Einführung in die Funktionale Programmierung mit Haskell

Funktionen höherer Ordnung

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

29. Mai 2013

Planung

- 30. & 31. Mai: keine Veranstaltung
- 5. Juni: Übung statt Vorlesung

Vorläufige Planung:

Mi Vorlesung	Do Vorlesung	Fr Übung
17. Apr 13	18. Apr 13	19. Apr 13
24. Apr 13	25. Apr 13	26. Apr 13
1. Mai 13	2. Mai 13	3. Mai 13
8. Mai 13	9. Mai 13	10. Mai 13
15. Mai 13	16. Mai 13	17. Mai 13
22. Mai 13	23. Mai 13	24. Mai 13
29. Mai 13	30. Mai 13	31. Mai 13
5. Jun 13	6. Jun 13	7. Jun 13
12. Jun 13	13. Jun 13	14. Jun 13
19. Jun 13	20. Jun 13	21. Jun 13
26. Jun 13	27. Jun 13	28. Jun 13
3. Jul 13	4. Jul 13	5. Jul 13
10. Jul 13	11. Jul 13	12. Jul 13
17. Jul 13	18. Jul 13	19. Jul 13

Sollumfang: 3+2
Raumbuchung: 4+2

42 Termine, davon
3 Feiertage
3 Übung nachholen
6 Entfällt (1x?)
1 Klausurtermin
= 20V + 12Ü

Klausur: 19.7. 10h

Funktionstypen

Der Typ einer Funktion ist ein zusammengesetzter Funktionstyp, der immer aus genau zwei Typen mit einem Pfeil dazwischen besteht.

Funktionstypen sind implizit rechtsgeklammert, d.h. man darf die Klammern manchmal weglassen:

`Int -> Int -> Int` wird gelesen als `Int -> (Int -> Int)`

Entsprechend ist die Funktionsanwendung implizit linksgeklammert:

`bar 1 8` wird gelesen als `(bar 1) 8`

Das bedeutet: `(bar 1)` ist eine Funktion des Typs `Int -> Int`!
Funktionen sind also normale Werte in einer funktionalen Sprache!

Partielle Funktionsanwendung

Bei **partieller Funktionsanwendung** werden einer Funktion nicht alle Argumente übergeben. Das Ergebnis ist wieder eine Funktion.

Beispiel:

```
> let foo x y z = x * y + z
foo :: Num a => a ->  a ->  a  -> a
```

Partielle Funktionsanwendung

Bei **partieller Funktionsanwendung** werden einer Funktion nicht alle Argumente übergeben. Das Ergebnis ist wieder eine Funktion.

Beispiel:

```
> let foo x y z = x * y + z
foo :: Num a => a ->  a ->  a  -> a
> :type foo 1
foo 1 :: Num a => a ->  a  -> a
```

Partielle Funktionsanwendung

Bei **partieller Funktionsanwendung** werden einer Funktion nicht alle Argumente übergeben. Das Ergebnis ist wieder eine Funktion.

Beispiel:

```
> let foo x y z = x * y + z
foo :: Num a => a -> (a -> (a  -> a ))
> :type foo 1
foo 1 :: Num a => a -> (a  -> a)
> :type foo 1 2
foo 1 2 :: Num a => a -> a
```

Partielle Funktionsanwendung

Bei **partieller Funktionsanwendung** werden einer Funktion nicht alle Argumente übergeben. Das Ergebnis ist wieder eine Funktion.

Beispiel:

```
> let foo x y z = x * y + z
foo :: Num a => a -> (a -> (a  -> a ))
> :type foo 1
foo 1 :: Num a => a -> (a  -> a)
> :type foo 1 2
foo 1 2 :: Num a => a -> a
> let bar = foo 1 2
bar :: Integer -> Integer
```

Partielle Funktionsanwendung

Bei **partieller Funktionsanwendung** werden einer Funktion nicht alle Argumente übergeben. Das Ergebnis ist wieder eine Funktion.

Beispiel:

```
> let foo x y z = x * y + z
foo :: Num a => a -> (a -> (a  -> a ))
> :type foo 1
foo 1 :: Num a => a -> (a  -> a)
> :type foo 1 2
foo 1 2 :: Num a => a -> a
> let bar = foo 1 2
bar :: Integer -> Integer
> bar 3
5
```

Achtung: Nicht verwechseln mit partiellen Funktionen!

Partielle Funktionsanwendung

Bei **partieller Funktionsanwendung** werden einer Funktion nicht alle Argumente übergeben. Das Ergebnis ist wieder eine Funktion.

