Kapitel 1: Einführung und Grundlagen

- *Teil 1: Einführung und Motivation*
  - Funktionale Programmierung: Warum? Warum mit Haskell?
  - Erste Schritte in Haskell, erste Schritte mit Hugs
- *Teil 2: Grundlagen*
  - Elementare Datentypen
  - Tupel, Listen und Funktionen

Beachte: ...einige Begriffe werden heute im Vorgriff angerissen und erst im Lauf der Vorlesung genau geklärt!)

Teil 1: Einführung und Motivation

- Funktionale Programmierung: Warum überhaupt? Warum mit Haskell?
- Erste Schritte in Haskell, erste Schritte mit Hugs

Warum funktionale Programmierung?

Ein bunter Strauß an *Programmierparadigmen*, z.B.

- *imperativ*
  - prozedural (Pascal, Modula, C,...)
  - objektorientiert (Smalltalk, Oberon, C++, Java,...)
- *deklarativ*
  - funktional (Lisp, ML, Miranda, Haskell, Gofer,...)
  - logisch (Prolog und Varianten)
- Mischformen
  z.B. funktional/logisch, funktional/objektorientiert,...
- *visuell*
  Stichwort: Visual Programming Languages (VPLs),
  z.B. Forms/3, FAR,...
  Einstieg für mehr: `web.engr.oregonstate.edu/~burnett`
- ...

Ein Vergleich - prozedural vs. funktional

Gegeben eine Aufgabe A.

Prozedural: Typischer Lösungsablauf in folgenden Schritten:

1. Beschreibe eine(n) Lösung(sweg) L für A.
2. Gieße L in die Form einer Menge von Anweisungen (Kommandos) für den Rechner *unter expliziter Organisation der Speicherverwaltung*.

Zur Illustration ein einfaches Beispiel

Betrachte folgende Aufgabe:

"...bestimme die Werte aller Komponenten eines ganzzahligen Feldes, deren Werte kleiner oder gleich 10 *sind."*

Eine typische Lösung *prozedural*...

```
...
j := 1;
FOR i:=1 TO maxLength DO
    IF a[i] <= 10 THEN b[j] := a[i]; j := j+1 FI
OD;
```

Mögliches Problem bei großen Anwendungen:

...inadäquates Abstraktionsniveau $\leadsto$ *Softwarekrise!*

Softwarekrise

- Ähnlich wie objektorientierte Programmierung verspricht deklarative, insbesondere funktionale Programmierung ein angemesseneres Abstraktionsniveau zur Problemlösung zur Verfügung zu stellen

- ...und damit einen Beitrag zur Überwindung der vielzitierten Softwarekrise zu leisten

Zum Vergleich...

...eine typische Lösung *funktional*, hier in Haskell:

```
...
a :: [Int]
b :: [Int]
b = [ n | n <- a, n <= 10 ]
```

Vergleiche diese funktionale Lösung mit: $\{n \mid n \in a \wedge n \leq 10\}$

...und der Idee, etwas von der Eleganz der Mathematik in die Programmierung zu bringen!

Essenz funktionaler Programmierung

...statt des "wie" das "was" in den Vordergrund stellen!

Hilfsmittel im obigen Beispiel:

- *Listenkomprehension* (engl. list comprehension!)
  ...typisch und spezifisch für funktionale Sprachen!

Noch nicht überzeugt?

Betrachte *Quicksort*, ein komplexeres Beispiel...

Aufgabe: Sortiere eine Liste L ganzer Zahlen aufsteigend.

Lösung (mittels Quicksort):

- *Teile*: Wähle ein Element l aus L und partitioniere L in zwei (möglicherweise leere) Teillisten L_1 und L_2 so, dass alle Elemente von L_1 (L_2) kleiner oder gleich (größer) dem Element l sind.

- *Herrsche*: Sortiere L_1 und L_2 mit Hilfe rekursiver Aufrufe von Quicksort.

- *Zusammenführen der Teilergebnisse*: Trivial (die Gesamtliste entsteht durch Konkatenation der sortierten Teillisten).

Quicksort...

...eine typische *prozedurale* (Pseudocode-) Realisierung:

```
quickSort (L,low,high)
  if low < high
    then splitInd = partition(L,low,high)
         quickSort(L,low,splitInd-1)
         quickSort(L,splitInd+1,high) fi

partition (L,low,high)
  l = L[low]
  left = low
  for i=low+1 to high do
    if L[i] <= l then left = left+1
                      swap(L[i],L[left]) fi od
  swap(L[low],L[left])
  return left
```

...mit dem initialen Aufruf `quickSort(L,1,length(L))`.

Zum Vergleich...

...eine typische *funktionale* Realisierung von Quicksort, hier in Haskell:

```
quickSort :: [Int] -> [Int]

quickSort []     = []
quickSort (x:xs) =
    quickSort [ y | y<-xs, y<=x ] ++
                        [x] ++ quickSort [ y | y<-xs, y>x ]
```

Vorteile funktionaler Programmierung

- *Einfach(er) zu erlernen*
  ...da weniger Grundkonzepte (insbesondere: keine Zuweisung, keine Schleifen, keine Sprünge)

- *Höhere Produktivität*
  ...da Programme dramatisch kürzer als funktional vergleichbare imperative Programme (Faktor 5 bis 10)

- *Höhere Zuverlässigkeit*
  ...da Korrektheitsüberlegungen/-beweise einfach(er)
  (math. Hintergrund, keine durchscheinende Maschine)

Nachteile funktionaler Programmierung

- *Geringe*(*re*) *Effizienz*
  ...aber: enorme Fortschritte (Effizienz oft durchaus vergleichbar mit entsprechenden C-Implementierungen), zudem Korrektheit vorrangig gegenüber Geschwindigkeit, weiters einfache(re) Parallelisierbarkeit

- *Gelegentlich unangemessen*, oft für inhärent zustandsbasierte Anwendungen oder zur GUI-Programmierung
  ...aber: Anwendungseignung ist stets zu überprüfen; kein Spezifikum fkt. Programmierung

Warum Haskell?

Ein Blick auf andere funktionale (Programmier-)sprachen...

- λ-Kalkül (Ende der 30er-Jahre, Alonzo Church, Stephen Kleene)
- Lisp (frühe 60er-Jahre, John McCarthy)
- ML, SML (Mitte 70er-Jahre, Michael Gordon, Robin Milner)
- Hope (um 1980, Rod Burstall, David McQueen)
- Miranda (um 1980, David Turner)
- OPAL (Mitte der 80er-Jahre, Peter Pepper et al.)
- Haskell (Ende der 80er-Jahre, Paul Hudak, Philip Wadler et al.)
- Gofer (Anfang der 90er-Jahre, Mark Jones)
- ...

Warum etwa nicht Haskell?

Haskell ist...

- eine fortgeschrittene moderne funktionale Sprache
  - starke Typisierung
  - lazy evaluation
  - Funktionen höherer Ordnung
  - Polymorphie/Generizität
  - pattern matching
  - Datenabstraktion (abstrakte Datentypen)
  - Modularisierung (Programmierung im Großen)
  - ...
- eine Sprache für "real world" Probleme
  (s.a. `http://homepages.inf.ed.ac.uk/wadler/realworld/index.html` (URL noch gültig?))
  - mächtige Bibliotheken
  - Schnittstellen z.B. zu C
  - ...

Nicht zuletzt: Wenn auch reich, ist Haskell eine "gute" Lehrsprache, auch dank Hugs!

Fassen wir noch einmal zusammen...

Steckbrief: **Funktionale Programmierung**

Grundlage:	Lambda-Kalkül
Abstraktion:	Funktionen (höherer Ordnung)
Eigenschaft:	referentielle Transparenz
Historische Bedeutung:	Basis vieler Programmiersprachen
Anwendungsbereiche:	Theoretische Informatik Artificial Intelligence experimentelle Software Programmierunterricht
Programmiersprachen:	Lisp, ML, Miranda, Haskell,...

sowie...

Steckbrief: **Haskell**

benannt nach:	Haskell B. Curry (1900-1982) `http://www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Curry.html`
Paradigma:	rein funktionale Programmierung
Eigenschaften:	lazy evaluation, pattern matching
Typsicherheit:	stark typisiert, Typinferenz modernes polymorphes Typsystem
Syntax:	komprimiert, intuitiv
Informationen:	`http://haskell.org` `http://haskell.org/tutorial/`
Interpretierer:	Hugs (`http://haskell.org/hugs/`)

Erste Schritte in Haskell

Haskell-Programme...

...gibt es in zwei (notationellen) Varianten:

Als sog.

- (*Gewöhnliches*) *Haskell-Skript*

  Intuitiv ...alles, was nicht als Kommentar notationell ausgezeichnet ist, wird als Programmtext betrachtet.

- *Literate Haskell-Skript*

  Intuitiv ...alles, was nicht als Programmtext notationell ausgezeichnet ist, wird als Kommentar betrachtet.

Zur Illustration: Ein Programm als...

...(gewöhnliches) Haskell-Skript:

```
{-#######################################################
   FirstScript.hs  ...‘‘ordinary scripts’’ erhalten
                   die Dateiendung .hs
#######################################################-}

-- Die konstante Funktion sum17and4
sum17and4 :: Int
sum17and4 = 17+4

-- Die Funktion square zur Quadrierung einer ganzen Zahl
square :: Int -> Int
square n = n*n

-- Die Funktion double zur Verdopplung einer ganzen Zahl
double :: Int -> Int
double n = 2*n

-- Die Funktion doubleSquare, eine Anwendung der vorherigen
doubleSquare :: Int
doubleSquare = double (square (4711 - sum17and4))
```

Zum Vergleich das gleiche Programm...

...als literate Haskell-Skript:

```
########################################################
   FirstLiterate.lhs  ...``literate scripts'' erhalten
                      die Dateiendung .lhs
########################################################

Die konstante Funktion sum17and4
>     sum17and4 :: Int
>     sum17and4 = 17+4

Die Funktion square zur Quadrierung einer ganzen Zahl
>     square :: Int -> Int
>     square = n*n

Die Funktion double zur Verdopplung einer ganzen Zahl
>     double :: Int -> Int
>     double = 2*n

Die Funktion doubleSquare, eine Anwendung der vorherigen
>     doubleSquare :: Int
>     doubleSquare = double (square (4711 - sum17and4))
```

Kommentare in Haskell-Programmen

Kommentare in...

- (*gewöhnlichem*) *Haskell-Skript*
  - *einzeilig:* ...bis zum Rest der Zeile nach `--`
  - *mehrzeilig:* ...alles zwischen `{-` und `-}`
- *literate Haskell-Skript*
  - Jede nicht durch `>` eingeleitete Zeile

Konvention: Dateiendung...

- `.hs` für gewöhnliche
- `.lhs` für literate

Haskell-Skripte.

Erste Schritte mit Hugs

Hugs: Der Haskell-Interpretierer...

Aufruf von Hugs: `hugs <fileName>`

...und z.B. im Fall von `FirstScript` für `<fileName>` weiter mit:

```
Main> double (sum17and4)
42
```

Wichtige Kommandos in Hugs:

`:?`	Liefert Liste der Hugs-Kommandos
`:load <fileName>`	Lädt die Haskell-Datei `<fileName>` (erkennbar an Endung `.hs` bzw. `.lhs`)
`:reload`	wiederholt letztes Ladekommando
`:quit`	Beendet den aktuellen Hugs-Lauf
`:info name`	Liefert Information über das mit `name` bezeichnete "Objekt"
`:type exp`	Liefert den Typ des Argumentausdrucks `exp`
`:edit <fileName>.hs`	Öffnet die Datei `<fileName>.hs` enthaltende Datei im voreingestellten Editor
`:find name`	Öffnet die Deklaration von `name` im voreingestellten Editor
`!<com>`	Ausführen des Unix- oder DOS-Kommandos `<com>`

...mehr dazu: `http://www.haskell.org/hugs/`

Fehlermeldungen&Warnungen in Hugs

- Fehlermeldungen
  - Syntaxfehler
    `Main> sum17and4 == 21)` ...liefert
    `ERROR: Syntax error in input (unexpected ‘)’)`
  - Typfehler
    `Main> sum17and4 + False` ...liefert
    `ERROR: Bool is not an instance of class ‘‘Num’’`
  - Programmfehler
    ...später
  - Modulfehler
    ...später
- Warnungen
  - Systemmeldungen
    ...später

...mehr zu Fehlermeldungen:

`http://www.cs.kent.ac.uk/people/staff/sjt/craft2e/errors.html`

Bequem...

Haskell stellt umfangreiche Bibliotheken (`Prelude.hs`,...) mit vielen vordefinierten Funktionen zur Verfügung, z.B. zum

- Umkehren von Zeichenreichen (`reverse`)
- Aufsummieren von Listenelementen (`sum`)
- Verschmelzen von Listen (`zip`)
- ...

Exkurs: Mögliche Namenskonflikte

...soll eine Funktion gleichen (bereits vordefinierten) Namens deklariert werden, können Namenskonflikte durch *Verstecken* (engl. *hiding*) vordefinierter Namen vermieden werden.

Am Beispiel von `reverse`, `sum`, `zip`:

Ergänze...

```
import Prelude hiding (reverse,sum,zip)
```

...am Anfang des Haskell-Skripts im Anschluss an die Modul-Anweisung (so vorhanden), wodurch die vordefinierten Namen `reverse`, `sum` und `zip` verborgen werden.

(Mehr dazu später im Zusammenhang mit dem Modulkonzept von Haskell).

Teil 2: Grundlagen

- Elementare Datentypen (`Bool`, `Int`, `Integer`, `Float`, `Char`)
- Tupel, Listen und Funktionen

Elementare Datentypen

...werden in der Folge nach nachstehendem Muster angegeben:

- Name des Typs
- Typische Konstanten des Typs
- Typische Operatoren (und Relatoren, so vorhanden)

(Ausgew.) Elementare Datentypen (1)

Wahrheitswerte

Typ	`Bool`	Wahrheitswerte
Konstanten	`True :: Bool`	Symbol für “wahr”
	`False :: Bool`	Symbol für “falsch”
Operatoren	`&& :: Bool -> Bool -> Bool`	logisches und
	`|| :: Bool -> Bool -> Bool`	logisches oder
	`not :: Bool -> Bool`	logische Negation

Elementare Datentypen (2)

Ganze Zahlen

Typ	`Int`	Ganze Zahlen(endl. Ausschn.)
Konstanten	`0 :: Int`	Symbol für “0”
	`-42 :: Int`	Symbol für “-42”
	`2147483647 :: Int`	Wert für “maxInt”
	...	
Operatoren	`+ :: Int -> Int -> Int`	Addition
	`* :: Int -> Int -> Int`	Multiplikation
	`^ :: Int -> Int -> Int`	Exponentiation
	`- :: Int -> Int -> Int`	Subtraktion (Infix)
	`- :: Int -> Int`	Vorzeichenwechsel (Prefix)
	`div :: Int -> Int -> Int`	Division
	`mod :: Int -> Int -> Int`	Divisionsrest
	`abs :: Int -> Int`	Absolutbetrag
	`negate :: Int -> Int`	Vorzeichenwechsel

Elementare Datentypen (3)

Ganze Zahlen (fortgesetzt)

Relatoren		
	`> :: Int -> Int -> Bool`	echt größer
	`>= :: Int -> Int -> Bool`	größer gleich
	`== :: Int -> Int -> Bool`	gleich
	`/= :: Int -> Int -> Bool`	ungleich
	`<= :: Int -> Int -> Bool`	keiner gleich
	`< :: Int -> Int -> Bool`	echt kleiner

...die Relatoren == und /= sind auf Werte aller Elementar- und vieler weiterer Typen anwendbar, beispielsweise auch auf Wahrheitswerte (Stichwort: *Überladen* (engl. *Overloading*)!

...mehr dazu später.

Elementare Datentypen (4)

Ganze Zahlen (Variante)

Typ	`Integer`	Ganze Zahlen
Konstanten	`0 :: Integer`	Symbol für “0”
	`-42 :: Integer`	Symbol für “-42”
	`21474836473853883234 :: Integer`	“Große” Zahl
	...	
Operatoren	...	

...wie `Int`, jedoch ohne a priori Beschränkung für eine maximal darstellbare Zahl.

Elementare Datentypen (5)

Gleitkommazahlen

Typ	`Float`	Gleitkommazahlen (endlicher Ausschnitt)
Konstanten	`0.123 :: Float`	Symbol für "0,123"
	`-42.4711 :: Float`	Symbol für "-42,4711"
	`123.6e-2 :: Float`	$123{,}6 \times 10^{-2}$
	...	
Operatoren	`+ :: Float -> Float -> Float`	Addition
	`* :: Float -> Float -> Float`	Multiplikation
	...	
	`sqrt :: Float -> Float`	(pos.) Quadratwurzel
	`sin :: Float -> Float`	sinus
	...	
Relatoren	`== :: Float -> Float -> Bool`	gleich
	`/= :: Float -> Float -> Bool`	ungleich
	...	

Elementare Datentypen (6)

Zeichen

Typ	`Char`	Zeichen (Literal)
Konstanten	`'a' :: Char`	Symbol für “a”
	...	
	`'Z' :: Char`	Symbol für “Z”
	`'\t' :: Char`	Tabulator
	`'\n' :: Char`	Neue Zeile
	`'\\' :: Char`	Symbol für “backslash”
	`'\'' :: Char`	Hochkomma
	`'\"' :: Char`	Anführungszeichen
Operatoren	`ord :: Char -> Int`	Konversionsfunktion
	`chr :: Int -> Char`	Konversionsfunktion

Zusammengesetzte Datentypen und Funktionen...

- Tupel
- Listen
  - *Spezialfall*: Zeichenreihen
- Funktionen

Tupel

Tupel ...fassen eine festgelegte Zahl von Werten möglicherweise verschiedener Typen zusammen.

$\leadsto$ Tupel sind *heterogen*!

Beispiele:

- ...Modellierung von Studentendaten

```
("Max Mustermann","e0123456@student.tuwien.ac.at",534) ::
                                (String, String, Int)
```

- ...Modellierung von Bibliotheksdaten

```
("PeytonJones","Implementing Funct. Lang.",1987,True) ::
                                (String, String, Int, Bool)
```

Tupel...

- Allgemeines Muster

```
(v1,v2,...,vk) :: (T1,T2,...,Tk)
```

  mit `v1,...,vk` Bezeichnungen von Werten und `T1,...,Tk` Bezeichnungen von Typen mit

```
v1 :: T1, v2 :: T2,..., vk :: Tk
```

  Lies: `vi` ist vom Typ `Ti`

- Standardkonstruktor

```
(. , . , ... , . )
```

Spezialfall: Paare (“Zweitupel”)

- Beispiele

```
type Point = (Float, Float)

(0.0,0.0) :: Point
(3.14,17.4) :: Point
```

- Standardselektoren (für Paare)

```
fst (x,y) = x
snd (x,y) = y
```

- Anwendung der Standardselektoren

```
fst (0.0,0.0) = 0.0
snd (3.14,17.4) = 17.4
```

Hilfreich...

Typsynonyme

```
type Student = (String, String, Int)
type Buch = (String, String, Int, Bool)
```

...erhöhen die Transparenz in Programmen.

Wichtig: Typsynonyme definieren *keine* neuen Typen, sondern einen Namen für einen schon existierenden Typ (später mehr dazu).

Tupel...

Selbstdefinierte Selektorfunktionen...

```
type Student = (String, String, Int)

name  :: Student -> String
email :: Student -> String
kennzahl :: Student -> Int

name (n,e,k)     = n
email (n,e,k)    = e
kennZahl (n,e,k) = k
```

...mittels *Mustererkennung* (engl. *pattern matching*) (später mehr dazu).

Selbstdefinierte Selektorfunktionen...

Ein weiteres Beispiel...

```
autor :: Buch -> String
kurzTitel :: Buch -> String
erscheinungsJahr :: Buch -> Int
ausgeliehen :: Buch -> Bool

autor (a,t,j,b)  = a
kurzTitel (a,t,j,b)  = t
erscheinungsJahr (a,t,j,b)  = j
ausgeliehen (a,t,j,b)  = b

autEntlehnt (a,t,j,b) = (autor (a,t,j,b), ausgeliehen (a,t,j,b))
```

...auch hier mittels Mustererkennung

Listen

Listen ...fassen eine beliebige/unbestimmte Zahl von Werten gleichen Typs zusammen.

$\leadsto$ Listen sind *homogen*!

Einfache Beispiele:

- Listen ganzer Zahlen
  `[2,5,12,42] :: [ Int ]`
- Listen von Wahrheitswerten
  `[True,False,True] :: [ Bool ]`
- Listen von Gleitkommazahlen
  `[3.14,5.0,12.21] :: [ Float ]`
- Leere Liste
  `[]`
- ...

Listen

Beispiele komplexerer Listen:

- Listen von Listen

  `[[2,4,23,2,5],[3,4],[],[56,7,6,]] :: [[Int]]`

- Listen von Paaren

  `[(3.14,42.0),(56.1,51.3)] :: [(Float,Float)]`

- ...

- *Ausblick*: Listen von Funktionen

  `[fac, abs, negate] :: [Integer -> Integer]`

Vordefinierte Funktionen auf Listen

Die Funktion `length` mit einigen Aufrufen:

```
length :: [a] -> Integer
length []     = 0
length (x:xs) = 1 + length xs

length [1, 2, 3] => 3
length ['a','b','c'] => 3
length [[1],[2],[3]] => 3
```

Die Funktionen `head` und `tail` mit einigen Aufrufen:

```
head :: [a] -> a
head (x:xs) = x

tail :: [a] -> [a]
tail (x:xs) = xs

head [[1],[2],[3]] => [1]
tail [[1],[2],[3]] => [[2],[3]]
```

Spezielle Notationen für Listen

- Spezialfälle (i.w. für Listen über Zahlen und Zeichen)

  ...`[2 .. 6]` kurz für `[2,3,4,5,6]`

  ...`[11,9 .. 2]` kurz für `[11,9,7,5,3]`

  ...`['a','d' .. 'j']` kurz für `['a','d','g','j']`

  ...`[0.0,0.3 .. 1.0]` kurz für `[0.0,0.3,0.6,0.9]`

- *Listenkomprehension*

  ...ein erstes Beispiel:

  `[3*n | n <- list]` kurz für `[3,6,9,12]`, wobei hier `list` vom Wert `[1,2,3,4]` vorausgesetzt ist.

$\leadsto$ Listenkomprehension ist ein sehr elegantes und ausdruckskräftiges Sprachkonstrukt!

Zeichenreihen

...in Haskell als spezielle Listen realisiert:

Typ	`String`	Zeichenreihen
	`type String = [Char]`	Deklaration (als Liste von Zeichen)
Konstanten	`"Haskell" :: String`	Zeichenr. für "Haskell"
	`""`	Leere Zeichenreihe
	...	
Operatoren	`++ :: String -> String -> String`	Konkatenation
Relatoren	`== :: String -> String -> Bool`	gleich
	`/= :: String -> String -> Bool`	ungleich

Weitere Beispiele zu Zeichenreihen

```
['h','e','l','l','o'] == "hello"
"hello" ++ " world" == "hello world"
```

Es gilt:

```
[1,2,3] == 1:2:3:[]
```

Funktionen in Haskell

...am Beispiel der Fakultätsfunktion:

Zur Erinnerung:

$$! : IN \rightarrow IN$$

$$n\,! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n-1)\,! & \text{sonst} \end{cases}$$

...und *eine* mögliche Realisierung in Haskell:

```
fac :: Integer -> Integer
fac n = if n == 0 then 1 else (n * fac(n-1))
```

Beachte: ...Haskell stellt eine Reihe, oft eleganterer, notationeller Varianten zur Verfügung!

Fkt. in Haskell: Notat. Varianten (1)

...am Beispiel der Fakultätsfunktion.

```
fac :: Integer -> Integer
```

(1) In Form "bedingter Gleichungen"

```
fac n
   | n == 0      = 1
   | otherwise   = n * fac (n - 1)
```

$\leadsto$ *Hinweis*: Variante (1) ist "der" Regelfall in Haskell!

Fkt. in Haskell: Notat. Varianten (1)

(2) *λ-artig*

fac = \n -> (if n == 0 then 1 else (n ∗ fac (n − 1)))

- Reminiszenz an den funktionaler Programmierung zugrundeliegenden λ-Kalkül ($\lambda\, x\, y.\ (x + y)$)
- In Haskell: $\backslash\ x\ y \rightarrow\ x{+}y$ sog. *anonyme* Funktion. Praktisch, wenn der Name keine Rolle spielt und man sich deshalb bei Verwendung anonymer Funktionen keinen zu überlegen braucht.

(3) *Gleichungsorientiert*

fac n = if n == 0 then 1 else (n ∗ fac (n − 1))

Fkt. in Haskell: Notat. Varianten (2)

...am Beispiel weiterer Funktionen.

kVA :: Float -> (Float, Float)
    -- Berechnung von Volumen (V) und Fläche (A)
       einer Kugel (K).
       Zur Erinnerung: $V = \frac{4}{3}\pi r^3 \quad A = 4\pi r^2$

Mittels *lokaler Deklarationen*...

(4a) *where*-Konstrukt

```
kVA r =
  ((4/3) * myPi * rcube r, 4 * myPi * square r)
  where
  myPi     = 3.14
  rcube x  = x * square x
  square x = x * x
```

Fkt. in Haskell: Notat. Varianten (3)

bzw...

(4b) *let*-Konstrukt

```
kVA r =
  let
  myPi     = 3.14
  rcube x  = x * square x
  square x = x * x
  in
  ((4/3) * myPi * rcube r, 4 * myPi * square r)
```

Fkt.in Haskell: Notat. Varianten (4)

In einer Zeile...

(5a) ...mittels “;”

```
kVA r =
  ((4/3) * myPi * rcube r, 4 * myPi * square r)
  where
  myPi = 3.14; rcube x  = x * square x; square x = x * x
```

(5b) ...mittels “;”

```
kVA r =
  let myPi = 3.14; rcube x = x * square x; square x = x * x
  in
  ((4/3) * myPi * rcube r, 4 * myPi * square r)
```

Fkt. in Haskell: Notat. Varianten (5)

Spezialfall: *binäre* (*zweistellige*) Funktionen...

```
imax :: Integer -> Integer -> Integer
imax p q
  | p >= q     = p
  | otherwise  = q

tripleMax :: Integer -> Integer -> Integer -> Integer
tripleMax p q r
  | (imax p q == p) && (p `imax` r == p)   = p
  | ...
  | otherwise                              = r
```

...`imax` in `tripleMax` als *Präfix*- und als *Infixoperator* verwandt

Fkt. in Haskell: Notat. Varianten (6)

Musterbasiert...

```
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib(n-2) + fib(n-1)

capVowels :: Char -> Char
capVowels 'a' = 'A'
capVowels 'e' = 'E'
capVowels 'i' = 'I'
capVowels 'o' = 'O'
capVowels 'u' = 'U'
capVowels  c  =  c
```

Fkt. in Haskell: Notat. Varianten (7)

Mittels *case*-Ausdrucks...

```
capVowels :: Char -> Char
capVowels letter
  = case letter of
     'a'     -> 'A'
     'e'     -> 'E'
     'i'     -> 'I'
     'o'     -> 'O'
     'u'     -> 'U'
     letter  -> letter
```

```
deCapVowels :: Char -> Char
deCapVowels letter
  = case letter of
     'A'         -> 'a'
     'E'         -> 'e'
     'I'         -> 'i'
     'O'         -> 'o'
     'U'         -> 'u'
     otherwise   -> letter
```

Fkt. in Haskell: Notat. Varianten (8)

Mittels *Muster* und *“wild cards”*...

```
add :: Integer -> Integer -> Integer
add n 0 = n
add 0 n = n
add m n = m+n

mult :: Integer -> Integer -> Integer
mult _ 0 = 0
mult 0 _ = 0
mult m n = m*n
```

Muster können (u.a.) sein...

- *Werte* (z.B. 0, 'c', True)
  ...ein Argument "passt" auf das Muster, wenn es vom entsprechenden Wert ist.

- *Variablen* (z.B. n)
  ...jedes Argument passt.

- *Wild card* "_"
  ...jedes Argument passt (sinnvoll für nicht zum Ergebnis beitragende Argumente)

- ...

$\rightsquigarrow$ mehr über Muster und musterbasierte Funktionsdefinitionen später...

Literaturhinweis

...auf den Haskell-Sprachreport:

- *Haskell 98: Language and Libraries. The Revised Report*. Simon Peyton Jones (Hrsg.), Cambridge University Press, 2003.

Fortf. von Kap. 1, Teil 2, Grundlagen

Beispiel

Ein einfacher Editor kann in Haskell wie folgt realisiert werden:

```
type Editor = [Char]
```

Schreiben Sie eine Haskell-Rechenvorschrift `ersetze` mit der Signatur

```
ersetze :: Editor -> Int -> String -> String -> Editor
```

die angesetzt auf einen Editor `e`, eine ganze Zahl `i`, eine Zeichenreihe `s` und eine Zeichenreihe `t` das `i`-te Vorkommen von `s` in `e` durch `t` ersetzt...

Beispiel (fortgef.)

Naheliegende Fragen:

- Warum so wenige Klammern?
- Warum so viele Pfeile (`->`) und warum so wenige Kreuze ($\times$)?
- Warum nicht folgende Signaturzeile?

  “`ersetze :: (Editor x Int x String x String) -> Editor`”

  Beachte: Haskell-korrekt wäre (d.h. “,” statt `x`)

  ```
  ersetze :: (Editor,Int,String,String) -> Editor
  ```

...die uns zu folgenden Thema führen

Mehr zu allg. Grundlagen und zu Haskell, insbesondere zu...

- Funktionen
  - ...und darüber wie man sie definieren/notieren kann
    - $\leadsto$ Notationelle Alternativen (bereits besprochen)
    - $\leadsto$ **Funktionssignaturen, Funktionsausdrücke, Klammereinsparungsregeln**
    - $\leadsto$ Abseitsregel und Layout-Konventionen

  und in der Folge weiterführend zu

  - Klassifikation von Rekursionstypen
  - Anmerkungen zu Effektivität und Effizienz
  - Komplexitätsklassen

Klammereinsparungsregeln in Funktionssignaturen

Konvention (von essentieller Bedeutung):

- Der *Typkonstruktor* $\rightarrow$ ist *rechtsassoziativ*!

Das bedeutet:

- Die Funktionssignatur

  ```
  f :: Int -> Float -> Int -> String -> Char
  ```

  steht abkürzend für

  ```
  f :: (Int -> (Float -> (Int -> (String -> Char))))
  ```

- Wann immer eine abweichende Klammerung intendiert ist, *muss* explizit geklammert werden!

  (*vgl. Klammereinsparungsregeln bei arithmetischen Ausdrücken*)

Funktionen und ihre Signaturen (1)

Zur Veranschaulichung, noch konkreter:

Die Signaturen der Funktionen `f`

```
f :: Int -> (Int -> Int)
```

(aufgrund der Klammereinsparungsregeln gleichbedeutend mit der ungeklammerten Kurzform `f :: Int -> Int -> Int`) und `g`

```
g :: (Int -> Int) -> Int
```

sind *grundsätzlich verschieden* und *unbedingt auseinanderzuhalten*!

Funktionen und ihre Signaturen (2)

Warum?

- `f` ist eine Funktion, die ganze Zahlen auf Abbildungen ganzer Zahlen in sich abbildet.

- `g` ist eine Funktion, die Abbildungen ganzer Zahlen in sich auf ganze Zahlen abbildet.

Zur Übung:

- Überlegen Sie sich, ob die Funktion $+$ (Addition auf ganzen Zahlen) dem Signaturschema von `f` oder dem von `g` folgt.

Funktionen und ihre Signaturen (3)

Ein weiteres Beispiel, noch konkreter und noch ein wenig komplexer...

