

Funktionale Programmierung

LVA 185.A03, VU 2.0, ECTS 3.0
WS 2020/2021

Vortrag II
Orientierung, Einordnung
15.10.2020

Jens Knoop



Technische Universität Wien
Information Systems Engineering
Compilers and Languages



Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Vortrag II

Orientierung, Einordnung

...zum selbstgeleiteten, eigenständigen Weiterlernen.

Teil II: Grundlagen

- Kapitel 2: Vordefinierte Datentypen
 - ↪ Zahlen, Zeichen, Wahrheitswerte, Tupel, Listen,...
- Kapitel 3: Funktionen
 - ↪ Syntaxvarianten, curryfiziert, uncurryfiziert, Stelligkeit,...
- Kapitel 4: Typsynonyme, Neue Typen, Typklassen
 - ↪ `type`, `newtype`, `class`, Überladung,...
- Kapitel 5: Algebraische Datentypdeklarationen
 - ↪ `data`, Funktionen auf alg. Datentypen, Feldsyntax,...
- Kapitel 6: Muster und mehr

Teil II

Grundlagen

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Kapitel 2

Vordefinierte Datentypen

Vortrag II

Teil II

Kap. 2

2.3

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Haskell bietet vordefiniert

...wie andere Programmiersprachen auch:

- **Ganze Zahlen:** Werte `0`, `1`, `-1`, `2`, `-2`, `3`, `-3`,...
- `Int` (bereichsbeschränkt, -2^n bis $2^n - 1$, $n \in \{31, 63\}$)
- `Integer` ('unbeschränkt')
- **Gleitkommazahlen:** Werte `0.0`, `3.14`, `-2.71828`,...
- `Float` (einfache Genauigkeit, 32 Bit)
- `Double` (doppelte Genauigkeit, 64 Bit)
- **Wahrheitswerte:** Werte `True`, `False`
- `Bool`
- **Zeichen:** Werte `'a'`, `'A'`, `'b'`, `'B'`, `'0'`, `'1'`, `'['`, `'@'`, `'#'`,...
- `Char`

mit den 'typüblichen' Operationen (`+`, `*`, `-`, `&&`, `||`, `==`, `>=`, `>`, etc.; s. [Haskell-Sprachbericht](#), [Standard-Präludium](#) f. Details).

Haskell bietet weiters vordefiniert

...Tupel-, Kreuzprodukttypen (Paare, Tripel, Quadrupel,...).

Paarwerte, Paartypen

`(2,3) :: (Int,Int)`

`('e',2.71828) :: (Char,Float)`

`(False,12345678901234567890) :: (Bool,Integer)`

Tripelwerte, Tripeltypen

`(1234567890,3.14,True) :: (Int,Float,Bool)`

`(1234567890,3.14,3.14) :: (Integer,Double,Double)`

`(True,False,True) :: (Bool,Bool,Bool)`

Randfall: Nulltupel oder leeres Tupel, Nulltupeltyp

`() :: ()` (wicht. Randf., s. Kap. 15 Ein-/Ausgabe)
Nulltupel Nulltupeltyp

...Gleichbezeichnung von Typ `()` und einzigem (!) Wert `()`.

Haskell bietet vordefiniert auch

...Listentypen, äußerst wichtige Datentypen in allen funktionalen Programmiersprachen (vgl. [Lisp](#): List Processing Language).

Liste von:

– Ganze Zahlen

```
[2,5,17,2,4,42,4711] :: [Int]
```

– Gleitkommazahlen

```
[3.14,5.0,-12.21] :: [Float]
```

– Wahrheitswerte

```
[True,False,True] :: [Bool]
```

– Zeichen

```
['a','B','c','$','D','e','@','$','#'] :: [Char]
```

– ohne Elemente, leere Liste (bel. Typs)

```
[] :: [a]
```

Weitere Beispiele (1)

Listen von

– Tupeln

```
[('a',True),('b',False),('c',False),  
 ('d',False),('e',True)] :: [(Char,Bool)]
```

```
[(3,5,4.0),(4,7,5.5),(2,8,5.0),(2,11,6.5)]  
 :: [(Int,Int,Float)]
```

– Listen

```
[[1,2,3],[9],[],[17,4,21],[],[3,2]] :: [[Int]]
```

```
[(['f','p'],2),(['h'],1),([],0)] :: [([Char],Int)]
```

```
[("fun",3),("h",1),("",0)] :: [([Char],Int)]
```

– Zeichenreihen

```
["sin","cos","tan","sqrt"] :: [Char]
```


Weitere Beispiele (2)

Listen von

- Funktionen

```
[sin,cos,tan,sqrt] :: [Float -> Float]
```

```
[(+),(*),ggf,mod] :: [Int -> Int -> Int]
```

```
[binom',binom''] :: [(Integer,Integer) -> Integer]
```

```
[binom,binom] :: [Integer -> Integer -> Integer]
```

- ...

Vielzahl vordefinierter Funktionen auf Listen

Name, Typ

Bedeutung, Beispiel

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

Anfügen eines Elements am Anfang einer Liste:

`5: [3,2] ->> [5,3,2]`

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

Aneinanderhängen zweier Listen:

`[11,7] ++ [5,3,2] ->> [11,7,5,3,2]`

`(!!)` `:: [a] -> Int -> a`

Zugreifen auf ein Listenelement:

`[5,3,2] !! 0 ->> 5`

`[5,3,2] !! 1 ->> 3`

`concat` `:: [[a]] -> [a]`

Verschmelzen einer Liste von Listen zu einer Liste:

`concat [[11,7], [5,3,2]]`

`->> [11,7,5,3,2]`

`reverse` `:: [a] -> [a]`

Umkehren einer Liste:

`reverse [5,3,2] ->> [2,3,5]`

...und viele weitere (siehe [Standard-Präludium](#)).

Einige davon jetzt als weitere Beispiele

Die Funktion `length` (Länge einer Liste):

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

Aufrufbeispiele:

```
length [1,2,3]           ->> 3
length [[1],[2,3],[4,5,6]] ->> 3
length [sin,cos,tan,sqr] ->> 4
length ["sin","cos","tan","sqrt"] ->> 4
length []                ->> 0
```

...und noch ein paar Beispiele

Die Funktionen `head` und `tail` (Kopf und Rest einer Liste):

```
head :: [a] -> a
```

```
head (x:_) = x
```

```
tail :: [a] -> [a]
```

```
tail (_:xs) = xs
```

Aufrufbeispiele:

```
head [1,2,3] ->> 1
```

```
head [[1],[2,3],[4,5,6]] ->> [1]
```

```
head [sin,cos,tan,sqrt] '->> sin'
```

```
head [sin,cos,tan,sqrt] (pi/2) ->> 1.0
```

```
tail [1,2,3] ->> [2,3]
```

```
tail [[1],[2,3],[4,5,6]] ->> [[2,3],[4,5,6]]
```

```
tail [sin,cos,tan,sqrt] '->> [cos,tan,sqrt]'
```

```
tail ["sin","cos","tan","sqrt"] ->> ["cos","tan","sqrt"]
```

Besonders nützl.: Listenaufzählungsausdrücke

...zur **automatischen Generierung** von Listen über Typen geordneter **aufzählbarer Werte** (Typen der Typklasse **Enum**):

```
[2..10]           ->> [2,3,4,5,6,7,8,9,10]
[2,4..10]        ->> [2,4,6,8,10]
[2,4..11]        ->> [2,4,6,8,10]
[2,4..12]        ->> [2,4,6,8,10,12]
[11,9..3]        ->> [11,9,7,5,3]
[11,9..2]        ->> [11,9,7,5,3]
[11,10..2]       ->> [11,10,9,8,7,6,5,4,3,2]
[11..2]          ->> []
['a','c'..'g']   ->> ['a','c','e','g'] ->> "aceg"
['a','c'..'h']   ->> ['a','c','e','g'] ->> "aceg"
[0.0,0.3..1.2]  ->> [0.0,0.3,0.6,0.9,1.2]
```

Äußerst nützlich auch: Listenkomprehension

...Zusammenfassung, Vereinigung von Mannigfaltigkeiten zu einer Einheit (Philos.):

- In funktionalen Sprachen ein weiteres wichtiges Sprachkonstrukt für automatische Listengenerierung!
- Alleinstellungsmerkmal funktionaler Sprachen!

Beispiele:

```
ns = [1..10] ( $\hat{=}$  [1,2,3,4,5,6,7,8,9,10])
```

```
[3*n | n <- ns] ->> [3,6,9,12,15,18,21,24,27,30]
```

```
[n | n <- ns, odd(n)] ->> [1,3,5,7,9]
```

```
[n | n <- ns, even(n), n > 5] ->> [6,8,10]
```

```
[n*(n+1) | n <- ns, (even(n) || n > 5)]  
->> [6,20,42,56,72,90,110]
```

```
[p | n <- ns, m <- ns, n <= 3, m >= 9, let p=m*n]  
->> [9,10,18,20,27,30]
```

Syntaktischer Zucker

Die Schreibweise:

$[1,2,3]$ (Syntaktischer Zucker)

ist Abkürzung für die Grund-/Standarddarstellung:

$(1:(2:(3:[])))$ (Standarddarstellung)

die dank **Klammereinsparungsregeln** (Rechtsassoziativität von $(:)$) gleichbedeutend ist zu:

$1:2:3:[]$ (ausgenutzte Klammereinsparungsregeln)

Es gilt:

$[1,2,3] == (1:(2:(3:[]))) == 1:2:3:[]$

Zeichenreihen

...spezielle Listen; **Listen über Zeichen**, d.h. über Elementen des Typs `Char`.

Beispiele:

`('F' : ('u' : ('n' : []))) :: [Char]` (Standarddarst.)

`'F' : 'u' : 'n' : [] :: [Char]` (Rechtsassoziativität)

`['F', 'u', 'n'] :: [Char]` (Syntaktischer Zucker)

`"Fun" :: [Char]` (Noch mehr syntaktischer Zucker)

`('F' : ('u' : ('n' : []))) == 'F' : 'u' : 'n' : []`
`== ['F', 'u', 'n'] == "Fun"`

Zusätzlich gibt es **syntaktischen Zucker** für den **Typnamen**:

`('F' : ('u' : ('n' : []))) :: String` (statt `[Char]`)

`'F' : 'u' : 'n' : [] :: String`

`['F', 'u', 'n'] :: String`

`"Fun" :: String`

Vordefinierte Funktionen auf Zeichenreihen

...alle auf **Listen** vordefinierten Operatoren und Relatoren stehen auch auf **Zeichenreihen** als speziellen Listen unmittelbar zur Verfügung.

Beispiele:

```
['H','e','l','l','o'] ++ "," ++ " " ++ "world!"  
->> "Hello, world!"
```

```
['H','e','l','l','o'] ++ "," ++ " " ++ "world!"  
== "Hello, world!" ->> True
```

```
length "Hello, world!" ->> 13
```

```
head "Hello, world!" ->> 'H'
```

```
tail (tail "Hello, world!") ->> "llo, world!"
```

```
head (tail (tail "Hello, world!")) ->> 'l'
```

Kapitel 2.3

Leseempfehlungen

Vortrag II

Teil II

Kap. 2

2.3

Kap. 3

Kap. 4

Kap. 5




Kap. 6

Umgekehrte
Klassen-
zimmer I




Hinweis

Aufgabe

Basiselesempfehlungen für Kapitel 2

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 2, Einfache Datentypen; Kapitel 5.1, Listen; Kapitel 5.2, Tupel; Kapitel 5.3, Zeichenreihen)
-  Richard Bird. *Introduction to Functional Programming using Haskell*. Cambridge University Press, 2. Auflage, 1998. (Kapitel 2, Simple datatypes; Kapitel 4, Lists)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 3, Basic types and definitions; Kapitel 5, Data types, tuples and lists)

Weiterführende Leseempfehlungen für Kap. 2

-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 3.1, Basic concepts; Kapitel 3.2, Basic types; Kapitel 3.3, List types; Kapitel 3.4, Tuple types; Kapitel 5, List comprehensions)
-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 1, An Intro to Lists, Tuples; Kapitel 2, Common Haskell Types)
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 2, Types and Functions – Useful Composite Data Types: Lists and Tuples, Functions over Lists and Tuples)

Kapitel 3

Funktionen

Vortrag II

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Kapitel 3.1

Definition, Schreibweisen, Sprachkonstrukte

Zentral: Alternativenauswahl auszudrücken

Nicht immer wie für `binom3`:

```
binom3 :: (Int,Int) -> Int
binom3 (m,n) = (m + n ) * (m - n)
```

kommt man **ohne Alternativen** aus:

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

Alternativen mit 'if-then-else'

...werden schnell unübersichtlich, wenn sie **geschachtelt** werden:

```
fib :: Int -> Int
fib n = if n == 0 then 0
        else if n == 1 then 1
              else fib (n-1) + fib (n-2)
```

```
maximum :: Int -> Int -> Int -> Int
maximum n m k = if n >= m
                  then (if n >= k then n else k)
                  else (if m >= k then m else k)
```


Alternativenauswahl: 2 Hauptmethoden

...wert- oder musterbasiert:

1. Wertbasierte Auswahl (Leitfrage: Welchen Wert hat das Argument?)

```
fib :: Int -> Int
fib n
  | n == 0 = 0           (wenn Wert gleich 0, dann...)
  | n == 1 = 1           (wenn Wert gleich 1, dann...)
  | True   = fib (n-1) + fib (n-2) (wenn Wert anders, dann...)
```

...entspricht 1-zu-1 'if-then-else', Unterschied ist nützlich, aber rein syntaktisch ('|' statt 'if-then-else')!

2. Musterbasierte Auswahl (Leitfrage: Wie sieht das Argument aus?)

```
fib :: Int -> Int
fib 0 = 0           (wenn das Arg. aussieht wie 0, dann...)
fib 1 = 1           (wenn das Arg. aussieht wie 1, dann...)
fib n = fib (n-1) + fib (n-2) (wenn das Arg. anders aussieht als 0 oder 1, dann...)
```

...substantiell neu! Musterausdrücke, Musterauswahl sind Alleinstellungsmerkmal funktionaler Programmierung!

Beide Methoden lassen sich gut kombinieren

3. Mischform: Muster- und wertbasierte Auswahl

```
fib :: Int -> Int
```

```
fib 0 = 0
```

```
fib 1 = 1
```

```
fib n
```

```
  | n > 1 = fib (n-1) + fib (n-2)
```

```
  | n < 0 = error "Argument unzuverlässig!"
```

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

```
binom (n,0)
```

```
  | n >= 0 = 1
```

```
  | n < 0 = error "Argument unzuverlässig!"
```

```
binom (n,k)
```

```
  | n == k = 1
```

```
  | (n>0) && (k>0) = binom (n-1,k-1) + binom (n-1,k)
```

```
  | True = error "Argument unzuverlässig!"
```

Darüberhinaus bietet Haskell

...viele weitere Varianten an, Funktionen zu definieren:

- Lokale Deklarationen (**where**, **let**):

```
quickSort :: [Integer] -> [Integer]
quickSort []      = []
quickSort (n:ns) = quickSort smaller
                  ++ [n]
                  ++ quickSort larger
                  where smaller = [m | m<-ns, m<=n]
                        larger  = [m | m<-ns, m>n]
```

```
quickSort :: [Integer] -> [Integer]
quickSort []      = []
quickSort (n:ns) = let smaller = [m | m<-ns, m<=n]
                  larger  = [m | m<-ns, m>n]
                  in (quickSort smaller
                    ++ [n]
                    ++ quickSort larger)
```

Mehr Varianten

...in einer Zeile, argumentfrei:

- In einer Zeile (;):

```
quickSort :: [Integer] -> [Integer]
quickSort []      = []
quickSort (n:ns) =
  quickSort smaller ++ [n] ++ quickSort larger
  where smaller = [m | m <- ns, m <= n]; larger = [m | m <- ns, m > n]
```

```
quickSort :: [Integer] -> [Integer]
quickSort []      = []
quickSort (n:ns) =
  let smaller = [m | m <- ns, m <= n]; larger = [m | m <- ns, m > n]
  in (quickSort smaller ++ [n] ++ quickSort larger)
```

- Argumentfrei (Anonyme λ -Abstraktion):

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

Anonyme λ -Abstraktion

...für noch mehr Varianten und Beispiele siehe [Kapitel 3.1](#).

Kapitel 3.2

Funktionssignaturen, Funktionsterme, Funktionsstelligkeiten

Überblick

Funktions-

- Signaturen
- Terme
- Stelligkeiten

und damit verbundene

- Klammereinsparungsregeln in Haskell.

Das Wichtigste auf einen Blick:

- (Funktions-) Signaturen sind rechtsassoziativ geklammert.
- (Funktions-) Terme sind linksassoziativ geklammert.
- (Funktions-) Stelligkeit ist 1.

Als Beispiel: Die Editorfunktion 'ersetze'

...eine Funktion, die in einem `Text` das n -te Vorkommen einer Zeichenreihe `s` durch eine Zeichenreihe `s'` ersetzt.

Implementierung in Haskell

```
type Txt = String
type Vork = Int
type Alt = Txt
type Neu = Txt
ersetze :: (Txt -> (Vork -> (Alt -> (Neu -> Txt))))
```

...angewendet auf einen Text `t`, eine Vorkommensnummer `n` und zwei Zeichenreihen `s` und `s'` ist das Resultat der Anwendung von `ersetze` ein Text `t'`, in dem das n -te Vorkommen von `s` in `t` durch `s'` ersetzt ist.

Eine Anwendung, ein Aufruf von `ersetze`

Die `Funktion`, noch einmal wiederholt:

```
type Txt = String
type Vork = Int
type Alt = Txt
type Neu = Txt
ersetze :: (Txt -> (Vork -> (Alt -> (Neu -> Txt))))
```

Konkrete `Argumentwerte`:

```
"Ein alter Text" :: Txt
1                :: Vork
"alter"          :: Alt
"neuer"          :: Neu
```

Die `Auswertung`:

```
ersetze "Ein alter Text" 1 "alter" "neuer"
≡ (((ersetze "Ein alter Text") 1) "alter") "neuer"
->> "Ein neuer Text" :: Txt
```


Schrittweise Auswertung (1)

...**ersetze** und die nach fortgesetzter Argumentkonsumation entstehenden **Funktionsterme** sind mit Ausnahme des letzten von **funktionalem Typ**.

