Heutige Themen

- **Teil 1:** Funktionen höherer Ordnung (*kurz*: Funktionale)
  - Funktionen als Argumente
  - Funktionen als Resultate
  - Spezialfall: Funktionale auf Listen
  - Anwendungen

  ...und ihre Vorteile für die Programmierung.
- **Teil 2:** Ein- und Ausgabe
- **Teil 3:** Fehlerbehandlung

Zuvor: Hinweis zu Aufgabenblatt 6 (1)

```
data Graph b = Nil |
               Node b [(b,Int)] (Graph b) deriving (Eq,Show)
```

Wir können unmittelbar definieren:

```
isEq g h = g == h
```

wobei

```
isEq :: Eq a => a -> a -> Bool
```

die allgemeinste (und automatisch inferierte) Typisierung von `isEq` ist und

```
isEq :: Eq (Graph b) => (Graph b) -> (Graph b) -> Bool
```

eine ebenfalls mögliche speziellere Typisierung wäre.

Zuvor: Hinweis zu Aufgabenblatt 6 (2)

Einige Aufrufbeispiele:

```
g1 = Nil :: Graph Integer
g2 = Node 5 [] Nil :: Graph Integer
g3 = Node 6 [] Nil :: Graph Integer
g4 = Node 6 [] Nil :: Graph Integer
g5 = Nil :: Graph Float

isEq g1 g2 == False
isEq g3 g4 == True

isEq Nil Nil == True
isEq (Node 5 [] Nil) Nil == False

isEq g1 g5 => Typfehler!
```

Zuvor: Hinweis zu Aufgabenblatt 6 (3)

```
isDummyEq1 :: Eq b => (Graph b) -> (Graph b) -> Bool
isDummyEq1 Nil Nil                          = True
isDummyEq1 (Node n l1 g1) (Node m l2 g2) = n == m
isDummyEq1 _ _                              = False

isDummyEq2 :: Eq (Graph b) => (Graph b) -> (Graph b) -> Bool
isDummyEq2 Nil Nil                          = True
isDummyEq2 (Node n l1 g1) (Node m l2 g2) = (Node n [] Nil)
                                               == (Node m [] Nil)
isDummyEq2 _ _                              = False
```

Einige Aufrufbeispiele:

```
isDummyEq1 g1 g3 == False
isDummyEq1 g3 g4 == True

isDummyEq2 g1 g1 == True
isDummyEq2 g2 g3 == False
```

Zuvor: Hinweis zu Aufgabenblatt 6 (4)

Gefälliger ist eine "Wrapper"-Lösung:

```
isDummyEq3 :: Eq (Graph b) => (Graph b) -> (Graph b) -> Bool
isDummyEq3 Nil Nil                          = True
isDummyEq3 (Node n l1 g1) (Node m l2 g2) = (wrapper n)
                                              == (wrapper m)
isDummyEq3 _ _                              = False

wrapper :: b -> Graph b
wrapper x = Node x [] Nil
```

Einige Aufrufbeispiele:

```
isDummyEq3 g1 g3 == False
isDummyEq3 g3 g4 == True
```

Kontexte bei Aufgabenblatt 6 entsprechend ergänzen!

Teil 1: 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:

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

- Informatik: *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 Σ mit $\Sigma =_{def} \{\sigma \,|\, \sigma \in [Var \ \to \ Data]\}$.

    (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!

Des Pudels Kern

...bei Funktionalen:

Wiederverwendung!

...wie auch schon bei Funktionsabstraktion und Polymorphie.

Dies wollen wir in der Folge genauer herausarbeiten...

Funktionale – Motivation 1(6)

(siehe Fethi Rabhi, Guy Lapalme. Algorithms - A Functional Approach, Addison-Wesley, 1999, S. 7f.)