Mit folgenden Deklarationen für `f` und `g`

```
f :: Int -> (Int -> Int -> Int)
f 1 = (+)
f 2 = (-)
f 3 = (*)
f _ = div

g :: (Int -> Int -> Int) -> Int
g h = h 6 3
```

...liefern die nachstehenden Aufrufe von `f` und `g` die angegebenen Resultate:

```
Main> f 1 2 3                              kurz fuer: (((f 1) 2) 3)
5

Main> f 3 2 3
6

Main> g (+)
9

Main> g (*)
18
```

Funktionen und ihre Signaturen (4)

Offenbar gilt: ...in `g` sind die Argumente 6 und 3 fest vorgegeben. Betrachte deshalb jetzt die folgende "Erweiterung" `k` von `g`, die das vermeidet:

```
k :: (Int -> Int -> Int) -> Int -> Int -> Int
k h x y = h x y
```

Beachte: ...aufgrund der Klammereinsparungsregeln für Funktionsterme (linksassoziativ) und $\rightarrow$ (rechtsassoziativ) steht obige Deklaration von `k` abkürzend für:

```
k :: ((Int -> (Int -> Int)) -> (Int -> (Int -> Int)))
(((k h) x) y) = ((h x) y)
```

Für `k` sind jetzt folgende Aufrufe mit variablen Argumenten möglich:

```
Main> k (*) 3 5
15

Main> k (+) 17 4
21

Main> k div 42 5
8
```

Funktionen und ihre Signaturen (5)

Zur Übung:

Vergleichen Sie die Deklaration der Funktion `f`

```
f :: Int -> (Int -> Int -> Int)
f 1 = (+)
f 2 = (-)
f 3 = (*)
f _ = div
```

...mit der Deklaration ihrer scheinbar naheliegenden “dualen Variante” `g`:

```
g :: (Int -> Int -> Int) -> Int
g (+) = 1
g (-) = 2
g (*) = 3
g div = 42
g _   = 99
```

- Was beobachten Sie, wenn Sie die Funktionen `f` und `g` aufrufen?
- Haben Sie (schon) eine Erklärung dafür?

Funktionen und ihre Signaturen (6)

Bleiben Sie auch an folgender Frage dran...

- Warum möglicherweise sind die Klammereinsparungsregeln für $\rightarrow$

  ```
  f :: Int -> Int -> Int -> Int
  ```

  zugunsten der *Rechtsassoziativität* von $\rightarrow$

  ```
  f :: (Int -> (Int -> (Int -> Int)))
  ```

  und nicht der *Linksassoziativität* gefallen?

  ```
  f :: (((Int -> Int) -> Int) -> Int)
  ```

Funktionen und ihre Signaturen (7)

In jedem Falle gilt:

Die Einsicht in den Unterschied

- von

  ```
  f :: Int -> Int -> Int -> Int
  ```

  ...aufgrund der *Rechtsassoziativität* von $\rightarrow$ abkürzend und gleichbedeutend mit der vollständig, aber nicht überflüssig geklammerten Version

  ```
  f :: (Int -> (Int -> (Int -> Int)))
  ```

- und von

  ```
  f :: (((Int -> Int) -> Int) -> Int)
  ```

ist *essentiell* und von absolut *zentraler* Bedeutung!

Funktionen und ihre Signaturen (8)

Bewusst pointiert...

Ohne diese Einsicht ist erfolgreiche Programmierung (speziell) im funktionalen Paradigma

- nicht möglich
- oder allenfalls Zufall!

Bestandsaufnahme (1)

- Bis jetzt:
  ...Konzentration auf Funktionsdeklarationen und ihre *Signaturen* bzw. *Typen*

- Ab jetzt:
  ...Konzentration auf *Funktionsterme* und ihre *Signaturen* bzw. *Typen*

Bestandsaufnahme (2)

Tatsache:

Wir sind gewohnt, mit Ausdrücken der Art

```
add 2 3
```

umzugehen. (Auch wenn wir gewöhnlich `2+3` statt `add 2 3` schreiben.)

Frage:

- Warum könnte es sinnvoll sein, auch mit (*scheinbar unvollständigen*) Ausdrücken wie

  ```
  add 2
  ```

  umzugehen?

- Entscheidend für die Antwort: Können wir einem Ausdruck wie `add 2` sinnvoll eine Bedeutung geben und wenn ja, welche?

Funktionsterme und ihre Typen (1)

Betrachten wir die Funktion `add` zur Addition ganzer Zahlen noch einmal im Detail:

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

abkuerzend fuer: ((add m) n) = m+n
```

Dann sind die Ausdrücke `add`, `add 2` und `add 2 3` von den Typen:

```
add :: Int -> Int -> Int
add 2 :: Int -> Int
add 2 3 :: Int
```

Funktionsterme und ihre Typen (2)

Erinnerung:

```
add :: Int -> Int -> Int
```

entspricht wg. vereinbarter Rechtsassoziativität von $\rightarrow$

```
add :: Int -> (Int -> Int)
```

Somit *verbal* umschrieben:

- `add :: Int -> Int -> Int`
  ...bezeichnet eine Funktion, die ganze Zahlen auf Funktionen von ganzen Zahlen in ganze Zahlen abbildet (*Rechtsassoziativität von* $\rightarrow$!).

- `add 2 :: Int -> Int`
  ...bezeichnet eine Funktion, die ganze Zahlen auf ganze Zahlen abbildet.

- `add 2 3 :: Int`
  ...bezeichnet eine ganze Zahl (nämlich 5).

Funktionsterme und ihre Typen (3)

Damit haben wir eine Antwort auf unsere Ausgangsfrage...

- Warum könnte es sinnvoll sein, auch mit (*scheinbar unvollständigen*) Ausdrücken wie

  ```
  add 2
  ```

  umzugehen?

- Entscheidend für die Antwort: Können wir einem Ausdruck wie `add 2` sinnvoll eine Bedeutung geben und wenn ja, welche?

Nämlich:

Es ist sinnvoll, mit Ausdrücken der Art `add 2` umzugehen, weil

- wir ihnen sinnvoll eine Bedeutung zuordnen können!

- im Falle von `add 2`:
  ...`add 2` bezeichnet eine Funktion auf ganzen Zahlen, die angewendet auf ein Argument dieses Argument um 2 erhöht als Resultat liefert.

Funktionsterme und ihre Typen (4)

Betrachte auch folgendes Beispiel von vorhin unter dem neuen Blickwinkel auf Funktionsterme und ihre Typen:

```
k :: (Int -> Int -> Int) -> Int -> Int -> Int
k h x y = h x y
```

Dann gilt:

```
k :: (Int -> Int -> Int) -> Int -> Int -> Int
k add :: Int -> Int -> Int
k add 2 :: Int -> Int
k add 2 3 :: Int
```

Zur Übung:

- Ausprobieren! In Hugs lässt sich mittels des Kommandos `:t <Ausdruck>` der Typ eines Ausdrucks bestimmen!

  Bsp.: `:t k add 2` liefert `k add 2 :: Int -> Int`

Funktionsterme und ihre Typen (5)

Beachte:

Der Ausdruck (Funktionsterm)

```
k add 2 3
```

steht kurz für

```
(((k add) 2) 3)
```

Analog stehen die Ausdrücke (Funktionsterme)

```
k add
k add 2
```

kurz für

```
(k add)
((k add) 2)
```

Funktionsterme und ihre Typen (6)

Beobachtung (anhand des vorigen Beispiels):

- Funktionen in Haskell sind grundsätzlich *einstellig*!
- Wie die Funktion `k` zeigt, kann dieses Argument komplex sein, bei `k` z.B. eine Funktion, die ganze Zahlen auf Funktionen ganzer Zahlen in sich abbildet.

Beachte:

Die Sprechweise, Argument der Funktion `k` sei eine zweistellige Funktion auf ganzen Zahlen, ist *lax* und *unpräzise*, gleichwohl (aus Gründen der Einfachheit und Bequemlichkeit) üblich.

Funktionsterme und ihre Typen (7)

Konsequenz aus voriger Beobachtung:

- Wann immer man nicht durch Klammerung etwas anderes erzwingt, ist (aufgrund der vereinbarten Rechtsassoziativität des Typoperators $\rightarrow$) das "eine" Argument der in Haskell grundsätzlich einstelligen Funktionen von demjenigen Typ, der links vor dem ersten Vorkommen des Typoperators $\rightarrow$ in der Funktionssignatur steht.

- Wann immer dies nicht erwünscht ist, muss dies durch explizite Klammerung in der Funktionssignatur ausgedrückt werden.

Funktionsterme und ihre Typen (8)

Beispiele:

- *Keine Klammerung* ($\leadsto$ Konvention greift!)

  ```
  f :: Int -> Tree -> Graph -> ...
  ```

  f ist einstellige Funktion auf ganzen Zahlen, nämlich `Int`, die diese abbildet auf...

- *Explizite Klammerung* ($\leadsto$ Konvention aufgehoben, wo gewünscht!)

  ```
  f :: (Int -> Tree) -> Graph -> ...
  ```

  f ist einstellige Funktion auf Abbildungen von ganzen Zahlen auf Bäume, nämlich `Int -> Tree`, die diese abbildet auf...

Hinweis: Wie wir Bäume und Graphen in Haskell definieren können, lernen wir bald.

Funktionsterme und ihre Typen (9)

Auch noch zu...

- ...
- Wann immer dies nicht erwünscht ist, muss dies durch explizite Klammerung in der Funktionssignatur erzwungen werden.

Beispiele:

- *Keine Klammerung*

  ```
  f :: Int -> Tree -> Graph -> ...
  ```

  f ist einstellige Funktion auf ganzen Zahlen, nämlich `Int`, die diese abbildet auf...

- *Explizite Klammerung*

  ```
  f :: (Int,Tree) -> Graph -> ...
  ```

  f ist einstellige Funktion auf Paaren aus ganzen Zahlen und Bäumen, nämlich `(Int,Tree)`, die diese abbildet auf...

Funktionsterme und ihre Typen (10)

Noch einmal zurück zum Beispiel der Funktion `k`:

```
k :: (Int -> Int -> Int) -> Int -> Int -> Int
```

...`k` ist eine einstellige Funktion, die eine zweistellige Funktion auf ganzen Zahlen als Argument erwartet (*lax!*) und auf eine Funktion abbildet, die ganze Zahlen auf Funktionen ganzer Zahlen in sich abbildet.

Zur Deutlichkeit die Signatur von `k` auch noch einmal vollständig, aber nicht überflüssig geklammert:

```
k :: ((Int -> (Int -> Int)) -> (Int -> (Int -> Int)))
```

Funktionsterme und ihre Typen (11)

Das Beispiel von `k` fortgesetzt:

```
k add :: Int -> Int -> Int
```

...`k add` ist eine einstellige Funktion, die ganze Zahlen auf Funktionen ganzer Zahlen in sich abbildet.

Zur Deutlichkeit auch hier noch einmal vollständig, aber nicht überflüssig geklammert:

```
(k add) :: (Int -> (Int -> Int))
```

Funktionsterme und ihre Typen (12)

Das Beispiel von `k` weiter fortgesetzt:

```
k add 2 :: Int -> Int
```

...`k add 2` ist eine einstellige Funktion, die ganze Zahlen in sich abbildet.

Zur Deutlichkeit auch hier wieder vollständig, aber nicht überflüssig geklammert:

```
((k add) 2) :: (Int -> Int)
```

Funktionsterme und ihre Typen (13)

Das Beispiel von `k` abschließend fortgesetzt:

```
k add 2 3 :: Int
```

`k add 2 3` bezeichnet ganze Zahl; in diesem Falle `5`.

Zur Deutlichkeit auch dieser Funktionsterm vollständig, aber nicht überflüssig geklammert:

```
(((k add) 2) 3) :: Int
```

Wichtige Vereinbarungen in Haskell

Wenn in Haskell durch Klammerung nichts anderes ausgedrückt wird, gilt für

- Funktionssignaturen *Rechtsassoziativität*, d.h.

  ```
  k :: (Int -> Int -> Int) -> Int -> Int -> Int
  ```

  steht für

  ```
  k :: ((Int -> (Int -> Int)) -> (Int -> (Int -> Int)))
  ```

- Funktionsterme *Linksassoziativität*, d.h.

  ```
  k add 2 3 :: Int
  ```

  steht für

  ```
  (((k add) 2) 3) :: Int
  ```

als vereinbart!

Zum Abschluss des Signaturthemas (1)

Frage:

- Warum mag uns ein Ausdruck wie

  ```
  add 2
  ```

  “unvollständig” erscheinen?

Zum Abschluss des Signaturthemas (2)

...weil wir im Zusammenhang mit der Addition tatsächlich weniger an Ausdrücke der Form

```
add 2 3
```

als vielmehr an Ausdrücke der Form

```
add' (2,3)
```

gewohnt sind!

Erinnern Sie sich?

$$+ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$$

Zum Abschluss des Signaturthemas (3)

Der Unterschied liegt in den Signaturen der Funktionen `add` und `add'`:

```
add  :: Int -> (Int -> Int)

add' :: (Int,Int) -> Int
```

Mit diesen Signaturen von `add` und `add'` sind einige Beispiele...

- *korrekter* Aufrufe:

```
add 2 3           => 5 :: Int
add' (2,3)        => 5 :: Int
add 2                  :: Int -> Int
```

- *inkorrekter* Aufrufe:

```
add (2,3)
add' 2 3     -- beachte: add' 2 3 steht kurz fuer (add' 2) 3
add' 2
```

Zum Abschluss des Signaturthemas (4)

Mithin...

- ...die Funktionen `+` und `add'` sind echte *zweistellige* Funktionen

wohingegen...

- ...die Funktion `add` einstellig ist und nur aufgrund der Klammereinsparungsregeln scheinbar ebenfalls "zweistellige" Aufrufe zulässt:

  ```
  add 17 4
  ```

  Aber: `add 17 4` steht kurz für `(add 17) 4`. Die geklammerte Variante macht deutlich: Ein Argument nach dem anderen und nur eines zur Zeit...

Fazit zum Signaturthema (1)

Wir müssen nicht nur sorgfältig

- zwischen

  ```
  f :: Int -> Int -> Int
  ```

  ...aufgrund der *Rechtsassoziativität* von $\rightarrow$ abkürzend und gleichbedeutend ist mit

  ```
  f :: Int -> (Int -> Int)
  ```

- und

  ```
  f :: (Int -> Int) -> Int
  ```

unterscheiden, sondern ebenso sorgfältig auch

- zwischen

  ```
  f :: (Int,Int) -> Int
  ```

- und

  ```
  f :: Int -> (Int,Int)
  ```

und nicht zuletzt zwischen allen vier Varianten insgesamt!

Fazit zum Signaturthema (2)

Mithin, schreiben Sie

```
f :: Int -> Int -> Int
```

nur, wenn Sie auch wirklich

```
f :: Int -> (Int -> Int)
```

meinen und nicht etwa

```
f :: (Int -> Int) -> Int
```

oder

```
f :: (Int,Int) -> Int
```

oder

```
f :: Int -> (Int,Int)
```

Es macht einen Unterschied!

Und deshalb die Bitte:

- Gehen Sie die vorausgegangenen Beispiele noch einmal Punkt für Punkt durch und vergewissern Sie sich, dass Sie sie im Detail verstanden haben.

Das ist wichtig, weil...

- dieses Verständnis und der aus diesem Verständnis heraus mögliche kompetente und selbstverständliche Umgang mit komplexen Funktionssignaturen und Funktionstermen essentiell für alles weitere ist!

Ein kurzer Ausblick

Wir werden auf die Unterschiede und die Vor- und Nachteile von Deklarationen in der Art von

```
add :: Int -> (Int -> Int)
```

und

```
add' :: (Int,Int) -> Int
```

im Verlauf der Vorlesung unter den Schlagwörtern *Funktionen höherer Ordnung*, *Currifizierung*, *Funktionen als "first class citizens*" wieder zurückkommen.

Behalten Sie die Begriffe im Hinterkopf und blättern Sie zu gegebener Zeit in Ihren Unterlagen wieder hierher zurück.

Ergänzungen zu Funktionstermen (1)

Betrachten wir noch einmal die Funktion `add`:

```
add  :: Int -> (Int -> Int)
add m n = m+n
```

...und die Frage nach der "Existenz(berechtigung)" von

```
add 2 :: Int -> Int
```

...welches eine Funktion auf ganzen Zahlen ist, die ihr um `2` erhöhtes Argument als Resultat liefert.

Wir können diese Funktion `doubleInc` nennen...

Ergänzungen zu Funktionstermen (2)

...und in natürlicherweise wie folgt definieren:

```
doubleInc :: Int -> Int
doubleInc n = 2+n
```

Wir können die Definition von `doubleInc` aber auch auf die Funktion `(add 2)` abstützen:

```
doubleInc :: Int -> Int
doubleInc n = (add 2) n
```

...oder noch kürzer argumentlos (als Identität von Funktionen) einführen:

```
doubleInc :: Int -> Int
doubleInc = (add 2)
```

Beobachtung: `doubleInc` ist (nur noch) ein anderer Name für die Funktion `(add 2)`, die hier und in den obigen Bsp. nur der Deutlichkeit halber geklammert ist.

Ergänzungen zu Funktionstermen (3)

Vergleiche `doubleInc`, `add 2`

```
doubleInc :: Int -> Int
doubleInc = add 2
```

mit

```
\n -> add 2 n
```

Beobachtung: `doubleInc`, `add 2` und `\n -> add 2 n` sind...

- i.w. gleichwertige Formulierungen derselben Funktion
- i.w. dadurch unterschieden, dass `doubleInc` eine herkömmlich und im gewohnten Sinn benannte Funktion ist, wohingegen `(add 2)` und `(\n -> (add 2) n)` unbenannt, zumindest nicht im gewohnten Sinn mit einem Namen benannt sind; die Funktion `(\n -> (add 2) n)` speziell ist im Haskell-Jargon eine sog. *anonyme Funktion*!

“Erfahrenheits”-Faustregel

Die Implementierung einer Funktion wie `doubleInc`

- durch

```
doubleInc :: Int -> Int
doubleInc n = 2+n
```

  ...deutet darauf hin, dass vermutlich noch wenig Erfahrung mit funktionaler Programmierung vorliegt

- durch

```
doubleInc :: Int -> Int        doubleInc :: Int -> Int
doubleInc = (+) 2              doubleInc = (+2)  -- sog. operator section
```

  ...deutet darauf hin, dass vermutlich bereits mehr Erfahrung mit funktionaler Programmierung vorliegt

- durch

```
\n -> 2+n
```

  ...deutet gleichfalls darauf hin, dass schon mehr Erfahrung mit funktionaler Programmierung vorliegt, und darüberhinaus, dass in der konkreten Anwendungssituation ein Name, unter dem auf die Funktion mit der Bedeutung “doubleInc” zugegriffen werden könnte, keine Rolle spielt.

Als Ausblick... (1)

...ein kleines Beispiel schon jetzt:

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

Anwendung:

```
map (\n -> 2+n) [1,2,3]   =>   [3,4,5]
```

...oder genausogut

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

Machen Sie sich klar, dass die Typisierung von `add’` folgendes nicht zulässt:

```
map (add’ 2) [1,2,3]
```

$\leadsto$ später mehr dazu unter dem Stichwort “Funktionale”, speziell Funktionale auf Listen...

Als Ausblick... (2)

Als Beispiel aussagekräftiger und überzeugender:

```
map (\n -> 3*n+42) [1,2,3]     =>   [45,48,51]
```

Wird eine Funktion mit der Abbildungsvorschrift von (`\n ->3*n+42`) ansonsten nicht gebraucht, spart man sich durch Verwendung der anonymen Funktion wie oben die Deklaration einer ansonsten nur genau einmal benutzten Funktion wie `dreifachPlus42`:

```
dreifachPlus42 :: Int -> Int
dreifachPlus42 n = 3*n+42

map dreifachPlus42 [1,2,3]   =>   [45,48,51]
```

Ein weiterer Nachtrag: Operatoren in Haskell

Operatoren in Haskell sind...

- ...grundsätzlich *Präfixoperatoren*, insbesondere alle selbstdeklarierten Operatoren (*vulgo*: selbstdeklarierte Funktionen)

  Beispiele: `fac 5`, `imax 2 3`, `tripleMax 2 5 3`,...

- ...in einigen wenigen Fällen *Infixoperatoren*, dies gilt insbesondere für arithmetische Operatoren

  Beispiele: `2+3`, `3*5`, `7-4`, `5^3`,...

Binäre Operatoren in Haskell: Infix- vs. Präfix

Für binäre Operatoren in Haskell gilt...

- Binäre Operatoren `bop`, die standardmäßig als...
  - Präfixoperatoren verwendet werden, können in der Form `'bop'` als Infixoperator verwendet werden

    Beispiel: `2 'imax' 3` (statt standardmäßig `imax 2 3`)
  - Infixoperatoren verwendet werden, können in der Form `(bop)` als Präfixoperator verwendet werden

    Beispiel: `(+) 2 3` (statt standardmäßig `2+3`)

Abschließend zu Funktionstermen (1)

Betrachten wir noch einmal die Funktionen `add` und `add'`:

```
add  :: Int -> (Int -> Int)

add' :: (Int,Int) -> Int
```

Abschließend zu Funktionstermen (2)

...hier noch einmal zusammen mit ihren Implementierungen:

```
add  :: Int -> (Int -> Int)
add m n = m+n

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

Sprechweise: Die Funktion...

- `add` ist *curryfiziert*
- `add’` ist *uncurryfiziert*

Curryfiziert vs. uncurryfiziert (1)

Idee: ...ziehe die Art der Konsumation mehrerer Argumente zur Klassifizierung von Funktionen heran

Erfolgt die Konsumation mehrerer Argumente durch Funktionen...

- einzeln Argument für Argument: *curryfiziert*
- gebündelt als Tupel: *uncurryfiziert*

Beispiele:

```
Funktion add curryfiziert:     add 2 3     bzw. (add 2) 3
Funktion add' uncurryfiziert:  add' (2,3)
```

Curryfiziert vs. uncurryfiziert (2)

Zentral sind die beiden *Funktionale* (synonym: *Funktionen höherer Ordnung*) `curry` und `uncurry`...

```
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* ersetzt Funktionspfeil “$\rightarrow$” durch Produkt-/Tupelbildung “$\times$”.

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

Curryfiziert vs. uncurryfiziert (3)

Die Funktionale `curry` und `uncurry` bilden...

- uncurryfiziert vorliegende Funktionen auf ihr curryfiziertes Gegenstück ab, d.h.
  ...für uncurryfiziertes `f :: (a,b) -> c` ist
  `curry f :: a -> b -> c` curryfiziert.

- curryfiziert vorliegende Funktionen auf ihr uncurryfiziertes Gegenstück ab, d.h.
  ...für curryfiziertes `g :: a -> b -> c` ist
  `uncurry g :: (a,b) -> c` uncurryfiziert.

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

curry f :: a -> b -> c

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

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

Im Beispiel...

```
add  :: Int -> (Int -> Int)
add m n = m+n

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

Damit gilt:

```
curry add' :: Int -> Int -> Int

uncurry add :: (Int,Int) -> Int
```

...und somit sind die folgenden Aufrufe gültige Aufrufe:

```
curry add' 17 4
   ⇒ add' (17,4) ⇒ 17+4 ⇒ 21

uncurry add (17,4)
   ⇒ add 17 4 ⇒ 17+4 ⇒ 21
```

Curryfiziert oder uncurryfiziert?

...das ist hier die Frage.

Zum einen...

- Geschmackssache (sozusagen eine notationelle Spielerei)
  ...sicher, auch das, aber: die Verwendung curryfizierter Formen ist in der Praxis vorherrschend
  $\leadsto$ `f x`, `f x y`, `f x y z`,... möglicherweise eleganter als `f x`, `f (x,y)`, `f (x,y,z)`,...?

Zum anderen (und weit wichtiger!) folgendes...

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

Beispiel: `add 4711 :: Int -> Int`

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

Layout-Konventionen für Haskell-Programme

Für die meisten gängigen Programmiersprachen gilt:

- Das Layout eines Programms hat Einfluss
  - auf seine Leserlichkeit, Verständlichkeit, Wartbarkeit
  - aber nicht auf seine Bedeutung

Für Haskell gilt das nicht!

- Das Layout eines Programms trägt in Haskell Bedeutung!
- Reminiszenz an Cobol, Fortran. Layoutabhängigkeit aber auch zu finden in modernen Sprachen wie z.B. occam.
- Für Haskell ist für diesen Aspekt des Sprachentwurfs eine grundsätzlich andere Entwurfsentscheidung getroffen worden als z.B. für Java, Pascal, C, etc.

Abseitsregel (engl. offside rule) (1)

...layout-abhängige Syntax als notationelle Besonderheit in Haskell

“Abseits”-Regel...

- Erstes Zeichen einer Deklaration (bzw. nach `let`, `where`):
  ...Startspalte neuer “Box” wird festgelegt
- Neue Zeile...
  - gegenüber der aktuellen Box nach rechts eingerückt: *...aktuelle Zeile wird fortgesetzt*
  - genau am linken Rand der aktuellen Box: *...neue Deklaration wird eingeleitet*
  - weiter links als die aktuelle Box: *...aktuelle Box wird beendet* (*“Abseitssituation”*)

Ein Beispiel zur Abseitsregel (1)

Unsere Funktion `kVA` zur Berechnung von Volumen und Oberfläche einer Kugel mit Radius `r`:

```
kVA r =
  ((4/3) * myPi * rcube r, 4 * myPi * square r)
  where
  myPi      = 3.14
  rcube x   = x *
     square x

square x = x * x
```

...nicht schön, aber korrekt. Das Layout genügt der Abseitsregel von Haskell und damit den Layout-Konventionen.

Abseitsregel (2)

Graphische Veranschaulichung der Abseitsregel...

```
-------------------------------------------------
|
kVA r =
|  ((4/3) * myPi * rcube r, 4 * myPi * square r)
   ---------------------------------
|  |
|  where
|  myPi     = 3.14
|  rcube x  = x *
|  |   square x
|  --------------------->
------------------------------------>
-------------------------------------------------
square x = x * x
|
\/
```

Layout-Konventionen

...bewährt hat es sich, eine Layout-Konvention nach folgendem Muster einzuhalten:

```
funName f1 f2... fn
  | g1   = e1
  | g2   = e2
  ...
  | gk   = ek

funName f1 f2... fn
  | diesIsteinGanz
    BesondersLanger
    Waechter
       = diesIstEinEbenso
         BesondersLangerAusdruck
  | g2        = e2
  ...
  | otherwise = ek
```

Verantwortung des Programmierers (1)

...die Auswahl einer angemessenen Notation. Vergleiche...

```
triMax :: Integer -> Integer -> Integer -> Integer

a) triMax = \p q r ->
     if p>=q then (if p>=r then p
                           else r)
             else (if q>=r then q
                           else r)
b) triMax p q r =
     if (p>=q) && (p>=r) then p
                         else
     if (q>=p) && (q>=r) then q
                         else r

c) triMax p q r
     | (p>=q) && (p>=r)    = p
     | (q>=p) && (q>=r)    = q
     | (r>=p) && (r>=q)    = r
```

Auswahlkriterium: Welche Variante lässt sich am einfachsten verstehen?

Verantwortung des Programmierers (2)

Hilfreich ist auch eine Richtschnur von C.A.R. Hoare:

Programme können grundsätzlich auf zwei Arten geschrieben werden:

- So einfach, dass sie offensichtlich keinen Fehler enthalten
- So kompliziert, dass sie keinen offensichtlichen Fehler enthalten

Es liegt am Programmierer, welchen Weg er einschlägt.

Rekursion

..speziell in funktionalen Sprachen

- Das zentrale Ausdrucksmittel/Sprachmittel, Wiederholungen auszudrücken. *Beachte:* Wir haben keine Schleifen in funktionalen Sprachen.
- Erlaubt oft sehr elegante Lösungen, oft wesentlich einfacher als schleifenbasierte Lösungen. Typisches Beispiel: *Türme von Hanoi*.
- Insgesamt so wichtig, dass eine *Klassifizierung* von Rekursionstypen angezeigt ist.

Eine solche Klassifizierung wird uns in der Folge beschäftigen.

Zuvor aber zwei Beispiele: *Quicksort* und *Türme von Hanoi*

Quicksort

...ein Beispiel, für das Rekursion auf eine elegante Lösung führt:

```
quickSort :: [Int] -> [Int]

quickSort []     = []
quickSort (x:xs) =
    quickSort [ y | y<-xs, y<=x ] ++
                     [x] ++ quickSort [ y | y<-xs, y>x ]
```

Türme von Hanoi (1)

...ein anderes Beispiel, für das Rekursion auf eine elegante Lösung führt:

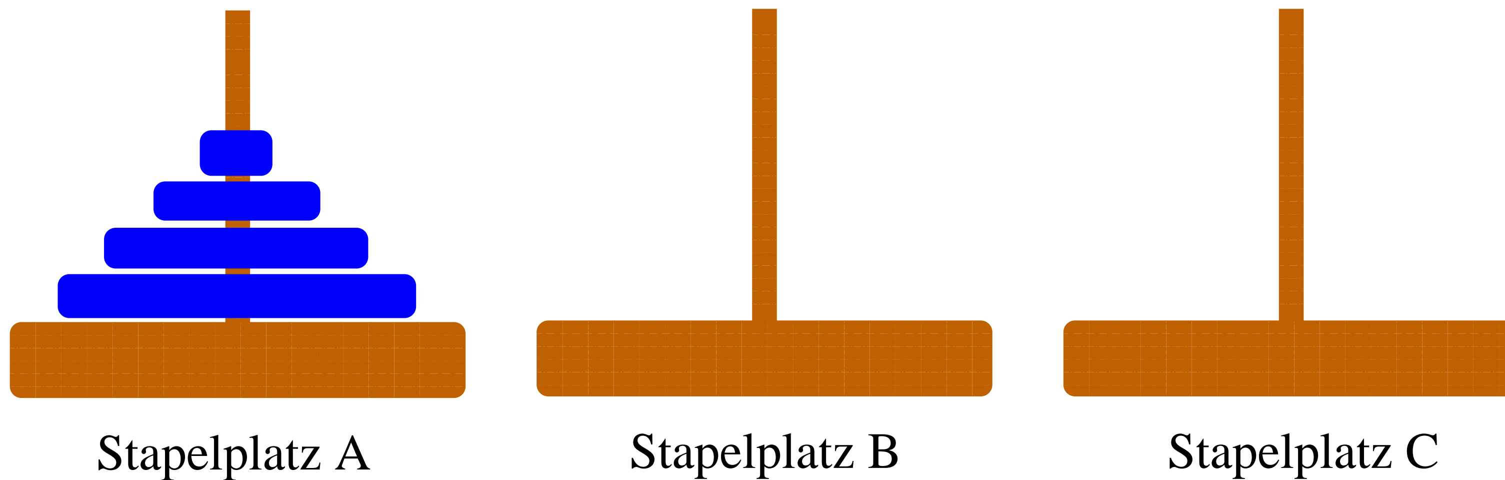
- *Ausgangssituation*:
  Gegeben sind drei Stapelplätze A, B und C. Auf Platz A liegt ein Stapel unterschiedlich großer Scheiben, die ihrer Größe nach sortiert aufgeschichtet sind, d.h. die Größe der Scheiben nimmt von unten nach oben sukzessive ab.

- *Aufgabe*: Verlege unter Zuhilfenahme von Platz B den Stapel von Scheiben von Platz A auf Platz C, wobei Scheiben stets nur einzeln verlegt werden dürfen und zu keiner Zeit eine größere Scheibe oberhalb einer kleineren Scheibe auf einem der drei Plätze liegen darf.

