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`

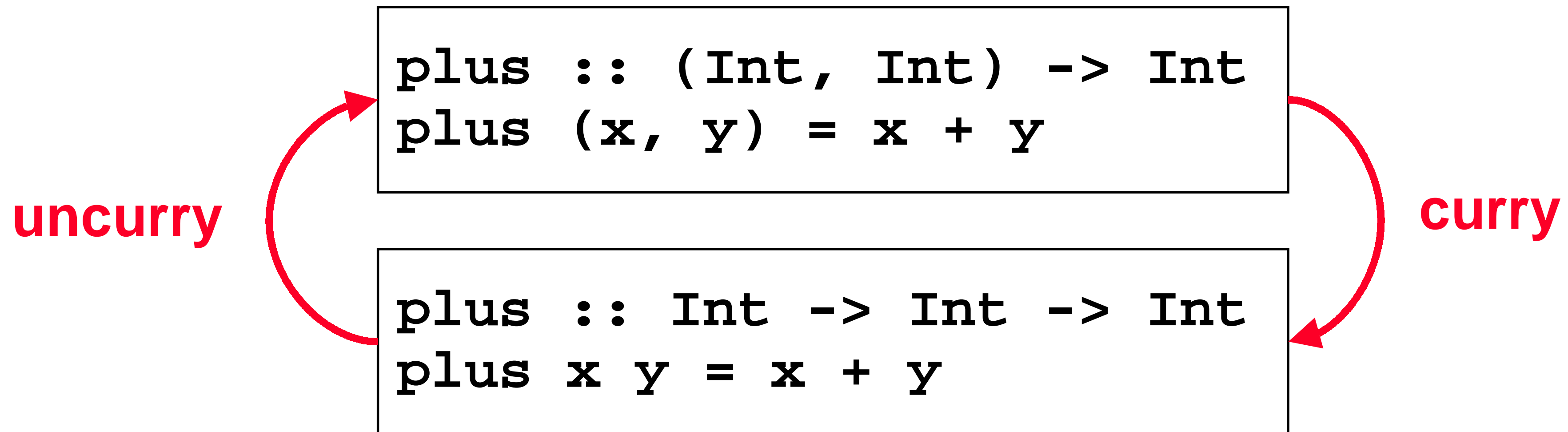
```
comp :: (b -> c) -> (a -> b) -> (a -> c)

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

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

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

Funktionen höherer Ordnung: `curry`




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

```
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 $[\,x_1, \ldots, x_n\,] = [\,\texttt{suc}\ x_1, \ldots, \texttt{suc}\ x_n\,]$

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

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]
suclist = map sqrt
```

suclist $[x_1, \ldots, x_n]$ = [suc x_1, ..., suc x_n]

sqrtlist $[x_1, \ldots, x_n]$ = [sqrt x_1, ..., sqrt x_n]

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

```
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
```