Es ist natürlich auch möglich, gar keine Argumente anzugeben.
Beispiel:

```
> let foo x y z = x * y + z
foo :: Num a => a -> a -> a -> a

> let bar = foo
bar :: Integer -> Integer -> Integer -> Integer
```

Bemerkung: Die hier von GHCI durchgeführte Spezialisierung des Typs von `Num a` nach `Integer` ist nicht zwingend. `foo` und `bar` dürfen auch den gleichen Typ haben.

Partielle Funktionsanwendung

Bei **partieller Funktionsanwendung** werden einer Funktion nicht alle Argumente übergeben. Das Ergebnis ist wieder eine Funktion.

Das Ergebnis ist insbesondere ein Wert.
Werte mit Funktionstyp sind Werte wie alle anderen auch.

Beispiel: Wir können Sie in listen packen.

```
> let myOps = [max,min,(+),(-),(*),(/)]
myOps :: [Double -> Double -> Double]

> head myOps 33 44
44.0

> (myOps !! 3) 100 11
89.0
```

Funktionen als Rückgabewert

```
> let divBy  x = \y -> (y/x)
divBy :: Fractional a => a -> a -> a

> let drittel = divBy 3
drittel :: Double -> Double
> drittel 9
3.0
> drittel 12
4.0
```

- Wert von Funktionstyp wird **Funktionsabschluss** genannt (engl.: closure) und besteht aus Kontext und Funktionsrumpf
  Kontext erklärt (schliesst offene) Variablen im Funktionsrumpf
- Werte im Kontext (wie überall in Haskell) unveränderlich, d.h. Funktion kann gefahrlos wiederverwendet werden
  $\Rightarrow$ Referentielle Transparenz

Funktionen als Rückgabewert

```
> let divBy  x = \y -> (y/x)
divBy :: Fractional a => a -> a -> a

> let drittel = divBy 3
drittel :: Double -> Double
> drittel 9
3.0
> drittel 12
4.0
```

- Wert von Funktionstyp wird **Funktionsabschluss** genannt (engl.: closure) und besteht aus Kontext und Funktionsrumpf
  Kontext erklärt (schliesst offene) Variablen im Funktionsrumpf
- Werte im Kontext (wie überall in Haskell) unveränderlich, d.h. Funktion kann gefahrlos wiederverwendet werden

  $\Rightarrow$ Referentielle Transparenz

Funktionen als Rückgabewert

```
> let divBy  x = \y -> (y/x)
divBy :: Fractional a => a -> a -> a

> let drittel = divBy 3
drittel :: Double -> Double
> drittel 9
3.0
> drittel 12
4.0
```

- Wert von Funktionstyp wird **Funktionsabschluss** genannt (engl.: closure) und besteht aus Kontext und Funktionsrumpf
  Kontext erklärt (schliesst offene) Variablen im Funktionsrumpf
- Werte im Kontext (wie überall in Haskell) unveränderlich, d.h. Funktion kann gefahrlos wiederverwendet werden

  $\Rightarrow$ Referentielle Transparenz

Funktionen als Rückgabewert

Alle diese Funktionsdefinitionen sind äquivalent:

```
divBy :: Fractional a => a -> (a -> a)

divBy_v1 x   = \y -> (y/x)
```

1. . . . mit anonymer Funktionsdefinition
2. . . . lokaler Funktionsdefinition
3. Partielle Anwendung von Infix-Funktionen mit Klammern ist auch möglich (**Sections**)
4. Vollständige Definition ist ja auch partiell anwendbar!

Funktionen als Rückgabewert

Alle diese Funktionsdefinitionen sind äquivalent:

```
divBy :: Fractional a => a -> (a -> a)

divBy_v1 x   = \y -> (y/x)
divBy_v2 x   = myDiv
  where myDiv y = (y/x)
```

1. . . . mit anonymer Funktionsdefinition
2. . . . lokaler Funktionsdefinition
3. Partielle Anwendung von Infix-Funktionen mit Klammern ist auch möglich (**Sections**)
4. Vollständige Definition ist ja auch partiell anwendbar!

Funktionen als Rückgabewert

Alle diese Funktionsdefinitionen sind äquivalent:

```
divBy :: Fractional a => a -> (a -> a)

divBy_v1 x   = \y -> (y/x)
divBy_v2 x   = myDiv
  where myDiv y = (y/x)
divBy_v3 x   = (/x)
```

1. . . . mit anonymer Funktionsdefinition
2. . . . lokaler Funktionsdefinition
3. Partielle Anwendung von Infix-Funktionen mit Klammern ist auch möglich (**Sections**)
4. Vollständige Definition ist ja auch partiell anwendbar!