Lösung: Übungsaufgabe

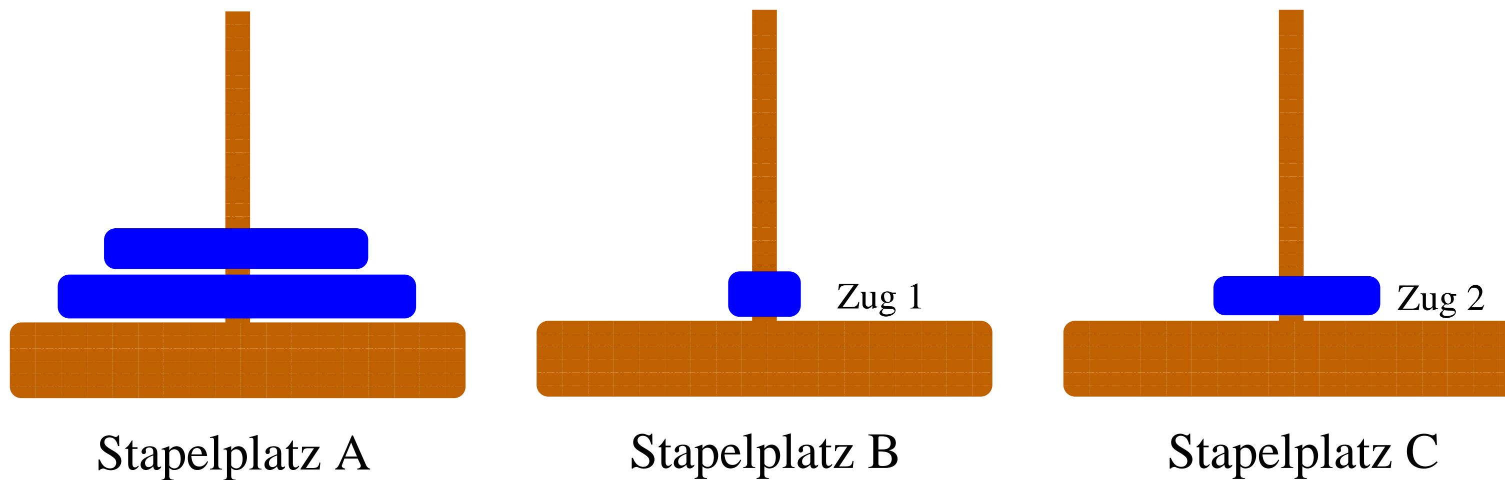
Türme von Hanoi (2)

Veranschaulichung:



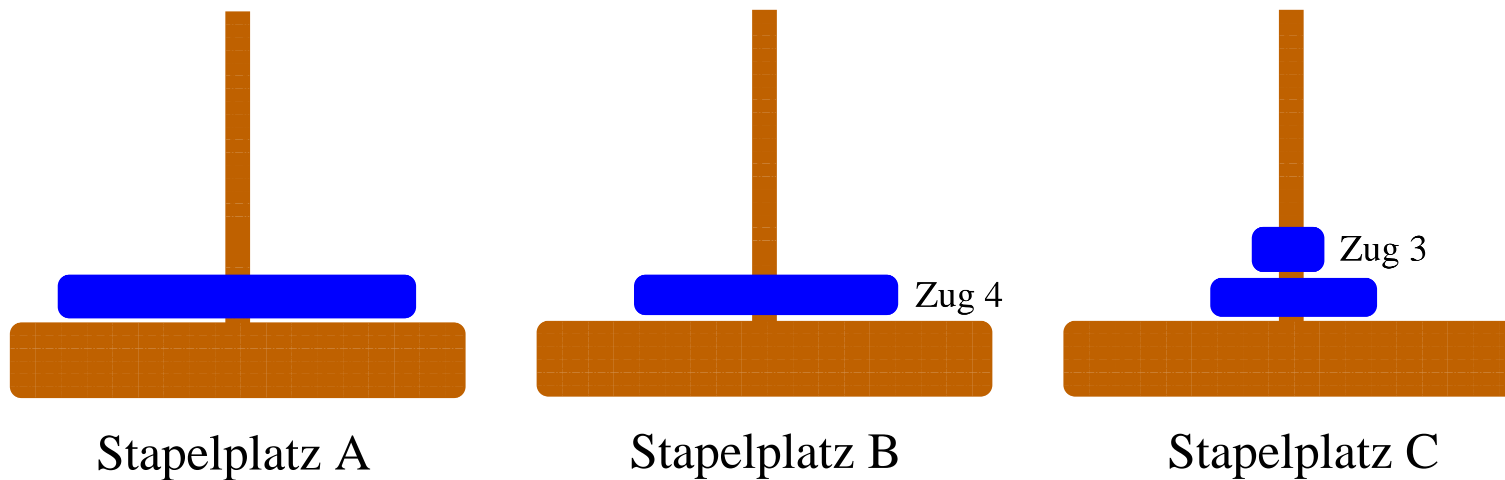
Türme von Hanoi (3)

Nach zwei Zügen:



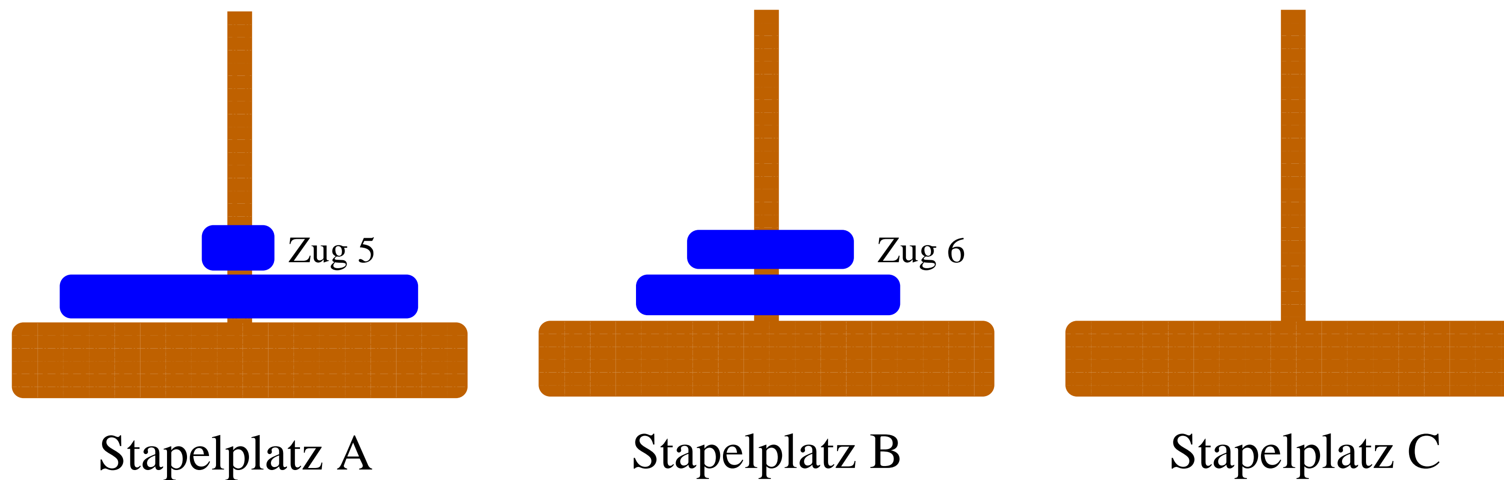
Türme von Hanoi (4)

Nach vier Zügen:



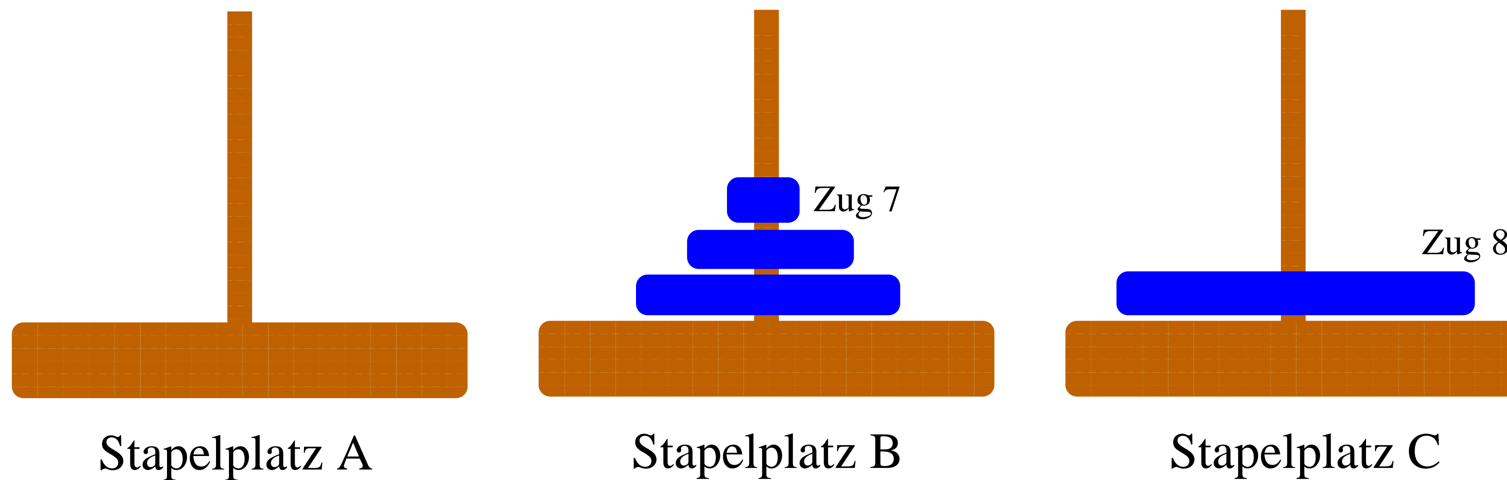
Türme von Hanoi (5)

Nach sechs Zügen:



Türme von Hanoi (6)

Nach acht Zügen:



Klassifikation der Rek.typen (1)

Generell...

...eine Rechenvorschrift heißt *rekursiv*, wenn sie in ihrem Rumpf (direkt oder indirekt) aufgerufen wird.

Dabei können wir unterscheiden...

- *Mikroskopische* Struktur
  ...betrachtet einzelne Rechenvorschriften und die syntaktische Gestalt der rekursiven Aufrufe

- *Makroskopische* Struktur
  ...betrachtet Systeme von Rechenvorschriften und ihre gegenseitigen Aufrufe

Rek.typen: Mikroskopische Struktur (2)

Üblich sind folgende Sprechweisen...

1. *Repetitive* (*schlichte*) *Rekursion*
   ...pro Zweig höchstens ein rekursiver Aufruf und zwar jeweils als äußerste Operation

   Bsp:

```
ggt :: Integer -> Integer -> Integer
ggt m n
  | n == 0          = m
  | m >= n          = ggt (m-n) n
  | m < n           = ggt (n-m) m
```

Rek.typen: Mikroskopische Struktur (3)

2. *Lineare Rekursion*

   ...pro Zweig höchstens ein rekursiver Aufruf, jedoch nicht notwendig als äußerste Operation

   Bsp:

```
powerThree :: Integer -> Integer
powerThree n
  | n == 0      = 1
  | n > 0       = 3 * powerThree (n-1)
```

Beachte: ...im Zweig $n > 0$ ist "$*$" die äußerste Operation, nicht `powerThree`!

Rek.typen: Mikroskopische Struktur (4)

3. *Geschachtelte Rekursion*

   ...rekursive Aufrufe enthalten rekursive Aufrufe als Argumente

   Bsp:

```
fun91 :: Integer -> Integer
fun91 n
  | n > 100    = n - 10
  | n <= 100   = fun91(fun91(n+11))
```

Preisfrage: Warum heißt die Funktion wohl `fun91`?

Rek.typen: Mikroskopische Struktur (5)

4. *Baumartige* (*kaskadenartige*) *Rekursion*

   ...pro Zweig können mehrere rekursive Aufrufe nebeneinander vorkommen

   Bsp:

```
binom :: (Integer,Integer) -> Integer
binom (n,k)
  | k==0 || n==k   = 1
  | otherwise      = binom (n-1,k-1) + binom (n-1,k)
```

Rek.typen: Makroskopische Struktur (6)

1. *Direkte Rekursion*
   ...entspricht Rekursion (Präzisierung!)

2. *Indirekte* oder auch *verschränkte* (*wechselweise*) *Rekursion*
   ...zwei oder mehr Funktionen rufen sich wechselweise auf

   Bsp:

```
isEven :: Integer -> Bool
isEven n
  | n == 0    = True
  | n > 0     = isOdd (n-1)

isOdd :: Integer -> Bool
isOdd n
  | n == 0    = False
  | n > 0     = isEven (n-1)
```

Anm. zu Effektivität & Effizienz (1)

Viele Probleme lassen sich...

- elegant rekursiv lösen (z.B. Türme von Hanoi)
- jedoch nicht immer effizient ($\neq$ effektiv!)

Als Faustregel gilt...

- Unter Effizienzgesichtspunkten ist...
  - repetitive Rekursion am (kosten-) günstigsten
  - geschachtelte und baumartige Rekursion am ungünstigsten

Anm. zu Effektivität & Effizienz (2)

(*Oft*) *folgende Abhilfe bei ineffizienten Implementierungen möglich*:

$\leadsto$ Umformulieren! Ersetzen ungünstiger durch günstigere Rekursionsmuster!

Etwa...

- Rückführung *linearer* Rekursion auf *repetitive* Rekursion

Anm. zu Effektivität & Effizienz (3)

...am Beispiel der Fakultätsfunktion:

Naheliegende Formulierung mit *linearem* Rekursionsmuster...

```
fac :: Integer -> Integer
fac n = if n == 0 then 1 else (n * fac(n-1))
```

Effizientere Formulierung mit *repetitivem* Rekursionsmuster...

```
fac :: Integer -> Integer
fac n = facRep (n,1)

facRep :: (Integer,Integer) -> Integer
facRep (p,r) = if p == 0 then r else facRep (p-1,p*r)
```

$\leadsto$ "*Trick*" ...Rechnen auf Parameterposition!

Aber: Überlagerungen mit anderen Effekten sind möglich, so dass sich der Effizienzgewinn nicht realisiert! (Zur Übung: Wie ist das im obigen Beispiel?)

Kaskaden- oder baumartige Rekursion

...oft anfällig für unnötige Mehrfachberechnungen.

...in der Folge illustriert am Beispiel der Berechnung der Folge der *Fibonacci-Zahlen*:

Die Folge $f_0, f_1, \ldots$ der *Fibonacci-Zahlen* ist definiert durch...

$$f_0 = 0, f_1 = 1 \quad \text{und} \quad f_n = f_{n-1} + f_{n-2} \quad \text{für alle } n \geq 2$$

Fibonacci-Zahlen (1)

Die naheliegende Implementierung...

```
fib :: Integer -> Integer
fib n
  | n == 0    = 0
  | n == 1    = 1
  | otherwise  = fib (n-1) + fib (n-2)
```

...führt auf kaskaden- bzw. baumartige Rekursion

$\leadsto$ ...und ist sehr, seeehr laaaangsaaaam (ausprobieren!)

Fibonacci-Zahlen (2)

Veranschaulichung ...durch manuelle Auswertung

```
fib 0   =>  0  --  1 Aufrufe von fib

fib 1   =>  1  --  1 Aufrufe von fib

fib 2   =>  fib 1 + fib 0
        =>  1 + 0
        =>  1  --  3 Aufrufe von fib

fib 3   =>  fib 2 + fib 1
        =>  (fib 1 + fib 0) + 1
        =>  (1 + 0) + 1
        =>  2  --  5 Aufrufe von fib
```

Fibonacci-Zahlen (3)

```
fib 4   =>  fib 3 + fib 2
        =>  (fib 2 + fib 1) + (fib 1 + fib 0)
        =>  ((fib 1 + fib 0) + 1) + (1 + 0)
        =>  ((1 + 0) + 1) + (1 + 0)
        =>  3  -- 9 Aufrufe von fib

fib 5   =>  fib 4 + fib 3
        =>  (fib 3 + fib 2) + (fib 2 + fib 1)
        =>  ((fib 2 + fib 1) + (fib 1 + fib 0))
                  + ((fib 1 + fib 0) + 1)
        =>  (((fib 1 + fib 0) + 1) + (1 + 0)) + ((1 + 0) + 1)
        =>  (((1 + 0) + 1) + (1 + 0)) + ((1 + 0) + 1)
        =>  5  --  15 Aufrufe von fib
```

Fibonacci-Zahlen (4)

```
fib 8  =>  fib 7 + fib 6
       =>  (fib 6 + fib 5) + (fib 5 + fib 4)
       =>  ((fib 5 + fib 4) + (fib 4 + fib 3))
           + ((fib 4 + fib 3) + (fib 3 + fib 2))
       =>  (((fib 4 + fib 3) + (fib 3 + fib 2))
             + (fib 3 + fib 2) + (fib 2 + fib 1)))
           + (((fib 3 + fib 2) + (fib 2 + fib 1))
             + ((fib 2 + fib 1) + (fib 1 + fib 0)))
       =>  ... --  60 Aufrufe von fib
```

Offensichtliche Probleme

- viele Mehrfachberechnungen
- exponentielles Wachstum!

Abhilfe

Programmiertechniken wie

- Dynamische Programmierung
- Memoization

Zentrale Idee:

- Speicherung und Wiederverwendung bereits berechneter (Teil-) Ergebnisse statt deren Wiederberechnung.

Komplexitätsklassen (1)

Nach P. Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*, 2. Auflage, 2003, Kapitel 11.

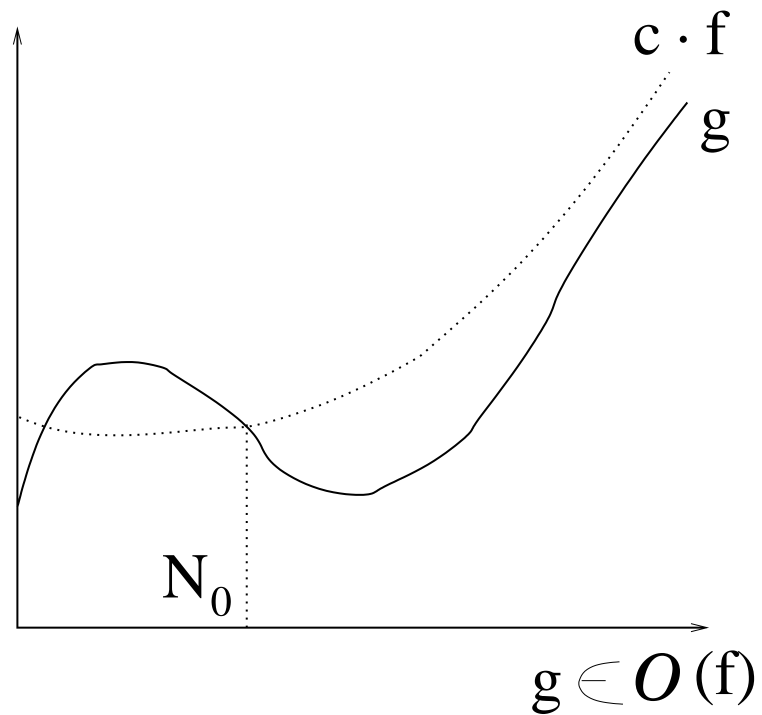
Erinnerung ...$\mathcal{O}$-Notation

- Sei f eine Funktion $f \; : \; \alpha \rightarrow I\!R^+$ von einem gegebenen Datentyp α in die Menge der positiven reellen Zahlen. Dann ist die Klasse $\mathcal{O}(f)$ die Menge aller Funktionen, die "langsamer wachsen" als f:

$$\mathcal{O}(f) =_{df} \{h \,|\, h(n) \leq c * f(n) \text{ für eine positive Konstante } c \text{ und alle } n \geq N_0\}$$

Komplexitätsklassen (2)

Veranschaulichung:



Komplexitätsklassen (3)

Beispiele häufig auftretender Kostenfunktionen...

Kürzel	Aufwand	Intuition: *vertausendfachte Eingabe heißt...*
$\mathcal{O}(c)$	konstant	... gleiche Arbeit
$\mathcal{O}(log\ n)$	logarithmisch	...nur zehnfache Arbeit
$\mathcal{O}(n)$	linear	...auch vertausendfachte Arbeit
$\mathcal{O}(n\ log\, n)$	"$n\ log\, n$"	...zehntausendfache Arbeit
$\mathcal{O}(n^2)$	quadratisch	...millionenfache Arbeit
$\mathcal{O}(n^3)$	kubisch	...milliardenfache Arbeit
$\mathcal{O}(n^c)$	polynomial	...gigantisch viel Arbeit (für großes c)
$\mathcal{O}(2^n)$	exponentiell	...hoffnungslos

Komplexitätsklassen (4)

...und was wachsende Eingaben in realen Zeiten in der Praxis bedeuten können:

n	linear	quadratisch	kubisch	exponentiell
1	1 μs	1 μs	1 μs	2 μs
10	10 μs	100 μs	1 ms	1 ms
20	20 μs	400 μs	8 ms	1 s
30	30 μs	900 μs	27 ms	18 min
40	40 μs	2 ms	64 ms	13 Tage
50	50 μs	3 ms	125 ms	36 Jahre
60	60 μs	4 ms	216 ms	36 560 Jahre
100	100 μs	10 ms	1 sec	$4 * 10^{16}$ Jahre
1000	1 ms	1 sec	17 min	sehr, sehr lange...

Fazit

Die vorigen Folien machen deutlich...

- ...Effizienz ist wichtig!
- ...Rekursionsmuster haben einen erheblichen Einfluss darauf (vgl. baumartig-rekursive Implementierung der Fibonacci-Funktion. Beachte aber: Nicht das baumartige Rekursionsmuster ist ein Problem an sich, sondern die unnötigen Mehrfachberechnungen von Werten im Falle der Fibonacci-Funktion!)

Allerdings...

- Baumartig rekursive Funktionsdefinitionen bieten sich zur *Parallelisierung* an!
  Stichwort: ...*divide and conquer*!

Zur Übung empfohlen...

- Wie könnte die Berechnung der Folge der Fibonacci-Zahlen

Struktur von Programmen

Programme funktionaler Programmiersprachen, speziell Haskell-Programme, sind zumeist

- Systeme (*wechselweiser*) *rekursiver* Rechenvorschriften, die sich *hierarchisch* oder/und *wechselweise* aufeinander abstützen.

Um sich über die *Struktur* solcher Systeme von Rechenvorschriften Klarheit zu verschaffen, ist neben der Untersuchung

- der *Rekursionstypen*

der beteiligten Rechenvorschriften insbesondere auch die Untersuchung

- ihrer *Aufrufgraphen*

geeignet.

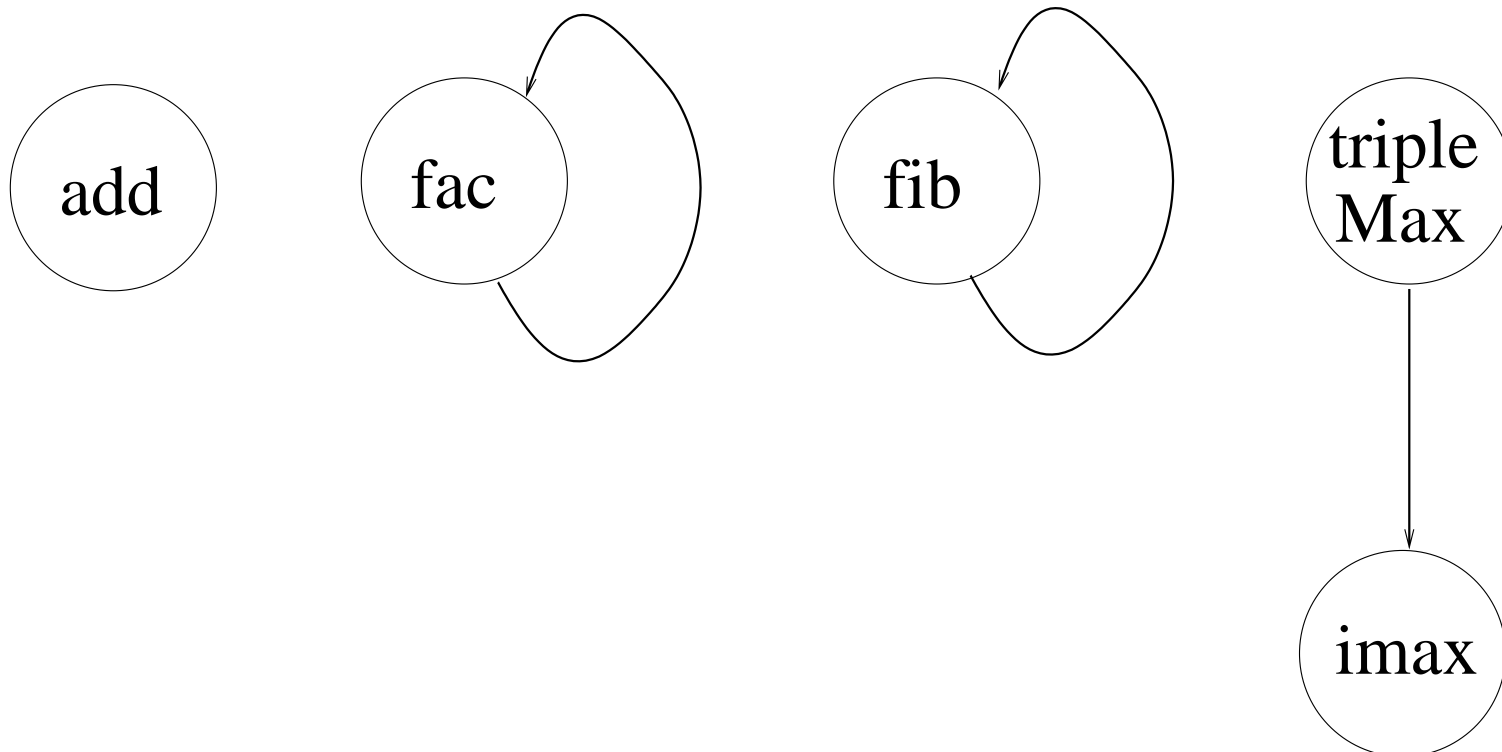
Aufrufgraphen

Der *Aufrufgraph* eines Systems S von Rechenvorschriften enthält

- einen *Knoten* für jede in S deklarierte Rechenvorschrift,
- eine gerichtete *Kante* vom Knoten f zum Knoten g genau dann, wenn im Rumpf der zu f gehörigen Rechenvorschrift die zu g gehörige Rechenvorschrift aufgerufen wird.

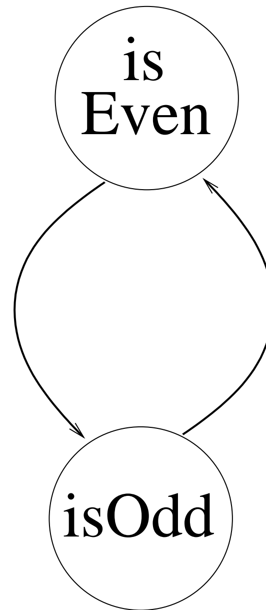
Beispiele von Aufrufgraphen (1)

...die Aufrufgraphen des Systems von Rechenvorschriften der Funktionen `add`, `fac`, `fib`, `imax` und `tripleMax`:



Beispiele von Aufrufgraphen (2)

...die Aufrufgraphen des Systems von Rechenvorschriften der Funktionen `isOdd` und `isEven`:



Beispiele von Aufrufgraphen (3)

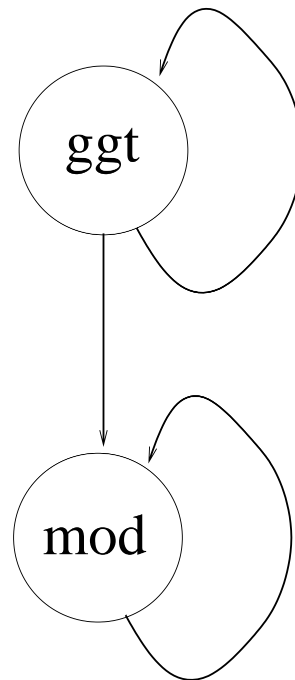
...das System von Rechenvorschriften der Funktionen `ggt` und `mod`:

```
ggt :: Int -> Int -> Int
ggt m n
    | n == 0 = m
    | n > 0  = ggt n (mod m n)

mod :: Int -> Int -> Int
mod m n
    | m < n  = m
    | m >= n = mod (m-n) n
```

Beispiele von Aufrufgraphen (3)

...und sein Aufrufgraph:



Auswertung von Aufrufgraphen

Aus dem Aufrufgraphen eines Systems von Rechenvorschriften ist u.a. ablesbar...

- *Direkte Rekursivität* einer Funktion: "Selbstkringel".
  ...z.B. bei den Aufrufgraphen der Funktionen `fac` und `fib`.

- *Wechselweise Rekursivität* zweier (oder mehrerer) Funktionen: Kreise (mit mehr als einer Kante)
  ...z.B. bei den Aufrufgraphen der Funktionen `isOdd` und `isEven`.

- *Direkte hierarchische Abstützung* einer Funktion auf eine andere: Es gibt eine Kante von Knoten f zu Knoten g, aber nicht umgekehrt.
  ...z.B. bei den Aufrufgraphen der Funktionen `tripleMax` und `imax`.

- *Indirekte hierarchische Abstützung* einer Funktion auf eine andere: Knoten g ist von Knoten f über eine Folge von Kanten erreichbar, aber nicht umgekehrt.

- *Wechselweise Abstützung*: Knoten g ist von Knoten f direkt oder indirekt über eine Folge von Kanten erreichbar und umgekehrt.

- *Unabhängigkeit/Isolation* einer Funktion: Knoten f hat (ggf. mit Ausnahme eines Selbstkringels) weder ein- noch ausgehende Kanten.
  ...z.B. bei den Aufrufgraphen der Funktionen `add`, `fac` und `fib`.

- ...

Kapitel 2: Datentypdeklarationen

Datenstrukturen in Haskell...

- Algebraische Datentypen (`data Tree = ...`)
- Typsynonyme (`type Student = ...`)
- Spezialitäten (`newtype State = ...`)

Datentypdeklarationen in Haskell

...selbstdefinierte (neue) Datentypen in Haskell!

$\leadsto$ Haskells Vehikel dafür: *Algebraische Typen*

Algebraische Typen erlauben uns zu definieren...

- Summentypen
  - *Spezialfälle*
    - Produkttypen
    - Aufzählungstypen

In der Praxis besonders wichtige Varianten...

- Rekursive Typen ($\leadsto$ "unendliche" Datenstrukturen)
- Polymorphe Typen ($\leadsto$ Wiederverwendung): *Später!*

Grundlegende Typmuster

Aufzählungs-, *Produkt-* und *Summentypen*:

- *Aufzählungstypen*
  $\rightsquigarrow$ Typen mit endlich vielen Werten

  ...typisches Beispiel: Typ `Jahreszeiten` mit Werten `Fruehling`, `Sommer`, `Herbst` und `Winter`.

- *Produkttypen* (synonym: *Verbundtypen)*
  $\rightsquigarrow$ Typen mit möglicherweise unendlich vielen Tupelwerten

  ...typisches Beispiel: Typ `Person` mit Werten `(Adam, maennlich, 27)`, `(Eva, weiblich, 25)`, etc.

- *Summentypen* (synonym: *Vereinigungstypen*)
  $\rightsquigarrow$ Vereinigung von Typen mit möglicherweise jeweils unendlich vielen Werten

  ...typisches Beispiel: Typ `Verkehrsmittel` als Vereinigung der (Werte der) Typen `Auto`, `Schiff`, `Flugzeug`, etc.

Zum Einstieg und Vergleich... (1)

Realisierung von Typdefinitionen in imperativen Sprachen

...hier am Bsp. von Pascal

- *Aufzählungstypen*

```
TYPE jahreszeiten = (fruehling, sommer, herbst, winter);
     spielkartenfarben = (kreuz, pik, herz, karo);
     werktage = (montag, dienstag, mittwoch,
                 donnerstag, freitag);
     transportmittel = (fahrrad, auto, schiff, flugzeug);
     form = (kreis, rechteck, quadrat, dreieck);
```

- *Produkttypen*

```
TYPE person = RECORD
                name: ARRAY [1..42] OF char;
                geschlecht: (maennlich, weiblich);
                alter: integer
              END;
```

Zum Einstieg und Vergleich... (2)

- *Summentypen*

```
TYPE verkehrsmittel =
       RECORD
         CASE vkm: transportmittel OF
           fahrrad: (tandem: Boolean);
           auto: (hersteller: ARRAY [1..30] OF char;
                  hubraum: real);
           schiff: (name: ARRAY [1..30] OF char;
                    tiefgang: real;
                    heimathafen: ARRAY [1..50] OF char);
           flugzeug: (reichweite: real;
                      sitzplaetze: integer)
       END;

     geometrischefigur =
       RECORD
         CASE fgr: form OF
           kreis: (radius: real);
           rechteck : (breite, hoehe: real);
           quadrat : (seitenlaenge, diagonale: real);
           dreieck: (s1, s2, s3: real; rechtwkg: boolean);
       END;
```

Zum Einstieg und Vergleich... (3)

Aufzählungstypen, Produkttypen, Summentypen...