Schritt 1: Der Funktionsterm **ersetze** konsumiert **ein** Argument, den Wert "**Ein alter Text**" vom Typ **Txt**. Der dadurch entstehende **Funktionsterm** ist von funktionalem Typ:

```
(ersetze "Ein alter Text") ::  
      (Vork -> (Alt -> (Neu -> Txt)))
```

Schritt 2: Der Funktionsterm **(ersetze "Ein alter Text")** konsumiert **ein** Argument, den Wert **1** vom Typ **Vork**. Der dadurch entstehende **Funktionsterm** ist von funktionalem Typ:

```
((ersetze "Ein alter Text") 1) :: (Alt -> (Neu -> Txt))
```

Schrittweise Auswertung (2)

Schritt 3: Der Funktionsterm `((ersetze "Ein alter Text") 1)` konsumiert **ein** Argument, den Wert `"alter"` vom Typ **Alt**. Der dadurch entstehende Funktionsterm ist von funktionalem Typ:

```
((ersetze "Ein alter Text") 1) "alter" :: (Neu -> Txt)
```

Schritt 4: Der Funktionsterm `((ersetze "Ein alter Text") 1) "alter"` konsumiert **ein** Argument, den Wert `"neuer"` vom Typ **Neu**. Der dadurch entstehende Term ist von **nichtfunktionalem** Typ:

```
((((ersetze "Ein alter Text") 1) "alter") "neuer") :: Txt
```

Insgesamt erhalten wir:

```
((((ersetze "Ein alter Text") 1) "alter") "neuer")  
->> "Ein neuer Text" :: Txt
```

Funktionssignaturen, Funktionsterme

Funktionssignaturen (oder syntaktische Funktionssignaturen oder Signaturen)

- geben den **Typ einer Funktion** an.

Funktionsterme

- sind aus **Funktionsaufrufen** aufgebaute **Ausdrücke**.

Beispiele:

- **Funktionssignatur**

```
ersetze :: Txt -> Vork -> Alt -> Neu -> Txt
```

- **Funktionsterme**

```
ersetze "Ein alter Text"
```

```
ersetze "Ein alter Text" 1
```

```
ersetze "Ein alter Text" 1 "alter"
```

```
ersetze "Ein alter Text" 1 "alter" "neuer"
```

Klammereinsparungsregeln

...für Funktionssignaturen und Funktionsterme.

Rechtsassoziativität für **Funktionssignaturen**:

ersetze `:: Txt -> Vork -> Alt -> Neu -> Txt`

...steht abkürzend für die vollständig, aber nicht überflüssig **rechtsassoziativ** geklammerte **Funktionssignatur**:

ersetze `:: (Txt -> (Vork -> (Alt -> (Neu -> Txt))))`

Linksassoziativität für **Funktionsterme**:

ersetze `"Ein alter Text" 1 "alter" "neuer"`

...steht abkürzend für den vollständig, aber nicht überflüssig **linksassoziativ** geklammerten **Funktionsterm**:

`((((ersetze "Ein alter Text") 1) "alter") "neuer")`

Hintergrund: Klammereinsparung

Die Festlegung von

- Rechtsassoziativität für Funktionssignaturen
- Linksassoziativität für Funktionsterme

dient der Einsparung von Klammern (vgl. Punkt- vor Strichrechnung in der Mathematik).

Die Festlegung erfolgt auf diese Weise, da so in

- Signaturen und Funktionstermen

meist möglichst wenige, oft gar keine Klammern nötig sind.

Funktionspfeil vs. Kreuzprodukt (1)

Eine naheliegende Frage im Zshg. mit der Funktion `ersetze`:

- ▶ Warum so **viele Pfeile** (`->`), warum so **wenige Kreuze** (`×`) in der Signatur von `ersetze`?
- ▶ Warum nicht

`'ersetze :: (Txt × Vork × Alt × Neu) -> Txt'`
statt

`ersetze :: Txt -> Vork -> Alt -> Neu -> Txt?`

Beachte: Das Kreuzprodukt in Haskell wird durch Tupelbeistrich ausgedrückt, d.h. `,` statt `×`. Die korrekte **Haskell-Spezifikation** für die Kreuzproduktvariante lautete daher:

`ersetze :: (Txt, Vork, Alt, Neu) -> Txt`

Funktionspfeil vs. Kreuzprodukt (2)

Beide Formen

- sind möglich, sinnvoll und berechtigt.

Funktionspfeil

- führt jedoch zu höherer (Anwendungs-) Flexibilität als Kreuzprodukt, da partielle Auswertung von Funktionen möglich ist.
- ist daher in funktionaler Programmierung die weitaus häufiger verwendete Form.

Zur Illustration:

- Berechnung der Binomialkoeffizienten.

Funktionspfeil vs. Kreuzprodukt (3)

Vergleiche die **Funktionspfeilform**:

```
binom :: Integer -> Integer -> Integer
```

```
binom n k
```

```
| k==0 || n==k = 1
```

```
| otherwise     = binom (n-1) (k-1) + binom (n-1) k
```

...mit der **Kreuzproduktform**:

```
binom' :: (Integer,Integer) -> Integer
```

```
binom' (n,k)
```

```
| k==0 || n==k = 1
```

```
| otherwise     = binom' (n-1,k-1) + binom' (n-1,k)
```


Funktionspfeil vs. Kreuzprodukt (4)

Die höhere Flexibilität der Funktionspfeilform zeigt sich in der Anwendungssituation:

Der Funktionsterm `(binom 45)`

- ist von funktionalem Typ `(Integer -> Integer)`, eine Funktion, die ganze Zahlen in sich abbildet.
- liefert angewendet auf eine natürliche Zahl k die Anzahl der Möglichkeiten, auf die man k Elemente aus einer 45-elementigen Grundgesamtheit herausgreifen kann:
`((binom 45)` entspricht der Funktion `k_aus_45)`

Funktionspfeil vs. Kreuzprodukt (5)

Wir können den Funktionsterm `(binom 45)` deshalb auch benutzen, um *in argumentfreier Weise* eine neue Funktion zu definieren, z.B. die Funktion `k_aus_45` (vgl. Kap. 1.1.1):

```
k_aus_45 :: Integer -> Integer
k_aus_45 = binom 45 -- arg.frei: k_aus_45 ist nicht
                    -- von einem Arg. gefolgt
```

Die Funktion `k_aus_45` und der Funktionsterm `(binom 45)` bezeichnen *dieselbe Funktion*; sie sind Synonyme.

Aufrufe folgender Form sind deshalb möglich:

```
(binom 45) 6 ->> 8.145.060
binom 45 6   ->> 8.145.060 -- Klammereinsparungsr.
k_aus_45 6   ->> binom 45 6 ->> 8.145.060
```

Funktionspfeil vs. Kreuzprodukt (6)

Beachte: Auch die Funktion

```
binom' :: (Integer,Integer) -> Integer
```

ist im Haskell-Sinn einstellig.

Folgende Schreibweise macht dies besonders deutlich:

```
type IntPair = (Integer,Integer)
```

```
binom' :: IntPair -> Integer -- 1 Argument: 1-stellig
```

```
binom' p
```

```
  | snd(p) == 0 || fst(p)==snd(p) = 1
```

```
  | otherwise = binom' (fst(p)-1,snd(p)-1)  
                + binom' (fst(p)-1,snd(p))
```

`p` vom Typ `IntPair`, das eine Argument von `binom'` ist von einem `Paartyp`.

Funktionspfeil vs. Kreuzprodukt (7)

Beachte: `binom'` bietet nicht die Flexibilität von `binom`:

- `binom'` konsumiert ihr `eines` Argument `p` vom Paartyp `(Integer,Integer)` und liefert unmittelbar ein Resultat vom elementaren Typ `Integer`.

`binom' (45,6) ->> 8.145.060 :: Integer`

- ein funktionales Zwischenresultat entsteht anders als bei `binom` nicht.
- Eine lediglich `teilweise Versorgung mit Argumenten` und damit `partielle Auswertung` von `binom'` ist `nicht möglich`.

Aufrufe der Form:

`binom' 45`

sind `syntaktisch inkorrekt` und führen zu Fehlermeldungen.

Vordef. arithmetische Operationen in Pfeilform

Auch die arithmetischen (und viele weitere) Operationen sind in Haskell aus diesem Grund in der Funktionspfeilform vordefiniert:

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

Nachstehend instantiiert für den Typ `Int`:

```
(+) :: Int -> Int -> Int  
(* ) :: Int -> Int -> Int  
(- ) :: Int -> Int -> Int  
...
```

Funktionsstelligkeiten: Mathematik vs. Haskell

...unterschiedliche Sichtweisen und Akzentsetzungen.

Mathematik: Betonung der 'Teile' – eine Funktion der Form:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

wird **zweistellig** angesehen: $\binom{\cdot}{\cdot} : \mathbb{IN} \times \mathbb{IN} \rightarrow \mathbb{IN}$

Allgemein: $f : M_1 \times \dots \times M_n \rightarrow M$ hat Stelligkeit n .

Haskell: Betonung des 'Ganzen' – eine Funktion der Form:

```
type I = Integer
```

```
binom' (n,k)
```

```
  | k==0 || n==k = 1
```

```
  | otherwise = binom' (n-1,k-1) + binom' (n-1,k)
```

wird **einstellig** angesehen: $\text{binom}' :: (I,I) \rightarrow I$

Allgemein: $f :: (M_1, \dots, M_n) \rightarrow M$ hat Stelligkeit 1 .

Zusammenfassung (1)

Für Haskell gilt:

Die Klammerung unvollständig geklammerter

- Funktionssignaturen ist rechtsassoziativ
- Funktionsterme ist linksassoziativ

zu vervollständigen.

Funktionen sind

- einstellig; sie konsumieren stets ein Argument zur Zeit.

Argumente und Werte von Funktionen und Funktionstermen

- können elementaren, zusammengesetzten oder funktionalen Typs sein.

Zusammenfassung (2)

...exakte vs. saloppe Lesart für curryfizierte Funktionen.

Exakt: `binom` ist eine 1-stellige Funktion:

```
binom :: Integer -> (Integer -> Integer)
binom n = g where g k = if ... then ... else ...
:: (Integer -> Integer)
```

...die ganze Zahlen als Argument auf 1-stellige Funktionen abbildet, die ganze Zahlen auf ganze Zahlen abbilden.

Salopp: `binom` ist eine 2-stellige Funktion:

```
binom :: Integer -> Integer -> Integer
binom n k = k' where k' = if ... then ... else ...
:: Integer
```

...die Paare ganzer Zahlen als Argument auf ganze Zahlen abbildet.

Kapitel 3.3

Curryfizierte, uncurryfizierte Funktionen

Vortrag II

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Curryfiziert und uncurryfiziert

...bezeichnen bestimmte ineinander überführbare Deklarationsweisen für Funktionen.

Entscheidend für die Unterscheidung ist die

- Art der Konsumation der Argumente.

Erfolgt die Konsumation

- Einzeln Argument für Argument: **curryfiziert**
- Alle auf einmal als Tupel: **uncurryfiziert**

Implizit liefert dies eine Unterscheidung in

- **curryfizierte** Funktionen
- **uncurryfizierte** Funktionen

Ein Beispiel

...die Funktionen `binom` und `binom'`:

– `binom :: Integer -> Integer -> Integer`

...ist **curryfiziert** deklariert.

– `binom' :: (Integer,Integer) -> Integer`

...ist **uncurryfiziert** deklariert.

Das Beispiel: Vollständig ausformuliert

...Funktion `binom`, `curryfiziert` deklariert:

```
binom :: Integer -> Integer -> Integer
```

```
binom n k
```

```
| k==0 || n==k = 1
```

```
| otherwise = binom (n-1) (k-1) + binom (n-1) k
```

```
binom 45 6 ->> (binom 45) 6 ->> 8.145.060
```

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

...Funktion `binom'`, `uncurryfiziert` deklariert:

```
binom' :: (Integer,Integer) -> Integer
```

```
binom' (n,k)
```

```
| k==0 || n==k = 1
```

```
| otherwise = binom' (n-1,k-1) + binom' (n-1,k)
```

```
binom' (45,6) ->> 8.145.060
```

```
      :: Integer
```

Informell

Curryfizieren ersetzt

- Produkt-/Tupelbildung ' \times ' durch Funktionspfeil ' \rightarrow '.

Uncurryfizieren ersetzt

- Funktionspfeil ' \rightarrow ' durch Produkt-/Tupelbildung ' \times '.

Bemerkung: Die Bezeichnung erinnert an Haskell B. Curry; die Idee selbst ist älter und geht auf Moses Schönfinkel und die Mitte der 1920er-Jahre zurück.

Die Funktionale `curry` und `uncurry`

...die Mittler zwischen `curryfizzierter` und `uncurryfizzierter` Darstellung von Funktionen:

Das Funktional `curry`:

`curry` :: $\underbrace{((a,b) \rightarrow c)}$ \rightarrow $\underbrace{(a \rightarrow b \rightarrow c)}$
Argumenttyp von `curry`: Resultattyp von `curry`:
uncurryfiziert! curryfiziert!

`curry` `f` = `g`
where `g x y = f (x,y)`

Das Funktional `uncurry`:

`uncurry` :: $\underbrace{(a \rightarrow b \rightarrow c)}$ \rightarrow $\underbrace{((a,b) \rightarrow c)}$
Argumenttyp von `uncurry`: Resultattyp von `uncurry`:
curryfiziert! uncurryfiziert!

`uncurry` `g` = `f`
where `f (x,y) = g x y`

Beachte: Auflösen der where-Klausel

...liefert kürzere Funktionsdefinitionen.

Statt:

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

```
curry f = g where g x y = f (x,y)
```

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

```
uncurry g = f where f (x,y) = g x y
```

können wir bedeutungsgleich kürzer schreiben:

```
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
```

Die Implementierungen v. `curry` u. `uncurry` (1)

Das Funktional `curry`:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)    -- x, y wird zu (x,y)
                        -- zusammengesetzt und so
                        -- für f verarbeitbar
```

Das Funktional `uncurry`:

```
uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry g (x,y) = g x y  -- (x,y) wird in x, y
                        -- getrennt und so
                        -- für g verarbeitbar
```


Die Implementierungen v. **curry** u. **uncurry** (2)

...in größerem Detail:

Das Funktional **curry**:

$$\begin{array}{l} \text{curry} :: ((a,b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c) \\ \text{curry } f \quad \quad \quad x \quad y = f(x,y) \\ \underbrace{\text{:: } ((a,b) \rightarrow c)} \quad \underbrace{\text{:: } a} \quad \underbrace{\text{:: } b} \quad \underbrace{\text{:: } (a,b)} \\ \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \underbrace{\text{:: } c} \end{array}$$

Das Funktional **uncurry**:

$$\begin{array}{l} \text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow ((a,b) \rightarrow c) \\ \text{uncurry } g \quad \quad \quad (x,y) = g \quad x \quad y \\ \underbrace{\text{:: } (a \rightarrow b \rightarrow c)} \quad \underbrace{\text{:: } (a,b)} \quad \quad \quad \underbrace{\text{:: } a} \quad \underbrace{\text{:: } b} \\ \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \underbrace{\text{:: } c} \end{array}$$

curry: Schritt für Schritt zur Definition

$$\text{curry} :: ((a,b) \rightarrow c) \rightarrow (a \rightarrow (b \rightarrow c))$$
$$\text{curry } f \ x \ y = f \ (x,y)$$

Sei f eine Funktion mit Signatur

$$f :: ((a,b) \rightarrow c)$$

Mit f erhalten wir für die Signaturen der Funktionsterme:

$$\text{curry} :: ((a,b) \rightarrow c) \rightarrow (a \rightarrow (b \rightarrow c))$$
$$(\text{curry } f) :: (a \rightarrow (b \rightarrow c))$$
$$((\text{curry } f) \ x) :: (b \rightarrow c)$$
$$(((\text{curry } f) \ x) \ y) :: c$$

Entsprechend erhalten wir für den Typ des rechtss. Fkt-Terms:

$$(((\text{curry } f) \ x) \ y) = f \ (x,y) :: c$$

Nach Einsparung von Klammern erhalten wir insgesamt:

$$\text{curry} :: ((a,b) \rightarrow c) \rightarrow (a \rightarrow (b \rightarrow c))$$
$$\text{curry } f \ x \ y = f \ (x,y)$$

Anwendungen von `curry` und `uncurry`

Betrachte:

```
binom :: Integer -> Integer -> Integer
```

```
binom' :: (Integer,Integer) -> Integer
```

und

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

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

Anwendung von `curry` und `uncurry` liefert:

```
curry binom' :: Integer -> Integer -> Integer
```

```
uncurry binom :: (Integer,Integer) -> Integer
```

Somit sind folgende Aufrufe möglich und gültig:

```
curry binom' 45 6    ->> binom' (45,6) ->> 8.145.060
```

```
uncurry binom (45,6) ->> binom 45 6   ->> 8.145.060
```

Curryfiziert oder uncurryfiziert?

...das ist die Frage.

Geschmackssache? Notationelle Spielerei?

- $f\ x, f\ x\ y, f\ x\ y\ z, \dots$ vs. $f(x), f(x,y), f(x,y,z), \dots$
Allenfalls bei oberflächlicher Betrachtung.

Denn es gilt: Nur **curryfizierte** Funktionen unterstützen das

- Prinzip partieller Auswertung und damit das Prinzip:
 \rightsquigarrow **Funktionen liefern Funktionen als Ergebnis!**

Beispiel: Die für das Argument 45 partiell ausgewertete Funktion `binom` liefert als Resultat eine einstellige Funktion, die Funktion `k_aus_45 :: Integer -> Integer` definiert durch `k_aus_45 = (binom 45)`.

Die Bevorzugung **curryfizierter** Formen ist deshalb sachlich gut begründet, vorteilhaft und in der Praxis vorherrschend.

Faustregel: Funktionsdefinitionen

...curryfiziert, wo möglich, uncurryfiziert nur dort, wo nötig.

Vortrag II

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Kapitel 3.4

Operatoren, Präfix- und Infixverwendung

Vortrag II

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Unterschiedliche Operatorverwendungsarten

Präfix

- Operator ist den Operanden vorangestellt:

Beispiele: `fac 5`, `binom (45,6)`, `reverse "desserts"`,
`quickSort [4,2,1,9,3,7,5]`,...

Infix

- Operator ist zwischen die Operanden gestellt:

Beispiele: `2 + 3`, `5 * 7`, `5 ^ 3`, `4 : [3,2,1]`, `[1,2,3,4] !! 2`,
`[3,2,1] ++ [1,2,3]`,...

Postfix

- Operator ist den Operanden nachgestellt:

Beispiele: In Haskell keine; in der Mathematik wenige, etwa die Fakultätsfunktion “!”; regelmäßig bei Verwendung “umgekehrt polnischer Notation”.

In Haskell

...ist **Präfixverwendung**

- ▶ Regelfall, insbesondere für alle selbstdeklarierten Operatoren (d.h. selbstdeklarierte Funktionen).

Beispiele:

- Vordefinierte Funktionen: `div`, `reverse`, `zip`,...
- Selbstdefinierte Funktionen: `fac`, `binom`, `quicksort`,...

...ist **Infixverwendung**

- ▶ Regelfall für einige vordefinierte Operatoren und Relatoren, darunter viele arithmetische Operatoren und Relatoren.

Beispiele: `2 + 3`, `5 * 7`, `5 ^ 3`, `4 : [3,2,1]`, `[1,2,3,4] !! 2`,
`[3,2,1] ++ [1,2,3]`, `[3,2,1] == [1,2,3]`,
`4 <= 5`, `"Fun" < "More Fun"`,...

Für binäre Operatoren

...ist in Haskell **Infix-** und **Präfixverwendung** möglich, gleich ob

- ▶ vor- oder selbstdefiniert.

Allgemein: Wird der Binäroperator `bop` im Regelfall als

- ▶ **Präfixoperator** verwendet, so kann `bop` mit Hochkommata als **Infixoperator** `'bop'` verwendet werden.

Beispiele: `45 'binom' 6, 3 'mult' 5`
(statt standardmäßig: `binom 45 6, mult 3 5`)

- ▶ **Infixoperator** verwendet, so kann `bop` geklammert als **Präfixoperator** `(bop)` verwendet werden.

Beispiele: `(+) 2 3, (++) [3,2,1] [1,2,3]`
(statt standardmäßig: `2 + 3, [2,1] ++ [1,2]`)

Beispiel

...berechne das **Maximum dreier ganzer Zahlen**:

```
maximum :: Int -> Int -> Int -> Int
```

```
maximum p q r
```

```
| (mx p q == p) && (p 'mx' r == p) = p
```

```
| (mx p q == q) && (q 'mx' r == q) = q
```

```
| otherwise = r
```

```
where mx :: Int -> Int -> Int
```

```
    mx p q
```

```
    | p >= q = p
```

```
    | otherwise = q
```

Beachte: Der Binäroperator `mx` wird in `maximum` als **Präfixoperator** (`mx p q`) und **Infixoperator** (`p 'mx' r`) verwendet.

Kapitel 3.5

Operatorabschnitte

Vortrag II

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Operatorabschnitte (1)

Partiell ausgewertete Binärooperatoren heißen in Haskell

- ▶ Operatorabschnitte (engl. *operator sections*)

Beispiele:

- (*2) `dbl`, die Funktion, die ihr Argument verdoppelt
($\lambda x. x * 2$)
- (2*) `dbl`, s.o. ($\lambda x. 2 * x$)
- (<2) `x_kleiner_als_2`, das Prädikat, das überprüft, ob sein Argument kleiner als 2 ist ($\lambda x. x < 2$)
- (2<) `2_kleiner_als_x`, das Prädikat, das überprüft, ob 2 kleiner als sein Argument ist ($\lambda x. 2 < x$)
- (2:) `headAppend`, die Funktion, die 2 an den Anfang einer typkompatiblen Liste setzt ($\lambda xs. (2 : xs)$)
- ...

Operatorabschnitte (2)

Beispiele (f.g.s.):

- (+1), (1+) `inc`, die Funktion, die ihr Argument um 1 erhöht ($\lambda x. x + 1$) bzw. ($\lambda x. 1 + x$)
- (1-) `eins_minus`, die Funktion, die ihr Argument von 1 abzieht ($\lambda x. 1 - x$)
- (-1) kein Operatorabschn., sondern d. Zahl '-1'.
- (+(-1)) `dec`, die Funktion, die ihr Argument um 1 erniedrigt ($\lambda x. x + (-1)$) bzw. ($\lambda x. x - 1$)
- ('div' 2) `hlv`, die Funktion, die ihr Argument ganzzahlig halbiert ($\lambda x. x \text{ div } 2$)
- (2 'div') `zwei_durch`, die Funktion, die 2 ganzzahlig durch ihr Argument teilt ($\lambda x. 2 \text{ div } x$)
- ...
- (div 2),
div 2 `zwei_durch`, s.o. ($\lambda x. 2 \text{ div } x$); keine echten Operatorabschnitte, sondern gewöhnliche Präfixoperatorverwendung.

Operatorabschnitte (3)

Operatorabschnitte können in Haskell gebildet werden mit

1. vordefinierten
2. selbstdefinierten

binären Operatoren.

Beispiele für die curryfizierte Funktion `binom` (vgl. Kap. 3.2):

- `(binom 45)` `45_über_k`, die Funktion `k_aus_45`.
- `(45 'binom')` `45_über_k`, s.o.
- `('binom' 6)` `n_über_6`, die Funktion `6_aus_n`.
- ...

Beachte: Mit der uncurryfizierten Funktion `binom'` (vgl. Kapitel 3.2) können keine Operatorabschnitte gebildet werden.

Anwendung: Punktfreie, argumentlose

...Funktionsdefinitionen mit Operatorabschnitten:

- `45_über_k` bzw. `k_aus_45`
`k_aus_45 :: Integer -> Integer`
`k_aus_45 = binom 45`
`k_aus_45 :: Integer -> Integer`
`k_aus_45 = (45 'binom')`
- `n_über_6` bzw. `6_aus_n`
`sechs_aus_n :: Integer -> Integer`
`sechs_aus_n = ('binom' 6)`
- **Inkrement**
`inc :: Integer -> Integer`
`inc = (+1)`
- **Verdoppeln**
`dbl :: Integer -> Integer`
`dbl = (2*)`
- ...

Nichtkommutative Operatoren

...benötigen Obacht bei der **Bildung von Operatorabschnitten**.

Infix- und Präfixbenutzung hat für **nichtkommutative Operatoren** einen **Bedeutungsunterschied**. Am Beispiel von `div`:

- ▶ **Infix**verwendung führt zu den Funktionen `hlv` und `zwei_durch`.
- ▶ **Präfix**verwendung führt zur Funktion `zwei_durch`.

bei ansonsten gleicher partieller Auswertung.

Zusammenfassung

...**Operatorabschnitte**: Notationelle Abkürzungen ('syntaktischer Zucker') für **anonyme λ -Abstraktionen**.

Im Detail: Ist **op** ein Binäroperator und sind **x** und **y** typgeeignete Operanden für **op**, dann heißen die Ausdrücke:

- **(op)**, **(x op)**, **(op y)**

Operatorabschnitte, die für folgende Funktionen stehen:

- **(op)** = $(\lambda x. (\lambda y. x \text{ op } y))$
- **(x op)** = $(\lambda y. x \text{ op } y)$
- **(op y)** = $(\lambda x. x \text{ op } y)$

und somit besonders knappe Funktionsdefinitionen erlauben (Sonderfall: Der Subtraktionsoperator **(-)**).

Kapitel 3.6

Angemessene, unangemessene Funktionsdefinitionen

Vortrag II

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Funktionen

...wie die **Fakultäts-** und **Fibonacci-Funktion** sind (auf den natürlichen Zahlen) total definiert:

$$! : \mathbb{IN} \rightarrow \mathbb{IN}$$

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

$$fib : \mathbb{IN} \rightarrow \mathbb{IN}$$

$$fib(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ fib(n - 2) + fib(n - 1) & \text{sonst} \end{cases}$$

Ihre naheliegenden Implementierungen

...in einer Programmiersprache sind hingegen häufig nur partiell definiert. So terminieren folgende Implementierungen **nur für nichtnegative Argumente** und sind **für negative Argumente nicht definiert**:

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n - 1)

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

In Programmiersprachen sind **total definierte Funktionen** die Ausnahme, **partiell definierte Funktionen** die Regel.

Guter Prog.-Stil: Offenlegen der Partialität

- ...die Partialität der Implementierungen `fac` und `fib` ist
- i.w. **technisch induziert** (Abwesenheit eines Datentyps für natürliche Zahlen).

Explizite, transparente Sichtbarmachung der Partialität ist jedoch **sinnvoll, angemessen** und auch **einfach möglich**:

```
fac :: Int -> Int
fac n | n == 0 = 1
      | n >= 1 = n * fac (n - 1)
      | otherwise = error "undefiniert"
```

```
fib :: Int -> Int
fib n | n == 0 = 1
      | n == 1 = 1
      | n >= 2 = fib (n-2) + fib (n-1)
      | otherwise = error "undefiniert"
```

Auch die Funktionen f , g und h

...sind **partiell** definiert:

$$f : \mathbb{Z} \rightarrow \mathbb{Z}$$

$$f(z) = \begin{cases} 2 & \text{falls } z \geq 1 \\ \text{undef} & \text{sonst} \end{cases}$$

$$g : \mathbb{Z} \rightarrow \mathbb{Z}$$

$$g(z) = \begin{cases} 2^z & \text{falls } z \geq 1 \\ \text{undef} & \text{sonst} \end{cases}$$

$$h : \mathbb{Z} \rightarrow \mathbb{Z}$$

$$h(z) = \begin{cases} 2^1 & \text{falls } z = 1 \\ 2^{(|z|+2)} & \text{falls } z \leq 0 \\ \text{undef} & \text{sonst} \end{cases}$$

Wir können sie angemessen implementieren

...so dass die Partialität der Implementierungen transparent und offen zutagelegt:

```
f :: Integer -> Integer
f z | z >= 1      = 2
    | otherwise = error "undefiniert"
```

```
g :: Integer -> Integer
g z | z >= 1      = 2^z
    | otherwise = error "undefiniert"
```

```
h :: Integer -> Integer
h z | z == 1      = 2
    | z <= 0      = 2^((abs z)+2)
    | otherwise = error "undefiniert"
```

...oder unangemessen, so dass das nicht gilt:

Betrachte dazu folgende **intransparente**, die **Partialität verschleiende Implementierung** von **f**:

```
f :: Integer -> Integer
f 1 = 2
f x = 2 * (f x)
```

Auch wenn man sich durch Nachrechnen vergewissern kann, dass die **Auswertung** von **f** terminiert für **n = 1**:

```
f 1 ->> 2
```

...aber **für keinen** von **1** verschiedenen Argumentwert, ist die Implementierung diesbezüglich **intransparent** und **verschleiend**.

Auswertungsbeispiele für f

Die **Auswertung** von f terminiert für $n = 1$:

$f\ 1 \quad \rightarrow 2$

Die **Auswertung** von f terminiert nicht für $n \neq 1$:

$f\ (-9) \rightarrow 2 * (f\ (-9)) \rightarrow 2 * (2 * (f\ (-9)))$
 $\rightarrow 2 * (2 * (2 * (f\ (-9)))) \rightarrow \dots$

$f\ (-1) \rightarrow 2 * (f\ (-1)) \rightarrow 2 * (2 * (f\ (-1)))$
 $\rightarrow 2 * (2 * (2 * (f\ (-1)))) \rightarrow \dots$

$f\ 0 \rightarrow 2 * (f\ 0) \rightarrow 2 * (2 * (f\ 0))$
 $\rightarrow 2 * (2 * (2 * (f\ 0))) \rightarrow \dots$

$f\ 2 \rightarrow 2 * (f\ 2) \rightarrow 2 * (2 * (f\ 2))$
 $\rightarrow 2 * (2 * (2 * (f\ 2))) \rightarrow \dots$

$f\ 3 \rightarrow 2 * (f\ 3) \rightarrow 2 * (2 * (f\ 3))$
 $\rightarrow 2 * (2 * (2 * (f\ 3))) \rightarrow \dots$

$f\ 9 \rightarrow 2 * (f\ 9) \rightarrow 2 * (2 * (f\ 9))$
 $\rightarrow 2 * (2 * (2 * (f\ 9))) \rightarrow \dots$

Kapitel 3.7

Funktions- und Programmformatierung, Abseitsregel

Vortrag II

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Programmformatierung, Programmbedeutung

Für die meisten Programmiersprachen gilt:

- ▶ Die Formatierung (engl. layout) des Programmtexts beeinflusst
 - seine Lesbarkeit, Verständlichkeit, Wartbarkeit
 - aber nicht seine Bedeutung

Nicht so für Haskell – für Haskell gilt:

- ▶ Die Formatierung des Programmtexts trägt Bedeutung!

Dieser Aspekt des Sprachentwurfs

- ist für Haskell grundsätzlich anders entschieden worden als für Sprachen wie Java, Pascal, C und viele andere.
- ersetzt `begin/end`- oder `{/}`-Paare durch Formatierungsanforderungen.
- kann als Reminiszenz an Sprachen wie Cobol, Fortran gesehen werden, findet sich aber auch in anderen neueren Sprachen wie z.B. occam, Miranda, Curry und anderen.

Graphische Veranschaulichung

```
type Radius      = Float -- Radius, Oberflaeche, Volumen und pi
type Oberflaeche = Float -- global im Gesamtprogramm sichtbar
type Volumen     = Float
pi = 3.14 :: Float
```

```
-----
| -- kugel_OV global im Gesamtprogramm sichtbar
kugel_OV :: Radius -> (Oberflaeche,Volumen)
kugel_OV =
|(oberflaeche r,
|  volumen r)
|
| -----
|   | -- oberflaeche lokal in kugelOV sichtbar
|   where oberflaeche :: Radius -> Oberflaeche
|           oberflaeche r = 4 * pi *
|             square r
|           ----->
|           -----
|           | -- volumen lokal in kugelOV sichtbar
|           volumen :: Radius -> Volumen
|           volumen r = (4/3)
|             * pi * cubic r
|           -----
|           | -- cubic lokal in volumen sichtbar
|           | where cubic x = x * square x
|           | ----->
|           | ----->
|           ----->
| ----->
|
| -----
| -- square global im Gesamtprogramm sichtbar
square :: Float -> Float
square x = x^2
| ----->
```

Vortrag II

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

kugel_OV korrekt, aber 'unschön' formatiert

```
type Radius      = Float
type Oberflaeche = Float
type Volumen     = Float

pi = 3.14 :: Float

kugel_OV :: Radius -> (Oberflaeche,Volumen)
kugel_OV r =
  (oberflaeche r,
   volumen r)
  where oberflaeche :: Radius -> Oberflaeche
        oberflaeche r = 4 * pi *
          square r
        volumen :: Radius -> Volumen
        volumen r = (4/3)
          * pi * cubic r
          where cubic x = x * square x

square :: Float -> Float
square x = x^2
```

Vortrag II

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Bewährte Formatierungskonventionen (1)

...zur Einhaltung der Abseitsregel für Funktionsdefinitionen:

```
funktionsName parameter_1 parameter_2... parameter_n
| waechter_1 = ausdruck_1
| waechter_2 = ausdruck_2
...
| otherwise = ausdruck_k
where
v_1 a_1 ... a_n = r_1      -- v_1, v_2,..., sichtbar
v_2                = r_2  -- in der gesamten Funk-
...                    -- tion funktionsName,
                        -- aber nicht außerhalb.
```

Vortrag II

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Umgekehrtes
Klassenzimmer I

Hinweis

Aufgabe

Bewährte Formatierungskonventionen (2)

...für lange Bedingungen und Ausdrücke:

```
funktionsName parameter_1 parameter_2... parameter_n
| waechter_1 = ausdruck_1
| waechter_2 = ausdruck_2
| diesIsteineGanz
  BesondersLangeMehrzeilige
    BedingungAlsWaechter
      = diesIstEinBesonders
        LangerMehrzeiliger
          AusdruckZurWertfestlegung
| waechter_4 = ausdruck_4
...
| otherwise = ausdruck_k
where...
```

Vortrag II

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Sprachkonstrukt- und Formatierungswahl

...nach **Angemessenheitserwägungen**:

- ▶ Was ist gut und einfach lesbar und verständlich?

Zwei Bsp.: Drei Implementierungen für fib

1. Mit Mustern:

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

2. Mit bewachten Ausdrücken:

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

3. Mit geschachtelten bedingten Ausdrücken:

```
fib :: Int -> Int
fib n = if n == 0 then 1
        else if n == 1 then 1
             else fib (n-2) + fib (n-1)
```

Drei Implementierungen für maximum

1. Mit **anonymer λ -Abstraktion** u. **geschachtelten bedingten Ausdrücken**:

```
maximum :: Int -> Int -> Int -> Int
maximum = \p q r -> if p>=q then (if p>=r then p else r)
                    else (if q>=r then q else r)
```

2. Mit **geschachtelten bedingten Ausdrücken**:

```
maximum :: Int -> Int -> Int -> Int
maximum p q r = if (p>=q) && (p>=r) then p
                else if (q>=p) && (q>=r) then q else r
```

3. Mit **bewachten Ausdrücken**:

```
maximum :: Int -> Int -> Int -> Int
maximum p q r
  | (p>=q) && (p>=r) = p
  | (q>=p) && (q>=r) = q
  | otherwise      = r
```

Programme können grundsätzlich
auf zwei Arten geschrieben werden:

So **einfach**, dass sie **offensichtlich keinen** Fehler enthalten;
so **kompliziert**, dass sie **keinen offensichtlichen** Fehler enthalten.

C.A.R. 'Tony' Hoare (* 1934)
Turing Award Preisträger 1980

Gut gewählte Sprachkonstrukte u. gut gewählte Formatierung

- unterstützen dabei, Programme **einfach** und **offensichtlich fehlerfrei** zu schreiben (vgl. [Kap. 3.6](#))!

In Haskell heißt das

- 'schönes' Einrücken und zumeist die Verwendung **bewachter Ausdrücke** und **Muster** anstelle (**geschachtelter**) **bedingter Ausdrücke** (vgl. [Kap. 3.1](#)).

Kapitel 3.8

Leseempfehlungen

Vortrag II

Teil II

Kap. 2

Kap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

3.8

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I




Hinweis

Aufgabe

Basiselesempfehlungen für Kapitel 3

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 3, Funktionen und Operatoren; Kapitel 4, Rekursion als Entwurfstechnik)
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 3.5, Function types; Kapitel 3.6, Curried functions; Kapitel 4, Defining functions; Kapitel 6, Recursive functions)
-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 3, Syntax in Functions; Kapitel 4, Hello Recursion!)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 7, Defining functions over lists)

Weiterführende Leseempfehlungen für Kap. 3

-  Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 2, Expressions, types and values)
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 4, Functional Programming - Partial Function Application and Currying)
-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 6, Ein bisschen syntaktischer Zucker)

Kapitel 4

Typsynonyme, Neue Typen, Typklassen

Kapitel 4.1

Typsynonyme

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Wozu? Was 'macht' Funktion f?

