# Funktionen höherer Ordnung

# Funktionen höherer Ordnung

## Funktionen höherer Ordnung

Funktionen, die andere Funktionen als Parameter erhalten oder Funktionen als Rückgabewerte liefern, heißen Funktionen höherer Ordnung.

Differenzialoperator: $\frac{\mathrm{d}}{\mathrm{d}x} = \lambda f.\left(\lambda x.\left(\lim_{h\to 0}\frac{f(x+h)-f(x)}{h}\right)\right)$

- Parameter: Funktionen $f$, Rückgabe: Ableitung $f'$
- Punktweise Definition:

$$\left(\frac{\mathrm{d}}{\mathrm{d}x}(f)\right)(x) = \lim_{h\to 0}\frac{f(x+h)-f(x)}{h}$$

Volle Unterstützung in Haskell: Funktionen sind <u>erstklassig</u>

- Definitionen von Operatoren und Kombinatoren

$\Rightarrow$ Modularität und Lesbarkeit

# Funktionen als Parameter: `map`, `filter`

## Funktionsanwendung auf Listenelemente

```
map :: (s -> t) -> [s] -> [t]
map f []      = []
map f (x:xs) = f x : map f xs
```

```
toLower :: Char -> Char
toLowercase :: String -> String
toLowercase s = map toLower s
```

```
toUpper :: Char -> Char
toUppercase :: String -> String
toUppercase s = map toUpper s
```

## Filtern von Listen: Anhand Prädikat `pred :: t -> Bool`

- Behalte Elemente, die Prädikat erfüllen

```
filter :: (t -> Bool) -> [t] -> [t]
filter pred []     =  []
filter pred (x:xs) =  if pred x then x:(filter pred xs)
                     else filter pred xs
```

```
filter isDigit "0721 608-8350" ⇒+ "07216088350"
filter (>1) [1,2,3]            ⇒+ [2,3]
```

# Funktionen als Rückgabewerte

Haskell-Funktionen als Rückgabewert anderer Funktionen

## Lineare Funktionen: $f_a(x) = a \cdot x$

```
f :: Double -> (Double -> Double)
f a = \x -> a*x
```

## Funktionskomposition: $f \circ g$ (Infixnotation: `f . g`)

```
comp :: (u -> t) -> (s -> u) -> (s -> t)
comp f g = (\x -> f (g x))
```

## *n*-fache Funktionsanwendung: $f^n$

```
iter ::  (t -> t) -> Integer -> (t -> t)
iter f n
   | (n == 0)  = (\x -> x)
   | otherwise = f . (iter f (n - 1))
               == \x -> f ((iter f (n - 1)) x)
```

# Currying

## Lineare Funktionen: $f_a(x) = a \cdot x$

Definition mit $\lambda$

```
f :: Double -> (Double -> Double)
f a = \x -> a * x
```

als „mehrstellige“ Funktion:

```
f :: Double -> Double -> Double
f a x = a * x
```

- Definitionen äquivalent!
- `->` ist rechts-assoziativ:

  `Double -> Double -> Double` $\equiv$ `Double -> (Double -> Double)`
- Funktionsanwendung ist links-assoziativ:

  `f 3 7` $\equiv$ `(f 3) 7`
- Funktionen in Haskell sind gecurrieht

**Currying**

Ersetzung einer mehrstelligen Funktion durch Schachtelung einstelliger Funktionen

# Unterversorgung

Unterversorgung:

- Anwendung “mehrstelliger” Funktionen auf zu wenige Parameter
- Zusammen mit Kombinatoren: kompakte Schreibweise

Bei Infixoperatoren: Erhöhe Listeneinträge um 5

```
add5 :: [Integer] -> [Integer]
add5 list = map (5+) list
```

Noch kürzer: Unterversorgung von `map`

```
add5 :: [Integer] -> [Integer]
add5 = map (5+)
```

Anwendungsbeispiel: Erkenne Alphabetzeichen

```
isAlpha :: Char -> Bool
isAlpha = (isIn "abcdefghijklmnopqrstuvwxyz") . toLower
```

# Unterschied Currying/Tupelargumente

Funktion: Berechnung von $a \cdot x$

Gecurryt

```
f :: Double -> Double -> Double
f a x = a*x
```

Mit Tupeln

```
g :: (Double, Double) -> Double
g (a,x) = a*x
```

Definitionen verschieden!