- In Pascal ...drei verschiedene Sprachkonstrukte
- In Haskell ...ein *einheitliches* Sprachkonstrukt
  $\leadsto$ die *algebraische Datentypdefinition*

Zum Einstieg und Vergleich... (4)

Obige Einstiegsdatentypbeispiele in Haskell...

- Aufzählungstyp `Jahreszeiten`

```
data Jahreszeiten = Fruehling | Sommer | Herbst | Winter
data Werktage     = Montag | Dienstag | Mittwoch | Donnerstag | Freitag
data Bool         = True | False
```

- Produkttyp `Person`

```
data Person = Pers Name Geschlecht Alter
```

  mit

```
type Name  = String
type Alter = Int
data Geschlecht = Maennlich | Weiblich
```

Zum Einstieg und Vergleich... (5)

- Summentyp `Verkehrsmittel`

```
data Verkehrsmittel = Fahrrad Bool |
                      Auto String Float |
                      Schiff String Float String |
                      Flugzeug Float Int
```

In obiger Form offenbar wenig transparent im Vergleich zu:

```
TYPE verkehrsmittel =
     RECORD
       CASE vkm: transportmittel OF
         fahrrad: (tandem: Boolean);
         auto: (hersteller: ARRAY [1..30] OF char;
                hubraum: real);
         schiff: (name: ARRAY [1..30] OF char;
                  tiefgang: real;
                  heimathafen: ARRAY [1..50] OF char);
         flugzeug: (reichweite: real;
                    sitzplaetze: integer)
    END;
```

Zum Einstieg und Vergleich... (5)

- Summentyp `Verkehrsmittel`

```
data Verkehrsmittel = Fahrrad Tandem |
                      Auto Hersteller Hubraum |
                      Schiff Name Tiefgang Heimathafen |
                      Flugzeug Spannweite Sitzplaetze
```

  mit

```
type Tandem      = Bool
type Hersteller  = String
type Hubraum     = Float
type Name        = String
type Tiefgang    = Float
type Heimathafen = String
type Reichweite  = Float
type Sitzplaetze = Int
```

Man erkennt: Typsynonyme bringen *Transparenz* ins Programm!

Algebraische Datentypen in Haskell

...das allg. Muster der algebraischen Datentypdefinition:

```
data Typename
  = Con1 t11 ... t1k1 |
    Con2 t21 ... t2k2 |
        ...           |
    Conn tn1 ... tnkn
```

Sprechweisen:

- `Typename` ... *Typname/-identifikator*
- $\texttt{Con}_i$, $i = 1..n$ ...*Konstruktor*(*en*)/*-identifikatoren*
- k_i, $i = 1..n$ ...*Stelligkeit* des Konstruktors $\texttt{Con}_i$, $\mathsf{k}_i \geq 0$, $i = 1, \ldots, n$

Beachte: Typ- und Konstruktoridentifikatoren müssen mit einem Großbuchstaben beginnen (siehe z.B. `True`, `False`)!

Konstruktoren...

...können als Funktionsdefinitionen gelesen werden:

$\texttt{Con}_i$ :: $\texttt{t}_{i1}$ -> ... -> $\texttt{t}_{ik_i}$ -> Typename

Konstruktion von Werten eines algebraischen Datentyps durch...

...Anwendung eines Konstruktors auf Werte "passenden" Typs, d.h....

$\texttt{Con}_i$ $\texttt{v}_{i1}$ ... $\texttt{v}_{ik_i}$ :: Typname

wobei $\texttt{v}_{i1}$ :: $\texttt{t}_{i1}$ ... $\texttt{v}_{ik_i}$ :: $\texttt{t}_{ik_i}$, $j = 1, \ldots, k_i$

Beispiele:

- `Pers "Adam" Maennlich 27 :: Person`
- `Schiff "Donaukönigin" 2.74 "Wien" :: Verkehrsmittel`
- `Flugzeug 8540.75 275 :: Verkehrsmittel`

Aufzählungstypen (1)

Nullstellige Konstruktoren führen auf *Aufzählungstypen*...

Beispiele:

```
data Spielfarbe = Kreuz | Pik | Herz | Karo
data Wochenende = Sonnabend | Sonntag
data Geschlecht = Maennlich | Weiblich
data Form       = Kreis | Rechteck | Quadrat | Dreieck
```

Insbesondere ist der Typ der Wahrheitswerte...

```
data Bool = True | False
```

...Beispiel eines in Haskell vordefinierten Aufzählungstyps.

Aufzählungstypen (2)

Funktionsdefinitionen über Aufzählungstypen...
$\leadsto$ üblicherweise mit Hilfe von Pattern-matching.

Beispiele:

```
hatEcken :: Form -> Bool
hatEcken Kreis = False
hatEcken _     = True

istLandgebunden :: Verkehrsmittel -> Bool
istLandgebunden Fahrrad = True
istLandgebunden Auto    = True
istLandgebunden _       = False
```

Produkttypen

(Alternativenlose) mehrstellige Konstruktoren führen auf *Produkttypen*...

Beispiel:

```
type Name       = String
type Alter      = Int
data Geschlecht = Maennlich | Weiblich

data Person = Pers Name Geschlecht Alter
```

Beispiele: ...für Werte des Typs `Person`.

```
Pers "Paul Pfiffig" Maennlich 23  :: Person
Pers "Paula Plietsch" Weiblich 22 :: Person
```

Beachte: Funktionalität der Konstruktorfunktion ist hier...

```
Pers :: Name -> Geschlecht -> Alter -> Person
```

Summentypen (1)

Mehrere (null- oder mehrstellige) Konstruktoren führen auf Summentypen...

Beispiel:

```
type Radius       = Float
type Breite       = Float
type Hoehe        = Float
type Seite1       = Float
type Seite2       = Float
type Seite3       = Float
type Rechtwinklig = Bool

data XFigur = Kreis Radius |
              Rechteck Breite Hoehe |
              Quadrat Kantenlaenge |
              Dreieck Seite1 Seite2 Seite3 Rechtwinklig |
              Ebene
```

Die Varianten einer Summe werden durch "|" getrennt.

Summentypen (2)

Beispiele: ...für Werte des Typs erweiterte Figur `XFigur`

```
Kreis 3.14                   :: XFigur
Rechteck 17.0 4.0            :: XFigur
Quadrat 47.11                :: XFigur
Dreieck 3.0 4.0 5.0 True :: XFigur
Ebene                        :: XFigur
```

Zwischenfazit

Somit ergibt sich die eingangs genannte Taxonomie algebraischer Datentypen...

Haskell offeriert...

- Summentypen

mit den beiden *Spezialfällen*

- Produkttypen
  $\rightsquigarrow$ nur ein Konstruktor, mehrstellig

- Aufzählungstypen
  $\rightsquigarrow$ ein oder mehrere Konstruktoren, alle nullstellig

Rekursive Typen (1)

...der Schlüssel zu (potentiell) unendlichen Datenstrukturen.

Technisch:

...zu definierende Typnamen können rechtsseitig in der Definition benutzt werden.

Beispiel: ...(arithmetische) Ausdrücke

```
data Expr = Opd Int |
            Add Expr Expr |
            Sub Expr Expr |
            Squ Expr
```

Rekursive Typen (2)

Beispiele ...für Ausdrücke (lies --> als "entspricht").

```
Opd 42 :: Expr                                  -->  42
Add (Opd 17) (Opd 4) :: Expr                    -->  17+4
Add (Squ (Sub (Opd 42) (Squ (2)))) (Opd 12) :: Expr
                                    --> square(42-square(2))+12
```

...rekursive Typen ermöglichen potentiell unendliche Datenstrukturen!

Rekursive Typen (3)

Weiteres Beispiel:

Binärbäume, hier zwei verschiedene Varianten:

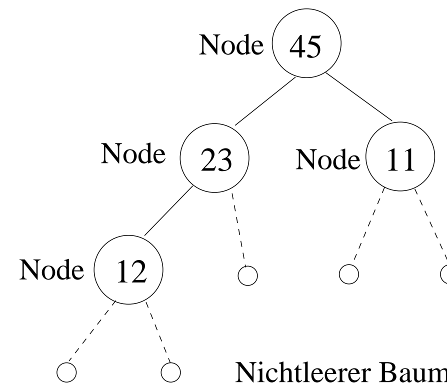
```
data BinTree1 = Nil | Node Int BinTree1 BinTree1

data BinTree2 = Leaf Int | Node Int BinTree2 BinTree2
```

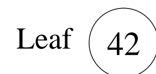
Rekursive Typen (4)

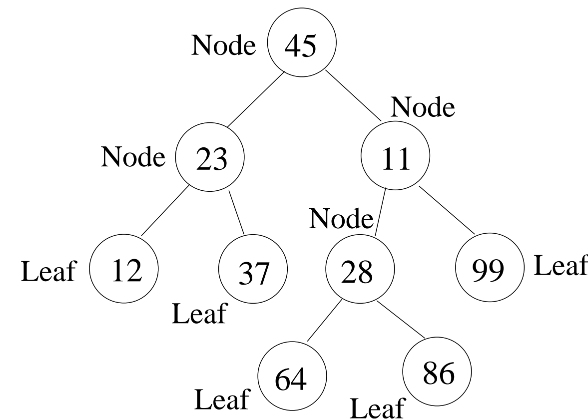
Veranschaulichung der Binärbaumvarianten 1&2 anhand eines Beispiels:

Variante 1



Variante 2





Rekursive Typen (5)

Beispiele ...für Funktionen über Binärbaumvariante 1.

```
valBinTree1 :: BinTree1 -> Int
valBinTree1 Nil              = 0
valBinTree1 (Node n bt1 bt2) = n + valBinTree1 bt1 + valBinTree1 bt2

depthBinTree1  :: BinTree1 -> Int
depthBinTree1 Nil = 0
depthBinTree1 (Node _ bt1 bt2)
                  = 1 + max (depthBinTree1 bt1) (depthBinTree1 bt2)
```

Mit diesen Definitionen...

```
valBinTree1 Nil                               ==  0
valBinTree1 (Node 17 Nil (Node 4 Nil Nil))    == 21
depthBinTree1 (Node 17 Nil (Node 4 Nil Nil))  ==  2
depthBinTree1 Nil                             ==  0
```

Rekursive Typen (6)

Beispiele ...für Funktionen über Binärbaumvariante 2.

```
valBinTree2 :: BinTree2 -> Int
valBinTree2 (Leaf n)         = n
valBinTree2 (Node n bt1 bt2) = n + valBinTree2 bt1 + valBinTree2 bt2

depthBinTree2  :: BinTree2 -> Int
depthBinTree2 (Leaf _) = 1
depthBinTree2 (Node _ bt1 bt2)
                       = 1 + max (depthBinTree2 bt1) (depthBinTree2 bt2)
```

Mit diesen Definitionen...

```
valBinTree2 (Leaf 3)                                        ==  3
valBinTree2 (Node 17 (Leaf 4) (Node 4 (Leaf 12) (Leaf 5)))   == 42
depthBinTree2 (Node 17 (Leaf 4) (Node 4 (Leaf 12) (Leaf 5))) ==  3
depthBinTree2 (Leaf 3)                                      ==  1
```

Wechselweise rekursive Typen

...ein Spezialfall rekursiver Typen.

Beispiel:

```
data Individual = Adult Name Address Biography |
                  Child Name

data Biography  = Parent String [Individual] |
                  NonParent String
```

Typsynonyme (1)

...hatten wir bereits kennengelernt bei der Einführung von Tupeltypen:

```
type Student = (String, String, Int)
type Buch = (String, String, Int, Bool)
```

...und auch in den Beispielen zu algebraischen Datentypen benutzt:

```
data Verkehrsmittel = Fahrrad Tandem |
                      Auto Hersteller Hubraum |
                      Schiff Name Tiefgang Heimathafen |
                      Flugzeug Spannweite Sitzplaetze

type Tandem         = Bool
type Hersteller     = String
type Hubraum        = Float
type Name           = String
type Tiefgang       = Float
type Heimathafen    = String
type Reichweite     = Float
type Sitzplaetze    = Int
```

Typsynonyme (2)

- Das Schlüsselwort `type` leitet die Deklaration von Typsynonymen ein
- Unbedingt zu beachten ist...
  - `type` ...führt neue Namen für bereits existierende Typen ein (Typsynonyme!), keine neuen Typen.

Somit gilt:

Durch `type`-Deklarationen eingeführte Typsyonyme...

- tragen zur Dokumentation bei und
- erleichtern (bei treffender Namenswahl) das Programmverständnis

aber...

- führen nicht zu (zusätzlicher) Typsicherheit!

Ein (pathologisches) Beispiel

```
type Euro        = Float
type Yen         = Float
type Temperature = Float

myPi    :: Float
daumen  :: Float
maxTemp :: Temperature
myPi    = 3.14
daumen  = 5.55
maxTemp = 43.2

currencyConverter :: Euro -> Yen
currencyConverter x = x + myPi * daumen
```

Mit obigen Deklarationen...

```
currencyConverter maxTemp => 60.627
```

...werden 43.2 °C in 60.627 Yen umgerechnet. Typsicher?

Ein reales Beispiel

Anflugsteuerung einer Sonde zum Mars...

```
type Geschwindigkeit = Float
type Meilen          = Float
type Km              = Float
type Zeit            = Float
type Wegstrecke      = Meilen
type Distanz         = Km

geschwindigkeit :: Wegstrecke -> Zeit -> Geschwindigkeit
geschwindigkeit w z = (/) w z

verbleibendeFlugzeit :: Distanz -> Geschwindigkeit -> Zeit
verbleibendeFlugzeit d g = (/) d g

verbleibendeFlugzeit 18524.34 1523.79
```

...durch Typisierungsprobleme dieser Art ging vor einigen Jahren eine Marssonde im Wert von mehreren 100 Mill. USD verloren.

Produkttypen vs. Tupeltypen (1)

Der Typ `Person` als...

- *Produkttyp*

```
data Person = Pers Name Geschlecht Alter
```

- *Tupeltyp*

```
type Person = (Name, Geschlecht, Alter)
```

Vordergründiger Unterschied:
...in der Tupeltypvariante fehlt der Konstruktor
(in diesem Bsp.: `Pers`)

Produkttypen vs. Tupeltypen (2)

...eine Abwägung von Vor- und Nachteilen.

Produkttypen und ihre typischen...

- *Vorteile* gegenüber *Tupeltypen*
  - Objekte des Typs sind mit dem Konstruktor “markiert” (trägt zur Dokumentation bei)
  - Tupel mit zufällig passenden Komponenten nicht irrtümlich als Elemente des Produkttyps manipulierbar (Typsicherheit! Vgl. früheres Beispiel zur Umrechnung Euro in Yen!)
  - Aussagekräftigere (Typ-) Fehlermeldungen (Typsynonyme können wg. Expansion in Fehlermeldungen fehlen).
- *Nachteile* gegenüber *Tupeltypen*
  - Produkttypelemente sind weniger kompakt, erfordern längere Definitionen (mehr Schreibarbeit)
  - Auf Tupeln vordefinierte polymorphe Funktionen (z.B. `fst`, `snd`, `zip`, `unzip`, `...`) stehen nicht zur Verfügung.
  - Der Code ist weniger effizient.

Andererseits...

Mit Produkttypen statt Typsynoymen...

```
data Euro        = EUR Float
data Yen         = YEN Float
data Temperature = Temp Float

myPi    :: Float
daumen  :: Float
maxTemp :: Temperature
myPi     = 3.14
daumen   = 5.55
maxTemp  = Temp 43.2
```

...wäre eine Funktionsdefinition im Stile von

```
currencyConverter :: Euro -> Yen
currencyConverter x = x + myPi * daumen
```

insbesondere auch ein Aufruf wie...

```
currencyConverter maxTemp
```

durch das Typsystem von Haskell verhindert!

Somit als kurzes Fazit... (1)

...unserer Überlegungen:

- *Typsynonyme* wie...

```
type Euro        = Float
type Yen         = Float
type Temperature = Float
```

  ...erben alle Operationen von `Float` und sind damit beliebig austauschbar – mit allen Annehmlichkeiten und Gefahren, sprich Fehlerquellen.

- *Produkttypen* wie...

```
data Euro        = EUR Float
data Yen         = YEN Float
data Temperature = Temp Float
```

  ...erben keinerlei Operationen von `Float`, bieten dafür aber um den Preis zusätzlicher Schreibarbeit und gewissen Performanzverlusts Typsicherheit!

Somit als kurzes Fazit... (2)

In ähnlicher Weise...

```
data Miles        = Mi Float
data Km           = Km Float
type Distance     = Miles
type Wegstrecke   = Km

...
```

...wäre auch der Verlust der Marssonde vermutlich vermeidbar gewesen.

Beachte:

- Typ- und Konstruktornamen dürfen in Haskell übereinstimmen (siehe z.B. `data Km = Km Float`)
- Konstruktornamen müssen global (d.h. modulweise) eindeutig sein.

Spezialitäten

...die `newtype`-Deklaration:

```
newtype Miles = Mi Float
```

`newtype`-Deklarationen sind im Hinblick auf...

- Typsicherheit
  ...mit `data`-Deklarationen vergleichbar
- Effizienz
  ...mit `type`-Deklarationen vergleichbar

Beachte: `newtype`-Deklarationen sind auf Typen mit nur einem Konstruktor eingeschränkt.

Polymorphe Typen

...in der Folge!

Rückblick: Das zentrale Thema zuletzt...

...selbstdefinierte (neue) Datentypen in Haskell!

$\leadsto$ Haskells Vehikel dafür: *Algebraische Typen*

Algebraische Typen erlauben uns zu definieren...

- Summentypen
  - Spezialfälle
    - Produkttypen
    - Aufzählungstypen

In der Praxis besonders wichtige Varianten...

- Rekursive Typen ($\leadsto$ "unendliche" Datenstrukturen)
- Polymorphe Typen ($\leadsto$ Wiederverwendung)

Offen geblieben ist bislang die Untersuchung und Diskussion *polymorpher Typen*!

Kapitel 3: Polymorphe Funktionen und Datentypen

Polymorphie

- Bedeutung lt. Duden:
  - *Vielgestaltigkeit*, *Verschiedengestaltigkeit*

    ...mit speziellen fachspezifischen Bedeutungsausprägungen

    * In der *Chemie*: das Vorkommen mancher Mineralien in verschiedener Form, mit verschiedenen Eigenschaften, aber gleicher chemischer Zusammensetzung
    * In der *Biologie*: Vielgestaltigkeit der Blätter oder der Blüte einer Pflanze
    * In der *Sprachwissenschaft*: das Vorhandensein mehrerer sprachlicher Formen für den gleichen Inhalt, die gleiche Funktion (z.B. die verschiedenartigen Pluralbildungen in: die Wiese*n*, die Feld*er*, die Tier*e*)
    * In der *Informatik*, speziell der *Theorie der Programmiersprachen*: $\leadsto$ das nächste Thema!

Polymorphie

...im programmiersprachlichen Kontext unterscheiden wir insbesondere zwischen

- Polymorphie
  - auf Datentypen
  - auf Funktionen
    - Parametrische Polymorphie ("Echte Polymorphie")
    - Ad hoc Polymorphie (synonym: Überladung)
      $\leadsto$ Haskell-spezifisch: Typklassen

Polymorphie

Wir beginnen mit...

- (Parametrischer) Polymorphie auf Funktionen

...die wir an einigen Beispielen schon kennengelernt haben:

- Die Funktionale `curry` und `uncurry`
- Die Funktionen `length`, `head` und `tail`

Rückblick (1)

Die Funktionale `curry` und `uncurry`...

```
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
```

Rückblick (2)

Die Funktionen `length`, `head` und `tail`...

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs

head :: [a] -> a
head (x:_) = x

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

Äußeres Kennzeichen parametrischer Polymorphie

Statt

- (ausschließlich) konkreter Typen (wie `Int`, `Bool`, `Char`,...)

treten in der (Typ-) Signatur der Funktionen

- (auch) *Typparameter*, sog. *Typvariablen*

auf.

Beispiele:

```
curry :: ((a,b) -> c) -> (a -> b -> c)

length :: [a] -> Int
```

Typvariablen in Haskell

Typvariablen in Haskell sind...

- freigewählte Identifikatoren, die mit einem Kleinbuchstaben beginnen müssen

  z.B.: `a`, `b`, aber auch `fp185161`

Im Unterschied dazu sind Typnamen, (Typ-) Konstruktoren in Haskell...

- freigewählte Identifikatoren, die mit einem Großbuchstaben beginnen müssen

  z.B.: `A`, `String`, `Node`, aber auch `Fp185161_WS0708`

Warum Polymorphie auf Funktionen?

...Wiederverwendung (durch Abstraktion)!

$\leadsto$ *ein typisches Vorgehen in der Informatik*!

Einfaches Bsp.: Funktionsabstraktion

Sind viele Ausdrücke der Art

```
(5 * 37 + 13) * (37 + 5 * 13)
(15 * 7 + 12) * (7 + 15 * 12)
(25 * 3 + 10) * (3 + 25 * 10)
...
```

zu berechnen, schreibe eine Funktion

```
f :: Int -> Int -> Int -> Int
f a b c = (a * b + c) * (b + a * c)
```

um die Rechenvorschrift `(a * b + c) * (b + a * c)` wiederverwenden zu können:

```
f 5 37 13
f 15 7 12
f 25 3 10
...
```

$\leadsto$ sog. *Funktionale Abstraktion* (verwandt: *Prozedurale Abstraktion*)

Zur Motivation param. Polymorphie (1)

Listen können Elemente sehr unterschiedlicher Typen zusammenfassen, z.B.

- Listen von Basistypelementen
  `[2,4,23,2,53,4] :: [Int]`
- Listen von Listen
  `[[2,4,23,2,5],[3,4],[],[56,7,6,]] :: [[Int]]`
- Listen von Paaren
  `[(3.14,42.0),(56.1,51.3),(1.12,2.22)] :: [Point]`
- Listen von Bäumen
  `[Nil,Node 42 Nil Nil), Node 17 (Node 4 Nil Nil) Nil)]`
- ...
- Listen von Funktionen
  `[fact, fib, fun91] :: [Integer -> Integer]`

Zur Motivation param. Polymorphie (2)

- *Aufgabe*: Bestimme die Länge einer Liste, d.h. die Anzahl ihrer Elemente.
- *Naive Lösung*: Schreibe für jeden Typ eine entsprechende Funktion.

Zur Motivation param. Polymorphie (4)

Umsetzung der naiven Lösung:

```
lengthIntLst :: [Int] -> Int
lengthIntLst []     = 0
lengthIntLst (_:xs) = 1 + lengthIntLst xs

lengthIntLstLst :: [[Int]] -> Int
lengthIntLstLst []     = 0
lengthIntLstLst (_:xs) = 1 + lengthIntLstLst xs

lengthPointLst :: [Point] -> Int
lengthPointLst []     = 0
lengthPointLst (_:xs) = 1 + lengthPointLst xs

lengthTreeLst :: [BinTree1] -> Int
lengthTreeLst []     = 0
lengthTreeLst (_:xs) = 1 + lengthTreeLst xs

lengthFunLst :: [Integer -> Integer] -> Int
lengthFunLst []     = 0
lengthFunLst (_:xs) = 1 + lengthFunLst xs
```

Zur Motivation param. Polymorphie (5)

Damit möglich:

```
lengthIntLst    [2,4,23,2,53,4] => 6
lengthIntLstLst [[2,4,23,2,5],[3,4],[],[56,7,6,]] => 4
lengthPointLst  [(3.14,42.0),(56.1,51.3),(1.12,2.22)] => 3
lengthTreeLst   [Nil,Node 42 Nil Nil, Node 17 (Node 4 Nil Nil) Nil)] => 3
lengthFunLst    [fact, fib, fun91] => 3
```

Zur Motivation param. Polymorphie (6)

Beobachtung:

- Die einzelnen Rechenvorschriften zur Längenberechnung sind i.w. identisch
- Unterschiede beschränken sich auf
  - Funktionsnamen und
  - Typsignaturen

Zur Motivation param. Polymorphie (7)

Sprachen, die parametrische Polymorphie offerieren, erlauben eine elegantere Lösung unserer Aufgabe:

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

```
length [2,4,23,2,53,4] => 6
length [[2,4,23,2,5],[3,4],[],[56,7,6,]] => 4
length [(3.14,42.0),(56.1,51.3),(1.12,2.22)] => 3
length [Nil,Node 42 Nil Nil, Node 17 (Node 4 Nil Nil) Nil)] => 3
length [fact, fib, fun91] => 3
```

Funktionale Sprachen, auch Haskell, offerieren parametrische Polymorphie!

Zur Motivation param. Polymorphie (8)

Unmittelbare Vorteile parametrischer Polymorphie:

- Wiederverwendung von
  - Rechenvorschriften und
  - Funktionsnamen (*Gute Namen sind knapp!*)

Monomorphie vs. Polymorphie

Rechenvorschriften der Form

- `length :: [a] -> Int` heißen *polymorph*.

Rechenvorschriften der Form

- `lengthIntLst :: [Int] -> Int`
- `lengthIntLstLst :: [[Int]] -> Int`
- `lengthPointLst :: [Point] -> Int`
- `lengthFunLst :: [Integer -> Integer] -> Int`
- `lengthTreeLst :: [BinTree1] -> Int` heißen *monomorph*.

Sprechweisen im Zshg. mit parametrischer Polymorphie

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

Sprechweisen:

- `a` in der Typsignatur von `length` heißt *Typvariable*. Typvariablen werden mit Kleinbuchstaben gewöhnlich vom Anfang des Alphabets bezeichnet: `a`, `b`, `c`,...

- Typen der Form...

  ```
  length :: [Point] -> Int
  length :: [[Int]] -> Int
  length :: [Integer -> Integer] -> Int
  ...
  ```

  heißen *Instanzen* des Typs `[a] -> Int`. Letzterer heißt *allgemeinster Typ* der Funktion `length`.

  Bem.: Das Hugs-Kommando `:t expr` liefert stets den (eindeutig bestimmten) *allgemeinsten* Typ eines (wohlgeformten) Haskell-Ausdrucks `expr`.

Weitere Bsp. polymorpher Funktionen

```
id :: a -> a                          -- Identitaetsfunktion
id x = x

id 3 => 3
id ["abc","def"] => ["abc","def"]


zip :: [a] -> [b] -> [(a,b)]          -- "Verschmelzen" von Listen
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _           = []

zip [3,4,5] ['a','b','c','d'] => [(3,'a'),(4,'b'),(5,'c')]
zip ["abc","def","geh"] [(3,4),(5,4)] => [("abc",(3,4)),("def",(5,4))]


unzip :: [(a,b)] -> ([a],[b])         -- "Aufreissen" von Listen
unzip []  = ([],[])
unzip ((x,y):ps) = (x:xs,y:ys)
                   where
                   (xs,ys) = unzip ps

unzip [(3,'a'),(4,'b'),(5,'c')] => ([3,4,5],['a','b','c'])
unzip [("abc",(3,4)),("def",(5,4))] => (["abc","def"],[(3,4),(5,4)])
```

Weitere in Haskell auf Listen vordefinierte (polymorphe) Funktionen

`:`	`:: a->[a]->[a]`	Listenkonstruktor (rechtsassoziativ)
`!!`	`:: [a]->Int->a`	Proj. auf i-te Komp., Infixop.
`length`	`:: [a]->Int`	Länge der Liste
`++`	`:: [a]->[a]->[a]`	Konkatenation zweier Listen
`concat`	`:: [[a]]->[a]`	Konkatenation mehrerer Listen
`head`	`:: [a]->a`	Listenkopf
`last`	`:: [a]->a`	Listenendelement
`tail`	`:: [a]->[a]`	Liste ohne Listenkopf
`init`	`:: [a]->[a]`	Liste ohne Listenendelement
`splitAt`	`:: Int->[a]->[[a],[a]]`	Aufspalten einer Liste an Stelle i
`reverse`	`:: [a]->[a]`	Umdrehen einer Liste
...		

Polymorphie

Soviel zu parametrischer Polymorphie auf Funktionen...

Wir fahren fort mit

- Polymorphie auf Datentypen
  - Algebraische Datentypen
  - Typsynonymen

Polymorphe algebraische Typen (1)

Der Schlüssel dazu...

$\leadsto$ Definitionen algebraischer Typen dürfen Typvariablen enthalten und werden dadurch polymorph.

Beispiele: Paare und Bäume...

```
data Pairs a = Pair a a

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

Polymorphe algebraische Typen (2)

Beispiele konkreter Werte von Paaren und Bäumen:

```
data Pairs a = Pair a a

Pair 17 4    :: Pairs Int
Pair [] [42] :: Pairs [Int]
Pair [] []   :: Pairs [a]


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

Node 'a' (Node 'b' Nil Nil) (Node 'z' Nil Nil)        :: Tree Char
Node 3.14 (Node 2.0 Nil Nil) (Node 1.41 Nil Nil)      :: Tree Float
Node "abc" (Node "b" Nil Nil) (Node "xyzzz" Nil Nil)  :: Tree [Char]
```

Polymorphe algebraische Typen (3)

Ähnlich wie parametrische Polymorphie unterstützt auch...

- Polymorphie auf algebraischen Datentypen

Wiederverwendung!

Vergleiche dies mit der schon bekannten Situation im Zshg. mit *polymorphen Listen*:

```
length :: [a] -> Int
```

Polymorphe Typen (4)

Ähnlich wie bei der Funktion *Länge* auf Listen...

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

...kann auf algebraischen Typen Typunabhängigkeit generell vorteilhaft ausgenutzt werden, wie hier das Bsp. der Funktion `depth` auf Bäumen zeigt:

```
depth :: Tree a -> Int
depth Nil = 0
depth (Node _ t1 t2) = 1 + max (depth t1) (depth t2)

depth (Node 'a' (Node 'b' Nil Nil) (Node 'z' Nil Nil))        => 2
depth (Node 3.14 (Node 2.0 Nil Nil) (Node 1.41 Nil Nil))      => 2
depth (Node "abc" (Node "b" Nil Nil) (Node "xyzzz" Nil Nil)) => 2
```

Heterogene algebraische Typen

...sind möglich, z.B. *heterogene* Bäume:

```
data HTree = LeafS String             |
             LeafI Int                |
             NodeF Float HTree HTree |
             NodeB Bool  HTree HTree

-- 2 Varianten der Funktion Tiefe auf Werten vom Typ HTree
depth :: HTree -> Int
depth (LeafS _) = 1
depth (LeafI _) = 1
depth (NodeF _ t1 t2) = 1 + max (depth t1) (depth t2)
depth (NodeB _ t1 t2) = 1 + max (depth t1) (depth t2)

depth :: HTree -> Int
depth (NodeF _ t1 t2) = 1 + max (depth t1) (depth t2)
depth (NodeB _ t1 t2) = 1 + max (depth t1) (depth t2)
depth _               = 1
```

Beachte: ...folgendes geht nicht $\leadsto$ *Syntaxfehler*!