Funktionen als Rückgabewert

Alle diese Funktionsdefinitionen sind äquivalent:

```
divBy :: Fractional a => a -> (a -> a)

divBy_v1 x   = \y -> (y/x)
divBy_v2 x   = myDiv
  where myDiv y = (y/x)
divBy_v3 x   = (/x)
divBy_v4 x y = (y/x)
```

1. . . . mit anonymer Funktionsdefinition
2. . . . lokaler Funktionsdefinition
3. Partielle Anwendung von Infix-Funktionen mit Klammern ist auch möglich (**Sections**)
4. Vollständige Definition ist ja auch partiell anwendbar!

Funktion als Argumente

Wir können Funktionstypen auch anders klammern:

```
twice :: (a -> a) -> (a -> a)
twice f = \x -> f (f x)

> twice (+3) 4
10

> twice reverse [1..3]
[1,2,3]

> twice ("ha " ++) "hi"
"ha ha hi"
```

Die Funktion `twice` nimmt also eine Funktion und wendet dieses zwei Mal an!

Funktion als Argumente

Wir können Funktionstypen auch anders klammern:

```
twice :: (a -> a) ->  a -> a
twice f = \x -> f (f x)
```

Die Funktion `twice` können wir natürlich auch wieder partiell anwenden, um eine neue Funktion zu erhalten:

```
> twice (twice (++"ha")) "Mua"
"Muahahahaha"
```

Dieses Muster können wir erneut abstrahieren und benennen:

```
quad :: (a -> a) ->  a -> a
quad f = twice (twice f)
> quad ('a':) "rgh!"
"aaaargh!"
```

Funktion als Argumente

Wir können Funktionstypen auch anders klammern:

```
twice :: (a -> a) ->  a -> a
twice f = \x -> f (f x)
```

Die Funktion `twice` können wir natürlich auch wieder partiell anwenden, um eine neue Funktion zu erhalten:

```
> twice (twice (++"ha")) "Mua"
"Muahahahaha"
```

Dieses Muster können wir erneut abstrahieren und benennen:

```
quad :: (a -> a) ->  a -> a
quad f = twice (twice f)
> quad ('a':) "rgh!"
"aaaargh!"
```

Funktion als Argumente

Wir können Funktionstypen auch anders klammern:

```
twice :: (a -> a) ->  a -> a
twice f = \x -> f (f x)
```

Die Funktion `twice` können wir natürlich auch wieder partiell anwenden, um eine neue Funktion zu erhalten:

```
> twice (twice (++"ha")) "Mua"
"Muahahahaha"
```

Dieses Muster können wir erneut abstrahieren und benennen:

```
quad :: (a -> a) ->  a -> a
quad f = twice (twice f)
> quad (quad ('a':)) "rgh!"
"aaaaaaaaaaaaaaaargh!"
```

Funktionen als Argumente

Ganzzahliges Maximum einer Funktion in einem Bereich bestimmen

```
fmax :: (Eq a, Enum a, Ord b) => (a -> b) -> (a,a) -> b
fmax f (bmin, bmax)
    | bmin ==  bmax  = f_bmin
    | f_bmin > f_max = f_bmin
    | otherwise      = f_max
  where
    f_max  = fmax f (succ bmin, bmax)
    f_bmin = f bmin
```

```
> fmax (\x-> 2*x^3 -30*x^2 +10) (-11,11)
10
```

```
> [ 2*x^3 -30*x^2 +10 | x<-[-5..5]]
[-990,-598,-314,-126,-22,10,-18,-94,-206,-342,-490]
```

Funktionen als Argumente

Ganzzahliges Maximum einer Funktion in einem Bereich bestimmen

```
fmax :: (Eq a, Enum a, Ord b) => (a -> b) -> (a,a) -> b
fmax f (bmin, bmax)
    | bmin ==  bmax  = f_bmin
    | f_bmin > f_max = f_bmin
    | otherwise      = f_max
  where
    f_max  = fmax f (succ bmin, bmax)
    f_bmin = f bmin

> fmax (\x-> 2*x^3 -30*x^2 +10) (-11,11)
10

> [ 2*x^3 -30*x^2 +10 | x<-[-5..5]]
[-990,-598,-314,-126,-22,10,-18,-94,-206,-342,-490]
```

Verallgemeinerung

In Aufgabe A1-5 sollen wir eine Funktion `oddCollatzStep` schreiben, welche jedes Element einer Liste mit 3 multipliziert und danach um eins erhöht.

```
oddCollatzStep :: [Int] -> [Int]
oddCollatzStep t = [3*n+1| n<-t]
```

Hier könnten wir verallgemeinern:

```
oddCollatzStep :: [Int] -> [Int]
oddCollatzStep = map collatzStep
  where

    map f l = [f n| n<-l]


    collatzStep n = 3*n+1
```