- `f` ist gecurriehte Funktion mit zwei Argumenten
- `g` ist Funktion mit einem Tupel als Argument

Vergleich:

- Tupelschreibweise entspricht mathematischer Schreibweise
- Kann aber nicht unterversorgt werden!

# Currying: Grundlagen

Mathematisch gilt der **mengentheoretische Isomorphismus**

$$(A \times B) \to C \cong A \to (B \to C)$$

Zu $f = \lambda a, b.F(a, b)$ gehört die „gecurryte“ Version $f_c = \lambda a.\lambda b.F(a, b)$.

Es ist $\forall a \in A, b \in B : f(a, b) = f_c(a)(b)$ Haskell: $f_c(a)(b) \cong$ `f a b`

Unterschied: $f_c$ kann unterversorgt werden!

Ferner gilt das **Extensionalitätsprinzip**: für $f, g : A \to B$ ist

$$f = g \iff \forall x \in A : f(x) = g(x)$$

Die hintere Form heisst **punktweise Definition**

Zusammen begründet es die Äquivalenz der Definitionen von `add5` (s.o.):

`add5 = map (+5)` $\cong$ `add5 list = map (\x -> 5+x) list`

# Lambda und Bindung

# Anonyme Funktionen

Karlsruher Institut für Technologie

Haskell verwendet Lambda-Notation für anonyme Funktionen

Beispiele:

| Funktion | $\lambda$-Notation | Haskell-Schreibweise |
|---|---|---|
| Quadratfunktion | $\lambda x.\, x^2$ | `(\x -> x*x)` |
| Identität | $\lambda y.\, y$ | `(\y -> y)` |

Vorteil: Funktionen als Parameter möglich z.B. in funktionalen Kombinatoren (s.u.)

# Anwendung Anonymer Funktionen

Anwendung von $\lambda$-Ausdrücken auf Argumente:

```
(\x -> x*x) 3        ⇒   3*3        ⇒ 9
(\y -> y) "Hallo"    ⇒   "Hallo"
```

Funktionsdefinitionen mit / ohne $\lambda$-Ausdrücke:

```
f = \x -> sin x / x            f x = sin x / x
g = \x -> x * (f (x*x))        g x = x * (f (x*x))
```

**Achtung**

Funktionsapplikation hat keine Syntax, nur „nebeneinander schreiben“.
Links Funktion, rechts Parameter.

Grund: Historische Entwicklung der kombinatorischen Logik.
Eindeutig, da jede Funktion genau einen Parameter hat.

# Bindungskonstrukte

## Bindungskonstrukte

Bindungskonstrukte legen Bedeutung und Geltungsbereich von Variablen fest.

Definitionen: `f x = x*x`     `pi = 3.14159`

- Bindung von `x` im Rumpf von `f`
- Globale Bindung von `f` und `pi`

$\lambda$-Abstraktion: `(\x -> x*x)`

- Bindung von `x` innerhalb des $\lambda$-Ausdrucks

Drei gültige Programme:

| | | |
|---|---|---|
| `y = 2`<br>`f x = y + x*x` | `f x = (\y -> y + x*x)` | `f x y = y + x*x` |

Im Rumpf ist `y`: frei | gebunden | gebunden.

# Bindung

Funktionsdefinition gültig?

```
f x = y + x*x
```

# Bindung

Funktionsdefinition gültig?

`f x = y + x*x` $\qquad$ `f 7` $\Rightarrow$ `y + 7*7` $\Rightarrow$ ???

# Bindung

Funktionsdefinition gültig?

```
f x = y + x*x
```

`f 7` $\Rightarrow$ `y + 7*7` $\Rightarrow$ ???