```
depth :: HTree -> Int
depth (_ _ t1 t2) = 1 + max (depth t1) (depth t2)
depth _           = 1
```

Heterogene polymorphe algeb. Typen

...sind ebenfalls möglich, z.B. *heterogene* polymorphe Bäume:

```
data PHTree a b c d = LeafA a                                           |
                      LeafB b                                           |
                      NodeC c   (PHTree a b c d) (PHTree a b c d) |
                      NodeD d c (PHTree a b c d) (PHTree a b c d)

-- 2 Varianten der Funktion Tiefe auf Werten vom Typ PHTree
depth :: (PHTree a b c d) -> Int
depth (LeafA _)         = 1
depth (LeafB _)         = 1
depth (NodeC _ t1 t2)   = 1 + max (depth t1) (depth t2)
depth (NodeD _ _ t1 t2) = 1 + max (depth t1) (depth t2)

depth :: (PHTree a b c d) -> Int
depth (NodeC _ t1 t2)   = 1 + max (depth t1) (depth t2)
depth (NodeD _ _ t1 t2) = 1 + max (depth t1) (depth t2)
depth _                 = 1
```

Das heißt...

- Auch *"mehrfach" polymorphe* Datenstrukturen sind möglich!

Polymorphe Typsynonyme

...auch *polymorphe Typsynonyme* und *Funktionen* darauf sind möglich:

Beispiel:

```
type List a = [a]

lengthList :: List a -> Int
lengthList []     = 0
lengthList (_:xs) = 1 + lengthList xs
```

Oder kürzer:

```
lengthList :: List a -> Int
lengthList = length
```

...abstützen auf Standardfunktion `length` möglich, da `List a` Typsynonym, kein neuer Typ.

Beachte: `type List = [a]` ist nicht möglich ($\leadsto$ *Typfehler!*)

Zusammenfassung und erste Schlussfolgerungen

Zu...

- Polymorphen Funktionen,
- Polymorphen Datentypen und
- Vorteilen, die aus ihrer Verwendung resultieren.

Polymorphe Typen

...ein (Daten-) Typ `T` heißt *polymorph*, wenn bei der Deklaration von `T` der Grundtyp oder die Grundtypen der Elemente (in Form einer oder mehrerer Typvariablen) als Parameter angegeben werden.

Zwei Beispiele:

```
data Tree a b = Leaf a |
                Node b (Tree a b) (Tree a b)

data List a = Empty |
              (Head a) (List a)
```

Polymorphe Funktionen

...eine Funktion `f` heißt *polymorph*, wenn deren Parameter (in Form einer oder mehrerer Typvariablen) für Argumente unterschiedlicher Typen definiert sind.

Typische Beispiele:

```
depth  :: (Tree a b) -> Int
depth Leaf _         = 1
depth (Node _ t1 t2) = 1 + max (depth t1) (depth t2)

lengthLst :: List a -> Int
lengthLst Empty        = 0
lengthLst (Head _ hs) = 1 + lengthLst hs

-- Zum Vergleich die polymorphe Standardfunktion length
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

Warum polymorphe Typen und Funktionen? (1)

...Wiederverwendung durch Parametrisierung!

$\leadsto$ *wie schon gesagt, ein typisches Vorgehen in der Informatik*!

...die Essenz eines Datentyps (einer Datenstruktur) ist wie die Essenz darauf arbeitender Funktionen oft unabhängig von bestimmten typspezifischen Details.

(Man vergegenwärtige sich das noch einmal z.B. anhand von Bäumen über ganzen Zahlen, Zeichenreihen, Personen,... oder Listen über Gleitkommazahlen, Wahrheitswerten, Bäumen,... und typischen Funktionen darauf wie etwa zur Bestimmung der Tiefe von Bäumen oder der Länge von Listen.)

Warum polymorphe Typen und Funktionen? (2)

- ...durch Parametrisierung werden gleiche Teile “ausgeklammert” und somit der Wiederverwendung zugänglich!
- ...(i.w.) gleiche Codeteile müssen nicht (länger) mehrfach geschrieben werden.
- ...man spricht deshalb auch von *parametrischer Polymorphie*.

Warum polymorphe Typen und Funktionen? (3)

Polymorphie und die mit ihr verbundene Wiederverwendung unterstützt die...

- Ökonomie der Programmierung (vulgo: "*Schreibfaulheit*")

Insbesondere aber trägt sie bei zu höherer...

- Transparenz und Lesbarkeit
  ...durch Betonung der Gemeinsamkeiten, nicht der Unterschiede!
- Verlässlichkeit und Wartbarkeit (ein Aspekt mit mehreren Dimensionen: Fehlersuche, Weiterentwicklung,...)
  ...als erwünschte Nebeneffekte!
- ...
- Effizienz (der Programmierung)
  $\leadsto$ *höhere Produktivität, früherer Markteintritt* (*time-to-market*)

Warum polymorphe Typen und Funktionen? (4)

Beachte...

- ...auch in anderen Paradigmen wie etwa imperativer und speziell objektorientierter Programmierung lernt man, den Nutzen und die Vorteile polymorpher Konzepte zunehmend zu schätzen!

  $\leadsto$ aktuelles Stichwort: *Generic Java*

Ad hoc Polymorphie

Bisher haben wir besprochen...

- Polymorphie in Form von...
  - (Parametrischer) Polymorphie
  - Polymorphie auf Datentypen

Jetzt ergänzen wir diese Betrachtung um...

- *Ad hoc* Polymorphie (Überladen, “unechte” Polymorphie)

Zur Motivation von Ad hoc Polymorphie

Ausdrücke der Form

```
(+) 2 3           => 5
(+) 27.55 12.8    => 39.63
(+) 12.42 3       => 15.42
```

...sind Beispiele wohlgeformter Haskell-Ausdrücke, wohingegen

```
(+) True False
(+) 'a' 'b'
(+) [1,2,3] [4,5,6]
```

...Beispiele nicht wohlgeformter Haskell-Ausdrücke sind.

Zur Motivation von Ad hoc Polymorphie

Offenbar...

- ist (+) nicht monomorph
  ...da (+) für mehr als einen Argumenttyp arbeitet

- ist der Typ von (+) verschieden von `a -> a -> a`
  ...da (+) nicht für jeden Argumenttyp arbeitet

Tatsächlich...

- ist (+) typisches Beispiel eines *überladenen* Operators.

Das Kommando `:t (+)` in Hugs liefert

- `(+) :: Num a => a -> a -> a`

Polymorphie vs. Ad hoc Polymorphie

Intuitiv

- Polymorphie

  Der polymorphe Typ (`a -> a`) wie in der Funktion
  `id :: a -> a`   steht abkürzend für:

  ∀(`a`) `a -> a`   "...für alle Typen"

- Ad hoc Polymorphie

  Der Typ (`Num a => a -> a -> a`) wie in der Funktion
  `(+) :: Num a => a -> a -> a`   steht abkürzend für:

  ∀(`a` ∈ `Num`) `a -> a -> a`   "...für alle Typen aus `Num`"

Im Haskell-Jargon ist `Num` eine sog.

- *Typklasse*

...eine von vielen in Haskell vordefinierten Typklassen.

Typklassen in Haskell

Informell

- Eine Typklasse ist eine Kollektion von Typen, auf denen eine in der Typklasse festgelegte Menge von Funktionen definiert ist.
- Die Typklasse `Num` ist die Kollektion der numerischen Typen `Int`, `Integer`, `Float`, etc., auf denen die Funktionen `(+)`, `(*)`, etc. definiert sind.

Hinweis: Vergleiche dieses Klassenkonzept z.B. mit dem Schnittstellenkonzept aus Java. Gemeinsamkeiten, Unterschiede?

Polymorphie vs. Ad hoc Polymorphie

Informell...

- (*Parametrische*) *Polymorphie* ...
  $\leadsto$ gleicher Code trotz unterschiedlicher Typen

- *ad-hoc Polymorphie* (synonym: *Überladen* (engl. *Overloading*))...
  $\leadsto$ unterschiedlicher Code trotz gleichen Namens (mit i.a. sinnvollerweise ähnlicher Funktionalität)

Ein erweitertes Bsp. zu Typklassen (1)

Wir nehmen an, wir seien an der Größe interessiert von

- Listen und
- Bäumen

Der Begriff "Größe" sei dabei typabhängig, z.B.

- Anzahl der Elemente bei Listen
- Anzahl der
  - Knoten
  - Blätter
  - Benennungen
  - ...

  bei Bäumen

Ein erweitertes Bsp. zu Typklassen (2)

Wir betrachten folgende Baumvarianten...

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

data Tree1 a b = Leaf1 b |
                 Node1 a b (Tree1 a b) (Tree1 a b)

data Tree2 = Leaf2 String |
             Node2 String Tree2 Tree2
```

...und den Haskellstandardtyp für Listen.

Ein erweitertes Beispiel zu Typklassen

Naive Lösung: Schreibe für jeden Typ eine passende Funktion:

```
sizeT :: Tree a -> Int          -- Zaehlen der Knoten
sizeT Nil            = 0
sizeT (Node n l r) = 1 + sizeT l + sizeT r

sizeT1 :: (Tree1 a b) -> Int   -- Zaehlen der Benennungen
sizeT1 (Leaf1 m)       = 1
sizeT1 (Node1 m n l r) = 2 + sizeT1 l + sizeT1 r

sizeT2 :: Tree2 -> Int          -- Summe der Laengen der Benennungen
sizeT2 (Leaf2 m)     = length m
sizeT2 (Node2 m l r) = length m + sizeT2 l + sizeT2 r

sizeLst :: [a] -> Int           -- Zaehlen der Elemente
sizeLst = length
```

Ein erweitertes Bsp. zu Typklassen (3)

“Smarte” Lösung mithilfe von Typklassen:

```
class Size a where                      -- Definition der Typklasse Size
  size :: a -> Int

instance Size (Tree a) where            -- Instanzbildung fuer (Tree a)
  size Nil          = 0
  size (Node n l r) = 1 + size l + size r

instance Size (Tree1 a b) where  -- Instanzbildung fuer (Tree1 a b)
  size (Leaf1 m)       = 1
  size (Node1 m n l r) = 2 + size l + size r

instance Size Tree2 where               -- Instanzbildung fuer Tree2
  size (Leaf2 m)     = length m
  size (Node2 m l r) = length m + size l + size r

instance Size [a] where
  size = length
```

Ein erweitertes Bsp. zu Typklassen (4)

Das Kommando `:t size` liefert:

```
size :: Size a => a -> Int
```

...und wir erhalten wie gewünscht:

```
size Nil => 0
size (Node "asdf" (Node "jk" Nil Nil) Nil) => 2

size (Leaf1 "adf") => 1
size ((Node1 "asdf" 3
          (Node1 "jk" 2 (Leaf1 17) (Leaf1 4))
          (Leaf1 21)) => 7

size (Leaf2 "abc") => 3
size (Node2 "asdf"
       (Node2 "jkertt" (Leaf2 "abc") (Leaf2 "ac"))
       (Leaf "xy"))      => 17

size [5,3,45,676,7] => 5
size [True,False,True] => 3
```

Definition von Typklassen

Allgemeines Muster einer Typklassendefinition...

```
class Name tv where
    ...signature involving the type variable tv
```

wobei

- `Name` ...Identifikator der Klasse
- `tv` ...Typvariable
- `signature` ...Liste von Namen zusammen mit ihren Typen

Schlussfolgerungen zu Typklassen

Intuitiv...

...Typklassen sind Kollektionen von Typen, für die eine gewisse Menge von Funktionen ("gleicher" Funktionalität) definiert ist.

Beachte...

..."Gleiche" Funktionalität kann nicht syntaktisch erzwungen werden, sondern liegt in der Verantwortung des Programmierers!

Mithin: ...Appell an die *Programmierdisziplin* unabdingbar!

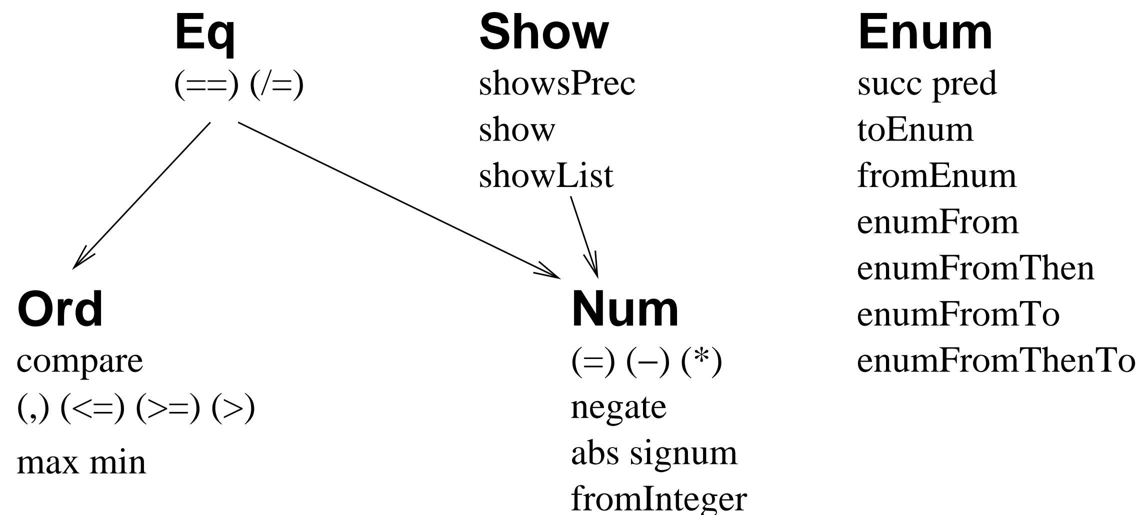
Bsp. einiger in Haskell vordef. Typklassen (1)

Vordefinierte Typklassen in Haskell...

- *Gleichheit* `Eq` ...die Klasse der Typen mit Gleichheitstest
- *Ordnungen* `Ord` ...die Klasse der Typen mit Ordnungsrelationen (wie etwa $<$, $\leq$, $>$, $\geq$, etc.)
- *Aufzählung* `Enum` ...die Klasse der Typen, deren Werte aufgezählt werden können (Bsp.: `[2,4..29]`)
- *Werte zu Zeichenreichen* `Show` ...die Klasse der Typen, deren Werte als Zeichenreihen dargestellt werden können
- *Zeichenreihen zu Werten* `Read` ...die Klasse der Typen, deren Werte aus Zeichenreihen herleitbar sind
- ...

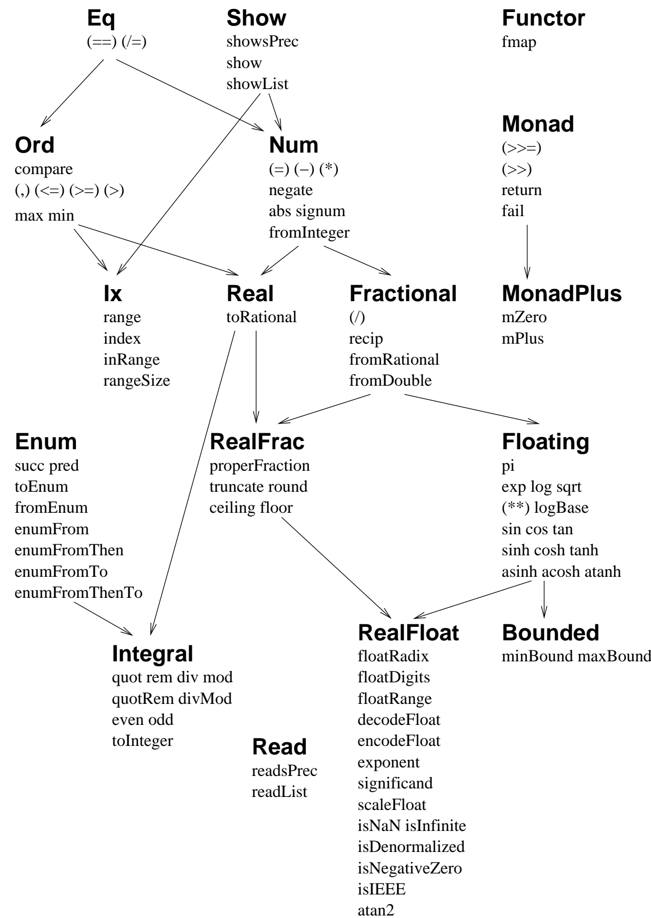
Bsp. einiger in Haskell vordef. Typklassen (2)

Auswahl vordefinierter Typklassen, ihrer Abhängigkeiten, Operatoren und Funktionen in Standard Prelude nebst Bibliotheken:



Quelle: Fethi Rabhi, Guy Lapalme. “Algorithms - A Functional Approach”, Addison-Wesley, 1999.

Typklassen: Ein größerer Ausschnitt vordef. Typklassen



Quelle: Fethi Rabhi, Guy Lapalme. “Algorithms - A Functional Approach”, Addison-Wesley, 1999, Figure 2.4.

Fortführung von Kapitel 3

Nachträge, Ergänzungen und Weiterführendes zu

- Ad hoc Polymorphie (Überladen)
  - Typklassen
  - Vererbung (Einfachvererbung, Mehrfachvererbung)
  - Überschreiben
  - Polymorphie vs. ad hoc Polymorphie
  - Vorteile für die Programmierung
- Listen, Muster und Funktionen
  - Listen und Muster
  - Listenkomprehension
  - Listenkonstruktoren vs. Listenoperatoren

Rückblick: Typklassenbeispiel (1)

Wir hatten angenommen, an der Größe interessiert zu sein von

- Listen und
- Bäumen

...wobei der Begriff “Größe” typabhängig gedacht war, z.B.

- Anzahl der
  - Elemente
  - *Tupelkomponenten* (als zusätzliches Beispiel)

  bei (*Tupel*-) Listen
- Anzahl der
  - Knoten
  - Blätter
  - Benennungen
  - ...

  bei Bäumen

Rückblick: Typklassenbeispiel (2)

- *Wunsch*: ...eine Funktion `size`, die mit Argumenten der verschiedenen Typen aufgerufen werden kann und das Gewünschte leistet.

- *Lösung*: ...ad hoc Polymorphie mittels Typklassen

Rückblick: Typklassenbeispiel (3)

Wir betrachteten folgende Baumvarianten...

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

data Tree1 a b = Leaf1 b |
                 Node1 a b (Tree1 a b) (Tree1 a b)

data Tree2 = Leaf2 String |
             Node2 String Tree2 Tree2
```

...und den Haskell-Standardtyp für Listen.

Rückblick: Typklassenbeispiel (4)

Haskells Typklassenkonzept erlaubte uns folgende Lösung:

```
class Size a where                -- Definition der Typklasse Size
  size :: a -> Int

instance Size (Tree a) where      -- Instanzbildung fuer (Tree a)
  size Nil          = 0
  size (Node n l r) = 1 + size l + size r

instance Size (Tree1 a b) where   -- Instanzbildung fuer (Tree1 a b)
  size (Leaf1 m)       = 1
  size (Node1 m n l r) = 2 + size l + size r

instance Size Tree2 where         -- Instanzbildung fuer Tree2
  size (Leaf2 m)     = length m
  size (Node2 m l r) = length m + size l + size r

instance Size [a] where
  size = length
```

Rückblick: Typklassenbeispiel (5)

Mit diesen Definitionen galt für den Typ der Funktion `size`:

```
size :: Size a => a -> Int
```

...und die Funktion `size` ermöglichte wie gewünscht:

```
size Nil => 0
size (Node "asdf" (Node "jk" Nil Nil) Nil) => 2

size (Leaf1 "adf") => 1
size (Node1 "asdf" 3
          (Node1 "jk" 2 (Leaf1 17) (Leaf1 4))
          (Leaf1 21) => 7

size (Leaf2 "abc") => 3
size (Node2 "asdf"
       (Node2 "jkertt" (Leaf2 "abc") (Leaf2 "ac"))
       (Leaf2 "xy"))      => 17

size [5,3,45,676,7] => 5
size [True,False,True] => 3
```

Als zusätzliches Beispiel (1)

...sei für Tupellisten "Größe" nicht durch die Anzahl der Listenelemente, sondern durch die Anzahl der Komponenten der tupelförmigen Listenelemente gegeben.

Lösung durch entsprechende Instanzbildung:

```
instance Size [(a,b)] where
  size = (*2) . length

instance Size [(a,b,c)] where
  size = (*3) . length
```

Beachte: Die Instanzbildung `instance Size [(a,b)]` geht über den Standard von Haskell 98 hinaus und ist nur in entsprechenden Erweiterungen möglich.

Als zusätzliches Beispiel (2)

Wie bisher gilt für den Typ der Funktion `size`:

```
size :: Size a => a -> Int
```

...und wir erhalten wie erwartet und gewünscht:

```
size [(5,"Smith"),(4,"Hall"),(7,"Douglas")] => 6
size [(5,"Smith",45),(4,"Hall",37),(7,"Douglas",42)] => 9
```

Anmerkungen

Sprechweisen:

- (*2), (*3) sind im Haskell-Jargon *operator sections*.
- "." bezeichnet in Haskell die aus der Mathematik bekannte Funktionskomposition:

  Sei $f : B \to C$ und $g : A \to B$, dann ist die *Funktionskomposition* $(f \circ g) : A \to C$ definiert durch:

  $$\forall\, a \in A.\ (f \circ g)(a) =_{df} f(g(a))$$

  Damit bedeutet z.B.

  ```
  ((*2) . length) [(5,"Smith"),(4,"Hall"),(7,"Douglas")]
  ```

  dasselbe wie

  ```
  (*2) (length [(5,"Smith"),(4,"Hall"),(7,"Douglas")])
  ```

Wermutstropfen (1)

Die Instanzbildungen

```
instance Size [a] where
  size = length

instance Size [(a,b)] where
  size = (*2) . length

instance Size [(a,b,c)] where
  size = (*3) . length
```

sind nicht gleichzeitig möglich.

Wermutstropfen (2)

Problem: Überlappende Typen!

```
ERROR "test.hs:45" - Overlapping instances for class "Size"
*** This instance   : Size [(a,b)]
*** Overlaps with   : Size [a]
*** Common instance : Size [(a,b)]
```

Konsequenz:

- Für Argumente von Instanzen des Typs `[(a,b)]` (und ebenso des Typs `[(a,b,c)]`) ist die Überladung des Operators `size` nicht mehr auflösbar

- Wünschenswert wäre:

```
instance Size [a] without [(b,c)],[(b,c,d)] where
  size = length
```

Beachte: In dieser Weise in Haskell nicht möglich.

Zusammenfassendes

...über die Funktion `size` und die Typklasse `Size`:

- die Typklasse `Size` stellt die Typspezifikation der Funktion `size` zur Verfügung
- jede Instanz der Typklasse `Size` muss eine instanzspezifische Implementierung der Funktion `size` zur Verfügung stellen
- Im Ergebnis ist die Funktion `size` wie auch z.B. die in Haskell vordefinierten Operatoren `+`, `*`, `-`, etc., oder die Relatoren `==`, `>`, `>=`, etc. *überladen*
- Synonym für *Überladen* ist *ad hoc Polymorphie*

Mehr zu Typklassen

Anders als die Typklasse `Size` können Typklassen auch

- Spezifikationen mehr als einer Funktion bereitstellen
- Standardimplementierungen (engl. *default implementations*) für (alle oder einige) dieser Funktionen bereitstellen
- von anderen Typklassen *erben*

In der Folge betrachten wir dies an ausgewählten Beispielen von in Haskell vordefinierten Typklassen...

Die vordefinierte Typklasse Eq (1)

Die in Haskell vordefinierte Typklasse `Eq`:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x==y)
  x == y = not (x/=y)
```

Die Typklasse `Eq` stellt

- Typspezifikationen von zwei Wahrheitswertfunktionen
- zusammen mit je einer Standardimplementierung

bereit.

Die vordefinierte Typklasse Eq (2)

Beachte:

- die Standardimplementierungen sind für sich allein nicht ausreichend, sondern stützen sich wechselweise aufeinander ab.

Trotz dieser Unvollständigkeit ergibt sich als Vorteil:

- Bei Instanzbildungen reicht es, entweder eine Implementierung für (==) oder für (/=) anzugeben. Für den jeweils anderen Operator gilt dann die vordefinierte Standard- (default) Implementierung.

- Auch für beide Funktionen können bei der Instanzbildung Implementierungen angegeben werden. In diesem Fall werden beide Standardimplementierungen *überschrieben*.

Übung: Vgl. dies z.B. mit Schnittstellendefinitionen und Definitionen abstrakter Klassen in Java. Welche Gemeinsamkeiten/Unterschiede gibt es?

Beispiele von Instanzbildungen der Typklasse Eq (1)

Am Beispiel des Typs der Wahrheitswerte:

```
instance Eq Bool where
  (==) True True   = True
  (==) False False = True
  (==) _ _         = False
```

Beachte: Der Ausdruck "Instanz" im Haskell-Jargon ist überladen!

- Bislang: Typ `T` ist Instanz eines Typs `U` (z.B. Typ `[Int]` ist Instanz des Typs `[a]`)
- Jetzt zusätzlich: Typ `T` ist Instanz einer (Typ-) Klasse `C` (z.B. Typ `Bool` ist Instanz der Typklasse `Eq`)

Beispiele von Instanzbildungen der Typklasse Eq (2)

Am Beispiel eines Typs für Punkte in der (x,y)-Ebene:

```
data Point = Point (Int,Int)

instance Eq Point where
  (==) (Point x) (Point y) = (fst(x)==fst(y)) &&
                              (snd(x)==snd(y))
```

Bemerkung: Mit feingranularenen Mustern lässt sich die Implementierung einfacher und transparenter realisieren:

```
instance Eq Point where
  (==) (Point (x,y)) (Point (u,v)) = (x==u) && (y==v))
```

Beachte: Typ- und Konstruktorname (`Point`!) dürfen übereinstimmen.

Beispiele von Instanzbildungen der Typklasse Eq (3)

Auch selbstdefinierte Typen können zu Instanzen vordefinierter Typklassen gemacht werden. Z.B. folgender Baumtyp zur Instanz der Typklasse `Eq`:

```
data Tree = Nil |
            Node Int Tree Tree
```

```
instance Eq Tree where
  (==) Nil Nil                = True
  (==) (Node m t1 t2) (Node n u1 u2)
                              = (m == n)   &&
                                (t1 == u1) &&
                                (t2 == u2)
  (==) _ _                    = False
```

Beispiele von Instanzbildungen der Typklasse Eq (4)

Das Vorgenannte gilt selbstverständlich auch für selbstdefinierte polymorphe Typen wie folgende Beispiele zeigen:

```
data Tree1 a   = Leaf1 a |
                 Node1 a (Tree1 a) (Tree1 a)

data Tree2 a b = Leaf2 b |
                 Node2 a b (Tree2 a b) (Tree2 a b)
```

Beispiele von Instanzbildungen der Typklasse Eq (5)

```
instance (Eq a) => Eq (Tree1 a) where
  (==) (Leaf1 s) (Leaf1 t)             = (s == t)
  (==) (Node1 s t1 t1) (Node1 t u1 u2) = (s == t)   &&
                                         (t1 == u1) &&
                                         (t2 == u2)
  (==) _ _                             = False

instance (Eq a, Eq b) => Eq (Tree2 a b) where
  (==) (Leaf2 q) (Leaf2 s)             = (q == s)
  (==) (Node2 p q t1 t1) (Node2 r s u1 u2)
                                       = (p == r)   &&
                                         (q == s)   &&
                                         (t1 == u1) &&
                                         (t2 == u2)
  (==) _ _                             = False
```

Bemerkungen:

- Getrennt durch Komma wie in `(Eq a, Eq b)` können in Kontexten mehrfache — konjunktiv zu verstehende — (Typ-) Bedingungen angegeben werden.

- Damit die Anwendbarkeit des Relators `(==)` auf Werte von Knotenbenennungen gewährleistet ist, muss die Instanzen der Typvariablen `a` und `b` selbst schon als Instanz der Typklasse `Eq` vorausgesetzt sein.

Einschub zu Sprechweisen

```
instance (Eq a) => Eq (Tree1 a) where
   (==) (Leaf1 s) (Leaf1 t)               = (s == t)
   (==) (Node1 s t1 t1) (Node1 t u1 u2) = (s == t)   &&
                                          (t1 == u1) &&
                                          (t2 == u2)
   (==) _ _                               = False
```

Sprechweisen und Vereinbarungen:

- `Tree1 a` ist Instanz der (gehört zur) Typklasse `Eq`, wenn `a` zu dieser Klasse gehört.
- Der Teil links von => heißt *Kontext*.
- Rechts von => dürfen ausschließlich Basistypen (z.B. `Int`), Typkonstruktoren beinhaltende Typen (z.B. `Tree a`, `[...]`) oder auf ausgezeichnete Typvariablen angewandte Tupeltypen (z.B. `(a,b,c,d)`) stehen.

Beispiele von Instanzbildungen der Typklasse Eq (6)

Instanzbildungen sind flexibel...

Abweichend von der vorher definierten Gleichheitsrelation auf Bäumen vom Typ (`Tree2 a b`), hätten wir den Gleichheitstest etwa auch so festlegen können, dass die Markierungen vom Typ `a` in inneren Knoten für den Gleichheitstest irrelevant sind:

```
instance (Eq b) => Eq (Tree2 a b) where
  (==) (Leaf2 q) (Leaf2 s)                = (q == s)
  (==) (Node2 _ q t1 t1) (Node2 _ s u1 u2)
                                          = (q == s)   &&
                                            (t1 == u1) &&
                                            (t2 == u2)
  (==) _ _                                = False
```

Beachte, dass für Instanzen des Typs `a` jetzt nicht mehr Mitgliedschaft in der Typklasse `Eq` gefordert werden muss.

Zusammenfassendes über den Relator (==) und die Typklasse Eq

Der Vergleichsoperator (==) ist...

- überladen (synonym: ad hoc polymorph), nicht echt polymorph

- in Haskell als Operation in der Typklasse `Eq` vorgegeben.

- damit anwendbar auf Werte aller Typen, die Instanz von `Eq` sind

- viele Typen sind bereits vordefinierte Instanz von `Eq`, z.B. alle elementaren Typen, Tupel und Listen über elementaren Typen

- auch selbstdefinierte Typen können zu Instanzen von `Eq` gemacht werden

Spezielle Frage

Ist es vorstellbar, jeden Typ zu einer Instanz der Typklasse `Eq` zu machen?

De facto hieße das, den Typ des Vergleichsoperators (==) von

```
(==) :: Eq a => a -> a -> Bool
```

zu

```
(==) :: a -> a -> Bool
```

zu verallgemeinern.

Zur Antwort (1)

Nein!

Der Grund ist im Kern folgender:

Anders als z.B. die Länge einer Liste, die eine vom konkreten Listentyp unabhängige Eigenschaft ist und deshalb eine (echt) polymorphe Eigenschaft ist und eine entsprechende Implementierung erlaubt

```
length :: [a] -> Int                -- echt polymorph
length []     = 0
length (_:xs) = 1 + length xs
```

ist Gleichheit eine typabhängige Eigenschaft, die eine typspezifische Implementierung verlangt.

Beispiel:

- unsere typspezifischen Implementierungen des Gleichheitstests auf Bäumen

Zur Antwort (3)

Warum ist nicht mehr möglich?

Im Sinne von Funktionen als *first class citizens* in funktionalen Sprachen wäre ein Gleichheitstest auch auf Funktionen sicher höchst wünschenswert.

Z.B.:

```
(==) fac fib                      =>  False
(==) (\x -> x+x) (\x -> 2*x)  =>  True
(==) (+2) doubleInc              =>  True
```

Zur Antwort (4)

In Haskell erforderte eine Umsetzung Instanzbildungen der Art:

```
instance Eq (Int -> Int) where
(==) f g = ...

instance Eq (Int -> Int -> Int) where
(==) f g = ...
```

Können wir die "Punkte" so ersetzen, dass wir einen Gleichheitstest für alle Funktionen der Typen `(Int -> Int)` und `(Int -> Int -> Int)` haben?

Zur Antwort (5)

Leider nein!

Zwar läßt sich für konkret vorgelegte Funktionen Gleichheit fallweise (algorithmisch) entscheiden, generell aber gilt folgendes aus der Theoretischen Informatik bekannte negative Ergebnis:

Theorem
Gleichheit von Funktionen ist unentscheidbar.

Zur Antwort (6)

Erinnerung:

“Gleichheit von Funktionen ist unentscheidbar” heißt informell, dass...

- es gibt keinen Algorithmus, der für zwei beliebig vorgelegte Funktionen stets nach endlich vielen Schritten entscheidet, ob diese Funktionen gleich sind oder nicht.

Machen Sie sich klar, dass daraus in der Tat nicht folgt, dass Gleichheit zweier Funktionen nie (*in endlicher Zeit*) *entschieden werden kann.*

Schlussfolgerung (1)

...anhand der Beobachtungen am Gleichheitstest (==):

- ...offenbar können Funktionen bestimmter Funktionalität nicht für jeden Typ angegeben werden, insbesondere lässt sich nicht für jeden Typ eine Implementierung des Gleichheitsrelators (==) angeben, sondern nur für eine Teilmenge aller möglichen Typen.

- ...die Teilmenge der Typen, für die das für den Gleichheitsrelator möglich ist, bzw. eine Teilmenge davon, für die das in einem konkreten Haskell-Programm tatsächlich gemacht wird, ist im Haskell-Jargon eine *Sammlung* (engl. *collection*) von Typen, eine sog. *Typklasse*.

Schlussfolgerung (2)

Auch wenn es schön wäre, eine (echt) polymorphe Implementierung von (==) zu haben zur Signatur

```
(==) :: a -> a -> Bool
```

und damit analog zur Funktion zur Längenbestimmung von Listen

```
length :: [a] -> Int
```

...ist eine Implementierung in dieser Allgemeinheit für (==) nicht möglich!

Die Typklassen, für die eine Implementierung von (==) angegeben werden kann, sind in Haskell in der Typklasse `Eq` zusammengefasst.

Typklassen und Vererbung (1)

Vererbung auf Typklassenebene...