Betrachte folgende Beispiele...

```
-- Fakultaetsfunktion
fact n  | n==0   = 1
        | n>0    = n * fact(n-1)

-- Aufsummieren der n ersten natuerlichen Zahlen
sumNat n | n==0  = 0
         | n>0   = n + sumNat(n-1)

-- Aufsummieren der n ersten Quadratzahlen natuerlicher Zahlen
sumSquNat n | n==0  = 0
            | n>0   = n*n + sumSquNat(n-1)
```

Funktionale – Motivation 2(6)

Beobachtung...

- Die Definitionen von `fact`, `sumNat` und `sumSquNat` folgen demselben *Rekursionsschema*.
- Dieses zugrundeliegende gemeinsame Rekursionsschema ist gekennzeichnet durch:
  - Triff eine Festlegung für den Wert der Funktion...
    - ...im *Basisfall*
    - ...im verbleibenden (rekursiven) Fall als *Kombination* des Argumentwerts `n` und des Funktionswerts für `n-1`

Funktionale – Motivation 3(6)

Diese Beobachtung legt nahe...

- Obiges Rekursionsschema, gekennzeichnet durch *Basisfall* und *Funktion zur Kombination von Werten*, herauszuziehen (zu abstrahieren) und musterhaft zu realisieren.

Wir erhalten...

- Realisierung des Rekursionsschemas

```
rekPatt base comb n | n==0  = base
                    | n>0   = comb n (rekPatt base comb (n-1))
```

Funktionale – Motivation 4(6)

Unmittelbare Anwendung des Rekursionsschemas...

```
fact n      = rekPatt 1 (*) n

sumNat n    = rekPatt 0 (+) n

sumSquNat n = rekPatt 0 (\x y -> x*x + y) n
```

...oder alternativ dazu in nichtargumentbehafteter Ausprägung:

```
fact      = rekPatt 1 (*)

sumNat    = rekPatt 0 (+)

sumSquNat = rekPatt 0 (\x y -> x*x + y)
```

Funktionale – Motivation 5(6)

Unmittelbarer Vorteil obigen Vorgehens...

- *Wiederverwendung* und dadurch...
  - *kürzerer*, *verlässlicherer*, *wartungsfreundlicherer* Code

Erforderlich für erfolgreiches Gelingen...

- *Funktionen höherer Ordnung* oder kürzer: *Funktionale*

  Intuition: Funktionale sind (spezielle) Funktionen, die Funktionen als Argumente erwarten und/oder als Resultat zurückliefern.

Funktionale – Motivation 6(6)

Illustriert am obigen Beispiel...

- Die Untersuchung des Typs von `rekPatt`...

  ```
  rekPatt :: Int -> (Int -> Int -> Int) -> Int
  ```

  zeigt:

  – Die Funktion `rekPatt` ist ein *Funktional*!

In der Anwendungssituation des Beispiels gilt weiter...

	Wert i. Basisf. (base)	Fkt. z. Kb. v. W. (comb)
`fact`	1	`(*)`
`sumNat`	0	`(+)`
`sumSquNat`	0	`\x y -> x*x + y`

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

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  _      _     = []
```

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
(constFun iterate (+) 3 (\x->x*x)) 2        =>  256
```

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 `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.

Bem.: `map`, `filter`, `foldl` und `flip` werden im Verlauf der heutigen Vorlesung noch genauer besprochen.

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 20 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]`,...)
- λ-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 auf Listen sind die Funktionale...

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

Das Standardfunktional `map` (1)

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

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

-- Implementierung 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:

```
map (\x -> x)     =  \x -> x
map (f . g)       =  map f . map g
map f . tail      =  tail . map f
map f . reverse   = reverse . map f
map f . concat    = concat . map (map f)
map f (xs ++ ys) = map f xs ++ map f ys
```

- (Nur) für strikte f:

```
f . head  =  head . (map f)
```

Das Standardfunktional `filter`

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

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

-- Implementierung 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

-- Implementierung 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] = (+ 2 (+ 4 (+ 6 (+ 8 (+ 10 0)))))
                      = (2 + (4 + (6 + (8 + (10 + 0))))) = 30
foldr (+) 0 []        = 0
```

In obiger Definition bedeuten:
`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] = (+ (+ (+ (+ (+ 0 2) 4) 6) 8) 10)
                      = (((((0 + 2) + 4) + 6) + 8) + 10) = 30
foldl (+) 0 []        = 0
```

In obiger Definition bedeuten:
`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

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

Der Vollständigkeit halber

Das vordefinierte Funktional `flip`:

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

Anwendungsbeispiel: Listenreversion

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

reverse [1,2,3]  =>  [3,2,1]
```

Zur Übung empfohlen: Nachvollziehen, dass `reverse` wie oben das Gewünschte leistet!

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
  - λ-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 (auf Funktionen und Datentypen)
- 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 eine detaillierte Diskussion: siehe Peter Pepper [4]

Teil 2: Ein- und Ausgabe

Die Behandlung von...

- Ein-/ Ausgabe in Haskell

  $\leadsto$ Einstieg in das *Monadenkonzept* von Haskell

...wird uns an die Schnittstelle von *funktionaler* und *imperativer* Programmierung führen!

Hello World!

```
helloWorld :: IO ()
helloWorld = putStr "Hello World!"
```

Hello World...

- ...gewöhnlich eines der ersten Beispielprogramme in einer neuen Programmiersprache
- ...in dieser LVA erst im letzten Drittel!

Ungewöhnlich?

Zum Vergleich...

Ein-/Ausgabe-Behandlung in...

- S. Thompson [3]: ...in Kapitel 18 (von 20)
- P. Pepper [4]: ...in Kapitel 21&22 (von 23)
- R. Bird [2]: ...in Kapitel 10 (von 12)
- A. J. T. Davie. "*An Introduction to Functional Programming Systems Using Haskell*", Cambridge, 1992. ...in Kapitel 7 (von 11)
- M. M. T. Chakravarty, G. C. Keller. "*Einführung in die Programmierung mit Haskell*", Pearson Studium, 2004. ...in Kapitel 7 (von 13)

Zufall?

...oder ist Ein-/Ausgabe möglicherweise

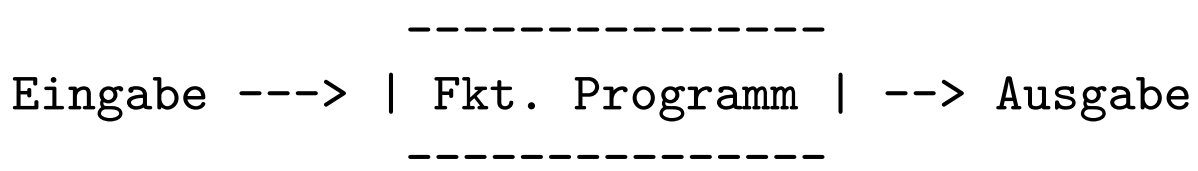
- ...weniger wichtig in funktionaler Programmierung?
- ...oder in besonderer Weise problembehaftet?

Tendenziell letzteres...

- Ein-/Ausgabe... führt uns an den Berührungspunkt von *funktionaler* und *imperativer* Programmierung!

Rückblick...

Unsere bisherige Sicht fkt. Programmierung...



In anderen Worten...

Unsere bisherige Sicht fkt. Programmierung ist...

- *stapelverarbeitungsfokussiert*
- nicht *dialog-* und *interaktionsorientiert*

...wie es heutigen Anforderungen und heutiger Programmierrealität entspricht.

Erinnerung

Im Vergleich zu anderen Paradigmen...

- Das funktionale Paradigma betont das "was" (Ergebnisse) zugunsten des "wie" (Art der Berechnung der Ergebnisse)

Von zentraler Bedeutung dafür...

- Der Wert eines Ausdrucks hängt nur von den Werten seiner Teilausdrücke ab
  Stichwort(*e*): ...Kompositionalität (ref. Transparenz)
  $\leadsto$ erleichtert Programmentwicklung und Korrektheitsüberlegungen

- Auswertungsabhängigkeiten, nicht aber Auswertungsreihenfolgen dezidiert festgelegt
  Stichwort(*e*): ...Flexibilität (Church-Rosser-Eigenschaft)
  $\leadsto$ erleichtert Implementierung einschl. Parallelisierung

- ...

Angenommen...

...wir hätten Konstrukte der Art (*Achtung*: Kein Haskell!)