Verallgemeinerung

In Aufgabe A1-5 sollen wir eine Funktion `oddCollatzStep` schreiben, welche jedes Element einer Liste mit 3 multipliziert und danach um eins erhöht.

```
oddCollatzStep :: [Int] -> [Int]
oddCollatzStep t = [3*n+1| n<-t]
```

Hier könnten wir verallgemeinern:

```
oddCollatzStep :: [Int] -> [Int]
oddCollatzStep = map collatzStep
  where
    map :: (Int -> Int) -> [Int] -> [Int]
    map f l = [f n| n<-l]

    collatzStep :: Int -> Int
    collatzStep n = 3*n+1
```

Verallgemeinerung

In Aufgabe A1-5 sollen wir eine Funktion `oddCollatzStep` schreiben, welche jedes Element einer Liste mit 3 multipliziert und danach um eins erhöht.

```
oddCollatzStep :: [Int] -> [Int] -- rekursive Version
oddCollatzStep   []  = []
oddCollatzStep (h:t) = 3*h+1 : oddCollatzStep t
```

Auch die rekursive Variante könnten wir verallgemeinern:

```
oddCollatzStep :: [Int] -> [Int]
oddCollatzStep = map (\n -> 3*n+1)
  where

    map _   []  = []
    map f (h:t) = f h : map f t
```

Verallgemeinerung

In Aufgabe A1-5 sollen wir eine Funktion `oddCollatzStep` schreiben, welche jedes Element einer Liste mit 3 multipliziert und danach um eins erhöht.

```
oddCollatzStep :: [Int] -> [Int] -- rekursive Version
oddCollatzStep   []  = []
oddCollatzStep (h:t) = 3*h+1 : oddCollatzStep t
```

Auch die rekursive Variante könnten wir verallgemeinern:

```
oddCollatzStep :: [Int] -> [Int]
oddCollatzStep = map (\n -> 3*n+1)
  where
    map :: (Int -> Int) -> [Int] -> [Int]
    map _   []  = []
    map f (h:t) = f h : map f t
```

Kommt dieses Muster bekannt vor?

Map

Standardbibliothek definiert polymorphe `map` Funktion:

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

Beispiele:

```
> map (compare 5) [3..7]
[GT,GT,EQ,LT,LT]

> map (replicate 3) [1..4]
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]

> map (map (+1)) [[1,2,3],[4,5]]
[[2,3,4],[5,6]]
```

Allgemein: $\text{map } f\ [a_1, \ldots, a_n] = [f(a_1), \ldots, f(a_n)]$

Map

Standardbibliothek definiert polymorphe `map` Funktion:

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

Beispiele:

```
> map (compare 5) [3..7]
[GT,GT,EQ,LT,LT]

> map (replicate 3) [1..4]
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]

> map (map (+1)) [[1,2,3],[4,5]]
[[2,3,4],[5,6]]
```

Allgemein: $\texttt{map}\ f\ [a_1, \ldots, a_n] = [f(a_1), \ldots, f(a_n)]$

Filter

Filtern von Listen ohne List-Comprehension, d.h. Entfernung von allen Elementen, welche einem Prädikat nicht genügen:

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

```
> filter (>= 5) [3..7]
[5,6,7]
```

```
> filter (`elem` ['A'..'Z']) "Obst Kaufen!"
"OK"
```

List-Comprehensions können kürzer sein; aber
Funktionen höherer Ordnung lassen sich oft schön kombinieren

Filter

Filtern von Listen ohne List-Comprehension, d.h. Entfernung von allen Elementen, welche einem Prädikat nicht genügen:

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

> filter (>= 5) [3..7]
[5,6,7]

> filter (`elem` ['A'..'Z']) "Obst Kaufen!"
"OK"
```

List-Comprehensions können kürzer sein; aber
Funktionen höherer Ordnung lassen sich oft schön kombinieren

Einführung in die Funktionale Programmierung mit Haskell

Funktionen höherer Ordnung

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

6. Juni 2013

Map & Filer

Map wendet Funktion auf jedes Element einer List an

- Ergebnis ist Liste **anderen Typs**
- Länge bleibt exakt **gleich**

Filter wendet Prädikat (Funktion mit Ergebnistyp `Bool`) auf jedes Element einer Liste an, und entfernt alle Elemente für die dabei `False` herauskommt

- Ergebnis ist Liste **gleichen** Typs
- Länge kann **kleiner** werden

`map` und `filter` sind wesentliche Bestandteile von List-Comprehensions (siehe Aufgabe H5-1)