```
f :: Float -> Float -> Float -> Float -> Float
f u x y z = u * x - (max ((z/100)*u) y)
```

Warum ist es schwer, die **Bedeutung** von `f` zu erkennen? Kein Name **'spricht'**!

Als **Verständnishilfe(n)** können wir einführen:

1. 'Sprechende' Funktionsnamen

```
wir_kaufen :: Float -> Float -> Float -> Float -> Float
wir_kaufen u x y z = u * x - (max ((z/100)*u) y)
```

2. 'Sprechende' Parameternamen:

```
wir_kaufen :: Float -> Float -> Float -> Float -> Float
wir_kaufen dollar eurowechselkurs zinsfuss mindestgebuehr
= dollar * eurowechselkurs
  - (max ((zinsfuss/100)*dollar) mindestgebuehr)
```

Offen bleibt: Der Typname `Float` **'spricht' nicht!**

In Haskell können wir diese Lücke schließen

...und als **Verständnishilfe** auch **'sprechende'** Typnamen einführen:

```
type Dollar           = Float
type Eurowechselkurs = Float
type Zinsfuss         = Float
type Mindestgebuehr  = Float
type Unser_Euro_Anbot = Float

wir_kaufen :: Dollar -> Eurowechselkurs -> Zinsfuss
              -> Mindestgebuehr -> Unser_Euro_Anbot

wir_kaufen dollar eurowechselkurs zinsfuss mindestgebuehr
= dollar * eurowechselkurs
  - (max ((zinsfuss/100)*dollar) mindestgebuehr)
```

Dank 'sprechender' Typnamen

...sog. **Typsynonyme**, reichen jetzt auch **kürzere Parameternamen**, ohne die Verständlichkeit zu beeinträchtigen:

```
wir_kaufen :: Dollar -> Eurowechselkurs -> Zinsfuss
             -> Mindestgebuehr -> Unser_Euro_Anbot
wir_kaufen d ewk zf mgb = d * ewk - (max ((zf/100)*d) mgb)
```

Wie funktionieren, was leisten Typsynonyme?

1. Wie funktionieren Typsynonyme?

- Typsynonyme werden eingeführt mit Schlüsselwort `type`:
`type <Alias-Name> = <existierender Typname>`

2. Was leisten Typsynonyme? Was nicht?

- Typsynonyme sind neue Namen, Aliasnamen für bereits existierende Typen, keine neuen Typen.
- Typ und Typsynonym dürfen sich wechselseitig vertreten!
- Typsynonyme führen (deshalb) nicht zu höherer Typsicherheit!

Stattdessen: Gut (!) gewählte Typsynonyme erhöhen gleichsam treffenden Kommentaren

- Lesbarkeit, Verständlichkeit

von Programmen für menschliche Leser!

Ein Antibeispiel: Eine formal korrekte, aber

...inhaltlich wenig sinnvolle Verwendung von Typsynonymen:

```
type Celsius = Float
type Meile   = Float
type Watt    = Float
type Lumen   = Float
```

```
c = 47.11 :: Celsius
m = 19.76 :: Meile
w = 17.4  :: Watt
l = 810.0 :: Lumen
```

wir_kaufen c m w l (Wohldefiniert! Kein Typfehler!)

```
->> wir_kaufen 47.11 19.76 17.4 810.0
->> 47.11 * 19.76 - (max ((17.4/100)*47.11) 810)
->> 930.8936 - (max 8.19714 810.0)
->> 930.8936 - 810.0
->> 120.8936
```

Zusammenfassung

Typsynonyme in Haskell (und anderen fkt. Sprachen)

- **erlauben** neue, sprechende Typnamen, sog. Typsynonyme, Typalias oder Typaliasnamen einzuführen.
- **erlauben nicht**, originär neue, bislang nicht existierende Typen einzuführen.
- **helfen** Haskell-Programme durch sprechende Namen für menschliche Leser einfacher lesbar und verständlich zu machen.
- **führen nicht** zu höherer Typsicherheit.

...sind bei

- **guter (!) Namenswahl**

unaufwändige, gute Programmkommentare!

Syntaktisch suggestivere Umsetzungen

...der **Typsynonymidee** hätten sein können:

```
synonyms_of Float = Dollar, Eurowechselkurs, Zinsfuss,  
                  Mindestgebuehr, Unser_Euro_Anbot,  
                  Celsius, Meile, Watt, Lumen
```

oder:

```
type Float aka Dollar, Eurowechselkurs, Zinsfuss,  
            Mindestgebuehr, Unser_Euro_Anbot,  
            Celsius, Meile, Watt, Lumen
```

oder wenigstens:

```
type Dollar, Eurowechselkurs, Zinsfuss, Mindestgebuehr,  
    Unser_Euro_Anbot, Celsius, Meile, Watt, Lumen = Float
```

statt wie in **Haskell** tatsächlich:

```
type Dollar           = Float  
type Eurowechselkurs = Float  
type Zinsfuss         = Float  
...
```

Drei komplexere Typsynonymbeispiele

► Tupeltypsynonyme für:

1. Studentendaten
2. Buchhandelsdaten

zusammen mit nützlichen Selektorfunktionen (kurz: Selektoren) unter Präfixverwendung der Tupelkonstruktoren (,,,) und (,,,,) zur Kreierung von Vier- und Fünftupeln.

► Abbildungstypsynonyme für:

3. Buchkäufe

zusammen mit datenbankabfrageähnlichen Funktionen.

Typsynonyme + Selektoren f. Studentendaten

Typsynonyme:

```
type Vorname      = String
type Nachname     = String
type Email        = String
type Studienkennzahl = Int
type Skz          = Studienkennzahl
type Student      = (Vorname, Nachname, Email, Skz)
```

Ein Student-Wert:

```
(,,,) "Max" "Mux" "e123456@stud.tuw.ac.at" 534
->> ("Max", "Mux", "e123456@stud.tuw.ac.at", 534) :: Student
```

Selektoren:

```
vorname :: Student -> Vorname           (Ausschließlich
vorname (v,n,e,k) = v                   Variablenmuster)
nachname :: Student -> Nachname
nachname (v,n,e,k) = n
email :: Student -> Email
email (v,n,e,k) = e
skz :: Student -> Studienkennzahl
skz (v,n,e,k) = k
```

Typsynonyme + Selektoren f. Buchhandelsdat.

Typsynonyme:

```
type Autor    = String
type Titel    = String
type Auflage  = Int
type Jahr     = Int
type Lagernd  = Bool
type Buch     = (Autor, Titel, Auflage, Jahr, Lagernd)
```

Ein Buch-Wert: (,,,) "S. Thompson" "Haskell" 3 2011 True

Selektoren: ->> ("S. Thompson", "Haskell", 3, 2011, True) :: Buch

```
autor :: Buch -> Autor           (Variablenmuster, wo nö-
autor (a,_,_,_,_) = a           tig, sonst 'wild card')
titel :: Buch -> Titel
titel (_,t,_,_,_) = t
auflage :: Buch -> Auflage
auflage (_,_,a,_,_) = a
erschieden :: Buch -> Jahr
erschieden (_,_,_,j,_) = j
lagernd :: Buch -> Lagernd
lagernd (_,_,_,_,l) = l
```

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.4

Kap. 5

Kap. 6

Umgekehrt
Klassen-
zimmer I

Hinweis

Aufgabe

Typsynonyme + Abfragefkt. für Buchkäufe

Typsynonyme:

```
type Kaufte      = Student -> [Buch]
type Verkauft_an = Buch    -> [Student]
```

Zwei datenbankähnliche Abfragefunktionen:

```
gib_titel_aller_buecher_gekauft_von :: Kaufte -> Student
                                     -> [Titel]
```

```
gib_titel_aller_buecher_gekauft_von kaufte stud
= [titel buch | buch <- kaufte stud]
```

```
gib_alle_kaeufer_von_Buch :: Verkauft_an -> Buch
                           -> [(Nachname,Email)]
```

```
gib_alle_kaeufer_von_Buch verkauft_an buch
= [(nachname stud, email stud) | stud <- verkauft_an buch]
```

...demonstrieren das Rechnen mit Funktionen und die Nützlichkeit von Listenkomprehensionen!

Kapitel 4.2

Neue Typen

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.4

Kap. 5

Kap. 6

Umgekehr

Klassen-

zimmer I

Hinweis

Aufgabe

Wozu? Um Typsicherheit zu erreichen!

...um 'Unfug' wie vorher besprochen auszuschließen:

```
type Celsius = Float
```

```
type Meile    = Float
```

```
type Watt     = Float
```

```
type Lumen    = Float
```

```
c = 47.11 :: Celsius
```

```
m = 19.76 :: Meile
```

```
w = 17.4  :: Watt
```

```
l = 810.0 :: Lumen
```

```
wir_kaufen c m w l
```

```
->> wir_kaufen 47.11 19.76 17.4 810.0
```

```
->> 47.11 * 19.76 - (max ((17.4/100)*47.11) 810)
```

```
->> 930.8936 - (max 8.19714 810.0)
```

```
->> 930.8936 - 810.0
```

```
->> 120.8936
```

...der Aufruf ist wohldefiniert und liefert keinen Typfehler!

Typsicherheit durch Neue Typen

...eingeführt durch `newtype`-Deklarationen:

```
newtype Dollar           = USD Float
```

```
newtype Eurowechselkurs = Ewk Float
```

```
newtype Zinsfuss        = Zfs Float
```

```
newtype Mindestgebuehr  = Mgb Float
```

```
newtype Unser_Euro_Anbot = UEA Float
```

```
wir_kaufen :: Dollar -> Eurowechselkurs -> Zinsfuss  
            -> Mindestgebuehr -> Unser_Euro_Anbot
```

```
wir_kaufen (USD x) (Ewk y) (Zfs z) (Mgb u)  
  = UEA (x * y - (max ((z/100)*x) u))
```

Deklaration Neuer Typen

Grundmuster:

```
newtype <freigewählter Typbezeichner>  
      = <freigewählter (Datenwert-) Konstruktorbez.>  
      <Bezeichner eines existierenden (!) Typs>
```

Beispiel:

```
newtype Dollar      = USD      Float  
      { Typbezeichner Konstruktor Exist. Typ }  
newtype Zinsfuss    = Zfs      Float  
      { Typbezeichner Konstruktor Exist. Typ }
```

(Datenwert-) Konstruktoren sind Funktionen:

```
USD :: Float -> Dollar
```

```
USD 4.2 :: Dollar
```

```
Zfs :: Float -> Zinsfuss
```

```
Zfs 4.2 :: Zinsfuss
```

newtype gibt Typsicherh., verhindert Unfug!

```
type Celsius = Float
type Meile   = Float
type Watt    = Float
type Lumen   = Float
c = 47.11 :: Celsius
m = 19.76 :: Meile
w = 17.4   :: Watt
l = 810.0  :: Lumen
```

wir_kaufen c m w l ->> Typfehler in Anwendung!

```
newtype Celsius = C Float
newtype Meile   = M Float
newtype Watt    = W Float
newtype Lumen   = L Float
```

```
c' = C 47.11 :: Celsius
m' = M 19.76 :: Meile
w' = W 17.4   :: Watt
l' = L 810.0  :: Lumen
```

wir_kaufen c' m' w' l' ->> Typfehler in Anwendung!

Neue Typen erreichen Typsicherheit!

Denn:

- **USD-Floats** sind verschieden von
- **Zfs-Floats** sind verschieden von
- **C-Floats** sind verschieden von
- **...-Floats** sind verschieden von
- **Floats** und allen Typsynonymen von **Floats**.

Eine (unerwünschte) Nebenwirkung der Typsicherheit:

- **Keine** der zahlreichen auf **Floats** vordefinierten Operationen und Relationen (**(+)**, **(*)**, **(-)**, **(==)**, **(<)**, **(<=)**, ...) steht auf **USD-Floats**, **Zfs-Floats**, ... zur Verfügung.
- **Alle** auf **USD-Floats**, **Zfs-Floats**, ... gewollten Operationen und Relationen (**addieren**, **multiplizieren**, **auf Gleichheit testen**, ...) müssen **selbst implementiert werden!**

Ärmel aufgekrempt und los!

Testen auf Gleichheit:

```
gleich_usd :: Dollar -> Dollar -> Bool
gleich_usd (USD x) (USD y) = x == y
```

Vergleichen nach Größe:

```
kleiner_usd :: Dollar -> Dollar -> Bool
kleiner_usd (USD x) (USD y) = x < y
```

Addieren:

```
addiere_usd :: Dollar -> Dollar -> Dollar
addiere_usd (USD x) (USD y) = USD (x+y)
```

Ausgeben am Bildschirm:

```
zeige_auf_bildschirm_usd :: Dollar -> String
zeige_auf_bildschirm_usd (USD x)
  = "USD" ++ " " ++ float_als_zeichenreihe x
  where float_als_zeichenreihe :: Float -> String
        float_als_zeichenreihe x = ...
```

...und genauso für [Zinsfüße](#), [Meilen](#), [Lumen](#), etc.

Mühsam? Haskell bietet einen besseren Weg!

...Wiederverwendung der (ohnehin) überladenen Relations- und Operationssymbole:

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

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

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

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

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

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

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

...die in `Typklassen` organisiert sind.

Kapitel 4.3

Typklassen

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.6

4.4

Kap. 5

Kap. 6

Umgekehr

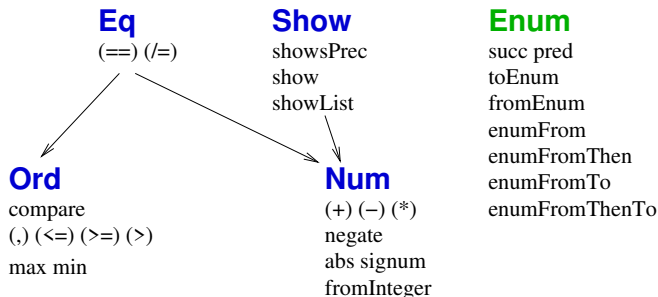
Klassen-
zimmer I

Hinweis

Aufgabe

Typklassen in Haskell

...ähneln **Schnittstellen** in **Java** und bilden eine **Hierarchie**:



Quelle: Fethi Rabhi, Guy Lapalme. [Algorithms - A Functional Approach](#). Addison-Wesley, 1999, Abb. 2.4 (Ausschnitt).

Typklassen – wozu?

...um **Überladung** von **Relatoren**, **Operatoren** zu organisieren!

Bisher liefern für die **Wertvereinbarungen** vom Typ **Dollar**:

```
d1 = USD 4.2 :: Dollar
d2 = USD 4.2 :: Dollar
d3 = USD 2.4 :: Dollar
```

folgende **Vergleiche** und **Operationen** unisono:

```
d1 == d2 ->> "Fehler: (==) unbekannt"
d2 == d3 ->> "Fehler: (==) unbekannt"
d2 /= d3 ->> "Fehler: (/=) unbekannt"
d1 >= d3 ->> "Fehler: (>=) unbekannt"
d3 < d2 ->> "Fehler: (<) unbekannt"
d1 + d2 ->> "Fehler: (+) unbekannt"
d1 - d2 ->> "Fehler: (-) unbekannt"
```

Relator- und Operatorsymbole

...wie (`==`), (`/=`), (`>=`), (`<`), ... und (`+`), (`-`), ... sind überladen und

- 'wissen'

mit den Werten vieler Typen (`Int`, `Integer`, `Float`, `Double`,...) umzugehen, aber

- nicht 'allwissend'!

Sie können nicht a priori 'wissen', wie mit Werten von uns neu eingeführter Typen wie z.B. `Dollar`:

```
newtype Dollar = USD Float
```

umgegangen werden soll.

...für Werte `Neuer Typen` muss dieses Wissen erst bereitgestellt werden in Form `zusätzlichen, neuen Überladungswissens`.

In Haskell dienen dazu Typklassen

...wie z.B. die Typklasse `Eq`:

```
class Eq a where
  (==) :: a -> a -> Bool  -- Funktionen d. Typklasse
  (/=) :: a -> a -> Bool  -- mit ihrer Signatur
  x /= y = not (x == y)   -- Protoimplementierungen
  x == y = not (x /= y)   -- für (/=) und (==)
```

Alle Typen, deren Werte mit `(==)`, `(/=)` verglichen werden sollen, müssen zu Elementen der Typklasse `Eq`, sog.

– Instanzen von `Eq`

gemacht werden.

Viele Typen wie `Int`, `Integer`, `Float`, `Double`, `Char`, `String`, `Bool`,... sind bereits vordefinierte Instanzen von `Eq`!

Eq-Instanzbildungen: 'Wissensbereitstellung'

1. Explizit durch `instance`-Deklaration:

```
instance Eq Dollar where
  (==) (USD x) (USD y) = x == y
  -- Impl. von (/=) vollautom. dank Proto.Impl.
```

Offenbar legt die Instanzbildung das 'Erwartete' fest. Das geht in `Haskell` auch einfacher. Vollautomatisch!

2. Automatisch bei Typdeklaration mit `deriving`-Klausel:

```
newtype Dollar = USD Float deriving Eq
```

Mit den vorherigen Vereinbarungen erhalten wir:

```
d1 == d2 ->> True
d2 == d3 ->> False
d2 /= d3 ->> True
d1 >= d3 ->> "Fehler: (>=) unbekannt"
d3 < d2 ->> "Fehler: (<) unbekannt"
d1 + d2 ->> "Fehler: (+) unbekannt"
d1 - d2 ->> "Fehler: (-) unbekannt"
```

Die Typklasse Ord: Vergleichsrelatoren

```
class Eq a => Ord a where
```

```
  compare           :: a -> a -> Ordering
```

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

```
  max, min         :: a -> a -> a
```

```
compare x y           -- Protoimplementierungen
```

```
  | x == y           = EQ           -- (==) aus Eq für a-Werte
```

```
  | x <= y           = LT
```


```
  | otherwise        = GT
```

```
x <= y               = compare x y /= GT
```

```
x < y                = compare x y == LT
```

```
x >= y               = compare x y /= LT
```

```
x > y                = compare x y == GT
```

```
max x y              Vergleiche auf Ordering-Werten
```

```
  | x <= y           = y
```

```
  | otherwise        = x
```

```
min x y
```

```
  | x <= y           = x
```

```
  | otherwise        = y
```

Ord-Instanzbildung: 'Wissensbereitstellung'

1. Explizit durch instance-Deklaration:

```
instance Ord Dollar where
  (<=) (USD x) (USD y) = x <= y
  -- alle anderen Impl. vollautom. dank Proto.Impl.
```

Offenbar legt die Instanzbildung wieder das 'Erwartete' fest.
Das geht in Haskell wieder einfacher. Vollautomatisch.

2. Automatisch bei Typdeklaration mit deriving-Klausel:

```
newtype Dollar = USD Float deriving (Eq,Ord)
```

Wir erhalten jetzt:

```
d1 == d2 ->> True
d2 == d3 ->> False
d2 /= d3 ->> True
d1 >= d3 ->> True
d3 < d2 ->> True
d1 + d2 ->> "Fehler: (+) unbekannt"
d1 - d2 ->> "Fehler: (-) unbekannt"
```

Die Typklasse Num: Arithmetische Operatoren

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

x - y      = x + negate y      -- Protoimplemen-
negate x = 0 - x              -- tierungen
```

Minimalvervollständigung bei Instanzbildungen für Num:

- Implementierung aller Funktionen mit Ausnahme von entweder `negate` oder `(-)`.