```
PRINT :: String -> a -> a
PRINT message value =
        << gib "message" am Bildschirm aus und liefere >>
        value

READFL :: Float
READFL = << lies Gleitkommazahl und liefere diese als Ergebnis >>
```

Mithin:

...Hinzunahme von Ein-/Ausgabe mittels seiteneffektbehafteter Funktionen!

Knackpunkt 1: Kompositionalität (1)

Vergleiche...

```
val :: Float
val = 3.14

valDiff :: Float
valDiff = val - val
```

mit...

```
readDiff :: Float
readDiff = READFL - READFL
```

und der Anwendung in...

```
constFunOrNot :: Float
constFunOrNot = valDiff + readDiff
```

Knackpunkt 1: Kompositionalität (2)

Beachte: ...der Wert von Ausdrücken hinge nicht länger nur von seinen Teilausdrücken ab (sondern auch von der Position im Programm)

$\leadsto$ ...Verlust von *Kompositionalität*

(...und der damit einhergehenden positiven Eigenschaften).

Knackpunkt 2: Flexibilität (1)

...oder der Verlust der Unabhängigkeit von der Auswertungsreihenfolge

Vom "Punkt"...

```
punkt r =
  let
  myPi = 3.14
  x    = r * myPi
  y    = r + 17.4
  z    = r * r
  in (x,y,z)
```

Knackpunkt 2: Flexibilität (2)

...zum "Knackpunkt" (*Achtung*: Kein Haskell!):

```
knackpunkt r =
  let
  myPi = PRINT "Constant Value" 3.14
  u    = PRINT "Erstgelesener Wert" dummy
  c    = READFL
  x    = r * c
  v    = PRINT "Zweitgelesener Wert" dummy
  d    = READFL
  y    = r + d
  z    = r * r
  in (x,y,z)
```

$\leadsto$ ...Verlust der *Auswertungsreihenfolgenunabhängigkeit*

Ergo...

Konzentration auf die Essenz der Programmierung wie im funktionalen Paradigma ("was" statt "wie") ist wichtig und richtig, aber...

- Kommunikation mit dem Benutzer (bzw. der Außenwelt) muss die zeitliche Abfolge von Aktivitäten auszudrücken gestatten.

In den Worten von P. Pepper [4]:

- ..."*der Benutzer lebt in der Zeit und kann nicht anders als zeitabhängig sein Programm beobachten*".

Konsequenz ...man (bzw. ein jedes Paradigma) darf von der Arbeitsweise des Rechners, nicht aber von der des Benutzers abstrahieren!

Haskells Ansatz zur Behandlung von Ein- und Ausgabe

Zentral...

- Elementare Ein-/Ausgabeoperationen (Kommandos) auf speziellen Typen (IO-Typen) sowie
- (Kompositions-) Operatoren, um Anweisungssequenzen (Kommandosequenzen) auszudrücken

Damit...

- Trennung von
  - funktionalem Kern und
  - imperativähnlicher Ein-/Ausgabe

...somit gelangen wir an die Schnittstelle von funktionaler und imperativer Welt!

(Ausgewählte) elementare Ein-/Ausgabeoperationen

```
-- Eingabe
getChar :: IO Char
getLine :: IO String

-- Ausgabe
putChar :: Char -> IO ()
putLine :: String -> IO ()
putStr  :: String -> IO ()
```

Bemerkung:

- `()`: ...spezieller einelementiger Haskell-Typ, dessen einziges Element (ebenfalls) mit `()` bezeichnet wird.

- `IO a`: ...spezieller Haskell-Typ "*I/O Aktion* (*Kommando*) *vom Typ* `a`" . `IO`: ...Typkonstruktor (ähnlich wie `[a]` für Listen oder `->` für Funktionstypen)

Kompositionsoperatoren

```
(>>)  :: IO a -> IO b -> IO b
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Intuitiv:

- `(>>)`: Wenn `p` und `q` Kommandos sind, dann ist `p >> q` das Kommando, das zunächst `p` ausführt, den Rückgabewert (`x` vom Typ `a`) ignoriert, und anschließend `q` ausführt.