```
foo  f p xs = [f x | x <- xs, x >= 0, p x]
foo' f p xs = map f (filter p (filter (>=0) xs))
```

Allerdings können List-Comprehensions mehr (z.B. refutable patterns, mehrfache Generatoren)

zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith _   _      _    = []
```

zipWith

`zipWith` verschmilzt zwei Listen mithilfe einer 2-stelligen Funktion, bei ungleicher Länge wird der Rest der längeren Liste ignoriert:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith _   _      _    = []
```

Beispiele:

```
> zipWith (,) [1..7] [20,19..0]
[(1,20),(2,19),(3,18),(4,17),(5,16),(6,15),(7,14)]

> zipWith (+) [10..20] [1..100]
[11,13,15,17,19,21,23,25,27,29,31]

> let chg x y = if x then y else negate y
> zipWith chg [True,False,False,True,False] [1..11]
[1,-2,-3,4,-5]
```

Falten auf Links

`foldl` faltet eine Liste mit einer binären Funktion zusammen. Die Klammerung "lehnt" dabei nach links:

$$\texttt{foldl}\ f\ a\ [b_1, \ldots, b_n] = f\ (\cdots(f\ (f\ a\ b_1)\ b_2)\cdots)\ b_n$$
$$= (\cdots((a\ `f`\ b_1)\ `f`\ b_2)\cdots)\ `f`\ b_n$$

Haskell code:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ acc   []  = acc
foldl f acc (h:t) = foldl (f acc h) t
```

Beispiel

```
> let sum     = foldl (+) 0
> sum [1..9]
45
```

Falten auf Links

`foldl` faltet eine Liste mit einer binären Funktion zusammen. Die Klammerung "lehnt" dabei nach links:

$$\texttt{foldl}\ f\ a\ [b_1, \ldots, b_n] = f\ (\cdots(f\ (f\ a\ b_1)\ b_2)\cdots)\ b_n$$
$$= (\cdots((a\ `f`\ b_1)\ `f`\ b_2)\cdots)\ `f`\ b_n$$

Haskell code:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ acc   []  = acc
foldl f acc (h:t) = foldl (f acc h) t
```

Beispiel

```
> let sum     = foldl (+) 0
> sum [1..9]
45

product = foldl (*) 1     :: [Double] -> Double
and     = foldl (&&) True :: [Bool] -> Bool
```

(Demo)

Falten auf rechts

`foldr` faltet auch Liste mit binären Funktion zusammen. Die Klammerung "lehnt" dabei aber nach rechts:

$$\begin{aligned}\texttt{foldr}\ f\ [b_1,\ldots,b_n]\ a &= f\ b_1\ (f\ b_2(\cdots(f\ b_n\ a)\cdots)\\ &= b_1\ `f`\ (b_2\ `f`\ (\cdots(b_n\ `f`\ a)\cdots)\end{aligned}$$

Haskell code:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _   []  acc = acc
foldr f (h:t) acc = f h (foldr f t acc)
```

Falten auf rechts

`foldr` faltet auch Liste mit binären Funktion zusammen. Die Klammerung "lehnt" dabei aber nach rechts:

$$\texttt{foldr}\ f\ [b_1, \ldots, b_n]\ a = f\ b_1\ (f\ b_2 (\cdots (f\ b_n\ a) \cdots)$$
$$= b_1\ `f`\ (b_2\ `f`\ (\cdots (b_n\ `f`\ a) \cdots)$$

Haskell code:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _   []  acc = acc
foldr f (h:t) acc = f h (foldr f t acc)
```

Beispiele

```
sum     = foldr (+)
and     = foldr (&&)
length  = foldr (\_ n -> succ n) 0
map f   = foldr (\x acc -> f x : acc) []
```

(Demo)

Foldl vs. Foldr

Wann verwendet man `foldl` und wann `foldr`?

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Das hängt von der Situation ab:

- Wenn binäre Funktion nicht assoziativ, dann oft keine Wahl
- `foldl` ist endrekursiv, und damit etwas effizienter
- `foldr` funktioniert gut wenn die binäre Funktion nicht immer beide Argumente inspiziert (z.B. `and`),
- `foldr` funktioniert im Gegensatz zu `foldl` auch mit unendlichen Listen $\Rightarrow$ verzögerte Auswertung

Varianten

Es gibt zahlreiche Varianten von `foldl` und `foldr`:

```
foldr1  :: (a -> a -> a) -> [a] -> a

foldl1  :: (a -> a -> a) -> [a] -> a
foldl'  :: (a -> b -> a) -> a -> [b] -> a -- strict
foldl1' :: (a -> a -> a) -> [a] -> a
```

So wie Varianten, welche auch die Zwischenergebnisse ausgeben:

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanl1 :: (a -> a -> a) -> [a] -> [a]
scanr1 :: (a -> a -> a) -> [a] -> [a]
```

Achtung: die ...1 Varianten liefern auf leeren Listen einen Fehler!

