

## Heutiges Thema...

Struktur, insbesondere...

- Struktur von *Programmen*
  - Klassifikation von Rekursionstypen  
~> Anmerkungen zu Effektivität und Effizienz  
~> Komplexitätsklassen
  - Aufrufgraphen
- Ergänzungen zu Funktionstermen
  - Currifizierte vs. uncurrifizierte Funktionen
- Struktur von *Daten*
  - Algebraische Datentypen (`data Tree = ...`)
  - Typsynonyme (`type Student = ...`)
  - Spezialitäten (`newtype State = ...`)

Hinweis: Die kursiv hervorgehobenen Punkte sind bereits in der Vorlesung am 03.11.2005 besprochen worden.

## Ergänzungen zu Funktionstermen (1)

Betrachten wir noch einmal die Funktion `add`:

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

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

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

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

Wir können diese Funktion `doubleInc` nennen...

## Ergänzungen zu Funktionstermen (2)

...und in natürlicherweise wie folgt definieren:

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

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

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

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

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

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

## Ergänzungen zu Funktionstermen (3)

Vergleiche `doubleInc`, `add 2`

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

mit

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

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

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

## "Erfahrenheits"-Faustregel

Die Implementierung einer Funktion wie `doubleInc`

- durch

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

...zeigt, dass vermutlich noch wenig Erfahrung mit funktionaler Programmierung vorliegt

- durch

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

...zeigt, dass bereits mehr Erfahrung mit funktionaler Programmierung vorliegt

- durch

```
\n -> 2*n
```

...zeigt gleichfalls, dass bereits mehr Erfahrung mit funktionaler Programmierung vorliegt, und darüberhinaus, dass in der konkreten Anwendungssituation ein Name, unter dem auf die Funktion mit der Bedeutung "doubleInc" zugegriffen werden könnte, keine Rolle spielt.

~> *Beispiele dafür später, aber...*

## Als Ausblick... (1)

...ein kleines Beispiel schon jetzt:

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

Anwendung:

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

...oder genauso gut

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

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

```
map (add' 2) [1,2,3]
```

~> später mehr dazu unter dem Stichwort "Funktionale", speziell Funktionale auf Listen...

## Als Ausblick... (2)

Als Beispiel aussagekräftiger und überzeugender:

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

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

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

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

## Ein anderer Nachtrag: Operatoren in Haskell

Operatoren in Haskell sind...

- ...grundsätzlich *Präfixoperatoren*, insbesondere alle selbst-deklarierten Operatoren (*vulgo*: selbstdeklarierte Funktionen)  
*Beispiele:* `fac 5`, `imax 2 3`, `tripleMax 2 5 3`,...
- ...in einigen wenigen Fällen *Infixoperatoren*, dies gilt insbesondere für arithmetische Operatoren  
*Beispiele:* `2+3`, `3*5`, `7-4`, `5^3`,...

## Binäre Operatoren in Haskell: Infix- vs. Präfix

Für binäre Operatoren in Haskell gilt...

- Binäre Operatoren `bop`, die standardmäßig als...
  - Präfixoperatoren verwendet werden, können in der Form `'bop'` als Infixoperator verwendet werden  
*Beispiel:* `2 'imax' 3` (statt standardmäßig `imax 2 3`)
  - Infixoperatoren verwendet werden, können in der Form `(bop)` als Präfixoperator verwendet werden  
*Beispiel:* `(+) 2 3` (statt standardmäßig `2+3`)

## Abschließend zu Funktionstermen (1)

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

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

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

## Abschließend zu Funktionstermen (2)

...hier noch einmal zusammen mit ihren Implementierungen:

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

```
add m n = m+n
```

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

```
add' (m,n) = m+n
```

*Sprechweise:* Die Funktion...

- `add` ist *curryfiziert*
- `add'` ist *uncurryfiziert*

## Curryfiziert vs. uncurryfiziert (1)

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

Erfolgt die Konsumation mehrerer Argumente durch Funktionen...

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

*Beispiele:*

```
Funktion add curryfiziert:    add 2 3    bzw. (add 2) 3
```

```
Funktion add' uncurryfiziert: add' (2,3)
```

## Curryfiziert vs. uncurryfiziert (2)

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

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

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

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

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

*Intuitiv:*

- *Curryfizieren* ersetzt Produkt-/Tupelbildung "x" durch Funktionspfeil "→".
- *Decurryfizieren* ersetzt Funktionspfeil "→" durch Produkt-/Tupelbildung "x".

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

## Curryfiziert vs. uncurryfiziert (3)

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

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

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

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

```
curry f :: a -> b -> c
```

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

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

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

## Im Beispiel...

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

```
add m n = m+n
```

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

```
add' (m,n) = m+n
```

Damit gilt:

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

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

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

```
curry add' 17 4
```

```
⇒ add' (17,4) ⇒ 17+4 ⇒ 21
```

```
uncurry add (17,4)
```

```
⇒ add 17 4 ⇒ 17+4 ⇒ 21
```

## Curryfiziert oder uncurryfiziert?

...das ist hier die Frage.