- `(>>=)`: Wenn `p` und `q` Kommandos sind, dann ist `p >>= q` das Kommando, das zunächst `p` ausführt, dabei den Rückgabewert `x` vom Typ `a` liefert, und daran anschließend `q x` ausführt und dabei den Rückgabewert `y` vom Typ `b` liefert.

Einfache Anwendungsbeispiele

```
-- Schreiben mit Zeilenvorschub (Standardoperation in Haskell)
putStrLn :: String -> IO ()
putStrLn = putStr . (++ "\n")

-- Lesen einer Zeile und Ausgeben der gelesenen Zeile
echo :: IO ()
echo = getLine >>= putLine
```

Weitere Ein-/Ausgabeoperationen

```
-- Schreiben und Lesen von Werten unterschiedlicher Typen
print :: Show a => a -> IO ()
print = putStrLn . show

read :: Read a => String -> a

-- Rueckgabewerterzeugung ohne Ein-/Ausgabe(aktion)
return :: a -> IO a
```

Erinnerung:

```
show :: Show a => a -> String
```

Die do-Notation: Bequemere (Kommando-) Sequenzenbildung

Komfortabler als (>>) und (>>=) ist Haskells do-Notation...

```
putStrLn :: String -> IO ()
putStrLn str = do putStr str
                  putStr "\n"

putTwice :: String -> IO ()
putTwice str = do putStrLn str
                  putStrLn str

putNtimes :: Int -> String -> IO ()
putNtimes n str = if n <= 1
                     then putStrLn str
                     else do putStrLn str
                             putNtimes (n-1) str
```

Weitere Beispiele zur `do`-Notation

```
putTwice = putNtimes 2

read2lines :: IO ()
read2lines = do getLine
                getLine
                putStrLn "Two lines read."

echo2times :: IO ()
echo2times = do line <- getLine
                putLine line
                putLine line

getInt :: IO Int
getInt = do line <- getLine
            return (read line :: Int)
```

“Single vs. updatable Assignment” (1)

Durch das Konstrukt

```
var <- ...
```

wird stets eine *frische* Variable eingeführt.

Sprechweise:

...Unterstützung des Konzepts des

- “single assignment”, nicht das des
- “updatable assignment” (destruktive Zuweisung wie aus imperativen Programmiersprachen bekannt)

“Single vs. updatable Assignment” (2)

...zur Illustration des Unterschieds betrachte:

```
goUntilEmpty :: IO ()
goUntilEmpty = do line <- getLine
                  while (return (line /= []))
                        (do putStrLn line
                            line <- getLine
                            return () )
```

wobei `while :: IO Bool -> IO () -> IO ()`

Abhilfe: ...Rekursion statt Iteration!

“Single vs. updatable Assignment” (3)

...z.B. auf folgende in S. Thompson [3] auf S. 393 vorgeschlagene Weise:

```
goUntilEmpty :: IO ()
goUntilEmpty =
    do line <- getLine
       if (line == [])
          then return ()
          else (do putStrLn line
                   goUntilEmpty)
```

Stichwort: Iteration

```
while :: IO Bool -> IO () -> IO ()

while test action
  = do res <- test
       if res then do action
                      while test action
              else return ()              -- "null I/O-action"
```

Erinnerung:

```
-- Rueckgabewerterzeugung ohne Ein-/Ausgabe(aktion)
return :: a -> IO a
```

Ein-/Ausgabe von und auf Dateien

...auch hierfür vordefinierte Standardoperatoren.

```
readFile   :: FilePath -> IO String
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

where

```
type FilePath = String     -- implementationsabhaengig

-- Anwendungsbeispiel: Bestimmung der Laenge einer Datei
size :: IO Int
size = do putLine "Dateiname = "
          name <- getLine
          text <- readFile name
          return(length(text))
```

Zusammenhang `do`-Konstrukt und `(>>)`, `(>>=)`-Operatoren

...illustriert anhand eines Beispiels:

Mittels `do`...

```
incrementInt :: IO ()
incrementInt = do line <- getLine
                  putStrLn (show (1 + read line :: Int))
```

Äquivalent dazu...

```
incrementInt = getLine >>= \line ->
               putStrLn (show (1 + read line :: Int))
```

Intuitiv...

```
"do = (>>=) plus anonyme lambda-Abstraktion"
```

Konvention in Haskell

- Hauptdefinition (übersetzter) Haskell-Programme ist (per Konvention) eine Definition `main` vom Typ `IO a`.

Beispiel:

```
main :: IO ()
main = do c <- getChar
          putChar c
```

...`main` ist Startpunkt eines (übersetzten) Haskell-Programms.
(intuitiv gilt somit: "Programm = Ein-/Ausgabekommando")

Fazit über Ein- und Ausgabe

Es gilt...

- Ein-/Ausgabe grundsätzlich unterschiedlich in funktionaler und imperativer Programmierung

  Am augenfälligsten:

  - *Imperativ*: Ein-/Ausgabe prinzipiell an jeder Programmstelle möglich
  - *Funktional*: Ein-/Ausgabe an bestimmten Programmstellen konzentriert

Häufige Beobachtung...

- ...die vermeintliche Einschränkung erweist sich oft als Stärke bei der Programmierung im Großen!

Ausblick

Das allgemeinere Konzept der

- Monaden in Haskell

und ihr Zusammenhang mit der Realisierung von

- Ein-/Ausgabe in Haskell

...nächstes Mal!

Teil 3: Fehlerbehandlung

... in Haskell:

- Bislang von uns nur rudimentär behandelt.

  Typische Formulierung aus den Aufgabenstellungen:

  ...so hat die Funktion als Ergebnis den aktualisierten Stimmzettel; ansonsten ist das Ergebnis die Zeichenreihe `Ungueltige Eingabe`.

- In der Folge Wege zu einem systematischeren Umgang mit unerwarteten Programmsituationen und Fehlern

Typische Fehlersituationen

- Division durch 0
- Zugriff auf das erste Element einer leeren Liste
- ...

In der Folge...

- 3 Varianten zum Umgang mit solchen Situationen

Variante 1: Panikmodus (1)

- *Berechnung anhalten und Fehlerursache melden.*

Hilfsmittel: Die Funktion `error`...

```
error :: String -> a
```

Aufruf von...

```
error "Unbehebbarer Fehler aufgetreten..."
```

liefert Ausgabe...

```
Program error: Unbehebbarer Fehler aufgetreten...
```

und Programmausführung stoppt.

Variante 1 (2)

Vor- und Nachteile von Variante 1:

- Schnell und einfach
- *Aber*: Die Berechung stoppt unwiderruflich. Jegliche (auch) sinnvolle Information über den Programmlauf ist verloren.

Ziel: Kein *Panikmodus*. Programmlauf nicht gänzlich abbrechen.

Variante 2: Dummy-Werte (1)

- *Verwendung von dummy-Werten im Fehlerfall.*

Statt...

```
tail :: [a] -> [a]
tail (_:xs) = xs
tail []     = error ‘‘PreludeList.tail: empty list’’
```

benutze zum Beispiel...

```
tl :: [a] -> [a]
tl (_:xs) = xs
tl []     = []
```

Variante 2 (2)

Vor- und Nachteile von Variante 2:

- Programmlauf wird nicht abgebrochen
- *Aber*: Ein geeigeter Default-Wert ist nicht immer offensichtlich.

  Betrachte z.B.:

```
hd :: [a] -> a
hd (x:_) = x
hd []    = ???????????
```

Möglicher Ausweg:

- ...jeweils gewünschten Wert als Parameter mitgeben.

  Im obigen Beispiel etwa:

```
hdy :: a -> [a] -> a
hdy y (x:_) = x
hdy y []    = y
```

Variante 2 (3)

Generelles Muster:

Ersetze übliche Implementierung einer (einstelligen) Funktion `f`...