FLIP

Die simple Funktion `flip` vertauscht die Reihenfolge der Argumente einer zweistelligen Funktion:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y  =  f y x
```

BEISPIEL:

```
subtract :: (Num a) => a -> a -> a
subtract = flip (-)
```

`subtract` ist nützlich, da `(-1)` ausnahmsweise keine Section `(\x -> (x-1))` darstellt, sondern die Konstante -1 ist.

Currying

Wegen partieller Applikation ist es nahezu unerheblich, ob eine Funktion ein Paar von Argumenten oder zwei Argumente nacheinander erhält. Wir können beliebig wechseln:

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

uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x,y) = f x y

> curry snd 42 69
69
> uncurry (+) (3,4)
7
```

- Benannt nach Haskell Curry (1900-82), amerikanischer Logiker
- Funktionstypen nach dem Muster `A -> (B -> C)` nennt man auch “curried function”

Pointfree Style

Currying erlaubt es uns oft, Funktionen als Komposition anderer Funktionen zu beschreiben, ohne dabei die Argument explizit zu erwähnen:

```
sum :: Num a => [a] -> a
sum xs = foldr (+) 0 xs
```

können wir einfacher schreiben als

```
sum    = foldr (+) 0
```

Argumente am Ende können wir einfach weglassen,
da Typ `a -> b -> c` ja identisch zu `a -> (b -> c)` ist!
`(\x y-> foobar 42 x y)` wird verwendet wie `(foobar 42)`

Dieser Programmierstil wird als **"Pointfree"** (Punktfrei) bezeichnet, und ist manchmal besser lesbarer

Begriffsbezeichnung in der Kategorientheorie begründet

Hintereinanderausführung

Mit der Infix Funktion `(.)` können wir zwei Funktionen miteinander verketten:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g = \x -> f (g x)

> ( (4+) . (10*) . succ . max 1 . \n -> n*n+1 ) 1
34
```

Damit können wir auch gleich eine Liste von Funktionen der Reihe nach ausführen:

```
compose :: [a -> a] -> a -> a
compose = foldr (.) id

> compose [(4+), (10*), succ, max 1, \n -> n*n+1] 1
34
```

Pointfree dank Punkt

Funktionskomposition ermöglicht oft den Punktfreien Stil:

```
bar x = ceiling (negate (cos x))
bar'  = ceiling . negate . cos

foo' :: (Int->b) -> (Int->Bool) -> [Int] -> [b]
foo'  f p xs = map f (filter p (filter (>=0) xs))
foo'' f p    = map f . filter p . filter (>=0)
```

Man sieht so oft besser, wie eine Funktion aus anderen Funktinen zusammensetzt wird.

Allgemein:

```
(\x -> f1(f2(f3(f4(f5 x))))) == f1.f2.f3.f4.f5
```

Ironischerweise nutzt der pointfree-style viele Punkt-Operatoren

Pointless Style

Vorsicht, der pointfree-style kann aber schnell pointless werden:

```
bar          = any . (==)
```

Pointless Style

Vorsicht, der pointfree-style kann aber schnell pointless werden:

```
bar  :: Eq a => a -> [a] -> Bool
bar        = any . (==)
```

Pointless Style

Vorsicht, der pointfree-style kann aber schnell pointless werden:

```
bar  :: Eq a => a -> [a] -> Bool
bar        = any . (==)
bar' x     = any   (==x)
```

Pointless Style

Vorsicht, der pointfree-style kann aber schnell pointless werden:

```
elem :: Eq a => a -> [a] -> Bool
elem       = any . (==)
elem' x    = any   (==x)
elem'' x l = any   (==x) l
```

Pointless Style

Vorsicht, der pointfree-style kann aber schnell pointless werden:

```
elem :: Eq a => a -> [a] -> Bool
elem       = any . (==)
elem' x    = any   (==x)
elem'' x l = any   (==x) l

sortAppend :: Ord a => [a] -> [a] -> [a]
sortAppend         = (sort .) . (++)
```

Pointless Style

Vorsicht, der pointfree-style kann aber schnell pointless werden:

```
elem :: Eq a => a -> [a] -> Bool
elem       = any . (==)
elem' x    = any   (==x)
elem'' x l = any   (==x) l

sortAppend :: Ord a => [a] -> [a] -> [a]
sortAppend         = (sort .) . (++)
sortAppend' xs ys = sort (xs ++ ys)
```

Funktionskomposition mit curried Funktionen oft schlechter lesbar!

- Übertrieben pointfree wird schnell "pointless"
- maximal 2-4 einstellige Funktionen kombinieren

Pointless Style