Zum einen...

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

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

- Sachargument  
... (nur) Funktionen in curryfizierter Darstellung unterstützen *partielle Auswertung*  
~> Funktionen liefern Funktionen als Ergebnis!

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

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

## Datentypdeklarationen in Haskell

...selbstdefinierte (neue) Datentypen in Haskell!

↪ Haskell's Vehikel dafür: *Algebraische Typen*

*Algebraische Typen* erlauben uns zu definieren...

- Summentypen
  - *Spezialfälle*
  - \* Produkttypen
  - \* Aufzählungstypen

In der Praxis besonders wichtige Varianten...

- Rekursive Typen (↪ "unendliche" Datenstrukturen)
- Polymorphe Typen (↪ Wiederverwendung): *Später!*

## Grundlegende Typmuster

*Aufzählungs-, Produkt- und Summentypen:*

- *Aufzählungstypen*
  - ↪ Typen mit endlich vielen Werten
  - ...typisches Beispiel: Typ Jahreszeiten mit Werten  
Fruehling, Sommer, Herbst und Winter.
- *Produkttypen* (synonym: *Verbundtypen*)
  - ↪ Typen mit möglicherweise unendlich vielen Tupelwerten
  - ...typisches Beispiel: Typ Person mit Werten  
(Adam, maennlich, 27), (Eva, weiblich, 25), etc.
- *Summentypen* (synonym: *Vereinigungstypen*)
  - ↪ Vereinigung von Typen mit möglicherweise jeweils unendlich vielen Werten
  - ...typisches Beispiel: Typ Verkehrsmittel als Vereinigung der  
(Werte der) Typen Auto, Schiff, Flugzeug, etc.

## Zum Einstieg und Vergleich... (1)

Realisierung von Typdefinitionen in imperativen Sprachen  
...hier am Bsp. von Pascal

- *Aufzählungstypen*

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

- *Produkttypen*

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

## Zum Einstieg und Vergleich... (2)

- *Summentypen*

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

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

## Zum Einstieg und Vergleich... (3)

Aufzählungstypen, Produkttypen, Summentypen...

- In Pascal ...drei verschiedene Sprachkonstrukte
- In Haskell ...ein *einheitliches* Sprachkonstrukt  
↪ die *algebraische Datentypdefinition*

## Zum Einstieg und Vergleich... (4)

Obige Einstiegsdatentypbeispiele in Haskell...

- Aufzählungstyp Jahreszeiten

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

- Produkttyp Person

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

mit

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

## Zum Einstieg und Vergleich... (5)

- Summentyp Verkehrsmittel

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

In obiger Form offenbar wenig transparent im Vergleich zu:

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

## Zum Einstieg und Vergleich... (5)

- Summentyp Verkehrsmittel

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

mit

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

*Man erkennt:* Typsynonyme bringen *Transparenz* ins Programm!

## Algebraische Datentypen in Haskell

...das allg. Muster der algebraischen Datentypdefinition:

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

Sprechweisen:

- Typename ... Typname/-identifikator
- $\text{Con}_i$ ,  $i = 1..n$  ... Konstruktoren/-identifikatoren
- $k_i$ ,  $i = 1..n$  ... Stelligkeit des Konstruktors  $\text{Con}_i$ ,  $k_i \geq 0$ ,  $i = 1, \dots, n$

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

## Konstruktoren...

...können als Funktionsdefinitionen gelesen werden:

```
Coni :: ti1 -> ... -> tiki -> Typename
```

Konstruktion von Werten eines algebraischen Datentyps durch...

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

```
Coni vi1 ... viki :: Typename
```

wobei  $v_{i1} :: t_{i1} \dots v_{ik_i} :: t_{ik_i}$ ,  $j = 1, \dots, k_i$

Beispiele:

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

## Aufzählungstypen (1)

Nullstellige Konstruktoren führen auf Aufzählungstypen...

Beispiele:

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

Insbesondere ist der Typ der Wahrheitswerte...

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

...Beispiel eines in Haskell vordefinierten Aufzählungstyps.

## Aufzählungstypen (2)

Funktionsdefinitionen über Aufzählungstypen...

~> üblicherweise mit Hilfe von Pattern-matching.

Beispiele:

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

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

## Produkttypen

(Alternativenlose) mehrstellige Konstruktoren führen auf Produkttypen...

