# Heutiges Thema

- Funktionen höherer Ordnung (*kurz*: Funktionale)
  - Funktionen als Argumente
  - Funktionen als Resultate
  - Spezialfall: Funktionale auf Listen
  - Anwendungen

  ...und ihre Vorteile für die Programmierung.

# Funktionale

Funktionen, unter deren Argumenten oder Resultaten Funktionen sind, heißen *Funktionen höherer Ordnung* oder kurz *Funktionale*.

*Mithin*...

**Funktionale sind spezielle Funktionen!**

...also nichts Besonderes, oder?

# Funktionale nichts Außergewöhnliches?

Im Grunde nicht...

Drei kanonische Beispiele aus Mathematik und Informatik:

- *Differential- und Integralrechnung*
  - $\frac{df(x)}{dx}$ $\rightsquigarrow$ `diff f a`
    ...Ableitung von `f` an der Stelle `a`
  - $\int_a^b f(x)dx$ $\rightsquigarrow$ `integral f a b`
    ...Integral von `f` zwischen `a` und `b`

- *Semantik von Programmiersprachen*
  - Denotationelle Semantik der while-Schleife

    $\mathcal{S}_{ds}[\![$ `while b do S od` $]\!] : \Sigma \to \Sigma$

    ...kleinster Fixpunkt eines Funktionals auf der Menge der Zustandstransformationen $[\Sigma \to \Sigma]$ über der Menge der Zustände $\Sigma$ mit $\Sigma =_{def} \{\sigma \mid \sigma \in [Var \to Data]\}$.

    (Siehe z.B. VU 185.183 Theoretische Informatik 2)

# Aber...

*The functions I grew up with, such as the sine, the cosine, the square root, and the logarithm were almost exclusively real functions of a real argument.*

*... I was really ill-equipped to appreciate functional programming when I encountered it: I was, for instance, totally baffled by the shocking suggestion that the value of a function could be another function.*[(*)]

Edsger W. Dijkstra (11.5.1930-6.8.2002)

*1972 Recipient of the ACM Turing Award*

[(*)] Zitat aus: Introducing a course on calculi. Ankündigung einer Lehrveranstaltung an der University of Texas at Austin, 1995.

# Feststellung

Der systematische Umgang mit Funktionen höherer Ordnung...
(Funktionen als "first-class citizens")

- ist charakteristisch für funktionale Programmierung
- hebt funktionale Programmierung von anderen Programmierparadigmen ab
- ist der Schlüssel zu extrem eleganten und ausdruckskräftigen Programmiermethoden

# Ein Ausflug in die Philosophie...

*Der Mensch wird erst durch Arbeit zum Menschen.*
Georg W.F. Hegel (27.8.1770-14.11.1831)

Frei nach Hegel...

*Funktionale Programmierung wird erst durch Funktionale zu funktionaler Programmierung!*

# Funktionale, Teil 1 ...Funktionen als Argumente (1)

Anstatt zweier spezialisierter Funktionen...

```
max :: Ord a => a -> a -> a
max x y
  | x > y     = x
  | otherwise = y

min :: Ord a => a -> a -> a
min x y
  | x < y     = x
  | otherwise = y
```

# Funktionale, Teil 1 ...Funktionen als Argumente (2)

...eine mit einem *Funktions-/Prädikatsargument parametrisierte Funktion*:

```
extreme :: Num a => (a -> a -> Bool) -> a -> a -> a
extreme p m n
| p m n     = m
| otherwise = n
```

*Anwendungsbeispiele*:

```
extreme (>) 17 4 = 17
extreme (<) 17 4 = 4
```

Dadurch wird folgende alternative Definitionen von `max` und `min` möglich...

```
max x y = extreme (>) x y
min x y = extreme (<) x y
```