```
foo :: (Int->b) -> (Int->Bool) -> [Int] -> [b]
foo'  f p xs = map f (filter p (filter (>=0) xs))
foo'' f p    = map f . filter p . filter (>=0)
```

Längere Kompositionsketten werden schnell unlesbar; dann ist es oft besser, den Zwischenergebnissen sprechende Namen zu geben:

```
foo''' f p xs_input =
    let xs_positives = filter (>=0) xs_input
        xs_filtered  = filter p xs_positives
        xs_mapped    = map f xs_filtered
    in xs_mapped
```

Vorsicht: Man darf nicht überall den gleichen Bezeichner verwenden, weil es dann als rekursive Definition verstanden wird und damit nicht mehr terminiert!

$ Funktion

Was macht diese Infix-Funktion?

```
($)    :: (a -> b) -> a -> b
f $ x =  f x
```

$ Funktion

Was macht diese Infix-Funktion?

```
($)    :: (a -> b) -> a -> b
f $ x =  f x
```

Antwort: Funktionsanwendung / Klammern sparen!

Im Gegensatz zu dem Leerzeichen als Funktionsanwendung, hat `$` eine sehr niedrige Präzedenz (Bindet schwach).

Merke: `$` ersetzt Klammer, welche so spät wie möglich schliesst

Beispiel:

```
sum (filter (> 10) (map (^2) [1..10]))
```

ist gleichwertig zu

```
sum $ filter (>10) $ map (^2) [1..10]
```

$ Funktion

Was macht diese Infix-Funktion?

```
infixr 0  $

($)   :: (a -> b) -> a -> b
f $ x =  f x
```

Weiterhin erlaubt `$` auch die Verwendung der Funktionsanwendung selbst als Funktion:

```
> map ($ 3) [(4+), (10*), succ, max 1, \n -> n*n+1, id]
[7,30,4,3,10,3]
```

Hinweis:

`$` keine eingebaute Syntax, sondern gewöhnliche Infix Funktion!

$ Funktion

Damit haben wir noch eine Variante:

```
foo'''' f p xs = map f $ filter p $ filter (>=0) xs

foo'    f p xs = map f (filter p (filter (>=0) xs))

foo''   f p    = map f . filter p . filter (>=0)

foo'''  f p xs_input =
    let xs_positives = filter (>=0) xs_input
        xs_filtered  = filter p xs_positives
        xs_mapped    = map f xs_filtered
    in xs_mapped