- Variable `x` ist formaler Parameter von `f`
  Im Funktionsaufruf bezeichnet `x` den Wert des Arguments
- Variable `y` nicht festgelegt
- $\Longrightarrow$ Funktionsdefinition ungültig

In der Definition von `f` ist Variable `x` gebunden, Variable `y` ist frei

# Lokale Bindung

## Lokale Namensbindung: `let` und `where`

```
energy m = let c = 299792458
           in  m * c * c
```

```
energy m = m * c * c
  where c = 299792458
```

# Lokale Bindung

## Lokale Namensbindung: `let` und `where`

```
energy m = let c = 299792458
           in  m * c * c
```

```
energy m = m * c * c
  where c = 299792458
```

## Anwendung: lokale Hilfsfunktionen

```
energy m = let c = 299792458
               square x = x * x
           in m * (square c)
```

```
energy m = m * (square c)
  where c = 299792458
        square x = x * x
```

## Auch rekursiv:

```
fak n = f n 1
   where f n a = if (n==0) then a else f (n - 1) (n * a)
```

# Verdeckung

## Innere Bindungen verdecken äußere:

```
f = (\x -> ((\x -> x*x) 3)+x)
```

```
f 1 ⇒ ((\x -> x*x) 3)+1
    ⇒ 9+1
    ⇒ 10
```

## Vorsicht bei Verdeckung:

```
unknown = let x = 3 in
            let x = 3*x in 4+x
```

- Variable `x` in `3*x` gebunden durch <u>inneres</u> **`let`**

$\Longrightarrow$ rekursive Definition von `x`

$\Longrightarrow$ Auswertung terminiert nicht

# Referenzielle Transparenz

Wird dieser Java-Code fehlerfrei laufen?

```
if (f(x) == f(x)) return x;
else              throw new Exception("==");
```

# Referenzielle Transparenz

Wird dieser Java-Code fehlerfrei laufen?

```
if (f(x) == f(x)) return x;
else              throw new Exception("==");
```

Abhängig von `f`

```
int z;
int f(int x) {
  z = z + 1;
  return x + z;
}
```

- `f` verändert Attribut `z`
- `f(x)` hängt von `x` und `z` ab

⇒ Vorkommen von `f(x)` werten verschieden aus,
obwohl im gleichen Geltungsbereich!

# Referenzielle Transparenz

Wird dieser Haskell-Code fehlerfrei laufen?

```
if (f x == f x) then x
else (error "==")
```

# Referenzielle Transparenz

Wird dieser Haskell-Code fehlerfrei laufen?

```
if (f x == f x) then x
else (error "==")
```

Ja, egal wie `f` definiert ist!

- `f` kann keinen Zustand verändern!
- `f x` hängt allein von `x` ab!

$\Rightarrow$ `f x` wertet im gleichen Gültigkeitsbereich stets gleich aus

**Referenzielle Transparenz**

Im gleichen Gültigkeitsbereich bedeuten gleiche Ausdrücke stets das gleiche. Zwei verschiedene Ausdrücke, die zum gleichen Wert auswerten, können stets durch den anderen ersetzt werden, ohne die Bedeutung des Programms zu verändern.

# Referenzielle Transparenz

Referenzielle Transparenz dient

- besserer Lesbarkeit, und dadurch
- geringerer Fehlerquote

Programme leichter zu verstehen, weil

- Teilprogramme analysierbar, ohne Seiteneffekte auf einen globalen Zustand beachten zu müssen
- Werte von Funktionsaufrufen ausschließlich von ihren Parametern abhängen

# Kombinatoren

# Folds

Summe/Produkt von Listen. Variationspunkte: **Initialwert**, **Operator**

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + (sum xs)
```

```
product :: [Int] -> Int
product []     = 1
product (x:xs) = x * (product xs)
```

# Folds

Summe/Produkt von Listen. Variationspunkte: **Initialwert**, **Operator**

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + (sum xs)
```

```
product :: [Int] -> Int
product []     = 1
product (x:xs) = x * (product xs)
```

Verallgemeinere Struktur als Fold

```
foldr op i    []  = i
foldr op i (x:xs) = op x (foldr op i xs)
```

# Folds

Summe/Produkt von Listen. Variationspunkte: **Initialwert**, **Operator**

```
sum :: [Int] -> Int
sum = foldr (+) 0
```

```
product :: [Int] -> Int
product  = foldr (*) 1
```

Verallgemeinere Struktur als Fold

```
foldr op i    []   = i
foldr op i (x:xs)  = op x (foldr op i xs)
```

