# Programmierung und Modellierung mit Haskell

## Benutzerdefinierte Datentypen

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

28. April 2014

# Tupel / Kartesisches Produkt



Kartesisches Produkt: $A \times B = \{(a, b) \mid a \in A \text{ und } b \in B\}$

Beispiel:

$$\{1, 7\} \times \{\diamondsuit, \heartsuit, \clubsuit\} = \{(1, \diamondsuit), (1, \heartsuit), (1, \clubsuit), (7, \diamondsuit), (7, \heartsuit), (7, \clubsuit)\}$$

Für endliche Mengen gilt: $|A \times B| = |A| \cdot |B|$

In Haskell können wir Produkte einfach nutzen:

```
> :t (3,True)
(Integer, Bool)

> :t ("Hello",'a',(True,True))
(String, Char, (Bool, Bool))
```

Sowohl den Typ eines Produktes als auch dessen Werte schreiben wir mit runden Klammern.

# Typabkürzungen



Was bedeutet der Wert `(17.3, 17.3, 80.0)`?

Problem: Keine Typsicherheit! Don't do this!
Typakürzung sind transparent, d.h. Werte des Typs `Point3D` können (versehentlich) auch dort verwendet werden, wo `Circle` oder `Monster` erwartet wird!

Merke: Typabkürzungen mit `type` erhöhen die Lesbarkeit langer generischer Typen, eignen sich aber nicht für die Daten Modellierung!

# Typabkürzungen



Was bedeutet der Wert `(17.3, 17.3, 80.0)`?

- Kreis, repräsentiert mit Mittelpunkt-Koordinaten und Radius?
  `type Circle = (Double, Double, Double)`

Problem: Keine Typsicherheit! Don't do this!
Typakürzung sind transparent, d.h. Werte des Typs `Point3D` können (versehentlich) auch dort verwendet werden, wo `Circle` oder `Monster` erwartet wird!
Merke: Typabkürzungen mit `type` erhöhen die Lesbarkeit langer generischer Typen, eignen sich aber nicht für die Daten Modellierung!

# Typabkürzungen



Was bedeutet der Wert `(17.3, 17.3, 80.0)`?

- Kreis, repräsentiert mit Mittelpunkt-Koordinaten und Radius?
  `type Circle = (Double, Double, Double)`
- Ein Punkt in einem dreidimensionalen Raum?
  `type Point3D = (Double, Double, Double)`

Problem: Keine Typsicherheit! Don't do this!
Typakürzung sind transparent, d.h. Werte des Typs `Point3D` können (versehentlich) auch dort verwendet werden, wo `Circle` oder `Monster` erwartet wird!

Merke: Typabkürzungen mit `type` erhöhen die Lesbarkeit langer generischer Typen, eignen sich aber nicht für die Daten Modellierung!

# Typabkürzungen



Was bedeutet der Wert `(17.3, 17.3, 80.0)`?

- Kreis, repräsentiert mit Mittelpunkt-Koordinaten und Radius?
  `type Circle = (Double, Double, Double)`
- Ein Punkt in einem dreidimensionalen Raum?
  `type Point3D = (Double, Double, Double)`
- Eckdaten einer Spielfigur in einem Echtzeitstrategiespiel?
  `type Monster = (Double, Double, Double)`

Problem: Keine Typsicherheit! Don't do this!
Typakürzung sind transparent, d.h. Werte des Typs `Point3D` können (versehentlich) auch dort verwendet werden, wo `Circle` oder `Monster` erwartet wird!

Merke: Typabkürzungen mit `type` erhöhen die Lesbarkeit langer generischer Typen, eignen sich aber nicht für die Daten Modellierung!

# Typabkürzungen



Was bedeutet der Wert `(17.3, 17.3, 80.0)`?

- Kreis, repräsentiert mit Mittelpunkt-Koordinaten und Radius?
  `type Circle = (Double, Double, Double)`
- Ein Punkt in einem dreidimensionalen Raum?
  `type Point3D = (Double, Double, Double)`
- Eckdaten einer Spielfigur in einem Echtzeitstrategiespiel?
  `type Monster = (Double, Double, Double)`

Problem: Keine Typsicherheit! Don't do this!
Typakürzung sind transparent, d.h. Werte des Typs `Point3D` können (versehentlich) auch dort verwendet werden, wo `Circle` oder `Monster` erwartet wird!