```
class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min             :: a -> a -> a
  compare              :: a -> a -> Ordering
  x <= y        = (x<y) || (x==y)
  x > y         = y < x
  ...
  compare x y
    | x == y    = EQ
    | x <= y    = LT
    | otherwise = GT
```

Typklassen und Vererbung (2)

- Die (wie `Eq` vordefinierte) Typklasse `Ord` erweitert die Klasse `Eq`.
- Jede Instanz der Typklasse `Ord` muss Implementierungen für alle Funktionen der Klassen `Eq` und `Ord` bereitstellen.

Beachte:

- `Ord` stellt wie `Eq` für einige Funktionen bereits Standardimplementierungen bereit.
- Bei der Instanzbildung für weitere Typen reicht es deshalb, Implementierungen der Relatoren (==) und (<) anzugeben.
- Durch Angabe instanzspezifischer Implementierungen bei der Instanzbildung können diese Standardimplementierungen aber auch nach Wunsch überschrieben werden.

Typklassen und Vererbung (3)

Auch *Mehrfachvererbung* auf Typklassenebene ist möglich, wie Haskells vordefinierte Typklasse `Num` zeigt:

```
class (Eq a, Show a) => Num a where
(+), (-), (*) :: a -> a -> a
negate        :: a -> a
abs, signum   :: a -> a
fromInteger   :: Integer -> a    -- Typkonversionsfunktion!
fromInt       :: Int -> a        -- Typkonversionsfunktion!

x - y         = x + negate y
fromInt       = ...
```

- ...vgl. dies auch mit Vererbungskonzepten objekt-orientierter Sprachen!

Typklassen und Vererbung (4)

Überschreiben ererbter Funktionen am Beispiel der Instanz `Point` der Typklasse `Eq`:

- *Vererbung*:
  Für die Instanzdeklaration von `Point` zur Klasse `Eq`

  ```
  instance Eq Point where
    Point (x,y) == Point (w,z) = (x==w) && (y==z)
  ```

  erbt `Point` folgende Implementierung des Ungleichheitstests (/=) aus der Klasse `Eq`:

  ```
  Point x /= Point y = not (Point x == Point y)
  ```

- *Überschreiben*:
  Die ererbte (Standard-) Implementierung von (/=) kann überschrieben werden, z.B. wie unten durch eine (geringfügig) effizientere Variante:

  ```
  instance Eq Point where
    Point (x,y) == Point (w,z) = (x==w) && (y==z)
    Point x /= Point y         = if x/=w then True else y/=z
  ```

Typklassen und Vererbung (5)

(*Automatisch*) *abgeleitete Instanzen* von Typklassen...

```
data Spielfarbe = Kreuz | Pik | Herz | Karo
                  deriving (Eq,Ord,Enum,Bounded,Show,Read)
data Tree a = Nil |
              Node a (Tree a) (Tree a)
              deriving (Eq,Ord)
```

- Algebraische Typen können durch Angabe einer `deriving`-Klausel als Instanzen vordefinierter Klassen automatisch angelegt werden.

- Intuitiv ersetzt die Angabe der `deriving`-Klausel die Angabe einer `instance`-Klausel.

Typklassen und Vererbung (6)

So ist

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

gleichbedeutend zu:

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

instance Eq a => Eq (Tree a) where
  (==) Nil Nil                = True
  (==) (Node m t1 t2) (Node n u1 u2)
                              = (m == n)   &&
                                (t1 == u1) &&
                                (t2 == u2)
  (==) _ _                    = False
```

Typklassen und Vererbung (7)

Analog ist

```
data Tree2 a b = Leaf2 bl |
                 Node2 a b (Tree2 a b) (Tree2 a b) deriving Eq
```

gleichbedeutend zu:

```
data Tree2 a b = Leaf2 bl |
                 Node2 a b (Tree2 a b) (Tree2 a b)

instance (Eq a, Eq b) => Eq (Tree2 a b) where
  (==) (Leaf2 q) (Leaf2 s)                   = (q == s)
  (==) (Node2 p q t1 t1) (Node2 r s u1 u2)
                                             = (p == r)   &&
                                               (q == s)   &&
                                               (t1 == u1) &&
                                               (t2 == u2)
  (==) _ _                                   = False
```

Typklassen und Vererbung (8)

Möchten Sie hingegen Gleichheit wie folgt realisiert wissen

```
data Tree2 a b = Leaf2 bl |
                 Node2 a b (Tree2 a b) (Tree2 a b)

instance (Eq a, Eq b) => Eq (Tree2 a b) where
  (==) (Leaf2 q) (Leaf2 s)            = (q == s)
  (==) (Node2 _ q t1 t1) (Node2 _ s u1 u2)
                                      = (q == s)   &&
                                        (t1 == u1) &&
                                        (t2 == u2)
  (==) _ _                            = False
```

geht an obiger Instanzdeklaration kein Weg vorbei.

Typklassen und Vererbung (9)

Mehr zu Typklassen, alles zu Funktionen vordefinierter Typklassen und über Grenzen und Einschränkungen etwa automatischer Ableitbarkeit von Typklassen...

- ...in jedem guten Buch über Haskell!

Zum Abschluss dieses Themas

Polymorphie und *Überladen* auf Funktionen bedeuten...

- vordergründig
  ... ein Funktionsname kann auf Argumente unterschiedlichen Typs angewendet werden.

- präziser
  - *Polymorphe Funktionen...*
    - ...haben eine einzige Implementierung, die für alle (zugelassenen/abgedeckten) Typen arbeitet
      (Bsp.: `length :: [a] -> Int]`)
  - *Überladene Funktionen...*
    - ...arbeiten für Instanzen einer Klasse von Typen mit einer für jede Instanz spezifischen Implementierung
      (Bsp.: `size :: Size a => a -> Int`)

Vorteile des Überladens von Operatoren

- Ohne Überladen ginge es nicht ohne ausgezeichnete Namen für alle Operatoren.
- Das gälte auch für die bekannten arithmetischen Operatoren, so wären insbesondere Namen der Art $+_{Int}$, $+_{Float}$, $*_{Int}$, $*_{Float}$, etc. erforderlich.
- Deren zwangweiser Gebrauch wäre nicht nur ungewohnt und unschön, sondern in der täglichen Praxis auch lästig.
- Haskells Angebot, hier Abhilfe zu schaffen und Operatoren zu überladen, ist das Konzept der *Typklassen*.

Zum Selbststudium: Andere Sprachen wie z.B. ML und Opal gehen hier einen anderen Weg und bieten andere Konzepte.

Kapitel 4: Listen und Listenkomprehension, Muster

Nachträge und Ergänzungen...

- Listen, Muster und Funktionen
- Listenkomprehension *at work*

Zurück zu Listen, Mustern und Funktionen darauf (1)

Einige Beispiele...

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

head :: [a] -> a
head (x:_)  =  x

tail :: [a] -> [a]
tail (_:xs) = xs

null :: [a] -> Bool
null []    = True
null (_:_) = False
```

Zurück zu Listen, Mustern und Funktionen darauf (2)

Muster, Wild Cards und Typvariablen...

```
sign :: Integer -> Integer
sign x
  | x > 0  = 1
  | x == 0 = 0
  | x < 0  = -1

takeFirst :: Integer -> [a] -> [a]
takeFirst m ys =  case (m,ys) of
                    (0,_)    -> []
                    (_,[])   -> []
                    (n,x:xs) -> x : takeFirst (n - 1) xs

ifThenElse :: Bool -> a -> a -> a
ifThenElse c t e = case c of True  -> t
                             False -> e
```

Somit erhalten wir als Fortschreibung...

...des Musterbegriffs: Muster können sein...

Bislang:

- *Werte* (z.B. 0, 'c', True)
  ...ein Argument "passt" auf das Muster, wenn es vom entsprechenden Wert ist.
- *Variablen* (z.B. n)
  ...jedes Argument passt (und ist rechtsseitig verwendbar).
- *Wild card* "_"
  ...jedes Argument passt (aber ist rechtsseitig nicht verwendbar).

Jetzt zusätzlich:

- *Konstruktormuster* (hier über Listen; z.B. [], (p:ps))
  - Eine Liste passt auf das Muster [], wenn sie leer ist.
  - Eine Liste passt auf (p:ps), wenn sie nicht leer ist und der Listenkopf auf p und der Listenrest auf ps passt.

Hinweis: Im Fall von (p:ps) reicht es, dass die Argumentliste nicht leer ist.

Oft sehr nützlich

...das sog. `as`-Muster (mit `@` gelesen als "as"):

Beispiel:

```
listTransform :: [a] -> [a]
listTransform l@(x:xs) = (x : l) ++ xs
```

Zum Vergleich ohne `as`-Muster...

```
listTransform :: [a] -> [a]
listTransform (x:xs) = (x : (x : xs)) ++ xs
```

...weniger elegant und weniger gut lesbar.

Ein weiteres Beispiel

Auch Funktionsdeklarationen der Form...

```
binom :: (Integer,Integer) -> Integer
binom (n,k)
  | k==0 || n==k   = 1
  | otherwise      = binom (n-1,k-1) + binom (n-1,k)
```

...sind Beispiele *musterbasierter* Funktionsdefinitionen.

Zum Vergleich...

...mit Standardselektoren ohne Muster:

```
binom :: (Integer,Integer) -> Integer
binom p
  | snd(p)==0 || snd(p)==fst(p)  = 1
  | otherwise  = binom (fst(p)-1,snd(p)-1)
                               + binom (fst(p)-1,snd(p))
```

...offenbar auch hier weniger elegant und weniger gut lesbar.

Schlussfolgerung zwischendurch

Musterbasierte Funktionsdefinitionen sind (i.a.)...

- elegant und
- führen zu knappen, gut lesbaren Spezifikationen.

Zu beachten aber ist: Musterbasierte Funktionsdefinitionen...

- können zu subtilen Fehlern führen und
- erschweren (oft) Programmänderungen/-weiterentwicklungen ("bis hin zur Tortur" (vgl. Pepper [4]): denke etwa an das Hinzukommen eines weiteren Parameters)

Listen, Listenkonstruktoren, Listenoperatoren

Wir kennen den vordefinierten Listentyp `String`:

```
Type String = [Char]
```

...und Beispiele gültiger Werte des Typs `String`, etwa:

```
['h','e','l','l','o'] == "hello"
"hello" ++ " world" -> "hello world"  (++: vordef. Konkatenationsop.)
```

Wir hatten aber auch gesehen, dass Elementtypen weit komplexer sein dürfen, bis hin zu Funktionen (Funktionen als "first class citizens"):

- Listen von Listen
  `[[2,4,23,2,5],[3,4],[],[56,7,6,]] :: [ [Int] ]`
- Listen von Paaren
  `[(3.14,42.0),56.1,51.3),(1.12,2.22)] :: [ Point ]`
- ...
- Listen von Funktionen
  `[fac, fib, fun91] :: [ Integer -> Integer ]`

Ist die Zulässigkeit von `[fac, fib, fun91] :: [ Integer -> Integer ]` bemerkenswert?

Erinnerung aus der Mathematik...

Eine Funktion f ist ein Tripel (D, W, G) mit einer Definitionsmenge (-bereich) D, einer Wertemenge (-bereich) W und einer rechtseindeutigen Relation G mit $G \subseteq D \times W$, dem sog. Funktionsgraphen von f.

Mithin...

Funktionen sind spezielle Relationen; spezielle Teilmengen *eines kartesischen Produkts*

Damit intuitiv naheliegend...

Listen von Funktionen ... "Listen von Listen von Paaren"

Schlagwort...

Funktionen als *"first class citizens"*

Listenkomprehension (1)

...ein für funktionale Programmiersprachen charakteristisches Ausdrucksmittel.

- Listenkomprehension

  ...ein einfaches Beispiel:

  `[3*n | n <- list]` steht kurz für `[3,6,9,12]`, wobei `list` vom

  Wert `[1,2,3,4]` vorausgesetzt ist.

$\leadsto$ Listenkomprehension ist ein sehr elegantes und ausdruckskräftiges Sprachkonstrukt!

Listenkomprehension (2)

Weitere Anwendungsbeispiele:

...wobei `lst = [1,2,4,7,8,11,12,42]` vorausgesetzt ist:

a) `[ square n | n <- lst ]` $\Rightarrow$ `[1,4,16,49,64,121,144,1764]`

b) `[ n | n <- lst, isPowOfTwo n ]` $\Rightarrow$ `[1,2,4,8]`

c) `[ n | n <- lst, isPowOfTwo n, n>=5 ]` $\Rightarrow$ `[8]`

d) `[ isPrime n | n <- lst ]` $\Rightarrow$

`[False,True,False,True,False,True,False,False]`

Listenkomprehension (3)

e)
```
addCoordinates :: [Point] -> [Float]
addCoordinates pLst = [ x+y | (x,y)<-pLst, (x>0||y>0) ]
```

```
addCoordinates [(0.0,0.5),(3.14,17.4),(-1.5,-2.3)] ⇒
                                              [0.5,20.54]
```

f)
```
allOdd :: [Integer] -> Bool
allOdd xs = ( [ x | x<-xs, isOdd x ] == xs )
```

```
allEven :: [Integer] -> Bool
allEven xs = ( [ x | x<-xs, isOdd x ] == [] )
```

Listenkomprehension (4)

g)
```
grabCapVowels :: String -> String
grabCapVowels s = [ c | c<-s, isCapVowel c ]

isCapVowel :: Char -> Bool
isCapVowel 'A' = True
isCapVowel 'E' = True
isCapVowel 'I' = True
isCapVowel 'O' = True
isCapVowel 'U' = True
isCapVowel  c  = False
```

Listenkomprehension "at work" (1)

...am Beispiel von "Quicksort".

Aufgabe: Sortiere eine Liste L ganzer Zahlen aufsteigend.

Lösung mittels Quicksort:

- *Teile*: Wähle ein Element l aus L und partitioniere L in zwei (möglicherweise leere) Teillisten L_1 und L_2 so, dass alle Elemente von L_1 (L_2) kleiner oder gleich (größer) dem Element l sind.

- *Herrsche*: Sortiere L_1 und L_2 mit Hilfe rekursiver Aufrufe von Quicksort.

- *Zusammenführen der Teilergebnisse*: Trivial, die Gesamtliste entsteht durch Konkatenation der sortierten Teillisten.

Listenkomprehension “at work” (2)

QuickSort: Eine typische Realisierung in Haskell...

```
quickSort :: [Int] -> [Int]

quickSort []     = []
quickSort (x:xs) =
    quickSort [ y | y<-xs, y<=x ] ++
                     [x] ++ quickSort [ y | y<-xs, y>x ]
```

Beachte: Funktionsanwendung bindet stärker als Listenkonstruktion. Deshalb Klammerung des Musters `x:xs` in `quickSort (x:xs) = ...`

Listenkomprehension “at work” (3)

Zum Vergleich: Eine typische imperative Realisierung von QuickSort...

```
quickSort (L,low,high)
  if low < high
    then splitInd = partition(L,low,high)
         quickSort(L,low,splitInd-1)
         quickSort(L,splitInd+1,high) fi

partition (L,low,high)
  l = L[low]
  left = low
  for i=low+1 to high do
    if L[i] <= l then left = left+1
                      swap(L[i],L[left]) fi od
  swap(L[low],L[left])
  return left
```

...und dem initialen Aufruf `quickSort(L,1,length(L))`.

Rückblick auf Kap. 1, Teil 1 der Vorlesung

Imperative vs. funktionale Programmierung – Ein Vergleich:

Gegeben ein Problem P.

Imperativ: Typischer Lösungsablauf besteht aus folgenden Schritten

1. Beschreibe eine(n) Lösung(sweg) L für P
2. Gieße L in die Form einer Menge von Anweisungen für den Rechner und organisiere dabei die Speicherverwaltung

Funktional:

- ...das "was" statt des "wie" in den Vordergrund stellen
- $\rightsquigarrow$ etwas von der Eleganz der Mathematik in die Programmierung bringen!

Quicksort: Ein eindrucksvolles Beispiel? Urteilen Sie selbst...

Listenkonstruktoren vs. Listenoperatoren

Der Operator (:) ist Listenkonstruktor, (++) Listenoperator...

Abgrenzung: Konstruktoren führen zu eindeutigen Darstellungen, gewöhnliche Operatoren i.a. nicht.

Veranschaulicht am Beispiel von Listen:

```
42:17:4:[]  ==  (42:(17:(4:[])))          -- eindeutig

[42,17] ++ [] ++ [4] == [42,17,4] == [42] ++ [17,4] ++ []
                                          -- nicht eindeutig
```

Bemerkung: `(42:(17:(4:[])))` deutet an, dass eine Liste *ein* Objekt ist, erzwungen durch die Typstruktur. Anders in imperativen/objektorientierten Sprachen: Listen sind dort nur indirekt existent, nämlich bei "geeigneter" Verbindung von Elementen durch Zeiger.

Kapitel 5: Funktionen höherer Ordnung

Funktionen höherer Ordnung (*kurz*: Funktionale)

- Funktionen als Argumente
- Funktionen als Resultate
- Spezialfall: Funktionale auf Listen
- Anwendungen

...der Schritt von *applikativer* zu *funktionaler* 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:

- Mathematik: *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`
- Informatik: *Semantik von Programmiersprachen*
  - Denotationelle Semantik der while-Schleife

    $\mathcal{S}_{ds}[\![\, \texttt{while b do S od} \,]\!] : \Sigma \rightarrow \Sigma$

    ...kleinster Fixpunkt eines Funktionals auf der Menge der Zustandstransformationen $[\Sigma \rightarrow \Sigma]$ über der Menge der Zustände Σ mit $\Sigma =_{def} \{\sigma \,|\, \sigma \in [Var \;\rightarrow\; Data]\}$.

    (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 in Kürze 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 319 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]

Kapitel 6: 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 Berechnung 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.

Kapitel 7: Ausdrücke, Auswertung von Ausdrücken, Auswertungsstrategien

- Ausdrücke, Auswertung von Ausdrücken, Auswertungsstrategien

  Schlagwörter: applicative und normal order evaluation, eager und lazy evaluation, ...

Auswerten von Ausdrücken und Funktionsaufrufen

$e :: Float$

$e = 2.71828$

$res = 2 * e * e \Rightarrow \; \ldots$

simple	::	Int−>Int−>Int−>Int
simple x y z	=	(x + z) ∗ (y + z)

$simple\ 2\ 3\ 4 \Rightarrow \; \ldots$

fac	::	Int −> Int
fac n	=	if n == 0 then 1 else n ∗ fact (n − 1)

$fac\ 2 \Rightarrow \; \ldots$

Auswerten von Ausdrücken

Einfache Ausdrücke

Viele (*Simplifikations-*) Wege führen zum Ziel...

$3 * (9 + 5) \Rightarrow 3 * 14 \Rightarrow 42$

oder

$3 * (9 + 5) \Rightarrow 3 * 9 + 3 * 5 \Rightarrow 27 + 3 * 5 \Rightarrow 27 + 15 \Rightarrow 42$

oder ...

Auswerten von Ausdrücken und Funktionsaufrufen (1)

Grundlegend:

- *Expandieren*
- *Simplifizieren*

Auswerten von Ausdrücken und Funktionsaufrufen (2)

Beispiele:

simple x y z = (x + z) ∗ (y + z)

simple 2 3 4 ⇒ (2 + 4) ∗ (3 + 4) (Expandieren)

⇒ 6 ∗ (3 + 4) (Simplifizieren)

⇒ 6 ∗ 7 (Simplifizieren)

⇒ 42 (Simplifizieren)

oder

simple 2 3 4 ⇒ (2 + 4) ∗ (3 + 4) (Expandieren)

⇒ (2 + 4) ∗ 7 (Simplifizieren)

⇒ 6 ∗ 7 (Simplifizieren)

⇒ 42 (Simplifizieren)

oder ...

Funktionsaufrufe

fac n = if n == 0 then 1 else (n ∗ fac (n − 1))

fac 2 ⇒ if 2 == 0 then 1 else (2 ∗ fac (2 − 1))
⇒ 2 ∗ fac (2 − 1)

Weiter mit a)

⇒ 2 ∗ fac 1
⇒ 2 ∗ (if 1 == 0 then 1 else (1 ∗ fac (1−1)))
⇒ analog fortführen...

...oder mit b)

⇒ 2 ∗ (if (2−1) == 0 then 1 else ((2−1) ∗ fac ((2−1)−1)))
⇒ analog fortführen...

Schließlich...

⇒ 2

Auswertung gemäß Variante a)

fac n = if n == 0 then 1 else (n ∗ fac (n − 1))

$$
\begin{aligned}
\text{fac } 2 &\Rightarrow \text{if } 2 == 0 \text{ then } 1 \text{ else } (2 * \text{fac } (2 - 1)) \\
&\Rightarrow 2 * \text{fac } (2 - 1) \\
&\Rightarrow 2 * \text{fac } 1 \\
&\Rightarrow 2 * (\text{if } 1 == 0 \text{ then } 1 \text{ else } (1 * \text{fac } (1 - 1))) \\
&\Rightarrow 2 * (1 * \text{fac } (1 - 1)) \\
&\Rightarrow 2 * (1 * \text{fac } 0) \\
&\Rightarrow 2 * (1 * (\text{if } 0 == 0 \text{ then } 1 \text{ else } (0 * \text{fac } (0 - 1)))) \\
&\Rightarrow 2 * (1 * 1) \\
&\Rightarrow 2 * 1 \\
&\Rightarrow 2
\end{aligned}
$$

Auswertung gemäß Variante b)

$\text{fac n} = \text{if n} == 0 \text{ then } 1 \text{ else } (\text{n} * \text{fac } (\text{n} - 1))$

$$
\begin{aligned}
\text{fac } 2 &\Rightarrow \text{if } 2 == 0 \text{ then } 1 \text{ else } (2 * \text{fac } (2 - 1)) \\
&\Rightarrow 2 * \text{fac } (2 - 1) \\
&\Rightarrow 2 * (\text{if } (2-1) == 0 \text{ then } 1 \text{ else } ((2-1) * \text{fac } ((2-1)-1))) \\
&\Rightarrow 2 * ((2-1) * \text{fac } ((2-1)-1)) \\
&\Rightarrow 2 * (1 * \text{fac } ((2-1)-1)) \\
&\Rightarrow 2 * (1 * (\text{if } ((2-1)-1) == 0 \text{ then } 1 \\
&\Rightarrow \qquad\qquad \text{else } ((2-1)-1) * \text{fac } (((2-1)-1)-1))) \\
&\Rightarrow 2 * (1 * 1) \\
&\Rightarrow 2 * 1 \\
&\Rightarrow 2
\end{aligned}
$$

Freiheitsgrade...

Betrachte...

```
fac(fac(square(2+2))) * fact(fac(square(3)))
```

Zentral...

- **Wo** im Ausdruck mit der Auswertung fortfahren?
- **Wie** mit (Funktions-) Argumenten umgehen?

Gretchenfrage...

- **Welcher** Einfluss auf das Ergebnis?

Glücklicherweise...

Theorem
Jede terminierende Auswertungsreihenfolge endet mit demselben Ergebnis.

...Alonzo Church/J. Barclay Rosser (1936)

Auswertungsstrategien

In der Praxis sind besonders zwei Strategien von Bedeutung:

Um den Ausdruck $f(exp)$ auszuwerten...

a) *Applicative order evaluation*:
...berechne zunächst den Wert von exp und setze diesen Wert dann im Rumpf von f ein (s. Variante a))
   - $\leadsto$ applicative order evaluation, eager evaluation, call-by-value evaluation, leftmost-innermost evaluation

b) *Normal order evaluation*:
...setze exp unmittelbar im Rumpf von f ein und werte den so entstehenden Ausdruck aus (s. Variante b))
   - $\leadsto$ normal order evaluation, call-by-name evaluation, leftmost-outermost evaluation
   - $\leadsto$ *"Intelligente" Realisierung*: lazy evaluation, call-by-need evaluation

Ein Beispiel

Einige einfache Funktionen:

```
-- Die Funktion square zur Quadrierung einer ganzen Zahl
square :: Int -> Int
square n = n*n

-- Die Funktion first zur Projektion auf die erste Paarkomponente
first :: (Int,Int) -> Int
first (m,n) = m

-- Die Funktion infiniteInc zum "ewigen" Inkrement
infiniteInc :: Int
infiniteInc = 1 + infiniteInc
```

Auswertung gemäß...

...applicative order (leftmost-innermost):

$$
\begin{aligned}
& \texttt{square}(\texttt{square}(\texttt{square}(2))) \\
\Rightarrow\ & \texttt{square}(\texttt{square}(2 * 2)) \\
\Rightarrow\ & \texttt{square}(\texttt{square}(4)) \\
\Rightarrow\ & \texttt{square}(4 * 4) \\
\Rightarrow\ & \texttt{square}(16) \\
\Rightarrow\ & 16 * 16 \\
\Rightarrow\ & 256
\end{aligned}
$$

...6 Schritte.

Auswertung gemäß...

...normal order (leftmost-outermost):

$$
\begin{array}{ll}
& \texttt{square}(\texttt{square}(\texttt{square}(2))) \\
\Rightarrow & \texttt{square}(\texttt{square}(2)) * \texttt{square}(\texttt{square}(2)) \\
\Rightarrow & (\texttt{square}(2) * \texttt{square}(2)) * \texttt{square}(\texttt{square}(2)) \\
\Rightarrow & ((2 * 2) * \texttt{square}(2)) * \texttt{square}(\texttt{square}(2)) \\
\Rightarrow & (4 * \texttt{square}(2)) * \texttt{square}(\texttt{square}(2)) \\
\Rightarrow & (4 * (2 * 2)) * \texttt{square}(\texttt{square}(2)) \\
\Rightarrow & (4 * 4) * \texttt{square}(\texttt{square}(2)) \\
\Rightarrow & 16 * \texttt{square}(\texttt{square}(2)) \\
\Rightarrow & ... \\
\Rightarrow & 16 * 16 \\
\Rightarrow & 256
\end{array}
$$

...1+6+6+1=14 Schritte.

Applicative order effizienter?

Nicht immer...

$$\texttt{first}\ (42, \texttt{square}(\texttt{square}(\texttt{square}(2))))$$

...in applicative order

$$\begin{aligned} & \texttt{first}\ (42, \texttt{square}(\texttt{square}(\texttt{square}(2)))) \\ \Rightarrow\ & ... \\ \Rightarrow\ & \texttt{first}\ (42, 256) \\ \Rightarrow\ & 42 \end{aligned}$$

...1+6+1=8 Schritte.

...in normal order

$$\begin{aligned} & \texttt{first}\ (42, \texttt{square}(\texttt{square}(\texttt{square}(2)))) \\ \Rightarrow\ & 42 \end{aligned}$$

...1 Schritt.

Von Normal Order zu Lazy Evaluation

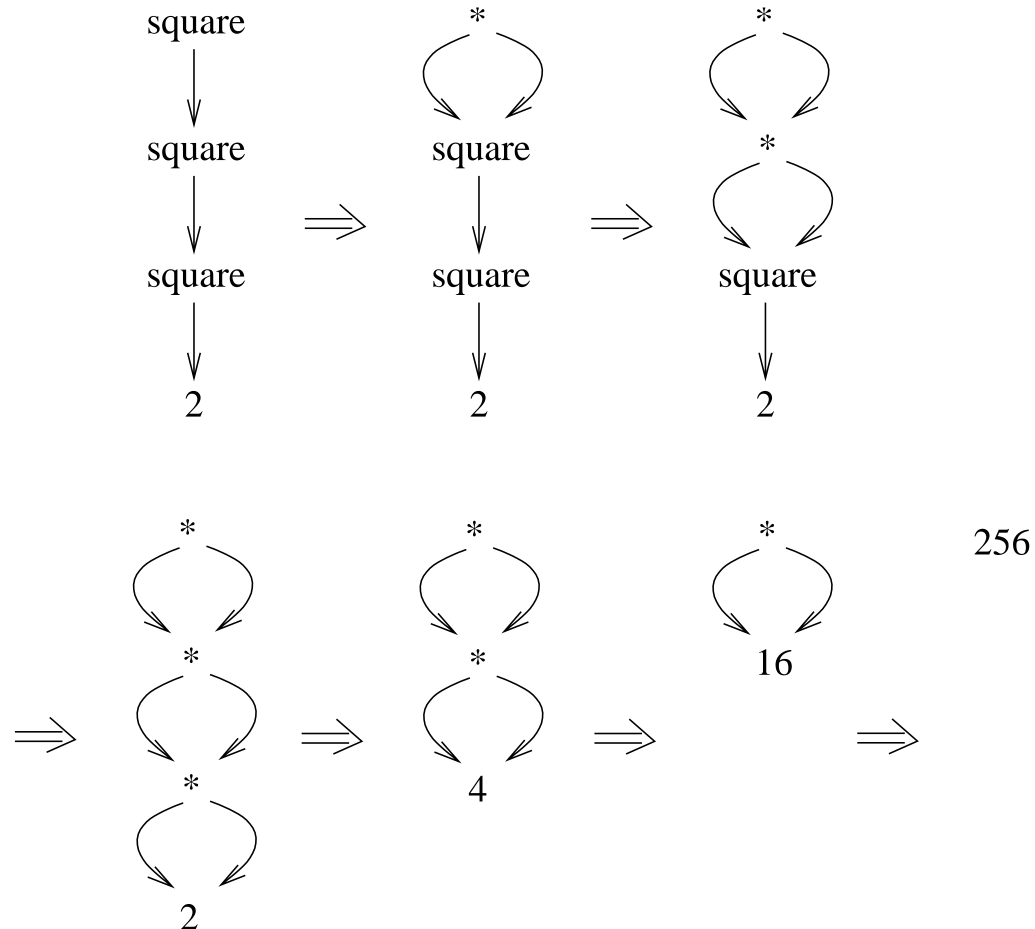
- *Problem*: Mehrfachauswertung von Ausdrücken bei *normal order Evaluation*

- *Ziel*: Vermeidung von Mehrfachauswertungen zur Effizienzsteigerung

- *Lösung*: *Lazy Evaluation*!

  ...Ausdrucksdarstellung und -auswertung basierend auf Graphen und Graphtransformationen.

Lazy evaluation (call-by-need)...



...6 Schritte, aber Graphtransformationen!

Lazy evaluation (call-by-need)...

- ...beruht (implementierungstechnisch) auf Graphtransformationen
- ...garantiert, dass Argumente höchstens einmal (möglicherweise also gar nicht) ausgewertet werden

Insgesamt:

$\leadsto$ ...effiziente Realisierung der normal order Strategie!

Zentrale Ergebnisse

Theorem

- ...alle terminierenden Auswertungsreihenfolgen enden mit demselben Ergebnis
  $\leadsto$ Konfluenz- oder Diamanteigenschaft

- ...wenn es eine terminierende Auswertungsreihenfolge gibt, so terminiert auch die normal order Auswertungsreihenfolge

...Church/Rosser (1936)

Insbesondere:
Lazy evaluation "vergleichbar effizient" wie applicative (eager) order, falls alle Argumente benötigt werden.

Frei nach Shakespeare

Eager or lazy evaluation: that is the question.

Quot capita, tot sensa.

Oder: Die Antworten sind verschieden:

- eager evaluation (z.B. ML, Scheme (abgesehen von Makros),...)
- lazy evaluation (z.B. Haskell, Miranda,...)

Lazy vs. Eager: Eine Abwägung (1)

Lazy Evaluation

- Vorteile
  - terminiert mit Normalform, wenn es eine terminierende Auswertungsreihenfolge gibt
  - wertet Argumente nur aus, wenn nötig
  - elegante Behandlung potentiell unendlicher Datenstrukturen
- Nachteile
  - konzeptuell und implementierungstechnisch anspruchsvoller
    * partielle Auswertung von Ausdrücken (Seiteneffekte! Beachte: Letztere nicht in Haskell! In Scheme: Verantwortung beim Programmierer.)
    * Graphtransformationen
    * Ein-/Ausgabe
  - Volles Verständnis: Domain-Theorie und λ-Kalkül im Detail

Lazy vs. Eager: Eine Abwägung (2)

Eager Evaluation

- Vorteile
  - Konzeptuell und implementierungstechnisch einfacher
  - Vom mathematischen Standpunkt oft “natürlicher” (Beispiel: `first (42,infiniteInc)`)
  - (Einfache(re) Integration imperativer Konzepte)

Mithin: eager oder lazy – eine Frage des Anwendungsprofils!

Bemerkung

Wäre ein Haskell-Compiler (Interpretierer) korrekt, der die Fakultätsfunktion applikativ auswertet?

Ja, weil die Funktion `fac` *strikt* in ihrem Argument ist...

$\leadsto$ eager evaluation oder auch strict evaluation!

Kapitel 8: λ-Kalkül

...formale Fundierung funktionaler Programmiersprachen

Intuitive vs. formale Berechenbarkeit

Ausgangspunkt:

Intuitiv berechenbar ...“wenn es eine *irgendwie machbare effektive mechanische Methode* gibt, die zu jedem Argument aus dem Definitionsbereich nach endlich vielen Schritten den Funktionswert konstruiert und die für alle anderen Argumente entweder mit einem speziellen Fehlerwert oder nie abbricht”.

Zentrale Frage...

Lässt sich der Begriff “intuitiver Berechenbarkeit” formal fassen?

Zur Beantwortung nötig...

Formale Berechnungsmodelle!

...d.h. *Explikationen* des Begriffs “intuitiver Berechenbarkeit”.

Der λ-Kalkül (1)

- ...ein spezielles formales Berechnungsmodell, wie viele andere auch, z.B.
  - allgemein rekursive Funktionen (Herbrand 1931, Gödel 1934, Kleene 1936)
  - Turing-Maschinen (Turing 1936)
  - μ-rekursive Funktionen (Kleene 1936)
  - Markov-Algorithmen (Markov 1951)
  - ...
- ...geht zurück auf Alonzo Church (1936)
- ...Berechnungen über Paaren, Listen, Bäumen, auch unendlichen, Funktionen höherer Ordnung einfach ausdrückbar
- ...in diesem Sinne "praxisnäher/realistischer" als andere formale Berechnungsmodelle

Der λ-Kalkül (2)

Church'sche These
Eine Funktion ist genau dann intuitiv berechenbar, wenn sie λ-definierbar ist (d.h. im λ-Kalkül ausdrückbar ist).

Beweis? ...schlechterdings unmöglich!

Aber...

Der λ-Kalkül (3)

Man hat bewiesen:

- Alle der obigen Berechnungsmodelle sind gleich mächtig.

Das kann als Hinweis darauf verstanden werden, dass alle der obigen Berechnungsmodelle den Begriff wahrscheinlich "gut" charakterisieren!

Aber: es schließt nicht aus, dass morgen ein mächtigeres formales Berechnungsmodell gefunden wird, das dann den Begriff der intuitiven Berechenbarkeit "besser" charakterisierte.

Präzedenzfall: Primitiv rekursive Funktionen

- ...bis Ende der 20er-Jahre als adäquate Charakterisierung intuitiver Berechenbarkeit akzeptiert
- ...tatsächlich jedoch: echt schwächeres Berechnungsmodell
- ...Beweis: Ackermann-Funktion ist berechenbar, aber nicht primitiv rekursiv (Ackermann 1928)

Die Ackermann-Funktion

... "berühmtberüchtigtes" Beispiel einer

- zweifellos (intuitiv) berechenbaren, aber nicht primitiv rekursiven Funktion!

```
ack :: (Integer,Integer) -> Integer
ack (m,n)
  | m == 0                 = n+1
  | (m > 0) && (n == 0)   = ack (m-1,1)
  | (m > 0) && (n /= 0)   = ack (m-1,ack(m,n-1))
```

⇝ ...hier in Haskell-Notation!

Der λ-Kalkül (4)

...ausgezeichnet durch

- *Einfachheit*
  ...nur wenige syntaktische Konstrukte, einfache Semantik

- *Ausdruckskraft*
  ...Turing-mächtig, alle "intuitiv berechenbaren" Funktionen im λ-Kalkül ausdrückbar

Darüberhinaus...

$\leadsto$ Bindeglied zwischen funktionalen Hochsprachen und ihren maschinennahen Implementierungen.

Wir unterscheiden...

- *Reiner* λ-Kalkül
  ...reduziert auf das "absolut Notwendige"
  $\leadsto$ besonders bedeutsam in Untersuchungen zur Theorie der Berechenbarkeit

- *Angewandte* λ-Kalküle
  ...syntaktisch angereichert, praxisnäher

Reiner λ-Kalkül: Syntax

Die Menge *Exp* der Ausdrücke des (reinen) λ-Kalküls, kurz λ-Ausdrücke, ist definiert durch:

- Jeder *Name* (*Identifier*) ist in *Exp*.
  (Bsp: $a, b, c, \ldots, x, y, z, \ldots$)

- *Abstraktion*: Wenn x ein Name und e aus *Exp* ist, dann ist auch (λx. e) in *Exp*. *Sprechweise*: Funktionsabstraktion mit formalem Parameter x und Rumpf e.
  (Bsp.: (λx.(x x)), (λx.(λy.(λz.(x (y z))))), ...)

- *Applikation*: Wenn f und e in *Exp* sind, dann ist auch (f e) in *Exp*; *Sprechweisen*: Anwendung von f auf e. f heißt auch *Rator*, e auch *Rand*.
  (Bsp.: ((λx.(x x)) y), ...)

Alternativ

...die Syntax in (modifizierter) Backus-Naur-Form (BNF):

$e ::=$		(λ-Ausdrücke)
$::=$	x	(Namen (Identifikatoren))
$::=$	$\lambda\, x.e$	(Abstraktion)
$::=$	$e\ e$	(Applikation)
$::=$	(e)	

Konventionen

- Überflüssige Klammern können weggelassen werden.
  Dabei gilt:
  - *Rechts*assoziativität für λ-Sequenzen in Abstraktionen
    Bsp.: – λx.λy.λz.(x (y z)) kurz für (λx.(λy.(λz.(x (y z))))),
    – λx. e kurz für (λx. e)
  - *Links*assoziativität für Applikationssequenzen
    Bsp.: – $e_1\ e_2\ e_3 \ldots e_n$ kurz für $(\ldots((e_1\ e_2)\ e_3) \ldots e_n)$,
    – $(e_1\ e_2)$ kurz für $e_1\ e_2$
- Der Rumpf einer λ-Abstraktion ist der längstmögliche dem Punkt folgende λ-Ausdruck
  Bsp.: – λx.e f entspricht λx.(e f), nicht (λx. e) f

Angewandte λ-Kalküle

Angewandte λ-Kalküle sind syntaktisch angereichert.

Beispielsweise:

- Auch Konstanten, Funktionsnamen oder "übliche" Operatoren können Namen (im weiteren Sinn) sein
(Bsp: 1, 3.14, $true$, $false$, $+$, $*$, $-$, fac, simple, ...)
- Ausdrücke können...
  - komplexer sein
  (Bsp.: if e then e_1 else e_2 fi ... statt cond e e_1 e_2 für cond geeignete Funktion)
  - getypt sein
  (Bsp.: 1 : IN, $true$: $Boole$, ...)
- ...

λ-Ausdrücke sind dann beispielsweise auch...

- Applikationen: fac 3, simple x y z (entspricht ((simple x) y) z), ...)
- Abstraktionen: λx.(x + x), λx.λy.λz.(x * (y - z)), 2 + 3,
(λx.if odd x then x*2 else x div 2 fi) 42, ...)

In der Folge

...erlauben wir uns die Annehmlichkeit, Ausdrücke, für die wir eine eingeführte Schreibweise haben (z.B. $n * fac\ (n-1)$), in dieser gewohnten Weise zu schreiben, auch wenn wir die folgenden Ergebnisse für den reinen λ-Kalkül formulieren.

Rechtfertigung:

- Resultate der theoretischen Informatik, insbesondere

  Alonzo Church. *The Calculi of Lambda-Conversion*. Annals of Mathematical Studies, Vol. 6, Princeton University Press, 1941

  ...zur Modellierung von ganzen Zahlen, Wahrheitswerten, etc. durch geeignete Ausdrücke des reinen λ-Kalküls

Freie und gebundene Variablen (1)

...in λ-Ausdrücken

Die Menge der *frei* vorkommenden Variablen...

$$free(\mathsf{x}) = \{\mathsf{x}\}, \text{ wenn x ein Variablenname ist}$$

$$free(\lambda \mathsf{x}.\mathsf{e}) = free(\mathsf{e}) \setminus \{\mathsf{x}\}$$

$$free(\mathsf{f}\ \mathsf{e}) = free(\mathsf{f}) \cup free(\mathsf{e})$$

Umgekehrt:

Die Menge der *gebunden* vorkommenden Variablen...

$$bound(\lambda \mathsf{x}.\mathsf{e}) = bound(\mathsf{e}) \cup \{\mathsf{x}\}$$

$$bound(\mathsf{f}\ \mathsf{e}) = bound(\mathsf{f}) \cup bound(\mathsf{e})$$

Beachte: gebunden $\neq$ nicht frei !
...sonst wäre etwa "x gebunden in y"

Freie und gebundene Variablen (2)

Betrachte: (λx. (x y)) x

- Gesamtausdruck
  - x kommt frei und gebunden in (λx. (x y)) x vor
  - y kommt frei in (λx. (x y)) x vor
- Teilausdrücke
  - x kommt gebunden in (λx. (x y)) vor und frei in (x y) und in x
  - y kommt frei in (λx. (x y)), (x y) und y vor

Gebunden vs. gebunden an...

Wir müssen unterscheiden:

- Eine *Variable* ist *gebunden*
- Ein *Variablenvorkommen* ist *gebunden an*

Gebunden und *gebunden an* ...unterschiedliche Konzepte!

Letzteres meint:

- Ein (definierendes oder angewandtes) Variablenvorkommen ist an ein definierendes Variablenvorkommen gebunden

Definition

- *Definierendes* V.vorkommen ...Vorkommen unmittelbar nach einem λ
- *Angewandtes* V.vorkommen ...jedes nicht definierende Vorkommen

Der λ-Kalkül: Vorwärts zur Semantik

Zentral sind folgende Begriffe:

- (Syntaktische) Substitution
- Konversionsregeln / Reduktionsregeln

Syntaktische Substitution

Syntaktische) *Substitution*, ein zentraler Begriff:

$$x[e/x] = e, \text{ wenn x ein Name ist}$$

$$y[e/x] = y, \text{ wenn y ein Name mit } x \neq y \text{ ist}$$

$$(f\ g)[e/x] = (f[e/x])\ (g[e/x])$$

$$(\lambda x.f)[e/x] = \lambda x.f$$

$$(\lambda y.f)[e/x] = \lambda y.(f[e/x]), \text{ wenn } x \neq y \text{ und } y \notin free(e)$$

$$(\lambda y.f)[e/x] = \lambda z.((f[z/y])[e/x]), \text{ wenn } x \neq y \text{ und } y \in free(e), \text{ wobei } x \neq z \text{ und } z \notin free(e) \cup free(f)$$

(Syntaktische) Substitution

Einige Beispiele zur Illustration:

- ((x y) (y z)) [a+b/y] = ((x (a+b)) ((a+b) z))
- λx. (x y) [a+b/y] = λx. (x (a+b))
- Aber: λx. (x y) [a+b/x] = λx. (x y)
- Achtung: λx. (x y) [x+b/y] $\leadsto$ λx. (x (x+b))
  ...ohne Umbenennung *Bindungsfehler*!

  Deshalb: λx. (x y) [x+b/y] = λz. ((x y)[z/x]) [x+b/y]
  = λz. (z y) [x+b/y]
  = λz. (z (x+b))
  ...dank Umbenennung kein Bindungsfehler!

Konversionsregeln

Ein zweiter zentraler Begriff: λ-*Konversionen*

- α-Konversion (Umbenennung formaler Parameter)

  λx.e $\Leftrightarrow$ λy.e[y/x], wobei y $\notin$ $free$(e)

- β-Konversion (Funktionsanwendung)

  (λx.f) e $\Leftrightarrow$ f[e/x]

- η-Konversion (Elimination redundanter Funktion)

  λx.(e x) $\Leftrightarrow$ e, wobei x $\notin$ $free$(e)

$\leadsto$ führen auf eine operationelle Semantik des λ-Kalküls.

Sprechweisen

...im Zusammenhang mit Konversionsregeln

- Von links nach rechts gerichtete Anwendungen der β- und η-Konversion heißen β- und η-Reduktion.
- Von rechts nach links gerichtete Anwendungen der β-Konversion heißen β-Abstraktion.

Intuition hinter den Konversionsregeln

Noch einmal zusammengefasst:

- α-Konversion: ...erlaubt die konsistente Umbenennung formaler Parameter von λ-Abstraktionen
- β-Konversion: ...erlaubt die Anwendung einer λ-Abstraktion auf ein Argument
(Achtung: Gefahr von Bindungsfehlern! Abhilfe: α-Konversion!)
- η-Konversion: ...erlaubt die Elimination redundanter λ-Abstraktionen

Bsp.: (λx.λy.x+y) (y$*$2) $\Rightarrow$ λy.(y$*$2)+y)
$\leadsto$ Bindungsfehler ("y wird eingefangen")

Beispiel für λ-Reduktion

$$
\begin{array}{ll}
(\lambda x.\lambda y.x*y)\ ((\lambda x.\lambda y.x+y)\ 9\ 5)\ 3 & (\beta\text{-Reduktion}) \\
\quad \Rightarrow\ (\lambda x.\lambda y.x*y)\ ((\lambda y.9+y)\ 5)\ 3 & (\beta\text{-Reduktion}) \\
\quad \Rightarrow\ (\lambda y.((\lambda y.9+y)\ 5)*y)\ 3 & (\beta\text{-Reduktion}) \\
\quad \Rightarrow\ (\lambda y.(9+5)*y)\ 3 & (\beta\text{-Reduktion}) \\
\quad \Rightarrow\ (9+5)*3 & (\beta\text{- und } \eta\text{-Reduktion nicht anwendbar})
\end{array}
$$

$\leadsto$ Weitere Regeln zur Reduktion primitiver Operationen in erweitertem λ-Kalkül (Auswertung arithmetischer Ausdrücke, bedingte Anweisungen, Listenoperationen, ...), sog. δ-Regeln.

$\leadsto$ solche Erweiterungen sind praktisch notwendig und einsichtig, aber für die Theorie (der Berechenkarkeit) kaum relevant.

Reduktionsfolgen & Normalformen (1)

- Ein λ-Ausdruck ist in *Normalform*, wenn er durch β-Reduktion und η-Reduktion nicht weiter reduzierbar ist.
- (Praktisch relevante) Reduktionsstrategien
  - normal order (leftmost-outermost)
  - applicative order (leftmost-innermost)

Reduktionsfolgen & Normalformen (2)

- *Beachte*: Nicht jeder λ-Ausdruck ist zu einem λ-Ausdruck in Normalform konvertierbar (Endlosrekursion).
  Bsp.: (1) $\lambda x.(x\ x)\ \lambda x.(x\ x) \;\Rightarrow\; \lambda x.(x\ x)\ \lambda x.(x\ x) \;\Rightarrow\; \ldots$
  (2) $(\lambda x.42)\ (\lambda x.(x\ x)\ \lambda x.(x\ x))$ (hat Normalform!)

Zentrale Resultate:

- Wenn ein λ-Ausdruck zu einem λ-Ausdruck in Normalform konvertierbar ist, dann führt jede terminierende Reduktion des λ-Ausdrucks zum (bis auf α-Konversion) selben λ-Ausdruck in Normalform (bis auf α-Konversion).

- Durch Reduktionen im λ-Kalkül sind genau jene Funktionen berechenbar, die Turing-, Markov-, ... berechenbar sind!

Church-Rosser-Theoreme

Seien e_1 und e_2 zwei λ-Ausdrücke...

Theorem 1
Wenn $e_1 \Leftrightarrow e_2$, dann gibt es einen λ-Ausdruck e mit $e_1 \Rightarrow^* e$ und $e_2 \Rightarrow^* e$
(sog. *Konfluenzeigenschaft*, *Diamanteigenschaft*)

Informell: ...wenn eine Normalform ex., dann ist sie eindeutig (bis auf α-Konversion)!

Theorem 2
Wenn $e_1 \Rightarrow^* e_2$ und e_2 in Normalform, dann gibt es eine normal order Reduktionsfolge von e_1 nach e_2
(sog. *Standardisierungstheorem*)

Informell: ...normal order Reduktion terminiert am häufigsten!

Semantik von λ-Ausdrücken

- λ-Ausdrücke in Normalform lassen sich (abgesehen von α-Konversionen) nicht weiter vereinfachen (reduzieren)
- Nach dem 1. Church-Rosser-Theorem ist die Normalform eines λ-Ausdrucks eindeutig bestimmt, wenn sie existiert (wieder abgesehen von α-Konversionen)

Das legt folgende Festlegung nahe:

- Besitzt ein λ-Ausdruck eine Normalform, so ist dies sein Wert.
- *Umgekehrt*: Die *Semantik* (*Bedeutung*) eines λ-Ausdrucks ist seine Normalform, wenn sie existiert, ansonsten ist sie undefiniert.

Rekursion. Und wie sie behandelt wird...

Erinnerung...

$$\text{fac n} = \text{if n} == 0 \text{ then } 1 \text{ else } (\text{n} * \text{fac } (\text{n} - 1))$$

oder alternativ:

$$\text{fac} = \lambda\text{n.if n} == 0 \text{ then } 1 \text{ else } (\text{n} * \text{fac } (\text{n} - 1))$$

"**Problem**"...

λ-Abstraktionen sind *anonym*.

Lösung: Der Y-Kombinator

Y-Kombinator: Y = λf.(λx.(f (x x)) λx.(f (x x)))

Es gilt: Für jeden λ-Ausdruck e ist (Y e) zu (e (Y e)) konvertierbar:

$$
\begin{array}{lll}
\mathsf{Y\ e} & \Rightarrow & \lambda \mathsf{x.(e\ (x\ x))}\ \lambda \mathsf{x.(e\ (x\ x))} \\
 & \Rightarrow & \mathsf{e}\ (\lambda \mathsf{x.(e\ (x\ x))}\ \lambda \mathsf{x.(e\ (x\ x))}) \\
 & \Leftrightarrow & \mathsf{e\ (Y\ e)}
\end{array}
$$

Mithilfe des Y-Kombinators lässt sich Rekursion realisieren.
Intuition:

$$
\begin{array}{l}
\quad \mathsf{f} = \cdots \mathsf{f} \cdots \text{ (rekursive Darstellung)} \\
\leadsto \mathsf{f} = \lambda \mathsf{f}.(\cdots \mathsf{f} \cdots)\ \mathsf{f} \ \ (\lambda\text{-Abstraktion}) \\
\leadsto \mathsf{f} = \mathsf{Y}\ \lambda \mathsf{f}.(\cdots \mathsf{f} \cdots) \text{ (nicht-rekursive Darstellung)}
\end{array}
$$

Bemerkung:
λ-Terme ohne freie Variablen heißen *Kombinatoren*.

Zur Übung empfohlen

Die Anwendung des Y-Kombinators anhand von Beispielen:

Betrachte:

$$\text{fac} = \text{Y}\ \lambda \text{f}.(\lambda \text{n}.\text{if n} == 0 \text{ then } 1 \text{ else n} * \text{f (n} - 1))$$

Rechne nach:

$$\text{fac } 1 \Rightarrow \ \ldots \ \Rightarrow 1$$

Überprüfe dabei:

Der Y-Kombinator realisiert Rekursion
durch wiederholtes Kopieren

Praktisch relevant: Typisierte λ-Kalküle

Jedem λ-Ausdruck ist ein Typ zugeordnet

Beispiele:

$$\begin{aligned} 3 &:: \text{Integer} \\ (*) &:: \text{Integer} -> \text{Integer} -> \text{Integer} \\ (\lambda x.2 * x) &:: \text{Integer} -> \text{Integer} \\ (\lambda x.2 * x)\ 3 &:: \text{Integer} \end{aligned}$$

Einschränkung: Typen müssen konsistent sein (wohltypisiert)

Problem jetzt: Selbstanwendung im Y-Kombinator

$\leadsto$ Y nicht endlich typisierbar!

Abhilfe: explizite Rekursion zum Kalkül hinzufügen mittels Hinzunahme der Reduktionsregel Y e $\Rightarrow$ e (Y e))

...nebenbei: zweckmäßig auch aus Effizienzgründen!

Zurück zu Haskell...

- Haskell beruht auf typisiertem λ-Kalkül
- Übersetzer/Interpretierer überprüft, ob alle Typen konsistent sind
- Programmierer kann Typdeklarationen angeben (Sicherheit), muss aber nicht (bequem, manchmal unerwartete Ergebnisse)
- fehlende Typinformation wird vom Übersetzer inferiert (berechnet)
- Rekursive Funktionen direkt verwendbar
  (daher in Haskell kein Y-Kombinator notwendig)

Ergänzende und weiterführende Literaturhinweise

- A. Church. *The Calculi of Lambda-Conversion*. Annals of Mathematical Studies, Vol. 6, Princeton University Press, 1941.

- H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. (Revised Edn.), North Holland, 1984.

Kapitel 9: 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...

```
                ----------------
Eingabe ---> | Fkt. Programm | --> Ausgabe
                ----------------
```

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:

- `(>>)` `(oft gelesen als ‘sequence’)`: 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.

- `(>>=)` `(oft gelesen als ‘then’ oder ‘bind’)`: 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

...in Kapitel 12!

Kapitel 10: Programmieren im Großen, Module, abstrakte Datentypen, reflektive Programmierung

Programmieren im Großen

Das Modulkonzept von Haskell...

Modularisierung im allgemeinen (1)

Intuitiv:

- Zerlegung großer Programm(systeme) in kleinere Einheiten, genannt *Module*

Ziel:

- Sinnvolle, über- und durchschaubare Organisation des Gesamtsystems

Modularisierung im allgemeinen (2)

Vorteile:

- *Arbeitsphysiologisch* ...Unterstützung arbeitsteiliger Programmierung
- *Softwaretechnisch* ...Unterstützung der Wiederbenutzung von Programmen und Programmteilen
- *Implementierungstechnisch* ...Unterstützung von "separate compilation"

Insgesamt:

- Höhere Effizienz der Softwareerstellung bei gleichzeitiger Steigerung der Qualität (Verlässlichkeit) und reduzierten Kosten

Modularisierung im allgemeinen (3)

Anforderungen an das Modulkonzept zur Erreichung vorgenannter Ziele:

- Unterstützung des Geheimnisprinzips
  ...durch Trennung von
  - Schnittstelle (Import/Export)
    ...*wie interagiert das Modul mit seiner Umgebung? Welche Funktionalität stellt es zur Verfügung* (*Export*), *welche Funktionalität erwartet es* (*Import*)*?*
  - Implementierung (Daten/Funktionen)
    ...*wie ist die Funktionalität des Moduls realisiert?*

  in einem Modul.

Module in Haskell – Allgemeiner Aufbau

```
module MyModule where

-- Daten- und Typdefinitionen
data D1 ... = ...
...
data Dn ... = ...

type T1 = ...
...
type Tn = ...

-- Funktionsdefinitionen
f1 :: ...
f1 ... = ...
...
fn :: ...
fn ... = ...
```

Das Modulkonzept von Haskell

...unterstützt:

- Export
  - Selektiv/Nicht selektiv
- Import
  - Selektiv/Nicht selektiv
  - Qualifiziert
  - Mit Umbenennung

...unterstützt nicht:

- automatischer Reexport!

Import: Nicht selektiv

```
module M1 where
...

module M2 where
import M1  -- Nicht selektiver Import:
           -- Alle im Modul M1 (sichtbaren) Bezeichner/
           -- Definitionen werden importiert und koennen
           -- in M2 benutzt werden.
```

Import: Selektiv

```
module M1 where
...                                -- wie auf voriger Folie

module M2 where
import M1 (D1, D2 (..), T1, f5)  -- Selektiver Import:
                                 -- Lediglich D1 (ohne Konstruktoren),
                                 -- D2 (einschliesslich Konstruktoren,
                                 -- T1 und f5 werden importiert und
                                 -- koennen in M2 benutzt werden.

module M3 where
import M1 hiding (D1, T2, f1)    -- Selektiver Import:
                                 -- Alle (sichtbaren) Bezeichner/
                                 -- Definitionen mit Ausnahme der
                                 -- explizit genannten werden importiert
                                 -- und koennen in M3 benutzt werden.
```

Erinnerung (vgl. Folie 26)

“Verstecken” von in `Prelude.hs` vordefinierten Funktionen...

Ergänze...

```
import Prelude hiding (reverse,tail,zip)
```

...am Anfang des Haskell-Skripts im Anschluss an die Modul-Anweisung (so vorhanden), um `reverse`, `tail` und `zip` zu verbergen.

Export: Nicht selektiv

```
module M1 where       -- Nicht selektiver Export:
data D1 ... = ...     -- Alle im Modul M1 (sichtbaren)
...                   -- Bezeichner/Definitionen werden
data Dn ... = ...     -- exportiert und koennen von anderen
                      -- Modulen importiert werden.
type T1 = ...
...
type Tn = ...

f1 :: ...
f1 ... = ...
...
fn :: ...
fn ... = ......
```

Export: Selektiv

```
module M1 (D1, D2 (..), T1, f2, f5) where
data D1 ... = ...
...                     -- Selektiver Export:
data Dn ... = ...       -- Nur die explizit genannten Bezeichner/
                        -- Definitionen werden exportiert und
type T1 = ...           -- koennen von anderen Modulen importiert
...                     -- werden.
type Tn = ...           -- Unterstuetzt Geheimnisprinzip!


f1 :: ...
f1 ... = ...
...
fn :: ...
fn ... = ......
```

Kein automatischer Reexport (1)

```
module M1 where ...

module M2 where
import M1      -- Nicht selektiver Import:
               -- Alle im Modul M1 (sichtbaren) Bezeichner/
fM2...         -- Definitionen werden importiert und koennen
               -- in M2 benutzt werden.

module M3 where
import M2      -- Alle im Modul M2 (sichtbaren) Bezeichner/
               -- Definitionen werden importiert und koennen
               -- in M3  benutzt werden, nicht jedoch von
               -- M2 aus M1 importierte Namen. Mithin: Kein
               -- automatischer Reexport!
```

Kein automatischer Reexport (2)

Abhilfe: Expliziter Reexport!

```
module M4 (D1 (..), f1, f2) where   -- Selektiver
import M1                           -- Reexport

module M2 (M1,fM2) where   -- Nicht selektiver
import M1                  -- Reexport
```

Sonderfälle: Namenskollisionen, Lokale Namen

- Namenskollisionen
  - Abhilfe/Auflösen: Qualifizierter Import

    ```
    import qualified M1
    ```

- Umbenennen importierter Module
  - Lokale Namen importierter
    - Module

      ```
      import MyM1 as M1
      ```

    - Bezeichner

      ```
      import M1 (f1,f2) renaming (f1 to g1, f2 to g2)
      ```

Konventionen und gute Praxis

- Konventionen
  - Pro Datei ein Modul
  - Modul- und Dateiname stimmen überein (abgesehen von der Endung `.hs` bzw. `.lhs` im Dateinamen).
  - Alle Deklarationen beginnen in derselben Spalte wie `module`.
- Gute Praxis
  - Module unterstützen *eine* (*!*) klar abgegrenzte Aufgabenstellung (vollständig) und sind in diesem Sinne in sich abgeschlossen; ansonsten Teilen (Teilungskriterium)
  - Module sind "kurz" (ideal: 2 bis 3 Druckseiten; prinzipiell: "so lang wie nötig, so kurz wie möglich")

Das Hauptmodul

Module `main`...

- ...muss in jedem Modulsystem als "top-level" Modul vorkommen und eine Definition namens `main` festlegen.

  $\leadsto$ ...ist der in einem übersetzten System bei Ausführung des übersetzten Codes zur Auswertung kommende Ausdruck.

Regeln “guter” Modularisierung (1)

(siehe dazu auch M. Chakravarty, G. Keller. *Einführung in die Programmierung mit Haskell*. Kapitel 10, Pearson Studium, 2004.)

Aus Modulsicht:

Module sollen...

- einen klar definierten, auch unabhängig von anderen Modulen verständlichen Zweck besitzen
- nur einer Abstraktion entsprechen
- einfach zu testen sein

Regeln "guter" Modularisierung (2)

Aus Gesamtprogrammsicht:

Aus Modulen aufgebaute Programme sollen so entworfen sein, dass...

- Auswirkungen von Designentscheidungen (z.B. Einfachheit vs. Effizienz einer Implementierung) auf wenige Module beschränkt sind

- Abhängigkeiten von Hardware oder anderen Programmen auf wenige Module beschränkt sind

Regeln "guter" Modularisierung (3)

Aus intra- und intermodularer Sicht:

Zwei zentrale Konzepte in diesem Zusammenhang sind...

- Intramodular: *Kohäsion*
- Intermodular: *Kopplung*

Regeln “guter” Modularisierung (4)

Aus intramodularer Sicht:

Kohäsion

- Anzustreben sind...
  - *Funktionale Kohäsion* (d.h. Funktionen ähnlicher Funktionalität sollten in einem Modul zusammengefasst sein, z.B. Ein-/Ausgabefunktionen, trigonometrische Funktionen, etc.)
  - *Datenkohäsion* (d.h. Funktionen, die auf den gleichen Datenstrukturen arbeiten, sollten in einem Modul zusammengefasst sein, z.B. Baummanipulationsfunktionen, Listenverarbeitungsfunktionen, etc.)
- Zu vermeiden sind...
  - *Logische Kohäsion* (d.h. unterschiedliche Implementierungen der gleichen Funktionalität sollten in verschiedenen Modulen untergebracht sein, z.B. verschiedene Benutzerschnittstellen eines Systems)
  - *Zufällige Kohäsion* (d.h. Funktionen sind ohne sachlichen Grund, zufällig eben, in einem Modul zusammengefasst)

Regeln “guter” Modularisierung (5)

Aus intermodularer Sicht:

Kopplung

- beschäftigt sich mit dem Import-/Exportverhalten von Modulen
  - Anzustreben ist...
    - *Datenkopplung* (d.h. Funktionen unterschiedlicher Module kommunizieren nur durch Datenaustausch (in funktionalen Sprachen per se gegeben))

Regeln “guter” Modularisierung (6)

Kennzeichen “guter” Modularisierung:

- *Starke Kohäsion*
  d.h. enger Zusammenhang der Definitionen eines Moduls

- *Lockere Kopplung*
  d.h. wenige Abhängigkeiten zwischen verschiedenen Modulen, insbesondere weder direkte noch indirekte zirkuläre Abhängigkeiten.

Abstrakte Datentypen, kurz: ADTs (1)

Ziel...

- Kapselung von Daten, Realisierung des Geheimnisprinzips auf Datenebene (engl. *information hiding*)

Implementierungstechnisch zentral...

- Haskells Modulkonzept, speziell “selektiver Export”

Abstrakte Datentypen, kurz: ADTs (2)

Hinweise auf drei klassische Literaturstellen:

- John V. Guttag. *Abstract Data Types and the Development of Data Structures*. Communications of the ACM, Vol. 20, No. 6, 396-404, 1977.
- John V. Guttag, J. J. Horning. *The Algebra Specification of Abstract Data Types*. Acta Informatica, Vol. 10, No. 1, 27-52, 1978.
- John V. Guttag, Ellis Horowitz, David R. Musser. *Abstract Data Types and Software Validation*. Communications of the ACM, Vol. 21, No. 12, 1048-1064, 1978.

Abstrakte Datentypen, kurz: ADTs (3)

Die grundlegende Idee am Beispiel des Typs *Schlange*:

```
Schnittstellen/Signatur:
NEW:                -> Queue
ADD:       Queue x Item -> Queue
FRONT:     Queue -> Item
REMOVE:    Queue -> Queue
IS_EMPTY: Queue -> Boolean

Axiome/Gesetze:
(1) IS_EMPTY(NEW) = true
(2) IS_EMPTY(ADD(q,i)) = false
(3) FRONT(NEW) = error
(4) FRONT(ADD(q,i)) = if IS_EMPTY(q) then i else FRONT(q)
(5) REMOVE(NEW) = error
(6) REMOVE(ADD(q,i)) = if IS_EMPTY(q) then NEW
                                       else ADD(REMOVE(q),i)
```

Bsp.: (Warte-) Schlangen als ADT (1)

Warteschlange...

...eine FIFO (first-in/first-out) Datenstruktur

```
module Queue ( Queue,       --  Kein Konstruktorexport!!!
               emptyQ,      --  Queue a
               isEmptyQ,    --  Queue a -> Bool
               joinQ,       --  a -> Queue a -> Queue a
               leaveQ,      --  Queue a -> (a, Queue a)
             ) where

...  -- Fortsetzung siehe naechste Folie
```

Bsp.: (Warte-) Schlangen als ADT (2)

```
... -- Fortsetzung der vorherigen Folie

data Queue = Qu [a]

emptyQ = Qu []

isEmptyQ (Qu []) = True
isEmptyQ _       = False

joinQ x (Qu xs) = Qu (xs++[x])

leaveQ q@(Qu xs)
  | not (isEmptyQ q)  =  (head xs, Qu (tail xs))
  | otherwise         =  error "Niemand wartet!"
```

Bsp.: (Warte-) Schlangen als ADT (3)

Notationelle Spielart des Mustervergleichs...