# Folds

Summe/Produkt von Listen. Variationspunkte: **Initialwert**, **Operator**

```
sum :: [Int] -> Int
sum = foldr (+) 0
```

```
product :: [Int] -> Int
product  = foldr (*) 1
```

Verallgemeinere Struktur als Fold

```
foldr op i    []   = i
foldr op i (x:xs) = op x (foldr op i xs)

foldl op i    []   = i
foldl op i (x:xs) = foldl op (op  i x) xs
```

Für die Liste `[1,2,3,4]` berechnet

- `foldr (+) 0` den Wert `(1+(2+(3+(4+0))))` – rechts-geklammert
- `foldl (+) 0` den Wert `((((0+1)+2)+3)+4)` – links-geklammert

Ergebnisse stimmen überein, da

- Addition `(+)` assoziativ ist
- `0` (rechts- und linksseitig) neutrales Element bzgl. `(+)` ist

# Anwendungen von Folds

Folds sind wichtige Kombinatoren:

- Komplexe Funktionen als Kombination einfacher Funktionen

```
foldr :: (s -> t -> t) -> t -> [s] -> t
foldr op i []       = i
foldr op i (x:xs) = op x (foldr op i xs)
```

Listenlänge:

```
length :: [t] -> Int
length list = foldr (+) 0 (map (\x ->1) list)
```

Listenlänge: mit zweistelligem $\lambda$-Ausdruck

```
length :: [t] -> Int
length = foldr (\x n -> n + 1) 0
```

Satzlänge: unterschiedliche Argumenttypen bei $\lambda$-Ausdruck, $s \neq t$

```
sentenceLength :: [String] -> Int
sentenceLength = foldr (\l n -> length l + n) 0
sentenceLength ["progpar", "ist", "toll"] ⇒+ 14
```

# Anwendungen von Folds

Folds sind wichtige Kombinatoren:

- Komplexe Funktionen als Kombination einfacher Funktionen

```
foldl :: (t -> s -> t) -> t -> [s] -> t
foldl op i []     = i
foldl op i (x:xs) = foldl op (op i x) xs
```

## Listenumkehrung:

```
rev :: [t] -> [t]
rev = foldl cons []
  where cons xs x = x:xs
```

```
rev [1,2,3] ⇒ foldl cons [] [1,2,3]
            ⇒ foldl cons (cons [] 1) [2,3]
            ⇒ foldl cons (cons (cons [] 1) 2) [3]
            ⇒ foldl cons (cons (cons (cons [] 1) 2) 3) []
            ⇒ (cons (cons (cons [] 1) 2) 3)
            ⇒ 3:(cons (cons [] 1) 2)
            ⇒ 3:2:(cons [] 1) ⇒+ [3,2,1]
```

# Anwendungen von Folds

Folds sind wichtige <u>Kombinatoren</u>:

- Komplexe Funktionen als Kombination einfacher Funktionen

```
foldr :: (s -> t -> t) -> t -> [s] -> t
foldr op i []     = i
foldr op i (x:xs) = op x (foldr op i xs)
```

## Listenkonkatenation:

```
app :: [t] -> [t] -> [t]
app left right = foldr (:) right left
```

```
app [1,2,3] [4,5] ⇒ foldr (:) [4,5] [1,2,3]
                  ⇒ 1:(foldr (:) [4,5] [2,3])
                  ⇒ 1:2:(foldr (:) [4,5] [3])
                  ⇒ 1:2:3:(foldr (:) [4,5] [])
                  ⇒ 1:2:3:[4,5]
                  =  [1,2,3,4,5]
```

# Anwendungen von Folds

Folds sind wichtige Kombinatoren:

- Komplexe Funktionen als Kombination einfacher Funktionen

```
foldr :: (s -> t -> t) -> t -> [s] -> t
foldr op i []      = i
foldr op i (x:xs) = op x (foldr op i xs)
```

Verflachen von Listen: (vordefiniert als `concat`)

```
flatten :: [[t]] -> [t]
flatten = foldr app []
```

$$
\begin{aligned}
\texttt{flatten [[1,2],[3]]} &\Rightarrow \texttt{foldr app [] [[1,2],[3]]}\\
&\Rightarrow^{+} \texttt{app [1,2] (foldr app [] [[3]])}\\
&\Rightarrow^{+} \texttt{app [1,2] (app [3] (foldr app [] []))}\\
&\Rightarrow^{+} \texttt{app [1,2] (app [3] [])}\\
&\Rightarrow^{+} \texttt{[1,2,3]}
\end{aligned}
$$

# Anwendungen von Folds

Folds sind wichtige Kombinatoren:

- Komplexe Funktionen als Kombination einfacher Funktionen

```
foldr :: (s -> t -> t) -> t -> [s] -> t
foldr op i []     = i
foldr op i (x:xs) = op x (foldr op i xs)
```

## Filtern von Listen:

```
filter :: (t -> Bool) -> [t] -> [t]
filter pred = foldr check []
   where check x xs = if (pred x) then x:xs else xs
```

```
filter (>1) [1,2,3] ⇒   foldr check [] [1,2,3]
                    ⇒+  if ((>1) 1) then 1:(foldr check [] [2,3])
                        else (foldr check [] [2,3])
                    ⇒+  foldr check [] [2,3]
                    ⇒+  2:(foldr check [] [3])
                    ⇒+  2:(3:[]) = [2,3]
```

# Kombination von Listen

## Kombination von Listen: `zipWith`

```
zipWith :: (s -> t -> u) -> [s] -> [t] -> [u]
zipWith f (x:xs) (y:ys) =  f x y : zipWith f xs ys
zipWith f xs     ys     =  []
```

## Zusammenfügen per Reißverschluss: `zip = zipWith (,)`

```
zip [1,2,3]
    [9,8,9] ⇒+ [(1,9),(2,8),(3,9)]

zip [1,2,3]
    [5]     ⇒+ [(1,5)]
```

# Kombination von Listen

## Kombination von Listen: `zipWith`

```
zipWith :: (s -> t -> u) -> [s] -> [t] -> [u]
zipWith f (x:xs) (y:ys) =  f x y : zipWith f xs ys
zipWith f xs     ys     =  []
```

## Vektor-Addition: `zipWith (+)`

```
zipWith (+) [ 2,4,6]
            [-1,3,0]
```

$\Rightarrow^{+}$ `[ 1,7,6]`

## Vektor-Skalarprodukt:

```
skalar v1 v2 = sum (zipWith (*) v1 v2)
```

# Kombination von Listen

## Kombination von Listen: `zipWith`

```
zipWith :: (s -> t -> u) -> [s] -> [t] -> [u]
zipWith f (x:xs) (y:ys) =  f x y : zipWith f xs ys
zipWith f xs     ys     =  []
```

## String-Vergleich: Hamming-Distanz

- Anzahl unterschiedlicher Stellen
- Bei Strings gleicher Länge

```
hamming l r = sum (zipWith differs l r)
    where differs x y = if (x == y) then 0 else 1

hamming "Saulus"
        "Paulus" ⇒+ 1

hamming [0,1,0]
        [1,1,1]  ⇒+ 2
```

# Intervalle, Summen

## Intervalle:

- Kurznotation: `[a..b]` $\Rightarrow^+$ `[a,a+1,a+2,...,b]`
- Intervalle + Listen-Kombinatoren
  $\Longrightarrow$ kompakte Programme

## Summen:

$\sum_{k=1}^{n} k$ `sum [1..n]`

$e = \lim_{n\to\infty} \sum_{k=0}^{n} \frac{1}{k!}$ `sum (map (\k -> 1 / fak k) [0..n])`

## Primzahlensieb: Liste aller Primzahlen $\leq n$

```
primes :: Integer -> [Integer]
primes n = sieve [2..n]
 where sieve []     = []
       sieve (p:xs) = p : sieve (filter (not . multipleOf p) xs)

       multipleOf p x = x `mod` p == 0
```

# List Comprehensions

## List Comprehensions

Schreibweise zur Generierung von Listen

$$[\texttt{e}|\texttt{q}_1, ..., \texttt{q}_m]$$

Die $\texttt{q}_i$ sind Tests, oder Generatoren der Form

- `p <- list`, mit Muster `p` und Listenausdruck `list`

Die durch die Muster gebundenen Variablen können in `e` und in den $\texttt{q}_i$ rechts vom Muster verwendet werden.