Beachte: Jeder Typ, der zu einer Instanz von Num gemacht werden soll, muss zuvor zu einer Instanz von Eq und Show gemacht worden sein.

Show, Num-Instanzbild.: 'Wissensbereitstellung'

1. Explizit durch instance-Deklarationen:

```
instance Show Dollar where
```

```
  show (USD x) = "USD" ++ " " ++ show x
```

show auf Dollar-Werten

show auf Float-Werten

```
instance Num Dollar where
```

```
  (+) (USD x) (USD y) = USD (x+y)
```

+ auf Dollar-Werten

+ auf Float-Werten

```
  (-) (USD x) (USD y) = USD (x-y)
```

```
  (*) (USD x) (USD y) = USD (x*y)
```

```
  abs (USD x) = USD (abs x)
```

```
  signum (USD x)           -- Vorzeichenfunktion
```

```
    | x < 0 = USD (-1.0)
```

```
    | x == 0 = USD 0.0
```

```
    | x > 0 = USD 1.0
```

```
  fromInteger n = USD (n :: Float)
```

Offenbar legen auch hier die Instanzbildungen

...das 'Erwartete' fest. Das geht in Haskell natürlich wieder einfacher. Vollautomatisch!

2. Automatisch bei Typdeklaration durch deriving-Klausel:

```
newtype Dollar = USD Float deriving (Eq, Ord, Show, Num)
```

Jetzt erhalten wir wie gewünscht u. schon anfänglich erwartet:

```
d1 == d2 ->> True
d2 == d3 ->> False
d2 /= d3 ->> True
d1 >= d3 ->> True
d3 < d2 ->> True
d1 + d2 ->> USD 8.4
d1 - d2 ->> USD 1.8
```

...und auch einige für Dollar nicht nötige oder sinnvolle Operationen:

```
d1 * d2 ->> USD 7.56           -- USD, nicht USD zum Quadrat
abs (USD (-4.2)) ->> USD 4.2    -- So verschwinden Schulden!
signum (USD 9999.99) ->> USD 1.0
```

Und wenn wir nicht das 'Erwartete' wollen?

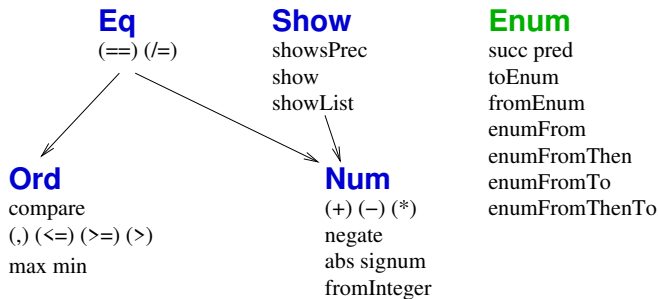
...müssen wir mit passenden Instanzbildungen für das gewollte 'Unerwartete' selbst sorgen. Z.B.:

```
newtype Dollar = USD Float
instance Show Dollar where
  show (USD x) = show x ++ "$"
instance Eq Dollar where
  (==) (USD x) (USD y) = abs (x-y) <= peanuts
  where peanuts = 2.0           -- Peanuts a la 'unsereins'
instance Ord Dollar where
  (>) (USD x) (USD y) = x-y > peanuts
  where peanuts = 2000000.0     -- Peanuts a la 'Trump'
```

Mit diesen Instanzbildungen für `Dollar` erhalten wir:

```
d1 == d2 ->> True    d2 /= d3 ->> False
d2 == d3 ->> True    d1 > d3 ->> False
show d1 ->> "4.2$"   show (d1 - d2) ->> "1.8$"
USD 10000.0 > USD 1.0 ->> False
```

Typklassen in Haskell: Eine Hierarchie



Quelle: Fethi Rabhi, Guy Lapalme. [Algorithms - A Functional Approach](#). Addison-Wesley, 1999, Abb. 2.4 (Ausschnitt).

Informelle Typklassenbeschreibungen

Bereits besprochen:

- **Eq**: Werte von **Eq**-Typen müssen auf Gleichheit und Ungleichheit vergleichbar sein.
- **Ord**: Werte von **Ord**-Typen müssen über Gleichheit und Ungleichheit hinaus bezüglich ihrer relativen Größe vergleichbar sein.
- **Show**: Werte von **Show**-Typen müssen eine Darstellung als Zeichenreihe besitzen.
- **Num**: Werte von **Num**-Typen müssen mit ausgewählten numerischen Operationen verknüpfbar sein, d.h. addiert, multipliziert, subtrahiert, etc. werden können.

Noch nicht besprochen:

- **Enum**: Werte von **Enum**-Typen müssen aufzählbar sein, d.h. einen Vorgänger- und Nachfolgerwert innerhalb des Typs besitzen.

Auch eigene Typklassen in Haskell definierbar

```
newtype Dollar = USD Float; newtype Euro = EUR Float
type Land      = String;    type Wert     = Float
```

```
class Waehrung a where
```

```
  -- 0-stellige Fkt., Konstanten
```

```
  ist_gesetzliches_Zahlungsmittel_in :: [Land]
```

```
  muenzen_im_Nennwert_von           :: [Wert]
```

```
  anzahl_verschiedene_Muenzen       :: Int
```

```
  -- 1-stellige Funktionen
```

```
  betrag_in_Muenzen_verschieden_zahlbar :: a -> Int
```

```
  kaufmaennisch_runden_auf_2_Kommastellen :: a -> Float
```

```
  -- Protoimplementierung
```

```
  anzahl_verschiedene_Muenzen x = length (muenzen_im_Nennwert_von x)
```

```
instance Waehrung Dollar where
```

```
  ist_gesetzliches_Zahlungsmittel_in = ["USA"]
```

```
  muenzen_im_Nennwert_von = [0.01,0.05,0.1,0.25,1.0,2.0]
```

```
  betrag_in_Muenzen_verschieden_zahlbar b = ...
```

```
  kaufmaennisch_runden_auf_2_Kommastellen b = ...
```

```
instance Waehrung Euro where...
```

Warum liefern Instanzbildungen wie

```
instance Eq Dollar where  
  (==) (USD x) (USD y) = x == y
```

```
instance Ord Dollar where  
  (USD x) <= (USD y) = x <= y
```

eigentlich **keine Fehlermeldungen?**

Aufgrund von **Überladungsvorwissen** von **Haskell-Übersetzer** und **Interpretierer**:

```
instance Eq Dollar where  
  (==) (USD x) (USD y) =  $\underbrace{\hspace{10em}}_{x == y}$   
       $\underbrace{\hspace{2em}}_{:: \text{Float}}$   $\underbrace{\hspace{2em}}_{:: \text{Float}}$   $\underbrace{\hspace{10em}}_{:: \text{Float} \rightarrow \text{Float} \rightarrow \text{Bool}}$ 
```

```
instance Ord Dollar where  
  (USD x) <= (USD y) =  $\underbrace{\hspace{10em}}_{x <= y}$   
       $\underbrace{\hspace{2em}}_{:: \text{Float}}$   $\underbrace{\hspace{2em}}_{:: \text{Float}}$   $\underbrace{\hspace{10em}}_{:: \text{Float} \rightarrow \text{Float} \rightarrow \text{Bool}}$ 
```

...viele Typen wie **Int**, **Integer**, **Float**, **Double**, **Char**, **String**
sind **vordefinierte Instanzen** von **Eq**, **Ord**, **Show**, etc.!

Kapitel 4.3.6

Zusammenfassung

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.3.6

4.4

Kap. 5

Kap. 6

Umgekehr

Klassen-
zimmer I

Hinweis

Aufgabe

Typklassen vs. objektorientierte Klassen

Haskells Typklassenkonzept unterscheidet sich wesentlich vom Klassenkonzept objektorientierter Sprachen.

In **objektorientierten Sprachen**: Klassen

- dienen der **Strukturierung** von Programmen.
- liefern **Blaupausen** zur **Generierung** von Werten.

In **Haskell**: Typklassen

- dienen **nicht** der Strukturierung von Programmen; liefern **keine** Blaupausen zur Generierung von Werten.
- dienen der **Organisation** und **Verwaltung** von **Überladung**.
- sind Mengen von Typen, deren Werte typspezifisch mit Funktionen gleichen Namens bearbeitet werden können ($(==)$, $(>)$, $(>=)$, $(+)$, $(*)$, $(-)$, etc.).
- erhalten Typen durch explizite Instanzbildung (**instance**-Deklaration) oder implizite automatische Instanzbildung (**deriving**-Klausel) als Elemente zugewiesen.

Faustregel zu Haskell's Typklassenphilosophie

Bei Einführung eines neuen Typs:

1. Überlege, welche Operationen/Relationen (semantische Begriffe: addiere, ist gleich) auf Werte dieses Typs anwendbar sind u. ob es bereits passende Operatoren/Relatoren (syntaktische Begriffe: (+), (==)) in exist. Typklassen dafür gibt.
2. Mache den neuen Typ zu Instanzen derjenigen Typklassen, in denen diese Operatoren/Relatoren eingeführt sind; oft reichen dafür deriving-Klauseln aus.
3. Sind auf die Werte des neuen Typs Operationen/Relationen anwendbar, für die es keine passenden Operatoren/Relatoren in existierenden Typklassen gibt, so
 - führe eine neue Typklasse (z.B. Waehrung) mit passenden Operatoren/Relatoren ein (wo möglich, zusammen mit vollständigen Implementierungen oder zu vervollständigenden Protoimplementierungen), wenn anzunehmen ist, dass diese konzeptuell auch für weitere erst noch zu definierende Datentypen relevant sein werden.

Woran erkennen wir überladene Funktionen?

...am nichtleeren **Signaturkontext**:

► Direkt überladene Funktionen

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

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

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

`betrag_in_Muenzen_verschieden_zahlbar` ::
`Waehrung a => a -> Int`

...sind unmittelbar in einer Typklasse eingeführt.

► Indirekt überladene Funktionen

`f` :: `(Num a, Waehrung a) => a -> a -> a`

`f x y`

= ...`betrag_in_muenzen_verschieden_zahlbar (x+y)`...

...sind nicht selbst in einer Typklasse eingeführt, stützen sich aber auf solche Funktionen ab.

Woran erkennen wir nicht überladene Fkt.?

...am leeren **Signaturkontext**:

Monomorpher Fall (nur konkrete Typen, keine Typvariablen):

```
fac    :: Integer -> Integer
fib    :: Integer -> Integer
binom  :: Int -> Int -> Int
```

Parametrisch polymorpher Fall (konkrete Typen, Typvariablen):

```
length :: [a] -> Int
last   :: [a] -> a
(++ )  :: [a] -> [a] -> [a]
first  :: (a,b) -> a
curry  :: ((a,b) -> c) -> (a -> b -> c)
uncurry :: (a -> b -> c) -> ((a,b) -> c)
```


Kapitel 4.4

Leseempfehlungen

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

4.1

4.2

4.3

4.4

Kap. 5




Kap. 6

Umgekehrte
Klassen-
zimmer I




Hinweis

Aufgabe

Basiselesempfehlungen für Kapitel 4

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 7.1, Typsynonyme mit `type`; Kapitel 7.2, Einfache algebraische Typen mit `data` und `newtype`; Kapitel 7.4, Automatische Instanzen von Typklassen; Kapitel 7.8, Eigene Klassen definieren)
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 3, Types and classes; Kapitel 8, Declaring types and classes)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 13.4, A tour of the built-in Haskell classes)

Weiterführende Leseempfehlungen für Kap. 4

-  Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 2, Expressions, types and values; Kapitel 3, Numbers)
-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 2, Believe the Type)
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 3, Defining Types, Streamlining Functions; Kapitel 6, Using Typeclasses)

Kapitel 5

Algebraische Datentypdeklarationen

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Umgekehrte
Klassen-
zimmer

Hinweis

Aufgabe

Kapitel 5.1

Überblick, Orientierung

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Umgekehrte
Klassen-
zimmer

Hinweis

Aufgabe

Algebraische Datentypdeklarationen in Haskell

...erlauben,

- ▶ originär neue Datentypen und ihre Werte einzuführen.

...heißen so,

- ▶ weil sie sich grob in
 - Summentypen
 - Produkttypen

einteilen lassen, deren Werte sich als **Summe** bzw. **Produkt** der Werte anderer Typen ergeben.

Was wir im Zshg. mit (Daten-) Typen

...bereits können: Existierenden Typen mittels

1. `type` zusätzliche Namen geben:

```
type Alter          = Int
type Schuhgroesse  = Int
a :: Alter
a = 42
s :: Schuhgroesse
s = a                (wohldefiniert, kein Typfehler!)
n :: Int
n = s                (wohldefiniert, kein Typfehler!)
a == s ->> True      (wohldefiniert, kein Typfehler!)
a /= n ->> False     (wohldefiniert, kein Typfehler!)
```

`Alter` und `Schuhgroesse` sind nichts als zusätzliche alternative Namen (Synonyme, Aliase) für den Typ `Int`, ohne eigene Typidentität: Alter- und Schuhgrößenwerte sind und bleiben Werte des Typs `Int` mit allen darauf vordef. Operationen und Relationen.

Was wir im Zshg. mit (Daten-) Typen

...auch schon können: Existierenden Typen mittels

2. `newtype` unverwechselbare neue Identitäten geben:

```
newtype Alter      = A Int deriving Eq
```

```
newtype Schuhgroesse = S Int deriving Eq
```

```
a :: Alter
```

```
a = A 42
```

```
s :: Schuhgroesse
```

```
s = a ->> Typfehler, Übersetzungsabbruch!
```

```
n :: Int
```

```
n = s ->> Typfehler, Übersetzungsabbruch!
```

```
a == s ->> Typfehler, Übersetzungsabbruch!
```

```
a /= n ->> Typfehler, Übersetzungsabbruch!
```

`Alter` und `Schuhgroesse` sind Typnamen mit neuer, eigener Typidentität: Alter- und Schuhgrößenwerte sind unverwechselbar, nicht länger gegeneinander austauschbar, nicht länger ganze Zahlen des Typs `Int`, sondern Alter- und Schuhgrößenfestkommazahlen, ohne darauf vordefinierte Operationen und Relationen.

Was wir im Zshg. mit (Daten-) Typen

...noch nicht können:

3. Originär neue, bisher nicht existierende Typen und ihre Werte einführen (z.B. für Bäume, Personen, Jahreszeiten,...):

```
data Baum          = ...
data Person        = ...
data Jahreszeit    = Fruehling
                   | Sommer
                   | Herbst
                   | Winter

data Spielfarbe    = Karo
                   | Herz
                   | Pik
                   | Kreuz
```

Die Aufgabe originär neue Datentypen und ihre Werte einzuführen, führt uns zu [Haskells](#) Konzept

- ▶ [algebraischer Datentypen](#).

Kapitel 5.2

Algebraische Datentypen

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.3

5.2.2

5.2.1

5.3

5.4

5.5

5.6

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Kapitel 5.2.3

Summentypen

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.3

5.2.2

5.2.1

5.3

5.4

5.5

5.6

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Bäume: Beispiel eines Summentyps

```
type Info = ...  
data Baum = Blatt Info  
          | Gabel Info Baum Baum deriving (Eq, Show)
```

Informell gelesen: Ein Wert des Typs **Baum** ist bzw. sieht aus wie

- ein mit einer **Information** benanntes **Blatt**

oder (ausgedrückt durch den Alternativenstrich |)

- eine mit einer **Information** benannte **Gabel** mit zwei **(Teil-) Bäumen**, einem linken und einem rechten.

Baumwerte sind also entweder Blatt oder Gabel mit zwei Teilbäumen. Die **Menge der Baumwerte** ist die Summe der Werte beider Baumausprägungen; deshalb **Summentyp!**

(Datenwert-) Konstruktoren

```
type Info = Int
data Baum = Blatt Info
           | Gabel Info Baum Baum deriving (Eq, Show)
```

Konstruktor

Konstruktor

Konstrukturen sind Abbildungen:

- Blatt, ein 1-stelliger Konstruktor:

`Blatt :: Info -> Baum`

...die Konstruktorabbildung `Blatt` liefert angewendet auf einen Wert vom Typ `Info` einen Wert vom Typ `Baum`.

- Gabel, ein 3-stelliger Konstruktor:

`Gabel :: Info -> Baum -> Baum -> Baum`

...die Konstruktorabbildung `Gabel` liefert angewendet auf einen Wert vom Typ `Info` und zwei Werte vom Typ `Baum` einen Wert vom Typ `Baum`.

Zwei Beispiele für Baumwerte

Blatt 3
:: Info
:: Baum

Gabel 2 (Blatt 3) (Gabel 5 (Blatt 7) (Blatt 11))
:: Info :: Baum
:: Info :: Baum :: Baum
:: Baum
:: Baum

Damit gilt:

Blatt 3 :: Baum

Gabel 2 (Blatt 3) (Gabel 5 (Blatt 7) (Blatt 11)) :: Baum

Schrift-, Bild-, Tonträger: Beispiel eines

...anderen **Summentyps** unter Benutzung der Typalias:

```
type Autor          = String      -- Buch-/E-Buchdaten
type Titel          = String
type Verlag         = String
type Auflage        = Int
type Lieferbar      = Bool
type LizenzBisJahr = Int

type Hauptdarsteller = [String]   -- Videodaten
type Regisseur       = String
type Sprachen        = [String]

type Kuenstler      = String      -- Audiodaten
type Std            = Int
type Min            = Int
type Sek            = Int
type Spieldauer     = (Std,Min,Sek)
```

Schrift-, Bild-, Tonträger: Ein Summentyp

```
data SchriftBildTontraeger =  
  Buch Autor Titel Verlag Auflage Lieferbar  
  | E_Buch Autor Titel Verlag LizenzBisJahr  
  | DVD Titel Hauptdarsteller Regisseur Sprachen  
  | CD Kuenstler Titel Spieldauer deriving (Eq,Show)
```

Vier Beispiele für Schrift-, Bild- und Tonträgerwerte:

```
Buch "Richard Bird" "Thinking Functionally"  
    "Cambridge University Press" 1 True  
:: SchriftBildTontraeger    (Buch: 5-stell. Konstruktor)  
E_Buch "Simon Thompson" "Haskell" "Pearson" 2018  
:: SchriftBildTontraeger    (E_Buch: 4-stell. Konstruktor)  
DVD "Der Pate" ["Marlon Brando","Al Pacino"]  
    "Francis Ford Coppola" ["Englisch","Deutsch","Italienisch"]  
:: SchriftBildTontraeger    (DVD: 4-stell. Konstruktor)  
CD "Angelika Nebel" "Klaviersonaten" (1,1,48)  
:: SchriftBildTontraeger    (CD: 3-stell. Konstruktor)
```


Anders als die

...Konstruktoren des Summentyps f. Schrift-, Bild-, Tonträger:

```
data SchriftBildTontraeger =  
  Buch Autor Titel Verlag Auflage Lieferbar  
  | E_Buch Autor Titel Verlag LizenzBisJahr  
  | DVD Titel Hauptdarsteller Regisseur Sprachen  
  | CD Kuenstler Titel Spieldauer deriving (Eq,Show)
```

ist der Konstruktor Gabel des Summentyps für Bäume:

```
data Baum = Blatt Info  
          | Gabel Info Baum Baum deriving (Eq,Show)
```

rekursiv definiert (im Zshg. mit Datentypen ist es allerdings üblich von induktiv statt wie bei Funktionen von rekursiv definiert zu sprechen):

- Induktiv (bzw. rekursiv) definierte Datentypen besitzen Werte 'beliebiger Größe', etwa Bäume 'beliebiger Größe' (allein die Speichergröße des Rechners ist die Grenze).

Induktiv vs. rekursiv

Im Zusammenhang mit

- **Funktionen** ist die Sprechweise **rekursiv definiert** üblich (vom Großen zum Kleinen; engl. top-down):

```
fac :: Int -> Int
```

```
fac n
```

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

```
  | n == 0 = 1
```

Rekursion: Eine **top/down-Sichtweise**.

- **Datentypen** ist hingegen die Sprechweise **induktiv** üblich (vom Kleinen zum Großen; engl. bottom-up):

```
data Baum = Blatt Info
```

```
          | Gabel Info Baum Baum deriving (Eq, Show)
```

Induktion: Eine **bottom/up-Sichtweise**.

...außer des eingebürgerten Sprachgebrauchs spricht nichts dagegen bei Funktionen auch von induktiv und bei Datentypen auch von rekursiv definiert zu sprechen.

Beispiele vordefinierter Summentypen

Der Möglicherweise-Typ (polymorph)

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

Der Entweder/Oder-Typ (polymorph)

```
data Either a b = Left a
                | Right b
                deriving (Eq,Ord,Read,Show)
```

Listen (polymorph)

```
data [a] = []
         | a : [a] deriving (Eq,Ord)

-- Kein gültiges Haskell; die genaue Implementierung
-- von Listen ist in der Sprachimplementierung verborgen,
-- aber konsistent mit obiger Typdeklaration.
```

Einige Wertbeispiele für

- den Möglicherweise-Typ:

```
Nothing :: Maybe a  
Just 42  :: Maybe Int  
Just 'a' :: Maybe Char
```

- den Entweder/Oder-Typ:

```
Left 42  :: Either Int Char  
Right 'a' :: Either Int Char  
Left 42  :: Either Int Baum  
Right (Blatt 42) :: Either Int Baum
```

- Listen:

```
[] :: []  
1 : (2 : (3 : [])) :: [Int]  
True : (False : (True : [])) :: [Bool]
```

Zusätzlicher syntaktischer Zucker erlaubt kürzer:

```
[1,2,3] :: [Int]  
[True,False,True] :: [Bool]
```

Das allgemeine Muster

...algebraischer Datentypdefinitionen:

```
data Typname = Kon_1 t_11 ... t_1k_1
              | Kon_2 t_21 ... t_2k_2
              ...
              | Kon_n t_n1 ... t_nk_n
```

Dabei sind:

- **Typname**: Freigewählter frischer **Identifikator** als Typname.
- **Kon_i**: Freigewählte frische **Identifikatoren** als (Datenwert-) **Konstruktornamen**.
- **k_i**: Stelligkeit des Konstruktors **Kon_i**.
- **t_ij**: Namen bereits existierender **Typen**.
- **Typ-** und **Konstruktornamen** müssen stets mit einem Großbuchstaben beginnen.

Anzahl und Stelligkeit der Konstruktoren

...liefern eine Aufteilung der **algebraischen Datentypen** in i.w.:

1. (Echte) Summentypen:

- Mindestens zwei Konstruktoren, mindestens ein nicht nullstelliger Konstruktor.

2. (Echte) Produkttypen:

- Exakt ein zwei- oder höherstelliger Konstruktor.

3. Aufzählungstypen:

- Ausschließlich nullstellige Konstruktoren.

Anmerkung: Die Aufteilung lässt folgenden Randfall. Ein algebraischer Datentyp mit genau einem einstelligen Konstruktor lässt sich in gleicher Weise als **unechter Summen-** wie als **unechter Produkt-**typ ansehen.

Kapitel 5.2.2

Produkttypen

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.3

5.2.2

5.2.1

5.3

5.4

5.5

5.6

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Einige Beispiele selbstdefinierter Produkttypen

```
type Nat0      = Int
type Vorname   = String      -- Personendaten
type Nachname  = String
type Alter     = Nat0
type Gemeinde  = String      -- Adressdaten
type Strasse   = String
type Hausnr    = Int
type Land      = String

data Person    = P Vorname Nachname Alter d...
data Anschrift = A Gemeinde Strasse Hausnr Land d...
data Einwohner = E Land Gemeinde [Person] deriving...
data Wohnsitze = W Land (Person -> [Anschrift])
data Gemeldet  = G (Land -> Gemeinde -> Strasse
                   -> Hausnr -> [Person])
```


Kapitel 5.2.1

Aufzählungstypen

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.3

5.2.2

5.2.1

5.3

5.4

5.5

5.6

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Beispiele selbstdefinierter Aufzählungstypen

```
data Jahreszeit = Fruehling | Sommer
                | Herbst | Winter
                deriving (Eq, Ord, Bounded,
                          Enum, Read, Show)

data Spielfarbe = Karo | Herz | Pik | Kreuz
                deriving (Eq, Ord, Bounded,
                          Enum, Read, Show)

data Werktag    = Montag | Dienstag | Mittwoch
                | Donnerstag | Freitag
                deriving (Eq, Ord, Bounded,
                          Enum, Read, Show)

data Wochenende = Samstag | Sonntag
                deriving (Eq, Ord, Bounded,
                          Enum, Read, Show)
```

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.2.3

5.2.2

5.2.1

5.3

5.4

5.5

5.6

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Beispiele vordefinierter Aufzählungstypen

Typ der Ordnungswerte, 3 Werte:

```
data Ordering = LT | EQ | GT deriving (Eq,Ord,  
                                         Bounded,  
                                         Enum, Read,  
                                         Show)
```

Typ der Wahrheitswerte, 2 Werte:

```
data Bool = False | True deriving (Eq,Ord,Bounded,  
                                     Enum,Read,Show)
```

Nulltupeltyp, 1 Wert:

```
data () = () deriving (Eq,Ord,Bounded,Enum,Read,  
                       Show)
```

...Nulltupeltyp und einziger (def.) Wert ident bezeichnet: `()`.

Kapitel 5.3

Funktionen auf algebraischen Datentypen

Algebraische Datentypen

...führen in natürlicher Weise zu **musterbasierten** Funktionsdefinitionen.

Leitfrage bei der Funktionsdefinition:

- Wenn der Wert meines algebraischen Datentyps aussieht wie ‘das und das Muster’, dann ist der Wert der Funktion ‘der und der’.

Dazu einige Beispiele zur Illustration.

Funktionen auf Bäumen

...zum **Aufsummieren** der Marken, zum Berechnen der **Tiefe** eines Baums:

```
summiere :: Baum -> Int
```

```
summiere (Blatt n) = n
```

```
summiere (Gabel n ltb rtb) = n + summiere ltb + summiere rtb
```

```
tiefe :: Baum -> Int
```

```
tiefe (Blatt _) = 1
```

```
tiefe (Gabel _ ltb rtb) = 1 + max (tiefe ltb) (tiefe rtb)
```

Aufrufbeispiele:

```
summiere (Blatt 42) ->> 42
```

```
summiere (Gabel 2 (Gabel 3 (Blatt 5) (Blatt 7)) (Blatt 11)) ->> 28
```

```
tiefe (Blatt 42) ->> 1
```

```
tiefe (Gabel 2 (Gabel 3 (Blatt 5) (Blatt 7)) (Blatt 11)) ->> 3
```

Funktion auf Schrift-, Bild-, Tonträgerwerten

...die Selektorfunktion `selektiereTitel`:

```
selektiereTitel :: SchriftBildTonTraeger -> Titel
selektiereTitel (Buch autor titel verlag auflage lieferbar) = titel
selektiereTitel (E_Buch _ titel _ _) = titel
selektiereTitel (DVD t _ _ _) = t
selektiereTitel (CD _ t _) = t
```

Aufrufbeispiele:

```
selektiereTitel (Buch "Richard Bird" "Thinking Functionally"
                  "Cambridge University Press" 1 True)
->> "Thinking Functionally" :: Titel
selektiereTitel (E_Buch "Simon Thompson" "Haskell" "Pearson" 2018)
->> "Haskell" :: Titel
selektiereTitel (DVD "Der Pate" ["Marlon Brando","Al Pacino"]
                 "Francis Ford Coppola" ["Englisch","Deutsch","Italienisch"])
->> "Der Pate" :: Titel
selektiereTitel (CD "Angelika Nebel" "Klaviersonaten" (1,1,48))
->> "Klaviersonaten" :: Titel
```

Kapitel 5.4

Feldsyntax

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Umgekehrte
Klassen-
zimmer

Hinweis

Aufgabe

Ziel: Transparente, sprechende Typdeklarationen

...in Haskell bieten sich dafür drei Möglichkeiten an:

1. Kommentierung
2. Typsynonyme
3. Feldsyntax (Verbundtypsyntax)

...mit dem Zusatzvorteil

- ‘geschenker’ Selektorfunktionen
- wesentlich vereinfachter weiterer Verarbeitungsfkt.

1.) Transparente, sprechende Typdeklarationen

...durch Kommentierung:

```
newtype Gb          = Gb (String,String,String)
                    deriving (Eq,Ord,Show)
data G              = M | W deriving (Eq,Ord,Show)
data Meldedaten = Md String -- Vorname
                    String  -- Nachname
                    Gb      -- Geboren (tt,mm,jjjj)
                    G       -- Geschlecht (m/w)
                    String  -- Gemeinde
                    String  -- Strasse
                    Int     -- Hausnummer
                    Int     -- PLZ
                    String  -- Land
                    deriving (Eq,Ord,Show)
```

2.) Transparente, sprechende Typdeklarationen

...durch Typsynonyme:

```
type Vorname      = String
type Nachname     = String
type Ziffernfolge = String
type Zf           = Ziffernfolge
newtype Gb        = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)
type Geboren      = Gb
data G             = M | W deriving (Eq,Ord,Show)
type Geschlecht   = G
type Gemeinde     = String
type Strasse      = String
type Hausnummer   = Int
type PLZ          = Int
type Land         = String

data Meldedaten   = Md Vorname Nachname Geboren
                  Geschlecht Gemeinde Strasse
                  Hausnummer PLZ Land deriving (Eq,Ord,Show)
```

3.) Transparente, sprechende Typdeklarationen

...durch **Feldsyntax** (oder: **Verbundtypsyntax**):

```
type Ziffernfolge = String
type Zf           = Ziffernfolge
data G            = M | W deriving (Eq,Ord,Show)
newtype Gb       = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)

data Meldedaten  = Md { vorname    :: String,
                       nachname   :: String,
                       geboren     :: Gb,
                       geschlecht  :: G,
                       gemeinde    :: String,
                       strasse     :: String,
                       hausnummer  :: Int,
                       plz         :: Int,
                       land        :: String
                       } deriving (Eq,Ord,Show)
```

Typgleiche Felder

...können in der **Feldsyntax** durch Beistrich getrennt **zusammengefasst** werden:

```
type Ziffernfolge = String
type Zf           = Ziffernfolge
data G            = M | W deriving (Eq,Ord,Show)
newtype Gb       = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)
data PersDaten   = PD { vorname,
                       nachname,
                       gemeinde,
                       strasse,
                       land      :: String,
                       geboren   :: Gb,
                       geschlecht :: G,
                       hausnummer,
                       plz       :: Int
                       } deriving (Eq,Ord,Show)
```

Feldnamen in Alternativen

...dürfen **wiederholt verwendet werden**, wenn ihr Typ für alle Vorkommen ident ist:

```
type Ziffernfolge = String
type Zf           = Ziffernfolge
data G            = M | W deriving (Eq,Ord,Show)
newtype Gb       = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)
data Meldedaten  = Md { vorname,
                       nachname,
                       gemeinde,
                       strasse,
                       land      :: String,
                       geboren   :: Gb,
                       geschlecht :: G,
                       hausnummer,
                       plz       :: Int
                       }
                 | KurzMd { vorname,
                           nachname :: String
                           } deriving (Eq,Ord,Show)
```

Insgesamt: Transparente, sprechende Typ-

...deklarationen durch **Kommentar**, **Typsynonyme**, **Feldsyntax**:

```
type Vorname      = String
type Nachname     = String
type Ziffernfolge = String
type Zf           = Ziffernfolge
newtype Gb       = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)
type Geboren     = Gb
data G           = M | W deriving (Eq,Ord,Show)
type Geschlecht  = G
type Gemeinde    = String
type Strasse     = String
type Hausnummer  = Int
type PLZ         = Int
type Land        = String
```

```
data Meldedaten = Md { vorname      :: Vorname,
                      nachname     :: Nachname,
                      geboren      :: Geboren, -- (tt,mm,jjjj)
                      geschlecht   :: Geschlecht,
                      gemeinde     :: Gemeinde,
                      strasse      :: Strasse,
                      hausnummer   :: Hausnummer,
                      plz          :: PLZ,
                      land         :: Land
                    } deriving (Eq,Ord,Show)
```

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Umgekehrt

Klassen-
zimmer I

Hinweis

Aufgabe

Mittels Kommentierung und Typsynonymen

...definierte Typen erfordern üblicherweise **musterdefinierte Selektor-, Wertsetzungs- und Werterzeugungsfunktionen**, etwa folgende **9 Selektorfunktionen**:

```
vornameVon :: Meldedaten -> Vorname
```

```
vornameVon (Md vn _ _ _ _ _ _) = vn
```

```
nachnameVon :: Meldedaten -> Nachname
```

```
nachnameVon (Md _ nn _ _ _ _ _ _) = nn
```

...

```
plzVon :: Meldedaten -> PLZ
```

```
plzVon (Md _ _ _ _ _ plz _) = hsnr
```

```
landVon :: Meldedaten -> Land
```

```
landVon (Md _ _ _ _ _ _ land) = land
```


In gleicher Weise

...sind 9 Wertsetzungsfunktionen zu schreiben:

```
setzeVorname :: Vorname -> Meldedaten -> Meldedaten
setzeVorname vn (Md _ nn geb gs gem str hsnr plz land)
  = Md vn nn geb gs gem str hsnr plz land
```

```
setzeNachname :: Nachname -> Meldedaten -> Meldedaten
setzeNachname nn (Md vn _ geb gs gem str hsnr plz land)
  = Md vn nn geb gs gem str hsnr plz land
```

...

```
setzePLZ :: PLZ -> Meldedaten -> Meldedaten
setzePLZ plz (Md vn nn geb gs gem str hsnr _ land)
  = Md vn nn geb gs gem str hsnr plz land
```

```
setzeLand :: Land -> Meldedaten -> Meldedaten
setzeLand land (Md vn nn geb gs gem str hsnr plz _)
  = Md vn nn geb gs gem str hsnr plz land
```

Und schließlich

...noch 9 Werterzeugungsfunktionen:

```
undef = undef          -- Auswertung terminiert nicht!
```

```
erzeugeMdMitVorname :: Vorname -> Meldedaten
```

```
erzeugeMdMitVorname vorname  
= Md vorname undef undef undef undef undef undef  
  undef undef
```

...

```
erzeugeMdMitLand :: Land -> Meldedaten
```

```
erzeugeMdMitLand land  
= Md undef undef undef undef undef undef undef  
  undef land)
```

Selektorfunktionen 'geschenkt'

...bei **Feldnamenverwendung** – die **Feldnamen** selbst sind die Selektorfunktionen:

```
vornameVon :: Meldedaten -> Vorname
```

```
vornameVon = vorname
```

```
nachnameVon :: Meldedaten -> Nachname
```

```
nachnameVon = nachname
```

```
...
```

```
plzVon :: Meldedaten -> PLZ
```

```
plzVon = plz
```

```
landVon :: Meldedaten -> Land
```

```
landVon = land
```

Beachte: Die Funktionen `vornameVon`, `nachnameVon`, etc., sind nur mehr Synonyme bzw. Aliase der Feldnamen `vorname`, `nachname`, etc.; ihre Einführung deshalb obsolet.