```
f x = ...
```

durch...

```
fErr y x
  | cond      = y
  | otherwise = f x
```

mit `cond` Charakterisierung der Fehlersituation.

Vor- und Nachteile:

- Generell, stets anwendbar
- Auftreten des Fehlerfalls nicht beobachtbar: `y` mag auch als gewöhnliches Ergebnis auftreten

Variante 3: Spezielle Fehlerwerte und -typen (1)

- *Fehlerwerte und -typen statt schlichter dummy-Werte*

```
data Maybe a = Nothing | Just a
               deriving (Eq, Ord, Read, Show)
```

...i.w. der Typ `a` mit dem Zusatzwert `Nothing`.

Damit...

```
fErr x
  | cond      = Nothing
  | otherwise = Just (f x)
```

...und anhand eines konkreten Beispiels:

```
errDiv :: Int -> Int -> Maybe Int
errDiv n m
  | (m == 0)  = Nothing
  | otherwise = Just (n 'div' m)
```

Variante 3 (2)

Vor- und Nachteile von Variante 3:

- Geänderte Funktionalität: statt `a`, jetzt `Maybe a`
- *Aber*:
  - Fehlerursachen können durch einen Funktionsaufruf "hindurchgereicht" werden (der Effekt der Funktion `mapMaybe`...)
  - Fehler können "gefangen" werden (die Rolle von der Funktion `maybe`...)

Variante 3 (3)

Die Funktionen `mapMaybe` und `maybe`:

```
mapMaybe :: (a -> b) -> Maybe a -> Maybe b

mapMaybe g Nothing  = Nothing
mapMaybe g (Just x) = Just (g x)


maybe :: b -> (a -> b) -> Maybe a -> b

maybe n f Nothing  = n
maybe n f (Just x) = f x
```

Variante 3 (4)

Anwendungsbeispiel(e):

Der Fehler wird "(auf-) gefangen":

```
maybe 42 (+1) (mapMaybe (*3) errDiv 9 0))
=> maybe 42 (+1) (mapMaybe (*3) Nothing)
=> maybe 42 (+1) Nothing
=> 42
```

Kein Fehlerfall, "alles geht gut":

```
maybe 42 (+1) (mapMaybe (*3) errDiv 9 1))
=> maybe 42 (+1) (mapMaybe (*3) (Just 9))
=> maybe 42 (+1) (Just 27)
=> 1 + 27
=> 28
```

Variante 3 (5)

Wesentlicher Vorteil der letzten Variante:

- Systementwicklung ohne explizite Fehlerbehandlung möglich.
- Fehlerbehandlung kann am Ende mithilfe der Funktionen `mapMaybe` und `maybe` ergänzt werden.

...für weitere Details siehe S. Thompson [3], Kapitel 14.3.

Vorschau auf die noch kommenden Aufgabenblätter...

Ausgabe des...

- siebenten Aufgabenblatts: Mo, den 27.11.2006
  ...Abgabetermine: Di, den 05.12.2006, und Di, den 12.12.2006, jeweils 15:00 Uhr

- achten Aufgabenblatts: Mo, den 04.12.2006
  ...Abgabetermine: Di, den 12.12.2006, und Di, den 09.01.2007, jeweils 15:00 Uhr

- neunten Aufgabenblatts: Mo, den 11.12.2006
  ...Abgabetermine: Di, den 09.01.2007, und Di, den 16.01.2007, jeweils 15:00 Uhr (letztes Aufgabenblatt)

Vorschau auf die noch kommenden Vorlesungstermine...

- Do, 30.11.2006, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal
- Do, 07.12.2006, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal
- Di, 12.12.2006, Vorlesung von 13:00 Uhr bis 14:00 Uhr im Informatik-Hörsaal, Treitlstr. (letzte Vorlesung)