foo     f p xs = [f x | x <- xs, x >= 0, p x]
```

⇒ Was jeweils am schönsten ist, hängt von der Situation ab.

Ordnung

Der Name "Funktion höherer Ordnung" stammt von üblichen Definition der **Typordnung**:

$$\mathrm{ord}(T) = \begin{cases} 0 & \text{falls } T \text{ ein Basistyp ist} \\ \max\big(1 + \mathrm{ord}(A), \mathrm{ord}(B)\big) & \text{falls } T \equiv A \to B \end{cases}$$

Beispiele:

$$\begin{aligned}
\mathrm{ord}\Big(\mathtt{Int} \to \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int}\Big) &= \\
\mathrm{ord}\Big(\mathtt{Int} \to (\mathtt{Int} \to \mathtt{Int}) \to \mathtt{Int}\Big) &= \\
\mathrm{ord}\Big((\mathtt{Int} \to (\mathtt{Int} \to \mathtt{Int})) \to \mathtt{Int}\Big) &= \\
\mathrm{ord}\Big(((\mathtt{Int} \to \mathtt{Int}) \to \mathtt{Int}) \to \mathtt{Int}\Big) &=
\end{aligned}$$

Ordnung

Der Name "Funktion höherer Ordnung" stammt von üblichen Definition der **Typordnung**:

$$\mathrm{ord}(T) = \begin{cases} 0 & \text{falls } T \text{ ein Basistyp ist} \\ \max\big(1 + \mathrm{ord}(A), \mathrm{ord}(B)\big) & \text{falls } T \equiv A \to B \end{cases}$$

Beispiele:

$$\mathrm{ord}\Big(\texttt{Int} \to \texttt{Int} \to \texttt{Int} \to \texttt{Int}\Big) = 1$$

$$\mathrm{ord}\Big(\texttt{Int} \to (\texttt{Int} \to \texttt{Int}) \to \texttt{Int}\Big) =$$

$$\mathrm{ord}\Big((\texttt{Int} \to (\texttt{Int} \to \texttt{Int})) \to \texttt{Int}\Big) =$$

$$\mathrm{ord}\Big(((\texttt{Int} \to \texttt{Int}) \to \texttt{Int}) \to \texttt{Int}\Big) =$$

Ordnung

Der Name "Funktion höherer Ordnung" stammt von üblichen Definition der **Typordnung**:

$$\mathrm{ord}(T) = \begin{cases} 0 & \text{falls } T \text{ ein Basistyp ist} \\ \max\big(1 + \mathrm{ord}(A), \mathrm{ord}(B)\big) & \text{falls } T \equiv A \to B \end{cases}$$

Beispiele:

$$\mathrm{ord}\Big(\texttt{Int} \to \texttt{Int} \to \texttt{Int} \to \texttt{Int}\Big) = 1$$

$$\mathrm{ord}\Big(\texttt{Int} \to (\texttt{Int} \to \texttt{Int}) \to \texttt{Int}\Big) = 2$$

$$\mathrm{ord}\Big((\texttt{Int} \to (\texttt{Int} \to \texttt{Int})) \to \texttt{Int}\Big) =$$

$$\mathrm{ord}\Big(((\texttt{Int} \to \texttt{Int}) \to \texttt{Int}) \to \texttt{Int}\Big) =$$

Ordnung

Der Name "Funktion höherer Ordnung" stammt von üblichen Definition der **Typordnung**:

$$\mathrm{ord}(T) = \begin{cases} 0 & \text{falls } T \text{ ein Basistyp ist} \\ \max\big(1 + \mathrm{ord}(A), \mathrm{ord}(B)\big) & \text{falls } T \equiv A \to B \end{cases}$$

Beispiele:

$$\mathrm{ord}\Big(\texttt{Int} \to \texttt{Int} \to \texttt{Int} \to \texttt{Int}\Big) = 1$$

$$\mathrm{ord}\Big(\texttt{Int} \to (\texttt{Int} \to \texttt{Int}) \to \texttt{Int}\Big) = 2$$

$$\mathrm{ord}\Big((\texttt{Int} \to (\texttt{Int} \to \texttt{Int})) \to \texttt{Int}\Big) = 2$$

$$\mathrm{ord}\Big(((\texttt{Int} \to \texttt{Int}) \to \texttt{Int}) \to \texttt{Int}\Big) =$$

Ordnung

Der Name "Funktion höherer Ordnung" stammt von üblichen Definition der **Typordnung**:

$$\mathrm{ord}(T) = \begin{cases} 0 & \text{falls } T \text{ ein Basistyp ist} \\ \max\big(1 + \mathrm{ord}(A), \mathrm{ord}(B)\big) & \text{falls } T \equiv A \to B \end{cases}$$

Beispiele:

$$\begin{aligned} \mathrm{ord}\Big(\texttt{Int} \to \texttt{Int} \to \texttt{Int} \to \texttt{Int}\Big) &= 1 \\ \mathrm{ord}\Big(\texttt{Int} \to (\texttt{Int} \to \texttt{Int}) \to \texttt{Int}\Big) &= 2 \\ \mathrm{ord}\Big((\texttt{Int} \to (\texttt{Int} \to \texttt{Int})) \to \texttt{Int}\Big) &= 2 \\ \mathrm{ord}\Big(((\texttt{Int} \to \texttt{Int}) \to \texttt{Int}) \to \texttt{Int}\Big) &= 3 \end{aligned}$$

Zusammenfassung

- Funktionen sind ganz normale Werte in Haskell
- Funktionen höherer Ordnung nehmen Funktionen als Argumente (und können Funktionen zurück liefern)
- Funktionen höherer Ordnung abstrahieren auf einfache Weise häufig verwendete Berechnungsverfahren;
  viele wichtige in Standardbibliothek verfügbar
- Funktionsanwendung darf partiell sein;
  partielle Funktionsanwendung liefert eine Funktion
- Die durch Funktionen höherer Ordnung gewonnene Modularität erlaubt sehr viele Kombinationsmöglichkeiten

Evolution of a Haskell Programmer

Funktionen höherer Ordnung sind ein mächtiges Werkzeug, welches sehr vielfältig benutzt werden kann. Welches Werkzeug für welchen Fall geeignet ist, muss von Fall zu Fall unterschieden werden.

Die humoristische Webseite Evolution of a Haskell Programmer fasst dies ganz gut zusammen. Auf welcher Stufe bist Du?

Wichtig ist, dass der Code lesbar bleibt, denn:

> *There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*
>
> *C.A.R. Hoare*
>
> *1980 ACM Turing Award Lecture*