Beispiel:

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

data Person = Pers Name Geschlecht Alter
```

Beispiele: ...für Werte des Typs Person.

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

Beachte: Funktionalität der Konstruktorenfunktion ist hier...

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

## Summentypen (1)

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

Beispiel:

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

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

Die Varianten einer Summe werden durch "|" getrennt.

## Summentypen (2)

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

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

## Zwischenfazit

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

Haskell offeriert...

- Summentypen

mit den beiden Spezialfällen

- Produkttypen  
~> nur ein Konstruktor, mehrstellig
- Aufzählungstypen  
~> ein oder mehrere Konstruktoren, alle nullstellig

## Rekursive Typen (1)

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

*Technisch:*

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

*Beispiel:* ...(arithmetische) Ausdrücke

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

## Rekursive Typen (2)

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

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

...rekursive Typen ermöglichen potentiell unendliche Datenstrukturen!

## Rekursive Typen (3)

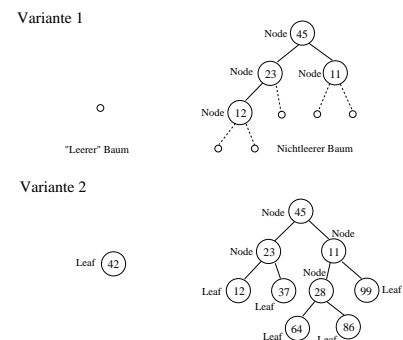
*Weiteres Beispiel:*

Binärbäume, hier zwei verschiedene Varianten:

```
data BinTree1 = Nil | Node Int BinTree1 BinTree1
data BinTree2 = Leaf Int | Node Int BinTree2 BinTree2
```

## Rekursive Typen (4)

Veranschaulichung der Binärbäumevarianten 1&2 anhand eines Beispiels:



## Rekursive Typen (5)

*Beispiele* ...für Funktionen über Binärbäumevariante 1.

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

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

Mit diesen Definitionen...

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

## Rekursive Typen (6)

*Beispiele* ...für Funktionen über Binärbäumevariante 2.

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

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

Mit diesen Definitionen...

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

## Wechselweise rekursive Typen

...ein Spezialfall rekursiver Typen.

*Beispiel:*

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

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

## Typsynonyme (1)

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

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

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

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

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

## Typsynonyme (2)

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

Somit gilt:

Durch `type`-Deklarationen eingeführte Typsynonyme...

- tragen zur Dokumentation bei und
- erleichtern (i.a.) das Programmverständnis

aber...

- führen nicht zu (zusätzlicher) Typsicherheit!

## Ein (pathologisches) Beispiel

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

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

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

Mit obigen Deklarationen...

```
currencyConverter maxTemp => 60.627
```

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

## Ein reales Beispiel

Anflugsteuerung einer Sonde zum Mars...

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

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

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

```
verbleibendeFlugzeit 18524.34 1523.79
```

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

## Produkttypen vs. Tupeltypen (1)

Der Typ `Person` als...

- *Produkttyp*

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

- *Tupeltyp*

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

*Vordergründiger Unterschied:*

...in der *Tupeltyp*-variante fehlt der Konstruktor (in diesem Bsp.: `Pers`)

## Produkttypen vs. Tupeltypen (2)

...eine Abwägung von Vor- und Nachteilen.

*Produkttypen* und ihre typischen...

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

## Andererseits...

Mit *Produkttypen* statt *Typsynonymen*...

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

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

...wäre eine Funktionsdefinition im Stile von

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

insbesondere auch ein Aufruf wie...

```
currencyConverter maxTemp
```

durch das Typsystem von Haskell verhindert!

## Somit als kurzes Fazit... (1)

...unserer Überlegungen:

- *Typsynonyme* wie...

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

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

- *Produkttypen* wie...

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

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

## Somit als kurzes Fazit... (2)

In ähnlicher Weise...

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

...

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

---

## Spezialitäten

...die `newtype`-Deklaration:

```
newtype Miles = Mi Float
```

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

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

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

---

## Polymorphe Typen

...demnächst!

---

## Vorschau auf die kommenden Aufgabenblätter...

Ausgabe des...

- vierten Aufgabenblatts: Mi, den 16.11.2005  
...Abgabetermine: Mi, den 23.11.2005, und Mi, den 30.11.2005, jeweils 12:00 Uhr
- fünften Aufgabenblatts: Mi, den 23.11.2005  
...Abgabetermine: Mi, den 30.11.2005, und Mi, den 07.12.2005, jeweils 12:00 Uhr
- sechsten Aufgabenblatts: Mi, den 30.11.2005  
...Abgabetermine: Mi, den 07.12.2005, und Mi, den 14.12.2005, jeweils 12:00 Uhr

---

## Vorschau auf die nächsten Vorlesungstermine...

- Do, 24.11.2005, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal
- Do, 01.12.2005, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal
- *Do, 08.12.2005: Keine Vorlesung: Feiertag!*
- Do, 15.12.2005, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal