f (x:xs) = g x : f xs
```

```
map :: (a -> b) -> [a] -> [b]
map g []     = []
map g (x:xs) = g x : map g xs
```

Funktionen höherer Ordnung: `filter`

```
dropEven :: [ Int ] -> [ Int ]
dropEven [] = []
dropEven (x:xs) |  odd  x    = x : dropEven xs
                | otherwise = dropEven xs
```

```
dropUpper :: [ Char ] -> [ Char ]
dropUpper [] = []
dropUpper (x:xs) | isLower x = x : dropUpper xs
                 | otherwise = dropUpper xs
```

vordefiniert im Modul **`Data.Char`**:

`import Data.Char`

```
f :: [ a ] -> [ a ]
f [] = []
f (x:xs) |     g   x   = x : f xs
         | otherwise = f xs
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter g [] = []
filter g (x:xs) | g x       = x : filter g xs
                | otherwise = filter g xs
```

Funktionen höherer Ordnung: `filter`

```
dropEven :: [ Int ] -> [ Int ]
dropEven = filter  odd
```

```
dropUpper :: [ Char ] -> [ Char ]
dropUpper = filter isLower
```

```
f :: [a] -> [a]
f [] = []
f (x:xs) |    g   x   = x : f xs
         | otherwise = f xs
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter g [] = []
filter g (x:xs) | g x        = x : filter g xs
                | otherwise = filter g xs
```

III. Funktionale Programmierung

- 1. Prinzipien der funktionalen Programmierung
- 2. Deklarationen
- 3. Ausdrücke
- 4. Muster (Patterns)
- 5. Typen und Datenstrukturen
- **6. Funktionale Programmiertechniken:** Unendliche Datenobjekte

Unendliche Datenobjekte

```
from :: Int -> [Int]
from x = x : from (x+1)
```

```
take :: Int -> [a] -> [a]
take 0    _    = []
take n (x:xs) = x : take (n-1) xs
```

```
     take 1 (from 5)

=    take 1 (5 : from (5+1))

=    5 : take (1-1) (from (5+1))

=    5 : take 0     (from (5+1))

=    5 : []

=    [5]
```

Sieb des Eratosthenes

1. Erstelle Liste aller natürlichen Zahlen ab 2.
2. Markiere die erste unmarkierte Zahl in der Liste.
3. Streiche alle Vielfachen der letzten markierten Zahl.
4. Gehe zurück zu Schritt 2.

```
drop_mult :: Int -> [Int] -> [Int]
drop_mult x xs = filter (\y -> mod y x /= 0) xs
```

```
dropall :: [Int] -> [Int]
dropall (x:xs) = x : dropall (drop_mult x xs)
```

```
primes :: [Int]
primes = dropall (from 2)
```

Sieb des Eratosthenes

```
primes = [2,3,5,7,11,13,17,19,23,29,31,...
```

```
take 5 primes = [2,3,5,7,11]
```

```
drop_mult :: Int -> [Int] -> [Int]
drop_mult x xs = filter (\y -> mod y x /= 0) xs
```

```
dropall :: [Int] -> [Int]
dropall (x:xs) = x : dropall (drop_mult x xs)
```

```
primes :: [Int]
primes = dropall (from 2)
```