Merke: Typabkürzungen mit `type` erhöhen die Lesbarkeit langer generischer Typen, eignen sich aber nicht für die Daten Modellierung!

# Typabkürzungen



Was bedeutet der Wert `(17.3, 17.3, 80.0)`?

- Kreis, repräsentiert mit Mittelpunkt-Koordinaten und Radius?
  `type Circle =  (Double, Double, Double)`
- Ein Punkt in einem dreidimensionalen Raum?
  `type Point3D = (Double, Double, Double)`
- Eckdaten einer Spielfigur in einem Echtzeitstrategiespiel?
  `type Monster = (Double, Double, Double)`

Problem: Keine Typsicherheit! Don't do this!
Typakürzung sind transparent, d.h. Werte des Typs `Point3D` können (versehentlich) auch dort verwendet werden, wo `Circle` oder `Monster` erwartet wird!

Merke: Typabkürzungen mit `type` erhöhen die Lesbarkeit langer generischer Typen, eignen sich aber nicht für die Daten Modellierung!

# Typabkürzungen



Was bedeutet der Wert `(17.3, 17.3, 80.0)`?

- Kreis, repräsentiert mit Mittelpunkt-Koordinaten und Radius?
  `type Circle =  (Double, Double, Double)`
- Ein Punkt in einem dreidimensionalen Raum?
  `type Point3D = (Double, Double, Double)`
- Eckdaten einer Spielfigur in einem Echtzeitstrategiespiel?
  `type Monster = (Double, Double, Double)`

Problem: Keine Typsicherheit! Don't do this!
Typakürzung sind transparent, d.h. Werte des Typs `Point3D` können (versehentlich) auch dort verwendet werden, wo `Circle` oder `Monster` erwartet wird!

Merke: Typabkürzungen mit `type` erhöhen die Lesbarkeit langer generischer Typen, eignen sich aber nicht für die Daten Modellierung!

# Benutzerdefinierte Datentypen

Haskell erlaubt benutzerdefinierte Datentypen. Der Programmierer erstellt einen neuen Typ und legt dessen Werte fest.

```
data MeinTyp = MeinKonstruktor Int Bool
```

Datentypdeklaration beginnen mit dem Schlüsselwort `data`, dann ein frischer Typname, der wie immer mit Großbuchstaben beginnt. Dahinter wird der **Konstruktor** mit den Typen seiner Argumente aufgelistet. Konstruktoren kann man als Funktionen betrachten, die Werte des Typs konstruieren:

```
> :t MeinKonstruktor
MeinKonstruktor :: Int -> Bool -> MeinTyp
```

- Konstruktoren beginnen mit einem Großbuchstaben.
- Infix-Konstruktoren beginnend mit `:` auch erlaubt.
- Bezeichner von Konstruktoren und Typen dürfen gleich sein, da Konstruktoren Ausdrücke sind.

# Konstruktoren in Mustervergleichen

Haskell kann mit derart konstruierten Werten benutzerdefinierter zuerst wenig anfangen; nicht einmal Bildschirmausgabe ist möglich.

Mithilfe von Pattern-Matching können wir jedoch die ursprünglichen Argumente eines Konstruktors wieder auspacken:

```
myShow :: MeinTyp -> String
myShow (MeinKonstruktor i True)  = "T(" ++ (show i)++")"
myShow (MeinKonstruktor i False) = "F(" ++ (show i)++")"
```

Konstruktoren mit Argumenten immer Klammern;
die Reihenfolge der Argumente ist zu beachten!

*Hinweis:* GHC kann solche Hilfsfunktionen automatisch erstellen

```
data MeinTyp = MeinKonstruktor Int Bool
  deriving (Eq, Show)
```

`Show` ist die notwendige Typklasse zur Anzeige von Werten.
Das Thema Typklassen wird demnächst behandelt werden.

# Konstruktoren in Mustervergleichen

Haskell kann mit derart konstruierten Werten benutzerdefinierter zuerst wenig anfangen; nicht einmal Bildschirmausgabe ist möglich.

Mithilfe von Pattern-Matching können wir jedoch die ursprünglichen Argumente eines Konstruktors wieder auspacken:

```
myShow :: MeinTyp -> String
myShow (MeinKonstruktor i True)  = "T(" ++ (show i)++")"
myShow (MeinKonstruktor i False) = "F(" ++ (show i)++")"
```

Konstruktoren mit Argumenten immer Klammern;
die Reihenfolge der Argumente ist zu beachten!

*Hinweis:* GHC kann solche Hilfsfunktionen automatisch erstellen

```
data MeinTyp = MeinKonstruktor Int Bool
  deriving (Eq, Show)
```

`Show` ist die notwendige Typklasse zur Anzeige von Werten.
Das Thema Typklassen wird demnächst behandelt werden.

# Ausblick: Typklassen

Eine **Typklasse** ist eine Mengen von Typen, wofür festgelegte Funktionen definiert sind.

Beispiel: Die Typklasse `Show` ist die Klasse aller Typen, für welche mindestens folgende Funktion definiert sein muss:

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

*Gelesen:* Für alle Typen `a` aus der Typklasse `Show`, bildet die Funktion `show` Werte des Typs `a` auf Werte des Typs `String` ab.
*Alternativ:* Funktion `show` bildet Werte aller **Instanzen** `a` der Typklasse `Show` auf Werte des Typs `String` ab.

Bearbeiten von Typklassen wird später behandelt. Einfache Instanzen können mit `derving` automatisch erzeugt werden.

Beispiele: Typklasse `Eq` ermöglicht Test auf Gleichheit; `Ord` ermöglicht Vergleiche; `Show` ermöglicht die Bildschirmausgabe.

# Beispiel

```
data Circle  = Circle  Double Double Double
data Point3D = Point3D Double Double Double
data Monster = Monster Double Double Double

hydralisk :: Monster
hydralisk = Monster 17.3 17.3 80.0

myCircle :: Circle
myCircle = Circle 17.3 17.3 80.0

area :: Circle -> Double
area (Circle _ _ r) = pi * r^2


> area myCircle
20106.192982974677

> area hydralisk

<interactive>:13:6:
    Couldn't match expected type `Circle' with actual type `Monster'
    In the first argument of `area', namely `hydralisk'
    In the expression: area hydralisk
    In an equation for `it': it = area hydralisk
```

# Records

Wenn ein Konstruktor viele Argumente hat, kann deren Auflistung schnell unübersichtlich werden.

```
data Person' = Person'  String Int Int Int Int
p0 = Person' "Tyrion" 135 26 7 0

data Person  = Person { name::String, height,age,mates,offspring::Int }
p2 = Person  {height=166, age=35, offspring=3, mates=3, name="Cersei"}
p3 = Person  "Jaimie" 187 35 1 3      -- alte Syntax auch noch erlaubt
```

**Record** Notation erlaubt es, die *Argumente eines Konstruktors* zu benennen; diese werden dann auch als **Felder** bezeichnet.

- Reihenfolge der Felder innerhalb geschweifter Klammern ist immer beliebig
- Datentypen können nachträglich zu Records gemacht werden: Werden keine geschweiften Klammern verwendet, gilt die Reihenfolge in der Definition wie üblich.

# Pattern Matching mit Records

Pattern Matching darf Record Syntax verwenden;
die Reihenfolge der Felder ist beliebig:

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}

showPerson :: Person -> String
showPerson Person { age=a, name=n } = n ++ ' ':show a

hasChildren :: Person -> Bool
hasChildren Person { offspring=n } | n > 0 = True
hasChildren _ = False

> showPerson p2
"Cersei 35"
```

Das Matching darf auch partiell sein, d.h. es müssen nicht alle Felder gemachted werden.

# Record Projektionen

**Projektionen** werden für jeden Feldnamen automatisch definiert:

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}
> :t name
name :: Person -> String

> name p3
"Jaimie"

> :t age
age :: Person -> Int

> age p3
35
```

# Record Pseudoupdate

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}
p1 = Person "Tyrion" 135 26 7 0
```

Funktionale "Field-updates" sind ebenfalls möglich. Dabei werden natürlich Kopien erstellt — denn bestehende Werte werden in der funktionalen Welt ja nie verändert!

```
p4 = p1 { name = "Imp" }
p5 = p1 { mates = 2 + mates p1 }

> p5
Person {name = "Tyrion", height = 135, age = 26,
        mates = 9, offspring = 0}
> p1
Person {name = "Tyrion", height = 135, age = 26,
        mates = 7, offspring = 0}
```

# Record Pseudoupdate

Records kann man bei Bedarf nachträglich leicht erweitern:

```
p6 = Person { name="Daenerys", height=157, offspring=0 }
```

ist identisch zu

```
p6 = Person "Daenerys" 157 undefined undefined 0
```

GHC gibt aber entsprechende Warnungen heraus, welche man abarbeiten sollte.

Hinweis:
Record-Syntax kann nur in Zusammenhang mit Konstruktoren verwendet werden. In anderen Sprachen entspricht dies eher benannten Argumenten für Funktionen/Methoden.
Dementsprechend gibt es auch kein Record-Subtyping in Haskell.

# NEWTYPE

Hat man dagegen nur *genau einen* Konstruktor mit *genau einem* Argument, dann bietet sich `newtype` an:

```
newtype Circle  = Circle  (Double, Double, Double)
newtype Point3D = Point3D (Double, Double, Double)
data    Monster = Monster (Double, Double, Double)
```

Anstatt dem Schlüsselwort `data` kann dann einfach `newtype` verwendet werden.

- `newtype` ist ein optimierter Spezialfall: zur Laufzeit keine Unterscheidungen zwischen den Typ und Newtyp.
- Nicht so strikt wie äquivalente Datentypdeklaration
- Eigene Klassen Instanzen möglich;
  ebenso automatische Übernahme der Klasseninstanzen des ursprünglichen Typs mit Erweiterung

# NEWTYPE

Hat man dagegen nur *genau einen* Konstruktor mit *genau einem* Argument, dann bietet sich `newtype` an:

```
newtype Circle  = Circle  (Double, Double, Double)
newtype Point3D = Point3D (Double, Double, Double)
data    Monster = Monster (Double, Double, Double)
```

Anstatt dem Schlüsselwort `data` kann dann einfach `newtype` verwendet werden.

- `newtype` ist ein optimierter Spezialfall: zur Laufzeit keine Unterscheidungen zwischen den Typ und Newtyp.
- Nicht so strikt wie äquivalente Datentypdeklaration
- Eigene Klassen Instanzen möglich;
  ebenso automatische Übernahme der Klasseninstanzen des ursprünglichen Typs mit Erweiterung

# Aufzählungen

Ein Datentyp darf auch mehrere Konstruktoren besitzen:

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

Das | liest man als "oder": Ein Wert des Typs `Bool` wurde entweder mit dem Konstruktor `True` oder dem Konstruktor `False` konstruiert.

Eine **Aufzählungen** (engl. **Ennumeration**) ist ein Datentyp, dessen Konstruktoren keinerlei Argumente besitzen.

Die Reihenfolge der Konstruktoren innerhalb der Deklaration impliziert üblicherweise eine Ordnung $\Rightarrow$ `deriving Ord`

# Beispiel

Aufzählungen werden wie gewohnt mit Pattern-Matching verarbeitet:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

next :: Day -> Day
next Mon = Tue
next Tue = Wed
next Wed = Thu
next Thu = Fri
next Fri = Sat
next Sat = Sun
next Sun = Mon
```

- Datentyp `Day` hat 7 Alternativen (Konstuktoren)
- Funktion `next` zählt einfach einen Wochentag weiter

# Beispiel

Aufzählungen werden wie gewohnt mit Pattern-Matching verarbeitet:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving (Enum)

next :: Day -> Day
next Sun = Mon
next d   = succ d

-- Unter A    nderen durch 'derving' generiert:
succ, pred :: Enum a => a -> a
toEnum     :: Enum a => Int -> a
```

- Datentyp `Day` hat 7 Alternativen (Konstuktoren)
- Funktion `next` zählt einfach einen Wochentag weiter

# Kombination: Alternativen & Argumente

```
data Früchte = Apfel (Int,Int)          -- 1 Argument
             | Birne  Int Int           -- 2 Argumente
             | Banane Int Int Double    -- 3 Argumente

meineFrüchte :: [Früchte]
meineFrüchte = [Apfel (2,90), Apfel (3,300),
                Birne 60 1, Banane 80 7 0.3]

hatApfel :: [Früchte] -> Bool
hatApfel           []  = False
hatApfel ((Apfel _):_) = True
hatApfel (  _      :t) = hatApfel t

> hatApfel meineFrüchte
True
> hatApfel (drop 2 meineFrüchte)
False
```

# Kombination: Alternativen & Argumente

```
data Früchte = Apfel (Int,Int)          -- 1 Argument
             | Birne  Int Int           -- 2 Argumente
             | Banane Int Int Double    -- 3 Argumente
```

Es empfiehlt sich meist, Alternativen in einer eigenen Funktion zu behandeln:

```
gesamtPreis :: [Früchte] -> Int
gesamtPreis []     = 0
gesamtPreis (h:t) = preis h + gesamtPreis t

preis :: Früchte -> Int
preis (Apfel (z,p))  = z * p
preis (Birne  p z )  = z * p
preis (Banane p z _) = z * p
```

```
> gesamtPreis meineFrüchte
1700
```

# Kombination: Alternativen & Argumente

```
data Früchte = Apfel  {preis::Int, anzahl::Int} -- 2 Arg.
             | Birne  {preis::Int, anzahl::Int} -- 2 Arg.
             | Banane {preis::Int, anzahl::Int,
                       krümmung::Double}        -- 3 Arg.
```

Noch einfacher geht es manchmal mit der Record-Syntax:

```
meineFrüchte :: [Früchte]
meineFrüchte = [Apfel 90 2, Apfel 300 3,
                Birne 60 1, Banane 80 7 0.3]

gesamtPreis :: [Früchte] -> Int
gesamtPreis []    = 0
gesamtPreis (h:t) = (preis h * anzahl h) + gesamtPreis t
```

Achtung: `krümmung :: Früchte -> Double` ist partielle Projektion — möglichst vermeiden, kann zu Abbruch führen!

# Rekursive Datentypen

Typdeklarationen dürfen auch (wechselseitig) rekursiv sein.

## Beispiel: Listen

```
data IntList = LeereListe | ListKnoten Int IntList

myList :: IntList
myList = ListKnoten 1 (ListKnoten 2 (LeereListe))

mySum :: IntList -> Int
mySum LeereListe = 0
mySum (ListKnoten h t) = h + mySum t

> mySum myList
3
```

# Rekursive Datentypen

## Beispiel: Binärbäume

```
data Baum = Blatt Char | Knoten Baum Char Baum
```

```
> :type Knoten
Knoten :: Baum -> Char -> Baum -> Baum
```

# Rekursive Datentypen

## Beispiel: Binärbäume

```
data Baum = Blatt Char | Knoten Baum Char Baum
```

```
> :type Knoten
Knoten :: Baum -> Char -> Baum -> Baum
```

```
myBaum :: Baum
myBaum = Knoten (Blatt 'a') 'T'
          (Knoten (Blatt 'z') 'n' (Blatt '!'))
```

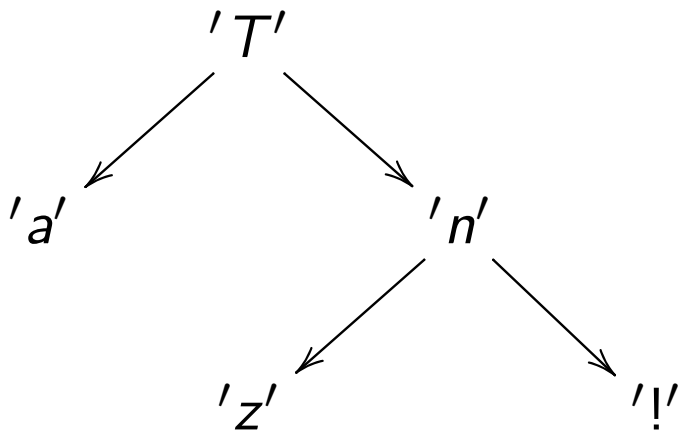
# Rekursive Datentypen

## Beispiel: Binärbäume

```
data Baum = Blatt Char | Knoten Baum Char Baum
```

```
> :type Knoten
Knoten :: Baum -> Char -> Baum -> Baum
```

```
myBaum :: Baum
myBaum = Knoten (Blatt 'a') 'T'
          (Knoten (Blatt 'z') 'n' (Blatt '!'))
```




## Terminologie

- **Wurzel**-knoten `'T'`
- **Blätter** `'a'`,`'z'`,`'!'`
- **linker Teilbaum** von **Knoten** `'n'` ist das Blatt `'z'`

# Rekursive Datentypen

## Beispiel: Binärbäume

```
data Baum = Blatt Char | Knoten Baum Char Baum

myBaum :: Baum
myBaum = Knoten (Blatt 'a') 'T'
          (Knoten (Blatt 'z') 'n' (Blatt '!'))

dfCollect :: Baum -> String
dfCollect (Blatt c) = [c]
dfCollect (Knoten links c rechts)
          = c : dfCollect links ++ dfCollect rechts
```

# Rekursive Datentypen

## Beispiel: Binärbäume

```
data Baum = Blatt Char | Knoten Baum Char Baum

myBaum :: Baum
myBaum = Knoten (Blatt 'a') 'T'
          (Knoten (Blatt 'z') 'n' (Blatt '!'))

dfCollect :: Baum -> String
dfCollect (Blatt c) = [c]
dfCollect (Knoten links c rechts)
          = c : dfCollect links ++ dfCollect rechts


> dfCollect myBaum
"Tanz!"
```

# Wechelseitig Rekursiv

Beispiel:

```
data Datei = Datei String | Verzeichnis Dir
data Dir   = Local String [Datei] | Remote String

data :: Dir
data = Dir "root"
       [Datei "info.txt"
       ,Verzeichnis (Remote "my.url/work")
       ,Verzeichnis (Local "tmp" [])
       ,Datei "help.txt"
       ]
```

- Reihenfolge der Definition ist egal
- Ganz normale Verwendung

Mann könnte in diesem Beispiel auch nur einen Datentyp mit 3 Alternativen definieren, doch dann könnte man kein Funktionen schreiben, welche nur gezielt Verzeichnisse bearbeiten.

# Typparameter

Datentypen können **Typvariablen** als Parameter verwenden:

## Beispiel: Listen

```
data List a = Leer | Element a (List a)

iList :: List Int
iList = Element 1 (Element 2 (Leer))

iSum :: List Int -> Int
iSum Leer = 0
iSum (Element h t) = h + iSum t

type IntList = List Int -- Typspezialisierung
```

- Typvariablen werden immer klein geschrieben
- `List` bezeichnen wir als **Typkonstruktor**. Nur durch Anwendung auf einen Typ wird ein Typ daraus: `List Int`.

# Typparameter

Datentypen können **Typvariablen** als Parameter verwenden

## Beispiel: Listen

```
data List a = Leer | Element a (List a)

iSum :: List Int -> Int

myLength :: (List a) -> Int
myLength Leer = 0
myLength (Element _ t) = 1 + myLength t

> myLength (Element 'a' (Element 'b' Leer))
2
```

Funktion `myLength` kann mit Listen umgehen, die einen beliebigen Typ in sich tragen; im Gegensatz zu `iSum`.
Solche Funktionen nennt man auch **polymorph**.

# Ausblick: Polymorphe Funktionen

Beispiele polymorpher Funktionen aus der Standardbibliothek:

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

fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd (_,y) = y

replicate :: Int -> a -> [a]

drop :: Int -> [a]-> [a]

zip :: [a] -> [b] -> [(a,b)]
```

# Maybe

`Maybe` ist ein wichtiger polymorpher Datentyp der Standardbibliothek

```
data Maybe a = Nothing | Just a
```

Damit können wir auf eine sichere Weise ausdrücken, dass eine Berechnungen fehlschlagen kann.

Beispiel

```
data Früchte = Apfel    {preis::Int, anzahl::Int}
             | Birne    {preis::Int, anzahl::Int}
             | Banane {preis::Int, anzahl::Int,
                       krümmung::Double}

getKrümmung :: Früchte -> Maybe Double
getKrümmung Banane {krümmung=k} = Just k
getKrümmung _                   = Nothing
```

# Maybe

## Beispiele

```
data Maybe a = Nothing | Just a

isJust           :: Maybe a -> Bool
isJust Nothing = False
isJust _       = True

fromMaybe :: a -> Maybe a -> a
fromMaybe def Nothing  = def
fromMaybe _   (Just a) = a

catMaybes :: [Maybe a] -> [a]
catMaybes ls = [x | Just x <- ls]
```

# Either

Polymorphe Datentypen können auch mehrere Typparameter haben. `Either` ist ein wichtiges Beispiel:

```
data Either a b  =  Left a | Right b
```

Zum Beispiel kann `Either a String` anstatt `Maybe a` für die Rückgabe von Fehlermeldungen verwendet werden, ohne die Berechnung abzubrechen.

`Either` ist Typ für disjunkte Vereinigungen:

$$A_1 \dot\cup A_2 = \{(i, a) \mid a \in A_i\}$$

Beispiel:

$$\{\heartsuit, \diamondsuit\} \dot\cup \{\heartsuit, \clubsuit, \spadesuit\} = \{(1, \heartsuit), (1, \diamondsuit), (2, \heartsuit), (2, \clubsuit), (2, , \spadesuit)\}$$

Für endliche Mengen gilt: $|A_1 \dot\cup A_2| = |A_1| + |A_2|$

Man spricht auch von "Summentypen" bei Alternativen

# Either

## Beispiele

```
data Either a b  =  Left a | Right b

isRight :: Either a b -> Bool
isRight (Left  _) = False
isRight (Right _) = True

lefts   :: [Either a b] -> [a]
lefts x = [a | Left a <- x]

partitionEithers :: [Either a b] -> ([a],[b])
partitionEithers [] = ([],[])
partitionEithers (h : t) =
  let (ls,rs) = partitionEithers t
  in case h of  Left l  -> (l:ls,   rs)
                Right r -> (  ls, r:rs)
```

# Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort data
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen
  - frischer Konstruktor – muss mit Großbuchstaben beginnen
  - beliebige Zahl bekannte Typen oder Typparameter
- optionale deriving-Klausel mit Liste von Typklassen

# Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

# Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

# Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

# Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen — lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

# Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

# Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen — lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

# Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
  deriving (class_1, ..., class_l)
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen — lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

# Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
  deriving (class_1, ..., class_l)
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

# Zusammenfassung: Datentypen

- Ein Typ (oder **Datentyp**) ist eine Menge von Werten
- Unter einer **Datenstruktur** versteht man einen Datentyp plus alle darauf verfügbaren Operationen
- Moderne Programmiersprachen ermöglichen, dass der Benutzer neue Typen definieren kann
- Datentypen können andere Typen als Parameter haben, **Konstruktoren** können als Funktionen betrachtetet werden
- **Records** erlauben Benennung dieser Typparameter
- Datentypen können (wechselseitig) rekursiv definiert werden
- Typdeklarationen:

| | |
|---|---|
| DATA | Deklaration wirklich neuer Typen |
| TYPE | Typabkürzungen für Lesbarkeit |
| NEWTYPE | Zur Unterscheidung verschiedener Nutzungen eines bereits existierenden Typs |