Andere Funktionen auf Meldedaten

...wie die Wertsetzungs- und Werterzeugungsfunktionen lassen sich dank Feldnamen wesentlich einfacher schreiben:

```
setzeVorname :: Vorname -> Meldedaten -> Meldedaten
```

```
setzeVorname vn md = md {vorname = vn}
```

```
setzeNachname :: Nachname -> Meldedaten -> Meldedaten
```

```
setzeNachname nn md = md {nachname = nn}
```

...

```
erzeugeMdMitVorname :: Vorname -> Meldedaten
```

```
erzeugeMdMitVorname vn = Md {vorname = vn}
```

```
erzeugeMdMitNachname :: Nachname -> Meldedaten
```

```
erzeugeMdMitNachname nn = Md {nachname = nn}
```

...nicht genannte Felder werden automatisch*) 'undefiniert' gesetzt.

*) Sprachimplementierungsabhängig: 'Gute' Übersetzer und Interpretierer sollten das jedenfalls tun.

Auch mehrere Felder

...können gleichzeitig gesetzt werden, hier sind es je zwei:

```
setzeName :: Vorname -> Nachname -> Meldedaten -> Meldedaten
setzeName vn nn md = md {vorname=vn, nachname=nn}
    -- Nicht genannte Felder behalten ihren Wert.
```

...

```
erzeugeMdMitName :: Vorname -> Nachname -> Meldedaten
erzeugeMdMitName vn nn = Md {vorname=vn, nachname=nn}
    -- Nicht genannte Felder werden 'undefiniert' gesetzt.
```

Weitere Beispiele von Feldnamenverwendungen

...liefere Vor- und Nachnamen, getrennt durch ein Leerzeichen:

```
vollerNameVon :: Meldedaten -> String
vollerNameVon md
  = vorname md ++ " " ++ nachname md

vollerNameVon' :: Meldedaten -> String
vollerNameVon' (Md {vorname = vn, nachname = nn})
  = vn ++ " " ++ nn
```

Gleichwertig ohne Feldnamenverwendung:

```
vollerNameVon'' :: Meldedaten -> String
vollerNameVon'' (Md vn nn _ _ _ _ _ _ _)
  = vn ++ " " ++ nn
```

doch weniger bequem, da

- die Zahl der Unterstriche **exakt** stimmen muss.
- **änderungsaufwändig** bei Hinzu-/Wegnahme von Feldkomponenten (alle Aufrufstellen müssen angepasst werden!)

Kapitel 5.5

Zusammenfassung, Anwendungshinweise

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.6

Kap. 6

Umgekehrte

Klassen-
zimmer I

Hinweis

Aufgabe

Zusammenfassung

1. `type` erlaubt existierenden Typen

- zusätzliche, neue Namen zu geben (Synonyme, Aliase).

Typ und Typsynonym sind ident; alle Funktionen auf dem Typ stehen daher auch auf jedem Typsynonym zur Verfügung; Typ und Typsynonyme können sich wechselseitig vertreten.

2. `newtype` erlaubt existierenden Typen

- unverwechselbare, neue Identitäten zu verleihen.

Typ und davon abgeleiteter Neuer Typ sind verschieden und unverwechselbar; keine auf dem Typ zur Verfügung stehende Funktion überträgt sich auf den abgeleiteten Neuen Typ; alle auf Werten des Neuen Typs benötigte Fkt. sind selbst zu implementieren; manchmal reicht eine `deriving`-Klausel dafür.

3. `data` erlaubt

- originär neue Typen und ihre Werte einzuführen.

Alle auf Werten des neuen Typs benötigte Fkt. sind selbst zu implementieren; manchm. reicht eine `deriving`-Klausel dafür.

Kapitel 5.5.1

Faustregeln

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.6

Kap. 6

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Die Faustregel zur Verwendung von type

Verwende **type-Deklarationen** wie

```
type Euro      = Float
type Celsius = Float
```

wenn

- **sprechendere Typnamen** die beabsichtigte Bedeutung von Typwerten anzeigen sollen.
- die Bequemlichkeit geschätzt wird, alle auf dem Grundtyp vorhandenen Funktionen **unmittelbar weiterverwenden** zu können (z.B. `(==)`, `(/=)`, `(+)`, `(-)`, ...).

unter bewusster Inkaufnahme, dass

- dies auch für semantisch unsinnige Fkt. gilt (z.B. Logarithmus, trigonometrische Fkt. für `Float`-Werte, die für Euro-Beträge oder Temperaturwerte stehen).
- auch semantisch unsinnige Verküpfungen möglich sind (z.B. Vergleich, Addition von Euro-, Temperaturwerten).

Die Faustregel zur Verwendung von `newtype`

Verwende `newtype`-Deklarationen wie

```
newtype Euro      = EUR Float
```

```
newtype Celsius  = C Float
```

wenn

- über sprechendere Typnamen hinaus die (Typ-) Sicherheit benötigt wird, dass semantisch unsinnige Verküpfungen durch das Typsystem verhindert werden u. ausgeschlossen sind (z.B. Vergleich, Addition von Euro-, Temperaturwerten).

unter bewusster Inkaufnahme, dass

- keine der auf dem Grundtyp vorhand. Fkt. unmittelbar weiterverwendet werden können (wie `(==)`, `(+)`,...), sondern alle auf dem Neuen Typ benötigten Fkt. **erst zu implementieren** sind
 - entweder unter neu erdachten Namen (`euro_gleich`, `celsius_gleich`, `euro_plus`, `celsius_plus`,...)
 - oder bei der Instanzbildung f. passende Typklassen (`instance Eq Euro...`, `instance Num Celsius...`) zur Wiederverw. überlad. Fkt.-Namen (wie `(==)`, `(+)`,...).

Die Faustregel zur Verwendung von data

Verwende `data`-Deklarationen wie für `Ausblick`, `Waehrung`:

```
newtype Zentralbank_Diskontzinssatz = ZS Float
type Diskont = Zentralbank_Diskontzinssatz
type Betrag = Float
data Ausblick = Steigt | Sinkt | Unveraendert
data Waehrung = EUR Betrag Diskont Ausblick
                | USD Betrag Diskont Ausblick
                | GBP Betrag Diskont Ausblick
```

wenn

- Namen und Werte `originär neuer Typen` benötigt werden (z.B. für Bäume, Personen, Netzwerke, Währungen,...)

und deshalb

- weder `type`- noch `newtype`-Deklarationen infrage kommen.

Kapitel 5.6

Leseempfehlungen

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

5.1

5.2

5.3

5.4

5.5

5.6

Kap. 6

Umgekehrte
Klassen-
zimmer

Hinweis

Aufgabe

Basiselesempfehlungen für Kapitel 5

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 7, Eigene Typen und Typklassen definieren)
-  Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 8, Benutzerdefinierte Datentypen)
-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 7, Making our own Types and Type Classes; Kapitel 12, Monoids – Wrapping an Existing Type into a New Type)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 14, Algebraic types)

Weiterführende Leseempfehlungen für Kap. 5

-  Ernst-Erich Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Oldenbourg Verlag, 2012. (Kapitel 4, Algebraische Datentypen)
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 8.1, Type declarations; Kapitel 8.2, Data declarations; Kapitel 8.3, Newtype declarations; Kapitel 8.4, Recursive types)
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 2, Types and Functions; Kapitel 3, Defining Types, Streamlining Functions – Defining a New Data Type, Type Synonyms, Algebraic Data Types)

Kapitel 6

Muster und mehr

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Muster...

Muster, Musterpassung (Kap. 6.1)

- Elementare Datentypen
- Tupeltypen
- Listentypen
- Algebraische Datentypen

...und mehr:

Listenkomprehension (Kap. 6.2)

- Alleinstellungsmerkmal funktionaler Programmiersprachen

Konstruktoren, Operatoren (Kap. 6.3)

- Begriffsbestimmung und Vergleich am Beispiel von Listen

Kapitel 6.1

Muster, Musterpassung

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.2

6.3

6.4

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Muster, Musterpassung

Muster sind

- (syntaktische) Ausdrücke, die die **Struktur von Werten** beschreiben.

Musterpassung (engl. **pattern matching**) dient

- in Funktionsdefinitionen durch Muster festgelegte Alternativen auszuwählen. Die Muster werden dabei in einer festen Reihenfolge (von oben nach unten) auf Passung ausprobiert; **passt** die Struktur eines (Argument-) Werts auf ein Muster, wird diese Alternative ausgewählt.

Beispiel

Musterbasierte Funktionsdefinition:

```
sortiere :: [Integer] -> [Integer]
sortiere []      = []      (Muster der leeren Liste)
sortiere (n:[]) = [n]     (Muster einelementiger Liste)
sortiere (n:ns) = ...     (Muster zwei- oder mehr-
                           elementiger Liste)
```

Aufrufe:

```
ns1 = []
ns2 = [2] (== 2:[])
ns3 = [2,3,1] (== 2:[3,1])

sortiere ns1 ->> []      (Wert ns1 passt auf Muster [])
sortiere ns2 ->> [2]    (Wert ns2 passt auf Muster (n:[]))
sortiere ns3 ->> ...    (Wert ns3 passt auf Muster (n:ns))
```

Kapitel 6.1.1

Muster für Werte elementarer Datentypen

Muster für Werte elementarer Datentypen

...mit **Konstanten**, **Variablen** und **Jokern** als Muster:

```
fib :: Int -> Int
```

```
fib 0 = 0
```

```
fib 1 = 1
```

```
fib n = fib (n-2) + fib (n-1)
```

```
potenz :: Integer -> Integer -> Integer
```

```
potenz _ 0 = 1
```

```
potenz m 1 = m
```

```
potenz m n = m * potenz m (n-1)
```

```
wenn_dann_sonst :: Bool -> a -> a -> a
```

```
wenn_dann_sonst True t _ = t
```

```
wenn_dann_sonst False _ e = e
```

Kapitel 6.1.2

Muster für Werte von Tupeltypen

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.2

6.3

6.4

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Muster für Werte von Tupeltypen

...mit **Konstanten**, **Variablen** und **Jokern** als Muster:

```
binom'' :: (Int,Int) -> Int      -- rein musterbasiert
```

```
binom'' (_,0) = 1
```

```
binom'' (1,1) = 1
```

```
binom'' (n,k) = binom'' (n-1,k-1) + binom'' (n-1,k)
```

```
binom'  :: (Int,Int) -> Int      -- musterbasiert mit
```

```
binom'  (n,k)                    -- Fallunterscheidung
```

```
  | k==0 || n==k = 1
```

```
  | otherwise     = binom' (n-1,k-1) + binom' (n-1,k)
```

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

```
fst_of_tripel (x,_,_) = x
```

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

```
snd_of_tripel (_,y,_) = y
```

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

```
thd_of_tripel (_,_,z) = z
```


Kapitel 6.1.3

Muster für Werte von Listentypen

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.2

6.3

6.4

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Muster für Werte von Listentypen (1)

...mit **Konstanten**, **Variablen** und **Jokern** als Muster:

```
add :: Num a => [a] -> a
```

```
add [] = 0
```

```
add (0:xs) = add xs
```

```
add (x:xs) = x + add xs
```

```
mult :: Num a => [a] -> a
```

```
mult [] = 1
```

```
mult (1:[]) = 1
```

```
mult (0:_) = 0
```

```
mult (1:xs) = mult xs
```

```
mult (x:xs) = x * mult xs
```

Muster für Werte von Listentypen (2)

...mit Konstanten und Variablen als Muster:

```
sortiere :: Ord a => [a] -> [a]
sortiere []           = []           (Muster der leeren Liste)
sortiere (n:[])      = [n]          (Muster einelementiger Liste)
sortiere (n:m:[])    = [n,m]        (Muster zweielementiger Liste)
  | n <= m = [n,m]
  | n > m  = [m,n]
sortiere (n:m:m':[]) (Muster dreielementiger Liste)
  | (n <= m) && (m <= m')           = [n,m,m']
  | (n > m) && (m > m')             = [m',m,n]
  | (n <= m) && (n <= m') && (m > m') = [n,m',m]
  | ...
...
sortiere (n:m:m':m':ns) (Muster vier- oder
  | ... (mehrelem. Liste)
...

```

Kapitel 6.1.4

Muster für Werte algebraischer Datentypen

Muster für Werte algebraischer Datentypen (1)

...mit 0-stelligen Konstruktoren entsprechend Konstanten als Muster:

```
type Zeichenreihe = [Char]
data Jahreszeit   = Fruehling
                  | Sommer
                  | Herbst
                  | Winter

wetter :: Jahreszeit -> Zeichenreihe
wetter Fruehling = "Launisch"
wetter Sommer  = "Sonnig"
wetter Herbst   = "Windig"
wetter Winter   = "Frostig"
```

Muster für Werte algebraischer Datentypen (2)

Konstruktormuster mit Variablen:

```
type Zett      = Int
data Ausdruck = Opd Zett
              | Add Ausdruck Ausdruck
              | Sub Ausdruck Ausdruck
              | Quad Ausdruck

eval :: Ausdruck -> Zett
eval (Opd z)      = z
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)
eval (Quad e)    = (eval e)^2
```

Muster für Werte algebraischer Datentypen (3)

Konstanten als Muster; Konstruktormuster mit Variablen und Jokern:

```
type Zett      = Int
data Baum a b = Blatt a
              | Wurzel b (Baum a b) (Baum a b)
data Liste a  = Leer
              | Kopf a (Liste a)

tiefe :: (Baum a b) -> Zett
tiefe (Blatt _)      = 1
tiefe (Wurzel _ l r) = 1 + max (tiefe l) (tiefe r)

laenge :: (Liste a) -> Zett
laenge Leer          = 0
laenge (Kopf _ xs)  = 1 + laenge xs
```

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.2

6.3

6.4

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Alternativenauswahl: Wert- vs. musterbasiert

- ▶ Wertbasiert (Leitfrage: Welchen Wert hat das Argument?)

```
fib :: Int -> Int
```

```
fib n
```

```
| n == 0 = 0           (wenn Wert gleich 0, dann...)  
| n == 1 = 1         (wenn Wert gleich 1, dann...)  
| True   = fib (n-1) + fib (n-2) (wenn anderer Wert, dann...)
```

- ▶ Musterbasiert (Leitfrage: Wie sieht das Argument aus?)

```
fib :: Int -> Int
```

```
fib 0 = 0
```

(wenn Arg. aussieht wie 0, dann...)

```
fib 1 = 1
```

(wenn Arg. aussieht wie 1, dann...)

```
fib n = fib (n-1) + fib (n-2) (wenn Arg. anders, dann...)
```

- ▶ Muster und wertbasiert kombiniert

```
fib :: Int -> Int
```

```
fib 0 = 0
```

```
fib 1 = 1
```

```
fib n
```

```
| n > 1 = fib (n-1) + fib (n-2)
```

```
| n < 0 = error "Argument unzulässig!"
```


Kapitel 6.1.5

Das als-Muster

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.1.1

6.1.2

6.1.3

6.1.4

6.1.5

6.2

6.3

6.4

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Beispiel: Berechne alle nichtleeren Postfixe

...einer Zeichenreihe:

```
nichtleere_postfixe "Curry"  
->> ["Curry", "urry", "rry", "ry", "y"]
```

Eine mögliche Implementierung:

```
nichtleere_postfixe :: String -> [String]  
nichtleere_postfixe (c:cs)  
  = (c:cs) : nichtleere_postfixe cs  
nichtleere_postfixe _ = []
```

Man erkennt: Die Implementierung nimmt Bezug auf

- das gesamte Argument: (c:cs)
- einen strukturellen Teil des Arguments: cs

Das **als-Muster** @ (engl. 'as') erlaubt das auszudrücken...

Das als-Muster

...lässt sich über **Muster für Listenwerte** hinaus generell für **Muster strukturierter Werte** verwenden, also die Werte von

- Tupeltypen
- Listentypen
- algebraischen Datentypen

Zwei Implementierungen

...von `nichtleere_postfixe` mit und ohne `als-Muster` @:

- Mit `als-Muster`:

```
nichtleere_postfixe :: String -> [String]
nichtleere_postfixe s@( _:cs)
  = s : nichtleere_postfixe cs
nichtleere_postfixe _ = []
```

- Ohne `als-Muster`:

```
nichtleere_postfixe :: String -> [String]
nichtleere_postfixe (c:cs)
  = (c:cs) : nichtleere_postfixe cs
nichtleere_postfixe _ = []
```

Kapitel 6.2

Listenkomprehension

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Listenkomprehension

...ein charakteristisches, elegantes und ausdruckskräftiges Sprachmittel

- ▶ funktionaler Programmiersprachen

das die Mengenbildungsoperation aus der Mathematik auf Listen nachbildet und ohne Parallele in Sprachen anderer Paradigmen ist:

- ▶ Alleinstellungsmerkmal funktionaler Sprachen!

Listenkompensation in Ausdrücken (1)

Zwei Listen:

```
ns1 = [1,2,3,4]
```

```
ns2 = [1,2,4,7,8,11,12,42]
```

Ein Generator, eine Transformation:

```
[ 3 * n | n <- ns1 ]
```

```
->> [3,6,9,12]
```

```
[ square n | n <- ns2 ]
```

```
->> [1,4,16,49,64,121,144,1764]
```

```
[ isPrime n | n <- ns2 ]
```

```
->> [False,True,False,True,False,True,False,False]
```

Listenkompensation in Ausdrücken (2)

Ein Generator, ein bzw. zwei Tests, eine Transformation:

```
[ fac n | n <- ns2, isPowOfTwo n ]  
->> [1,2,24,40320]
```

```
[ id n | n <- ns2, isPowOfTwo n, n>=5 ]      -- ','  
->> [8]                                       -- steht für 'und'
```

Zwei Generatoren, ein Filter, zwei Tests, eine Transformation:

```
[ ((m,n),m+n) | m <- ns1, n <- tail ns2, m<=2, n<=7 ]  
->> [((1,2),3),((1,4),5),((1,7),8),  
      ((2,2),4),((2,4),6),((2,7),9)]
```


Listenkompensation in Fkt.-Definitionen (1)

Abstandsberechnung vom Ursprung einer Liste von Punkten:

```
type Punkt = (Float,Float)
abstand_vom_ursprung :: [Point] -> [Float]
abstand_vom_ursprung ps
  = [sqrt (squ x + squ y) | (x,y) <- ps]
abstand_vom_ursprung [(3.0,4.0),(1.0,1.0),(-1.0,3.0)]
  ->> [5.0,1.414,3.1623]
```

Berechnung der nichtleeren Postfixe einer Zeichenreihe:

```
nichtleere_postfixe :: String -> [String]
nichtleere_postfixe s
  = [reverse (take n reverse s) |
      n <- [length s, (length s) - 1..1]]
nichtleere_postfixe "Curry"
  ->> ["Curry", "urry", "rry", "ry", "y"]
```

Kapitel 6.3

Konstruktoren, Operatoren

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3

6.4

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Konstruktoren vs. Operatoren

...am Beispiel von Listen.

Listen besitzen

- genau einen **Konstruktor**: `(:)`
- viele **Operatoren**: Z.B. `(++)`

Konstruktoren führen zu **eindeutigen** Darstellungen von Werten, **Operatoren** (i.a.) nicht:

```
[42,17,4] == (42:(17:(4:[])))           -- Eindeutige Dar-
                                     -- stellung von [42,17,4]
                                     -- mittels des Konstruktors (:).
```

```
[42,17,4] == [42,17] ++ [] ++ [4]      -- Viele Darstel-
                                     -- lungen von
                                     -- [42,17,4]
                                     -- mittels des
                                     -- Operators (++).
```

Verwendbarkeit von Konstruktoren, Operatoren

...in Mustern.

Operatoren (wie z.B. **(++)** für Listen) implizieren anders als **Konstruktoren** (**(:)** für Listen) (i.a.) keine **Zerlegungseindeutigkeit** von Werten.

In **Musterausdrücken** dürfen deshalb ausschließlich **Konstruk-toren**, keine **Operatoren** verwendet werden:

- `xs @ (x : (y : (z : zs)))` **zulässig** als Muster.
- `xs @ (ys ++ zs)` **unzulässig** als Muster.

Kapitel 6.4

Leseempfehlungen

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

6.1

6.2

6.3




6.4

Umgekehrte
Klassen-
zimmer I

Hinweis

Aufgabe

Basisleseempfehlungen für Kapitel 6

-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 4.4, Pattern matching; Kapitel 5, List comprehensions)
-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 3, Syntax in Functions – Pattern Matching)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 5.5, Lists in Haskell; Kapitel 5.6, List comprehensions; Kapitel 7.1, Pattern matching revisited; Kapitel 7.2, Lists and list patterns; Kapitel 10.1, Patterns of computation over lists; Kapitel 17.3, List comprehensions revisited)

Weiterführende Leseempfehlungen für Kap. 6

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 5.1.4, Automatische Erzeugung von Listen)
-  Antonie J. T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 7.4, List comprehensions)
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 12, Barcode Recognition – List Comprehensions)
-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 13, Mehr syntaktischer Zucker)

Umgekehrtes Klassenzimmer I

...zur Übung, Vertiefung

...nach Eigenstudium von Teil I 'Einführung':

- Zwar weiß ich viel...

Als Bonusthema, so weit die Zeit erlaubt:

- Programme erstellen und kommentieren, aber wie?

Zwar weiß ich viel...

doch möchte ich alles wissen.

Wagner, Assistent von Faust
Johann Wolfgang von Goethe (1749-1832)
dt. Dichter und Naturforscher

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Zwar weiß
ich viel...

Bonusthema:
Programme
kommentieren, aber
wie?

Hinweis

Aufgabe

Zeit für Ihren Zweifel, Ihre Fragen!

Der Zweifel ist der Beginn der Wissenschaft.

Wer nichts anzweifelt, prüft nichts.

Wer nichts prüft, entdeckt nichts.

Wer nichts entdeckt, ist blind und bleibt blind.

Pierre Teilhard de Chardin (1881-1955)

franz. Jesuit, Theologe, Geologe und Paläontologe

Die großen Fortschritte in der Wissenschaft
beruhen oft, vielleicht stets, darauf, dass man
eine zuvor nicht gestellte Frage doch,
und zwar mit Erfolg, stellt.

Carl Friedrich von Weizsäcker (1912-2007)

dt. Physiker und Philosoph

...entdecken Sie den **Wagner** in sich!

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Zwar weiß
ich viel...

Bonusthema:
Programme
kommentieren, aber
wie?

Hinweis

Aufgabe

Bonusthema

Programme erstellen und kommentieren,
aber wie?

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassenzimmer

Zwar weiß
ich viel...

Bonusthema:
Programme
kommentieren,
aber wie?

Hinweis

Aufgabe

Exzerpt von den Angaben:

Kommentieren Sie Ihr Programm zweckmäßig, aussagekräftig und problemangemessen. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten. Versehen Sie alle Funktionen, die Sie zur Lösung der Aufgaben benötigen, mit ihren Typspezifikationen, d.h. geben Sie deren syntaktische Signatur oder kurz, Signatur, explizit an.

Oder auch: *Beschreiben Sie knapp, aber gut nachvollziehbar, wie Ihre Rechenvorschrift vorgeht.*

Worauf ist dabei zu achten? Woran kann man sich orientieren?

Hieran: **Zehn Gebote** für die

► zweckmäßige, aussagekräftige und problemangemessene Erstellung und Kommentierung von Programmen.

Das erste Gebot

1. Gebot:

Verwende aussagekräftige Namen als Bezeichner für Funktionen, Parameter und Typen!

```
type Nat0 = Int
otherwise = True

fakultaet :: Nat0 -> Nat0
fakultaet n
  | n == 0    = 1
  | otherwise = n * fakultaet (n-1)
```

statt:

```
x :: Int -> Int
x y
  | y == 0 = 1
  | 1 > 0  = y * x (y-1)
```

Das zweite Gebot

2. Gebot:

Kommentiere Programmteile, Funktionen, deren Bedeutung nicht unmittelbar ersichtlich ist! Leitfragen dabei:

1. *Was macht die Funktion?* (Erklärung der logischen Funktionskomponente)
2. *Wie macht die Funktion das?* (Erklärung der operativen Funktionskomponente)

```
fakultaet :: Nat0 -> Nat0
fakultaet n | n == 0      = 1
             | otherwise = n * fakultaet (n-1)
```

- *Was macht die Funktion?* Angewendet auf eine positive ganze Zahl n , berechnet fakultaet den Wert $n!$; angewendet auf eine echt negative ganze Zahl, terminiert fakultaet nicht (regulär).
- *Wie macht die Funktion das?* Ist der Argumentwert 0, wird unmittelbar 1 und somit der Wert $0!$ zurückgegeben; ist der Argumentwert von 0 verschieden, so wird der Wert $n!$ gemäß der math. Definition von $!$ rekursiv als Produkt aus n und $(n - 1)!$ berechnet.

Das dritte Gebot

3. Gebot:

Formatiere den Programmtext entsprechend der logischen Struktur; verwende übersichtliches, 'sicheres' Layout!

```
fakultaet :: Nat0 -> Nat0
fakultaet n
  | n == 0    = 1
  | otherwise = n * fakultaet (n-1)
```

statt:

```
fakultaet :: Nat0 ->
  Nat0
fakultaet n | n == 0
  = 1
  | otherwise = n *
    fakultaet (n
      - 1)
```

Das vierte Gebot

4. Gebot:

Ist eine Funktion durch mehrere Gleichungen definiert, achte darauf, dass die Reihenfolge der Gleichungen irrelevant ist!

```
type Nat0 = Integer
fakultaet :: Nat0 -> Nat0
fakultaet n | n == 0 = 1
             | n > 0  = n * fakultaet (n-1)
```

statt:

```
type Nat0 = Integer
otherwise = True

fakultaet :: Nat0 -> Nat0
fakultaet n | n == 0    = 1
             | otherwise = n * fakultaet (n-1)
```

Korrekt, aber Reihenfolge relevant!

```
type Nat0 = Integer
fakultaet :: Nat0 -> Nat0
fakultaet n | n > 0  = n * fakultaet (n-1)
             | n == 0 = 1
```

```
type Nat0 = Integer
otherwise = True

fakultaet :: Nat0 -> Nat0
fakultaet n | otherwise = n * fakultaet (n-1)
             | n == 0    = 1
```

Inkorrekt! Terminiert nie (regulär)!

Zur **Überlegung**: Die rechte **grüne** Formulierung der Funktion **fakultaet** ist performanter als die linke. Warum?

Das fünfte Gebot

5. Gebot:

Vermeide, Funktionen unnötig zu schachteln!

```
ist_gerade :: Integer -> Bool
ist_gerade n
  | n == 0 = True
  | n > 0  = ist_ungerade (n-1)
ist_ungerade :: Integer -> Bool
ist_ungerade n
  | n == 0 = False
  | n > 0  = ist_gerade (n-1)
```

statt:

```
ist_gerade :: Integer -> Bool
ist_gerade n
  | n == 0 = True
  | n > 0  = ist_ungerade (n-1)
where
  ist_ungerade :: Integer -> Bool
  ist_ungerade n
    | n == 0 = False
    | n > 0  = ist_gerade (n-1)
```

Ohne Schachtelung: Beide Fkt. sind nutzbar!

```
ist_gerade 4 ->> True
ist_ungerade 5 ->> False
```

Mit Schachtelung: Nur eine Fkt. ist nutzbar!

```
ist_gerade 4 ->> True
ist_ungerade 5 ->> Fehler: Idf. unbekannt
```

Tipp: Ungewollte Sichtbarkeit von Bezeichnern kann mit Exportbeschränkungen in Moduldeklarationen (s. Kap. 17.3) angemessener und besser vermieden werden als durch Definitionslokalität.

Das sechste Gebot

6. Gebot:

Benenne konstante Ausdrücke und Typausdrücke!

```
type Eigenschaft    = (Double -> Bool)
type Transformator  = (Double -> Double)
g                   = 9.81      -- durchschn. Erdbeschleunigung in m/s2
mystisch            = 9.81      -- Zufällig wertgleich zu g, kein Zshg. mit g
pi                  = 3.141592  -- Kreiszahl
e                   = 2.71828   -- Eulersche Zahl
c                   = 299792.5  -- Lichtgeschwindigkeit im Vakuum in km/s
weltformelkonstante = mystisch2/(log c) * sqrt (pi*e/g2)
transformiere_liste :: Eigenschaft -> Transformator -> [Double] -> [Double]
  transformiere_liste _ _ [] = []
  transformiere_liste e t (z:zs)
    | e z      = ((t z) * weltformelkonstante) : (transformiere_liste e t zs)
    | not (e z) = weltformelkonstante : (transformiere_liste e t zs)

statt:
transformiere_liste :: (Double -> Bool) -> (Double -> Double) -> [Double] -> [Double]
transformiere_liste _ _ [] = []
transformiere_liste e t (z:zs)
  | e z      = ((t z) * (9.812/(log 299792.5) * sqrt (3.141592*2.71828/9.812))) : (transf... e t zs)
  | not (e z) = (9.812/(log 299792.5) * sqrt (3.141592*2.71828/9.812)) : (transf... e t zs)
```

Tipp: Konstante Ausdrücke sind weniger informativ als Bezeichner, Typausdrücke weniger als Typsynonyme. Werden Bezeichner und Typsynonyme verwendet, ist bei Programmänderungen nur an der Deklarationsstelle eine Änderung nötig, nicht an jeder Benutzungsstelle (Bsp.: `g` von `9.81` zu `9.8`).

Das siebente Gebot

7. Gebot:

Spezifiziere die Typen (zumindest) der Funktionen, deren Funktionalität nicht unmittelbar aus dem Programmtext ersichtlich ist!

```
-- Typspezifikation für add_at_rear_of_list offen-  
-- sichtlich und deshalb möglicherweise verzichtbar:  
add_at_rear_of_list xs x = xs ++ [x]  
  
-- Typspezifikation fuer magic nicht offensichtlich  
-- und deshalb unverzichtbar:  
magic = let pair x y z = z x y  
        f y = pair y y  
        g y = f (f y)  
        h y = g (g y)           (zum Typ von magic  
        in h (\x -> x)         s. Anfang v. Kap. 14)
```

Tipp: Typangaben erleichtern menschlichen Lesern das Lesen und Verstehen von Programmen; Übersetzern erleichtern sie die Ausgabe aussagekräftiger Meldungen im Fehlerfall.

Das achte Gebot

8. Gebot:

Vermeide rekursive Definitionen, wenn möglich!

```
type Nat0      = Integer
type Nat1      = Integer
type Faktor    = Nat1
type Faktorkandidat = Nat1

faktoren_von :: Nat0 -> [Faktoren]
faktoren_von n = [f | f <- [1..n], f 'ist_faktor_von' n]

ist_faktor_von :: Faktorkandidat -> Nat0 -> Bool
ist_faktor_von fk n = n mod fk == 0
```

statt:

```
faktoren :: Nat0 -> [Faktoren]
faktoren n = faktoren_von' 1 n
  where faktoren_von' fk n
        | fk > n           = []
        | n mod fk == 0   = fk : faktoren_von' (fk+1) n
        | True            = faktoren_von' (fk+1) n
```

Tipp: Definitionen mithilfe vordefinierter Funktionen aus Standard-Präludium oder anderer Bibliotheken oder mithilfe von Listenkomprehensionen sind oft einfacher lesbarer und verständlicher als rekursive 'ad hoc'-Formulierungen.

Das neunte Gebot

9. Gebot:

*Halte Informationen so lokal wie möglich! (engl. **information hiding**)*

```
module Main where
import Utils_1,Utils_2
f :: ...
f ... = ...useful_1...
g :: ...
g ... = ...useful_2...
h :: ...
h ... = ...useful...
statt:
module Main where
import Utils_1,Utils_2
f :: ...
f ... = ...useful_1...
g :: ...
g ... = ...useful_2...
h :: ...
h ... = ...useful...

module Utils_1 (useful_1,
                useful_2) where
useful_1 :: ...
useful_1 ... = ...help...
useful_2 :: ...
useful_2 ... = ...help...
help :: ...
help ... = ...

module Utils_2 (useful) where
useful :: ...
useful ... = ...help'
  where help' = ...

module Utils_1 where
useful_1 :: ...
useful_1 ... = ...help...
useful_2 :: ...
useful_2 ... = ...help...
help :: ...
help ... = ...

module Utils_2 where
useful :: ...
useful ... = ...help'
help' :: ...
help' ... = ...
```

(oder gar statt aller Deklarationen in Main)

Tipp: Exportbeschränkungen für Module (s. Kap. 17.3) und abstrakte Datentypen (s. Kap. 17.4) sind geeignete Mittel, um Implementierungsdetails zu verbergen und Programme robust gegenüber Änderungen zu machen.

Das zehnte Gebot

10. Gebot:

Fasse zusammengehörige häufig verwendbare Funktionen in Modulen/Bibliotheken zusammen!

```
module Superspeziell where
import Oftbrauchbar
superspeziell_1 :: ...
superspeziell_1 ... = ...oftbrauchbar_1...
superspeziell_2 :: ...
superspeziell_2 ... = ...oftbrauchbar_2...
superspeziell_3 :: ...
superspeziell_3...

statt:
module Superspeziell where
superspeziell_1 :: ...
superspeziell_1 ... = ...oftbrauchbar_1...
superspeziell_2 :: ...
superspeziell_2 ... = ...oftbrauchbar_2...
superspeziell_3 :: ...
superspeziell_3...

oftbrauchbar_1 :: ...
oftbrauchbar_2 :: ...

module Oftbrauchbar where
oftbrauchbar_1 :: ...
oftbrauchbar_2 :: ...
oftbrauchbar_3 :: ...
oftbrauchbar_4 :: ...
oftbrauchbar_5 :: ...
...

module Superspeziell'
import Oftbrauchbar
superspeziell :: ...
superspeziell ... =
...oftbrauchbar_1...
...

module Superspeziell' where
import Superspeziell
superspeziell :: ...
superspeziell ... = ...oftbrauchbar_1...
...
```

Tipp: Module/Bibliotheken unterstützen und erleichtern Wiederverwendung und verringern die Wahrscheinlichkeit unnötigen Wiedererfindens des Rades (s. [Kap. 17.2](#)).

Für die vorgestellten 10 Gebote

...und Richtlinien guter Programmerstellung und -kommentierung siehe:



Ralf Hinze. *Einführung in die funktionale Programmierung mit Miranda*. B.G. Teubner, 1992. (Kapitel 9.4, Programmierrichtlinien für Miranda)

...10 Gebote und Richtlinien, die nicht nur für **Miranda** und **Haskell** nützlich sind, sondern in gleicher oder angepasster Art und Weise auch für **andere Sprachen**.

Freier Platz

...für weitere sinnvolle Richtlinien und Gebote:

11. Gebot: ...

12. Gebot: ...

13. Gebot: ...

14. Gebot: ...

15. Gebot: ...

...

Vortrag II

Teil II

Kap. 2

Kap. 3

Kap. 4

Kap. 5

Kap. 6

Umgekehrte
Klassen-
zimmer I

Zwar weiß
ich viel...

Bonusthema:
Programme
kommentieren, aber
wie?

Hinweis

Aufgabe

Hinweis

...für das Verständnis von **Vorlesungsteil II** ist eine über den unmittelbaren Inhalt von **Vortrag II** hinausgehende weitergehende und vertiefende Beschäftigung mit dem Stoff nötig; siehe:

- ▶ **vollständige Lehrveranstaltungsunterlagen**

...verfügbar auf der Webseite der Lehrveranstaltung:

http://www.complang.tuwien.ac.at/knoop/fp185A05_ws2021.html

Aufgabe bis Dienstag, 27.10.2020

...selbstständiges Durcharbeiten von Teil II 'Grundlagen', Kap. 2 bis Kap. 6 und von Leit- und Kontrollfragenteil II zur Selbsteinschätzung und als Grundlage für die umgekehrte Klassenzimmersitzung am 27.10.2020:

Vortrag, umgek. Klassenz.	Thema Vortrag	Thema umgek. Klassenz.
Di, 06.10.2020, 08:15-09:45	Teil I	n.a. / Vorbesprechung
Di, 13.10.2020, 08:15-09:45	Teil II	Teil I
Di, 27.10.2020, 08:15-09:45	Teil III	Teil II
Mi, 04.11.2020, 08:15-09:45	Teil IV	Teil III
Mi, 18.11.2020, 08:15-09:45	Teil V	Teil IV
Mi, 02.12.2020, 08:15-09:45	Teil VI	Teil V
Mi, 16.12.2020, 08:15-09:45	Teil VII	Teil VI