# Funktionen als Argumente: Weitere Bsp.

(Gleichförmige) Manipulation der Marken eines benannten Baums bzw. Herausfiltern der Marken mit einer bestimmten Eigenschaft...

```
data Tree a = Nil |
              Node a (Tree a) (Tree a)

walkAndWork :: (a -> a) -> Tree a -> Tree a
walkAndWork f Nil = Nil
walkAndWork f (Node elem t1 t2) =
          (Node (f elem)) (walkAndWork f t1) (walkAndWork f t2)

filterTree :: (a -> Bool)  -> Tree a -> [a]
filtertree p Nil = []
filterTree p (Node elem t1 t2)
  | p elem    = [elem] ++ (filterTree p t1) ++ (filterTree p t2)
  | otherwise = (filterTree p t1) ++ (filterTree p t2)
```

...mithilfe von Funktionalen, die in Manipulationsfunktion bzw. Prädikat parametrisiert sind.

# Zwischenresümee 1: Funktionen als Argumente...

- ...erhöhen die Ausdruckskraft erheblich und
- ...unterstützen Wiederverwendung.

*Beispiel*:

Vergleiche

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip  _      _     = []
```

mit

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f  _      _     = []
```

# Curryfiziert vs. uncurryfiziert

(siehe auch Vorlesungsteil 2, Folie 17)

Übergabe von mehreren Argumenten an Funktionen...

- Argument für Argument einzeln (sog. *curryfizierte* Darstellung)
- Argumente gebündelt als Paar (sog. *uncurryfizierte* Darstellung)

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

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

*Intuitiv*: Curryfizieren ersetzt Produkt-/Tupelbildung "$\times$" durch Funktionspfeil "$\rightarrow$", decurryfizieren Funktionspfeil "$\rightarrow$" durch Produkt-/Tupelbildung "$\times$".

Die Bezeichnung geht auf Haskell B. Curry zurück, die (weit ältere) Idee auf M. Schönfinkel aus der Mitte der 20er-Jahre.

# Beispiele zu curry- und uncurryfiziert

```
curriedAdd :: Int -> Int -> Int
curriedAdd m n = m+n

uncurriedAdd :: (Int,Int) -> Int
uncurriedAdd (m,n) = m+n
```

Mit diesen Definitionen können wir schreiben...

```
curry uncurriedAdd 17 4
   ⇒ uncurriedAdd (17,4) ⇒ 17+4 ⇒ 21

uncurried curriedAdd (17,4)
   ⇒ curriedAdd 17 4 ⇒ 17+4 ⇒ 21
```

# Curryfiziert oder uncurryfiziert?

...das ist hier die Frage.

Zum einen...

- Geschmackssache (notationelle Spielerei)
  ...allerdings: die Verwendung curryfizierter Formen ist vorherrschend
  $\rightsquigarrow$ `f x`, `f x y`, `f x y z`,... eleganter als `f x`, `f (x,y)`, `f (x,y,z)`,...?

Zum anderen (und weit wichtiger!)...

- Sachargument
  ...(Nur) Funktionen in curryfizierter Darstellung unterstützen *partielle Auswertung*
  $\rightsquigarrow$Funktionen liefern Funktionen als Ergebnis!

  *Beispiel*: `curriedAdd 4711 :: Int -> Int`

  ...ist eine einstellige Funktion auf den ganzen Zahlen, die ihr Argument um `4711` erhöht und als Resultat zurückliefert.

# Funktionale, Teil 2 ...Funktionen als Resultate (1)

Auch diese Situation ist bereits aus der Mathematik vertraut...

Etwa in Gestalt der...

- Funktionskomposition (Komposition von Funktionen)

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

*Bsp.*:

**Theorem** [...bekannt aus Analysis 1]
Die Komposition stetiger Funktionen ist wieder eine stetige Funktion.

# Funktionale, Teil 2 ...Funktionen als Resultate (2)

...ermöglichen Funktionsdefinitionen auf dem (Abstraktions-) Niveau von Funktionen statt von (elementaren) Werten.

*Beispiel*:

```
giveFourthElem :: [a] -> a
giveFourthElem = head . tripleTail

tripleTail :: [a] -> [a]
tripleTail = tail . tail . tail
```

# Funktionale, Teil 2 ...Funktionen als Resultate (3)

In komplexen Situationen einfacher zu verstehen und zu ändern als die argumentversehenen Varianten...

Vergleiche folgende zwei argumentversehene Varianten der Funktion `giveFourthElem :: [a] -> a` ...

```
giveFourthElem ls = (head . tripleTail) ls  -- Variante 1
giveFourthElem ls = head (tripleTail ls)    -- Variante 2
```

...mit der argumentlosen Variante

```
giveFourthElem = head . tripleTail
```

# Funktionen als Resultate – Weitere Beispiele (1)

Iterierte Funktionsanwendung...

```
iterate :: Int -> (a -> a) -> (a -> a)
iterate n f
  | n > 0     = f . iterate (n-1) f
  | otherwise = id

id :: a -> a
id a = a

-- Anwendungsbeispiel
(iterate 3 square) 2
     --> (square . square . square . id) 2 --> 256
```

# Funktionen als Resultate – Weitere Beispiele (2)

Anheben (engl. *lifting*) eines Wertes zu einer (konstanten) Funktion...

```
constFun :: a -> (b -> a)
constFun c = \x -> c

-- Anwendungsbeispiele
constFun 42 "Die Anwort auf alle Fragen"    -->  42
constFun iterate giveFourthElem             -->  iterate
```

Vertauschen von Argumenten...

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

-- Anwendungsbeispiel und Eigenschaft von flip
flip . flip                                 --> id
```

# Funktionen als Resultate – Partielle Auswertung (1)

Insbesondere die Spezialfälle der sog. *operator sections*.

*Schlüssel*: ...partielle Auswertung / partiell ausgewertete Operatoren

- `(*2)` ...die Funktion, die ihr Argument verdoppelt.
- `(2*)` ...s.o.
- `(42<)` ...das Prädikat, das sein Argument daraufhin überprüft, größer oder gleich `42` zu sein.
- `(42:)` ...die Funktion, die `42` an den Anfang einer typkompatiblen Liste setzt.
- ...

# Funktionen als Resultate – Partielle Auswertung (2)

Partiell ausgewertete Operatoren...

...besonders elegant und ausdruckskräftig in Kombination mit Funktionalen und Funktionskomposition.

*Beispiel*:

```
fancySelect :: [Int] -> [Int]
fancySelect = filter (42<) . map (*2)
```

...multipliziert jedes Element einer Liste mit 2 und entfernt anschließend alle Elemente, die kleiner oder gleich 42 sind.

```
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []
```

...kehrt eine Liste um.

# Anm. zur Funktionskomposition

*Beachte*:

Funktionskomposition...

- ...ist assoziativ, d.h. `f . (g . h) = (f . g) . h = f . g. h`
- ...erfordert aufgrund der Bindungsstärke explizite Klammerung. (Bsp.: `head . tripleTail ls` in Variante 1 von Folie 16 führt zu Typfehler.)
- ...sollte auf keinen Fall mit Funktionsapplikation verwechselt werden: `f . g` (Komposition) ist verschieden von `f g` (Applikation)!

# Zwischenresümee 2: Funktionen als Resultate...

...von Funktionen (gleichberechtigt zu elementaren Werten) zuzulassen...

- ...ist der Schlüssel, Funktionen miteinander zu verknüpfen und in Programme einzubringen
- ...unterscheidet funktionale Programmierung signifikant von anderen Programmierparadigmen
- ...ist maßgeblich für die Eleganz und Ausdruckskraft und Prägnanz funktionaler Programmierung.

Damit bleibt (möglicherweise) die Frage...

- Wie erhält man funktionale Ergebnisse?

# Einige Standardtechniken zur Entwicklung...

...von Funktionen mit funktionalen Ergebnissen:

- Explizit (Bsp.: `extreme`, `iterate`,...)
- Partielle Auswertung (curryfiziert vorliegender Funktionen) (Bsp.: `curriedAdd 4711 :: Int->Int`, `iterate 5 :: (a->a)->(a->a)`,...)
  - Spezialfall: operator sections (Bsp.: `(*2)`, `(<2)`,...)
- Funktionskomposition (Bsp.: `tail . tail . tail :: [a]->[a]`,...)
- $\lambda$-Lifting (Bsp.: `constFun :: a -> (b -> a)`,...)

# Spezialfall: Funktionale auf Listen

Typische Problemstellungen...

- Behandlung aller Elemente einer Liste in bestimmter Weise
- Herausfiltern aller Elemente einer Liste mit bestimmter Eigenschaft
- Aggregation aller Elemente einer Liste mittels eines bestimmten Operators
- ...

# Standardfunktionale auf Listen

...werden in fkt. Programmiersprachen in großer Zahl offeriert, auch in Haskell.

Drei in der Praxis besonders häufig verwendete Funktionale sind die Funktionale...

- `map`
- `filter`
- `fold`

# Das Standardfunktional `map` (1)

```
-- Signatur
map :: (a -> b) -> [a] -> [b]

-- Mittels Listenkomprehension (Variante 1)
map f ls = [ f l | l <- ls ]

-- Mittels (expliziter) primitiver Rekursion (Variante 2)
map f []     = []
map f (l:ls) = f l : map f ls

-- Anwendungsbeispiel
map square [2,4..10] = [4,16,36,64,100]
```

# Das Standardfunktional `map` (2)

Einige Eigenschaften von `map`...

- Allgemein:

$$\texttt{map (\textbackslash x -> x)} = \texttt{\textbackslash x -> x}$$
$$\texttt{map (f . g)} = \texttt{map f . map g}$$
$$\texttt{map f . tail} = \texttt{tail . map f}$$
$$\texttt{map f . reverse} = \texttt{reverse . map f}$$
$$\texttt{map f . concat} = \texttt{concat . map (map f)}$$
$$\texttt{map f (xs ++ ys)} = \texttt{map f xs ++ map f ys}$$

- (Nur) für strikte f:

$$\texttt{f . head} = \texttt{head . (map f)}$$

# Das Standardfunktional `filter`

```
-- Signatur
filter :: (a -> Bool) -> [a] -> [a]

-- Mittels Listenkomprehension
filter p ls = [ l | l <- ls, p l ]

-- Mittels (expliziter) primitiver Rekursion
filter p []     = []
filter p (l:ls)
  | p l       = l : filter p ls
  | otherwise =     filter p ls

-- Anwendungsbeispiel
filter isPowerOfTwo [2,4..100] = [2,4,8,16,32,64]
```

# Das Standardfunktional `fold` (1)

...“Falten” von rechts: `foldr`.

```
-- Signatur ("folding from the right")
foldr :: (a -> b -> b) -> b -> [a] -> b

-- Mittels (expliziter) primitiver Rekursion
foldr f e []     = e
foldr f e (l:ls) = f l (foldr f e ls)

-- Anwendungsbeispiel
foldr (+) 0 [2,4..10] = 30
foldr (+) 0 []        = 0
```

*In obiger Definition*:
`f` ...binäre Funktion, `e` ...Startwert, und
`(l:ls)` ...Liste der zu aggregierenden Werte.

# Das Standardfunktional `fold` (2)

Anwendungen von `foldr` zur Definition einiger Standardfunktionen von Haskell...

```
concat :: [[a]] -> [a]
concat ls = foldr (++) [] ls

and :: [Bool] -> Bool
and bs = foldr (&&) True bs
```

# Das Standardfunktional `fold` (3)

...“Falten” von links: `foldl`.

```
-- Signatur ("folding from the left")
foldl :: (a -> b -> a) -> a -> [b] -> a

-- Mittels (expliziter) primitiver Rekursion
foldl f e []     = e
foldl f e (l:ls) = foldl f (f e l) ls

-- Anwendungsbeispiel
foldl (+) 0 [2,4..10] = 30
foldl (+) 0 []        = 0
```

*In obiger Definition*:
`f` ...binäre Funktion, `e` ...Startwert und
`(l:ls)` ...Liste der zu aggregierenden Werte.

# Das Standardfunktional `fold` (4)

...`foldr` vs. `foldl` – ein Vergleich:

```
-- Signatur ("folding from the right")
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e []     = e
foldr f e (l:ls) = f l (foldr f e ls)

foldr f e [a1,a2,...,an]
     ---> a1 'f' (a2 'f' ... 'f' (an-1 'f' (an 'f' e))...)

-- Signatur ("folding from the left")
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f e []     = e
foldl f e (l:ls) = foldl f (f e l) ls

foldr f e [b1,b2,...,bn]
     ---> (...((e 'f' b1) 'f' b2) 'f' ... 'f' bn-1) 'f' bn
```

# Zwischenresümee 3

Typisch für funktionale Programmiersprachen ist...

- Elemente (Werte/Objekte) aller (Daten-) Typen sind *Objekte erster Klasse* (engl. *first-class citizens*),

  Das heißt: Jedes Datenobjekt kann...

  - Argument und Wert einer Funktion sein
  - in einer Deklaration benannt sein
  - Teil eines strukturierten Objekts sein

# Folgendes Beispiel...

...illustriert dies sehr kompakt:

```
magicType = let
            pair x y z = z x y
            f y = pair y y
            g y = f (f y)
            h y = g (g y)
            in h (\x->x)
```

*Preisfragen*:

- Welchen Typ hat `magicType`?
- Wie ist es Hugs möglich, diesen Typ zu bestimmen?

# Zwischenresümee 4: Rechnen mit Funktionen...

Im wesentlichen folgende Quellen, Funktionen in Programme einzuführen...

- Explizite Definition im (Haskell-) Skript
- Ergebnis anderer Funktionen/Funktionsanwendungen
  - Explizit mit funktionalem Ergebnis
  - Partielle Auswertung
    - Spezialfall: Operator sections
  - Funktionskomposition
  - $\lambda$-Lifting

# Vorteile der Programmierung mit Funktionalen...

- Kürzere und i.a. einfacher zu verstehende Programme
  ...wenn man die Semantik (insbesondere) der grundlegenden Funktionen und Funktionale (`map`, `filter`,...) verinnerlicht hat.

- Einfachere Herleitung und Beweis von Programmeigenschaften (*Stichwort*: Programmverifikation )
  ...da man sich auf die Eigenschaften der zugrundeliegenden Funktionen abstützen kann.

- ...

- Wiederverwendung von Programmcode
  ...und dadurch Unterstützung des *Programmierens im Großen*.

# Stichwort Wiederverwendung

Wesentliche Quellen für Wiederverwendung in fkt. Programmiersprachen sind...

- Polymorphie
- Funktionale

# Stärken funktionaler Programmierung

...resultieren aus wenigen Konzepten

- sowohl bei Funktionen
- als auch bei Datentypen

Die Ausdruckskraft ergibt sich in beiden Fällen durch die Kombination der Einzelstücke.

$\leadsto$ ...*das Ganze ist mehr als die Summe seiner Teile!*

Für mehr Details: siehe Peter Pepper [4]

# Vorschau auf die nächsten Vorlesungstermine...

- **Vorgezogener Termin**: Do, 16.12.2004, 18-19 Uhr, Treitl-Hörsaal
- Di, 04.01.2005, keine Vorlesung, Ferialzeit
- Di, 11.01.2005, **keine** Vorlesung (vorgezogen auf den 16.12.2004)
- Di, 18.01.2005, 13-14 Uhr, Treitl-Hörsaal
- Di, 25.01.2005, 13-14 Uhr, Treitl-Hörsaal (letzte Vorlesung)