```
leaveQ q@(Qu xs)   --  argument as "q or as (Qu xs)"
```

Mittels...

- `q` Zugriff auf das Argument als Ganzes
- `(Qu xs)` Zugriff auf/über die Struktur des Arguments

Bsp.: (Warte-) Schlangen als ADT (2)

Programmiertechn. Vorteile aus der Benutzung von ADTs...

- *Geheimnisprinzip*: Nur die Schnittstelle ist bekannt, die Implementierung bleibt verborgen
  - Schutz der Datenstruktur vor unkontrolliertem oder nicht beabsichtigtem/zugelassenem Zugriff
  - Einfache Austauschbarkeit der zugrundeliegenden Implementierung
  - Arbeitsteilige Programmierung

Beispiel: `emptyQ == Qu []`

...führt in `Queue` importierenden Modulen zu einem Laufzeitfehler! (Die Implementierung und somit der Konstruktor `Qu` sind dort nicht sichtbar.)

Resümee algebraische vs. abstrakte Datentypen

- Algebraische Datentypen
  - ...werden durch die Angabe ihrer Elemente spezifiziert, aus denen sie bestehen.
- Abstrakte Datentypen
  - ...werden durch ihr Verhalten spezifiziert, d.h. durch die Menge der Operationen, die darauf arbeiten.

Kapitel 11: Programmierprinzipien

- Funktionen höherer Ordnung und Lazy Evaluation
  - Unendliche Listen: Programmieren mit Strömen
    - Generator/Selektor-, Generator/Filter-Prinzip
  - Teile und herrsche

  ...ermöglichen neue Modularisierungsstrategien

- Reflektive Programmierung

Streams

Jargon

Strom ...synonym für *unendliche Liste* (*lazy list*)

Ströme...

- (in Kombination mit lazy evaluation) erlauben viele Probleme elegant, knapp und effizient zu lösen

Das Sieb des Eratosthenes 1(3)

Intuition

1. Schreibe die natürlichen Zahlen beginnend mit 2 hintereinander auf.

2. Die kleinste noch nicht gestrichene Zahl ist eine Primzahl. Streiche alle Vielfachen dieser Zahl

3. Wiederhole Schritt 2 mit der kleinsten noch nicht gestrichenen Zahl.

Illustration

```
Schritt 1:      2  3  4  5  6  7  8  9  10  11  12  13...
Schritt 2:      2  3     5     7     9      11      13...
 ("mit 2")
Schritt 2:      2  3     5     7            11      13...
 ("mit 3")
...
```

Das Sieb des Eratosthenes 2(3)

Die Folge der Primzahlen...

```
primes :: [Int]
primes = sieve [2 ..]

sieve :: [Int] -> [Int]
sieve (x:xs) = x : sieve [ y | y <- xs, mod y x > 0 ]
```

Das Sieb des Eratosthenes 3(3)

Illustration ...durch händische Auswertung

```
primes
  => sieve [2 ..]
  => 2 : sieve [ y | y <- [3 ..], mod y 2 > 0 ]
  => 2 : sieve (3 : [ y | y <- [4 ..], mod y 2 > 0 ]
  => 2 : 3 : sieve [ z | z <- [ y | y <- [4 ..], mod y 2 > 0 ],
                                             mod z 3 > 0]
  => ...
  => 2 : 3 : sieve [ z | z <- [5, 7, 9 ..], mod z 3 > 0 ]
  => ...
  => 2 : 3 : sieve [5, 7, 11,...]
  => ...
```

Generator/Selektor-, Generator/Filter-Prinzip

Anwendung:

- *Generator/Selektor-Prinzip*: Die ersten 5 Primzahlen

```
take 5 primes
```

- *Generator/Selektor-Prinzip*: Alle Primzahlen zwischen 50 und 100

```
[ n | n <- primes, n > 49, n < 101 ]
```

- *Generator/Filter-Prinzip*: Alle Primzahlen mit mindestens drei Einsen in der Dezimaldarstellung

```
[ n | n <- primes, ... ]
```

Teile und Herrsche

```
divAndConquer :: (p -> Bool) -> (p -> s) -> (p -> [p]) -> (p -> [s] -> s) -> p -> s
divAndConquer ind solve divide combine initProblem
    = dac initProblem
    where dac problem
              | ind problem = solve problem
              | otherwise   = combine problem (map dac (divide problem))
```

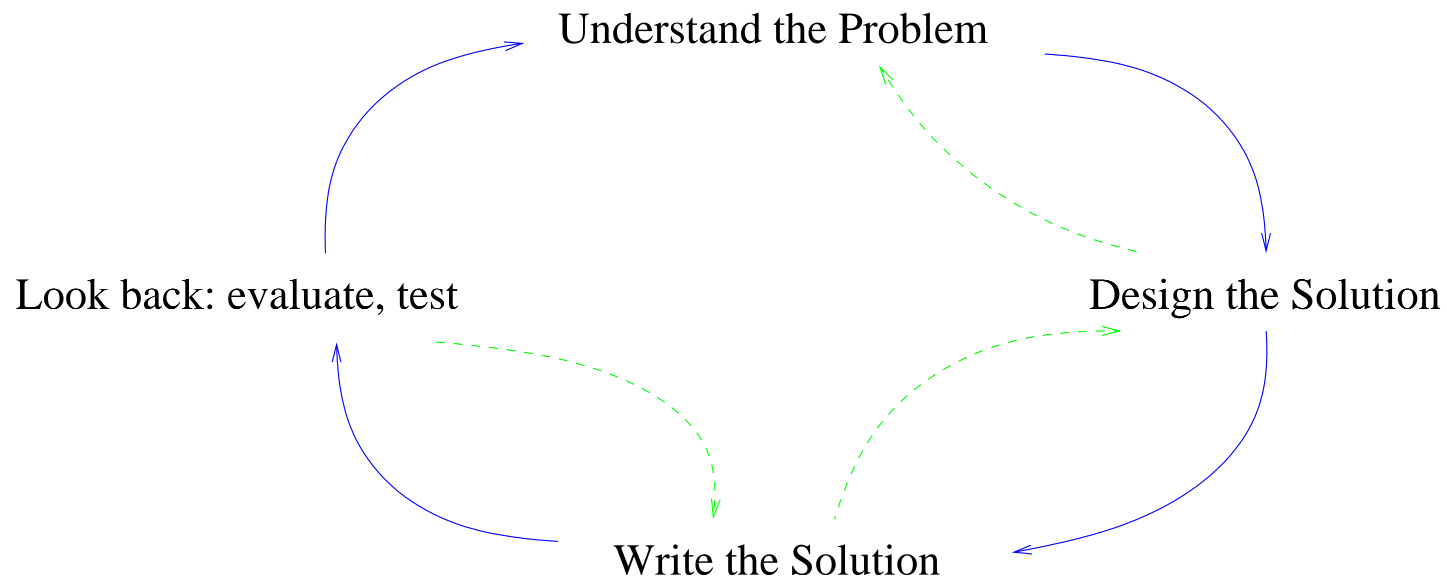
Anwendungen:

- Quicksort
- Binomialkoeffizienten
- ...

Bemerkung: Nicht immer ist ein Problem, dass sich auf "teile und herrsche" zurückführen lässt, auch wirklich sinnvoll dafür geeignet. Siehe dazu auch Aufgabenblatt 9!

Reflektives Programmieren

Der Entwicklungszyklus nach S. Thompson, Kap. 11 [3] ...



In jeder dieser Phasen ist es hilfreich, (sich) Fragen zu stellen!

Für eine beispielhafte Auswahl siehe z.B. S. Thompson, Kap. 11 [3]!

Kapitel 12: Monaden

Hintergrund und Grundlagen...

- Programmierung mit Monaden
  - Zusammenhang Monaden und Ein-/Ausgabe in Haskell

Ein-/Ausgabe in Haskell und Monaden

Ein-/Ausgabe in Haskell...

- realisiert als Spezialfall eines allgemeineren Konzepts, des Konzepts der *Monade*.

In der Folge deshalb: Haskells Monadenkonzept in größerem Detail

Monaden und Monadischer Programmierstil (1)

Monaden...

- erlauben die Reihenfolge, in der Operationen ausgeführt werden, explizit festzulegen.

Beispiel:

```
a-b -- Keine Festlegung der Auswertungsreihenfolge;
    -- kritisch, falls z.B. Ein-/Ausgabe involviert ist.

do a <- getInt  -- Reihenfolge explizit festgelegt
   b <- getInt
   return (a-b)
```

Monaden (1)

...sind *Konstruktorklassen*, Familien von Typen `m a` über einem polymorphen Typkonstruktor `m` mit den Funktionen `(>>=)`, `return`, `(>>)` und `fail`.

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a

  m >> k = m >>= \_ -> k       -- vordefiniert
  fail s = error s             -- vordefiniert
```

...wobei die Implementierungen der Funktionen gewissen Anforderungen genügen müssen.

Monaden (2)

Zentral:

- Die Funktionen `(>>=)` und `return` (und das abgeleitete (und in der Anwendung oft bequemere) `do`-Konstrukt)

...für die die Konstruktorklasse `Monad m` keine Standardimplementierung vorsieht.

Monaden (3)

Anforderungen (Gesetze) an die Monadenoperationen:

```
return a >>= f               =  f a
c >>= return                 =  c

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

Intuitiv:

- `return` gibt den Wert zurück, ohne einen weiteren Effekt.
- durch >>= gegebene Sequenzierungen sind unabhängig von der Klammerung (assoziativ)

Konstruktorklassen vs. Typklassen

Im Grundsatz ähnliche Konzepte, wobei...

- Konstruktorklassen
  - ...haben Typkonstruktoren als Elemente
- Typklassen (`Eq a, Ord a, Num a,...`)
  - ...haben Typen als Elemente

- Typkonstruktoren sind...
  - ...Funktionen, die aus gegebenen Typen neue Typen erzeugen
    (Bsp.: Tupelkonstruktor ( ), Listenkonstruktor `[ ]`,
    Funktionskonstruktor ->, aber auch:
    Ein-/Ausgabe `IO`,...)

Der abgeleitete Operator `(>@>)`...

...ist folgendermaßen definiert:

```
(>@>) :: Monad m => (a -> m b) ->
                    (b -> m c) ->
                    (a -> m c)

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

Hinweis:

...`return` ist vom Typ `a -> m a`!

Damit...

```
(1) return >@> f = f
(2) f >@> return = f

(3) (f >@> g) >@> h = f >@> (g >@> h)
```

Intuitiv...

- (1)&(2): `return` ist Einselement von `(>@>)`
- (3): `(>@>)` ist assoziativ

Beachte:
Obige Eigenschaften gelten nicht a priori, sondern sind durch die Implementierung sicherzustellen!

Beispiele von Monaden (1)

Die Identitätsmonade (mehr dazu auf Folie 497, Kapitel 12):

...einfachste aller Monaden.

```
(>>=)  :: m a -> (a -> m b) -> m b
m >>= f = f m

return :: a -> m a
return = id
```

Erinnerung:

- `(>>)` und `fail` implizit durch die Standarddefinition festgelegt.

Bemerkung:

- In diesem Szenario...
  `(>@>)` wird Vorwärtskomposition von Funktionen, `(>.>)`.
  Beachte: `(>.>)` ist assoziativ mit Einselement `id`.

Beispiele von Monaden (2)

Die Listenmonade:

```
instance Monad [] where
  xs >>= f = concat (map f xs)
  return x = [x]
  fail s   = []
```

Beispiele von Monaden (3)

Die “Maybe”-Monade:

```
instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing  >>= k = Nothing
  return         = Just
  fail s         = Nothing
```

Beispiele von Monaden (4)

Die Ein-/Ausgabe-Monade:

```
instance Monad IO where
(>>=) :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a   -- Rueckgabewerterzeugung
                      -- ohne Ein-/Ausgabe(aktion)
```

Standardfunktionen über Monaden

Kombination von Monaden...

- ...zum Aufbau komplexerer Effekte

```
mapF :: Monad m => (a -> b) -> m a -> m b
mapF f m = do x <- m
              return (f x)

joinM :: Monad m => m (m a) -> m a
joinM m = do x <- m
             x
```

Bemerkung:

- Aus (1), (2) und (3) folgt:
  `(4) mapF (f.g) = mapF f . mapF g`

Ein-/Ausgabe und Monaden

Erinnerung:

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

wobei...

- `(>>=)`: 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.

- `return`: Rückgabewerterzeugung ohne Ein-/Ausgabe(aktion)

Somit...

- Ein-/Ausgabe in Haskell Monad über dem Typkonstruktor `IO`

Programmieren mit Monaden (1)

Gegeben:

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

Aufgabe:

- Schreibe eine Funktion, die die Summe der Werte der Marken in einem Baum vom Typ `Tree Int` berechnet.

Programmieren mit Monaden (2)

Lösung 1: Monadenlos

```
sTree :: Tree Int -> Int
sTree Nil              = 0
sTree (Node n t1 t2) = n + sTree t1 + sTree t2
```

Beachte:

- Die Reihenfolge der Berechnung ist weitgehend nicht festgelegt (Freiheitsgrade!)

Programmieren mit Monaden (3)

Lösung 1: Monadenbehaftet

```
sumTree :: Tree Int -> Id Int
sumTree Nil = return 0
sumTree (Node n t1 t2) = do num <- return n
                            s1  <- sumTree t1
                            s2  <- sumTree t2
                            return (num + s1 + s2)
```

...wobei `Id` die Identitätsmonade bezeichnet.

Beachte:

- Die Reihenfolge der Berechnung explizit festgelegt (keine Freiheitsgrade!)

Programmieren mit Monaden (4)

Die Identitätsmonade:

```
data Id a = Id a

instance Monad Id where
  (>>=) (Id x) f = f x
  return         = Id
```

Programmieren mit Monaden (5)

Vergleich der monadenlosen und monadenbehafteten Lösung:

Es gilt:

- Anders als `sTree` hat `sumTree` einen "imperativen Anstrich" in etwa vergleichbar mit:

```
num  := n;
sum1 := sumTree t1;
sum2 := sumTree t2;
return (num + sum1 + sum2);
```

Programmieren mit Monaden: Ein Resümee (1)

Die Programmierung mit Monaden erlaubt...

- Berechnungsabläufe zu strukturieren.

Folgende Eigenschaften prädestinieren Monaden dafür in besonderer Weise:

- *Wohldefiniert*: ...Strategie sequentielle Programmteile systematisch zu spezifizieren.

- *Angemessen*: ...höhere Abstraktion durch Entkopplung der zugrundeliegenden Monade von der Struktur der Berechnung.

- *Fundiert*: ...Eigenschaften wie etwa (4) werden impliziert von den Monadforderungen (1), (2) und (3).

Programmieren mit Monaden: Ein Resümee (2)

Monaden sind...

- ein in der Kategorientheorie geprägter Begriff
  $\rightsquigarrow$ ...zur formalen Beschreibung der Semantik von Programmiersprachen (Eugenio Moggi, 1989)

- (ohne obigen Hintergrund) populär in der Welt funktionaler Programmierung, insbesondere weil (Philip Wadler, 1992)
  - erlauben gewisse Aspekte imperativer Programmierung in die funktionale Programmierung zu übertragen
  - eignen sich insbesondere zur Integration von Ein-/Ausgabe, aber auch für weitergehende Anwendungsszenarien
  - geeignete Schnittstelle zwischen funktionaler und effektbehafteter, z.B. imperativer und objektorientierter Programmierung.

Zum Abschluss von Kapitel 12

...λ-artige Funktionsnotation in Haskell

...am Beispiel der Fakultätsfunktion:

```
fac :: Int -> Int
fac = \n -> (if n == 0 then 1 else (n * fac (n - 1)))
```

Mithin in Haskell: "\" statt "λ" und "->" statt "."

Anekdote (vgl. P. Pepper [4]):

$(n.\widehat{n+}1) \leadsto (\wedge n.n+1) \leadsto (\lambda n.n+1) \leadsto \backslash n\text{-}{>}n+1$

Kapitel 13: Zusammenfassung und Ausblick

Abschluss und Rückblick...

- Blick über den Gartenzaun
  ...(ausgewählte) andere funktionale Programmiersprachen

- Rückblick auf die Vorlesung

Im Rückblick auf die Vorlesung...

- Welche Aspekte funktionaler Programmierung haben wir betrachtet?
  - ...paradigmentypische, sprachunabhängige Aspekte
- Welche nicht oder nur gestreift?
  - ...sprachabhängige, speziell Haskell-spezifische Aspekte

Frei nach (und im Sinne von) Dietrich Schwanitz...

"*Alles, was man wissen muss...*

...um selber weiter zu lernen."

Überblick ü. d. Vorlesungsinhalte... (1)

- Vorbesprechung
- Kapitel 1: Einführung und Grundlagen
  - Einführung und Motivation
    - Warum funktionale Programmierung
    - Warum Haskell
    - Einstieg in Haskell und Hugs
  - Grundlagen
    - Elementare Datentypen: Wahrheitswerte, ganze Zahlen,...
    - Tupel, Listen und Funktionen, insbesondere notationelle Varianten

Überblick ü. d. Vorlesungsinhalte... (2)

- Kapitel 1, Fortführung
  - Funktionen und Funktionssignaturen
  - Funktionen und Funktionsterme, Rekursionstypen, Komplexitätsklassen und Aufrufgraphen, Funktionen und Curryfizierung
- Kapitel 2: Datentypdeklarationen
  - Typsynonyme
  - Algebraische Datentypen
    - Summentypen
    - Spezialfälle: Aufzählungs- und Produkttypen
    - Wichtige Varianten: Rekursive und polymorphe Typen, `newtype`-Deklarationen

Überblick ü. d. Vorlesungsinhalte... (3)

- Kapitel 3: Polymorphe Funktionen und Datentypen
  - Parametrische und ad-hoc Polymorphie
  - Typklassen
  - Vererbung
- Kapitel 4: Listen und Listenkomprehension, Muster
- Kapitel 5: Funktionen höherer Ordnung
- Kapitel 6: Fehlerbehandlung
- Kapitel 7: Ausdrücke, Auswertung von Ausdrücken, Auswertungsstrategien für Funktionen:
  - lazy vs. eager, call by name vs. call by value, normal order vs. applicative order

Überblick ü. d. Vorlesungsinhalte... (4)

- Kapitel 8: λ-Kalkül
- Kapitel 9: Ein- und Ausgabe
- Kapitel 10: Programmieren im Großen
  - Das Modulkonzept von Haskell
  - Abstrakte Datentypen
  - Reflektives Programmieren
- Kapitel 11: Programmierprinzipien
- Kapitel 12: Monaden
  - Programmierung mit Monaden
- Kapitel 13: Zusammenfassung und Ausblick
  - Abschluss und Rückblick
  - Blick über den Gartenzaun und Resümee

Resümee (1)

Charakteristika fkt. und imp. Sprachen (P. Pepper [4])...

- Funktional...
  - Programm ist *Ein-/Ausgaberelation*
  - Programme sind *“zeit”-los*
  - Programmformulierung auf *abstraktem*, *mathematisch geprägten Niveau*
- Imperativ...
  - Programm ist *Arbeitsanweisung* für eine Maschine
  - Programme sind *zustands*- und *“zeit”-behaftet*
  - Programmformulierung konkret *mit Blick auf eine Maschine*

Resümee (2)

Die Fülle an Möglichkeiten (in funktionalen Programmiersprachen) erwächst aus einer kleinen Zahl von elementaren Konstruktionsprinzipien.

P. Pepper [4]

Im Falle von...

- Funktionen
  - (Fkt.-) Applikation, Fallunterscheidung und Rekursion
- Datenstrukturen
  - Produkt- und Summenbildung, Rekursion

Tragen bei zu Mächtigkeit und Eleganz und damit auch zu...

Functional programming is fun!

Rückschauend... war es das?

Blick über den Gartenzaun

...auf ausgewählte andere funktionale Programmiersprachen:

- ML: Ein “eager” Wettbewerber
- Lisp: Der Oldtimer
- APL: Ein Exot

...und einige ihrer Charakteristika.

ML: Ein “eager” Wettbewerber von Haskell

- ML ist eine strikte funktionale Sprache
- Lexical scoping, Curryfizieren (wie Haskell)
- stark typisiert mit Typinferenz, keine Typklassen
- umfangreiches Typkonzept für Module und ADT
- zahlreiche Erweiterungen (beispielsweise in OCAML) auch für imperative und objektorientierte Programmierung
- sehr gute theoretische Fundierung

Beispiel: Module/ADTs in ML

```
structure S = struct
     type 't Stack              = 't list;
     val  create                = Stack nil;
     fun  push x (Stack xs)     = Stack (x::xs);
     fun  pop (Stack nil)       = Stack nil;
     |    pop (Stack (x::xs))   = Stack xs;
     fun  top (Stack nil)       = nil;
     |    top (Stack (x:xs))    = x;
end;

signature st = sig type q; val push: 't -> q -> q; end;

structure S1:st = S;
```

Lisp: Der Oldtimer funktionaler Programmiersprachen

- Lisp ist eine noch immer häufig verwendete strikte funktionale Sprache mit imperativen Zusätzen
- umfangreiche Bibliotheken, leicht erweiterbar
- einfache, interpretierte Sprache, dynamisch typisiert
- Listen sind gleichzeitig Daten und Funktionsanwendungen
- nur lesbar, wenn Programme gut strukturiert
- in vielen Bereichen (insbesondere KI, Expertensysteme) erfolgreich eingesetzt
- sehr gut zur Metaprogrammierung geeignet

Ausdrücke in Lisp

Beispiele für Symbole:	A	(Atom)
	austria	(Atom)
	68000	(Zahl)

Beispiele für Listen: (plus a b)
((meat chicken) water)
(unc trw synapse ridge hp)
nil bzw. () entsprechen leerer Liste

Eine Zahl repräsentiert ihren Wert direkt —
ein Atom ist der Name eines assoziierten Wertes

(setq x (a b c)) bindet x global an (a b c)

(let ((x a) (y b)) e) bindet x lokal in e an a und y an b

Funktionen in Lisp

Erstes Element einer Liste wird normalerweise als Funktion interpretiert, angewandt auf restliche Listenelemente.

(quote a) bzw. 'a liefert Argument a selbst als Ergebnis.

Beispiele für primitive Funktionen:

(car '(a b c))	$\Rightarrow$ a	(atom 'a)	$\Rightarrow$ t
(car 'a)	$\Rightarrow$ error	(atom '(a))	$\Rightarrow$ nil
(cdr '(a b c))	$\Rightarrow$ (b c)	(eq 'a 'a)	$\Rightarrow$ t
(cdr '(a))	$\Rightarrow$ nil	(eq 'a 'b)	$\Rightarrow$ nil
(cons 'a '(b c))	$\Rightarrow$ (a b c)	(cond ((eq 'x 'y) 'b)	
(cons '(a) '(b))	$\Rightarrow$ ((a) b)	(t 'c))	$\Rightarrow$ c

Definition von Funktionen in Lisp

(lambda (x y) (plus x y)) ist Funktion mit zwei Parametern

((lambda (x y) (plus x y)) 2 3) wendet die Funktion an: ⇒ 5

(define (add (lambda (x y) (plus x y)))) definiert einen globalen Namen „add“ für die Funktion

(defun add (x y) (plus x y)) ist abgekürzte Schreibweise dafür

Beispiel:

```
(defun reverse (l) (rev nil l))
(defun rev (out in)
     (cond ((null in) out)
            (t (rev (cons (car in) out) (cdr in)))))
```

Closures

- kein Curryfizieren in Lisp, Closures als Ersatz
- Closures: lokale Bindungen behalten Wert auch nach Verlassen der Funktion

  Beispiel:
  ```
  (let ((x 5))
    (setf (symbol-function 'test)
      #'(lambda () x)))
  ```
- praktisch: Funktion gibt Closure zurück

  Beispiel:
  ```
  (defun create-function (x)
    (function (lambda (y) (add x y)))
  ```
- Closures sind flexibel, aber Curryfizieren ist viel einfacher

Dynamic Scoping — Static Scoping

- lexikalisch: Bindung ortsabhängig (Source-Code)
- dynamisch: Bindung vom Zeitpunkt abhängig
- normales Lisp: lexikalisches Binden

  Beispiel:
  ```
  (setq a 100)
  (defun test () a)
  (let ((a 4)) (test))
  ```
  $\Rightarrow$ 100

- dynamisches Binden durch (defvar a) möglich

  obiges Beispiel liefert damit 4

Makros

- Code expandiert, nicht als Funktion aufgerufen (wie C)
- Definition: erzeugt Code, der danach evaluiert wird

  Beispiel:

  ```
  (defmacro get-name (x n)
      (list 'cadr (list 'assoc x n)))
  ```

- Expansion und Ausführung:

  (get-name 'a b) $\Leftrightarrow$ (cadr (assoc 'a b))

- nur Expansion:

  (macroexpand '(get-name 'a b)) $\Rightarrow$ '(cadr (assoc 'a b))

Lisp vs. Haskell: Ein Vergleich

Kriterium	Lisp	Haskell
Basis	einfacher Interpreter	formale Grundlage
Zielsetzung	viele Bereiche	referentiell transparent
Verwendung	noch häufig	zunehmend
Sprachumfang	riesig (kleiner Kern)	moderat, wachsend
Syntax	einfach, verwirrend	modern, Eigenheiten
Interaktivität	hervorragend	nur eingeschränkt
Typisierung	dynamisch, einfach	statisch, modern
Effizienz	relativ gut	relativ gut
Zukunft	noch lange genutzt	einflussreich

APL: Ein Exot

- APL ist eine ältere applikative (funktionale) Sprache mit imperativen Zusätzen
- zahlreiche Funktionen (höherer Ordnung) sind vordefiniert, Sprache aber nicht einfach erweiterbar
- dynamisch typisiert
- verwendet speziellen Zeichensatz
- Programme sehr kurz und kompakt, aber kaum lesbar
- vor allem für Berechnungen mit Feldern gut geeignet

Beispiel: Programmentwicklung in APL

Berechnung der Primzahlen von 1 bis N

Schritt 1. $(\iota N) \circ.| (\iota N)$

Schritt 2. $0 = (\iota N) \circ.| (\iota N)$

Schritt 3. $+/[2]\ 0 = (\iota N) \circ.| (\iota N)$

Schritt 4. $2 = (+/[2]\ 0 = (\iota N) \circ.| (\iota N))$

Schritt 5. $(2 = (+/[2]\ 0 = (\iota N) \circ.| (\iota N)))\ /\ \iota N$

Erfolgreiche Einsatzfelder funktionaler Programmierung

- Compiler in compilierter Sprache geschrieben
- Theorembeweiser HOL und Isabelle in ML
- Modelchecker (z.B. Edinburgh Concurrency Workbench)
- Mobility Server von Ericson in Erlang
- Konsistenzprüfung mit Pdiff (Lucent 5ESS) in ML
- CPL/Kleisli (komplexe Datenbankabfragen) in ML
- Natural Expert (Datenbankabfragen Haskell-ähnlich)
- Ensemble zur Spezifikation effizienter Protokolle (ML)
- Expertensysteme (insbesondere Lisp-basiert)
- ...
- http://homepages.inf.ed.ac.uk/wadler/realworld/

Zum Schluss: Eine hoffnungsvolle Prognose...

The clarity and economy of expression that the language of functional programming permits is often very impressive, and, but for human inertia, functional programming can be expected to have a brilliant future.(∗)

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.

Lohnenswerte Literaturhinweise...

Bereits in der Vorbesprechung angegeben:

- Wadler, P. Why no one uses functional languages. ACM SIGPLAN Notices 33(8), 23-27, 1998

Aber man sollte stets beide Seiten der Medaille kennen...

- Wadler, P. An angry half-dozen. ACM SIGPLAN Notices 33(2), 25-30, 1998
- Hughes, J. Why Functional Programming Matters. Computer Journal 32(2), 98-107, 1989
- Philip Wadler. *The Essence of Functional Programming*. In Conference Record of the 19th Annual Symposium on Principles of Programming Languages (POPL'92), eingeladener Vortrag, 1992.
- Simon Peyton-Jones. *Wearing the Hair Shirt: A Retrospective on Haskell*, eingeladener Vortrag, 30th Annual Symposium on Principles of Programming Languages (POPL'03), 2003.

Organisatorisches zur schriftlichen LVA-Prüfung 1(3)

Hinweise zur schriftlichen LVA-Prüfung...

- Worüber...
  - Vorlesungsstoff, Übungsstoff, und folgender wissenschaftlicher (Übersichts-) Artikel: John Hughes. *Why Functional Programming Matters.* Research Topics in Functional Programming. D. Turner (Hrsg.), Addison Wesley, 1990.
- Wann, wo und wie lange...
  - Der Haupttermin ist am...
    - ∗ Do, den 20.01.2011, von 16:45 Uhr s.t. bis 18:45 Uhr, im Radinger-Hörsaal, Getreidemarkt 2; die Dauer beträgt 90 Minuten.
- Hilfsmittel...
  - Keine.

Organisatorisches zur schriftlichen LVA-Prüfung 2(3)

Hinweise zur schriftlichen LVA-Prüfung...

- Mitzubringen sind...
  - Papier, Stifte, **Studierendenausweis**.
- Anmeldung: Ist erforderlich...
  - *Wann*: Zwischen dem 10.01.2011 und dem 18.01.2011
  - *Wie*: Elektronisch über TISS
- Voraussetzung...
  - Mindestens 50% der Punkte aus der Laborübung

Organisatorisches zur schriftlichen LVA-Prüfung 3(3)

Neben dem Haupttermin wird es drei Nebentermine für die schriftliche LVA-Prüfung geben, und zwar...

- zu Anfang
- in der Mitte
- am Ende

der Vorlesungszeit im SS 2011.

Danach in jedem Fall Zeugnisausstellung.

Die genauen Termine werden rechtzeitig über TISS angekündigt!

Auch zur Teilnahme an der schriftlichen LVA-Prüfung an einem der Nebentermine ist eine Anmeldung über TISS erforderlich.