Inspiriert durch Mengenschreibweise der Mathematik:

- $\{x \mid x \in X \wedge P(x)\}$ bzw.
- $\{x \mid \exists y.F(x, y) \in M \wedge P(x, y)\}$

# List Comprehensions

## List Comprehensions

Schreibweise zur Generierung von Listen

$$[e|q_1, ..., q_m]$$

Die $q_i$ sind Tests, oder Generatoren der Form

- `p <- list`, mit Muster `p` und Listenausdruck `list`

Die durch die Muster gebundenen Variablen können in `e` und in den $q_i$ rechts vom Muster verwendet werden.

Alternative zu `filter` und `map`

- `[f x | x<-l]` $\Leftrightarrow$ `map (\x->f x) l` $\Leftrightarrow$ `map f l`
- `[x | x<-l, pred x]` $\Leftrightarrow$ `filter (\x -> pred x) l` $\Leftrightarrow$ `filter pred l`
- `[f x | x<-l, pred x]` $\Leftrightarrow$ `map f (filter pred l)`

# List Comprehensions

## List Comprehensions

Schreibweise zur Generierung von Listen

$$[\mathtt{e}|\mathtt{q}_1, ..., \mathtt{q}_m]$$

Die $\mathtt{q_i}$ sind Tests, oder Generatoren der Form

- `p <- list`, mit Muster `p` und Listenausdruck `list`

Die durch die Muster gebundenen Variablen können in `e` und in den $\mathtt{q_i}$ rechts vom Muster verwendet werden.

Programm: Erste *n* Quadratzahlen

```
squares n = [ x*x | x <- [0..n]]
```

`squares 10` $\Rightarrow^+$ `[0,1,4,9,16,25,36,49,64,81,100]`

- Generator `x <- [0..n]` bindet Elemente von `[0..n]` an Namen `x`

# List Comprehensions

## List Comprehensions

Schreibweise zur Generierung von Listen

$$[\mathtt{e} \mid \mathtt{q}_1, \ldots, \mathtt{q}_m]$$

Die $\mathtt{q_i}$ sind Tests, oder Generatoren der Form

- `p <- list`, mit Muster `p` und Listenausdruck `list`

Die durch die Muster gebundenen Variablen können in `e` und in den $\mathtt{q_i}$ rechts vom Muster verwendet werden.

Programm: Gerade Zahlen $\leq n$

```
evens n = [ x | x <- [0..n], x `mod` 2 == 0]
```

`evens 10` $\Rightarrow^{+}$ `[0,2,4,6,8,10]`

- Test ``x `mod` 2 == 0`` eliminiert ungerade `x`

# List Comprehensions

## List Comprehensions

Schreibweise zur Generierung von Listen

$$[e | q_1, \ldots, q_m]$$

Die $q_i$ sind Tests, oder Generatoren der Form

- `p <- list`, mit Muster `p` und Listenausdruck `list`

Die durch die Muster gebundenen Variablen können in `e` und in den $q_i$ rechts vom Muster verwendet werden.

Programm: Alle Quadrate von geraden Listenelementen

```
squaredEvens l = [ x*x | x <- l, x `mod` 2 == 0]
squaredEvens [0..10]   ⇒+ [0,4,16,36,64,100]
```

# List Comprehensions

## List Comprehensions

Schreibweise zur Generierung von Listen

$$[\mathtt{e} | \mathtt{q}_1, \ldots, \mathtt{q}_m]$$

Die $\mathtt{q}_i$ sind Tests, oder Generatoren der Form

- `p <- list`, mit Muster `p` und Listenausdruck `list`

Die durch die Muster gebundenen Variablen können in `e` und in den $\mathtt{q}_i$ rechts vom Muster verwendet werden.

Programm: Bestehende einer Prüfung

```
graduates :: Examination -> [Student]
graduates exam = [s | (s,a) <- exam, passed a ]
```

- Matche Elemente von `exam` mit Muster `(s,a)`
- Student `s` im Ergebnis nur falls zugehörige Bewertung `a` ausreicht