

PROGRAMMIERUNG UND MODELLIERUNG MIT HASKELL

TEIL 4: BENUTZERDEFINIERTER DATENTYPEN

Steffen Jost

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

25. April 2018



TEIL 4: BENUTZERDEFINIERTER DATENTYPEN

1 WIEDERHOLUNG

2 DATENTYPDEKLARATIONEN

- Konstruktoren
- Alternativen
- Record-Syntax
- Rekursive Datentypen
- Polymorphe Datentypen

3 WICHTIGE BEISPIELE

- Maybe
- Either

4 ZUSAMMENFASSUNG

- Typdeklaration
- Datentypen
- Pattern Matching
- Wächter



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

Beispiel:

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

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.



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

PROBLEM: Keine Typsicherheit! Don't do this!

Typabkürzungen 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 Typen, eignen sich aber nicht so gut für die Daten Modellierung.



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 Typen, eignen sich aber nicht so gut für die Daten Modellierung



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 Typen, eignen sich aber nicht so gut für die Daten Modellierung



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 Spiel?
`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 Typen, eignen sich aber nicht so gut für die Daten Modellierung.



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 Spiel?
`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 Typen, eignen sich aber nicht so gut für die Daten Modellierung!



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 Spiel?
`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 Typen, eignen sich aber nicht so gut für die Daten Modellierung!



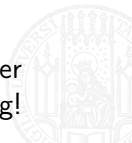
Was bedeutet der Wert (17.3, 17.3, 80.0)?

- 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 Spiel?
`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 Typen, eignen sich aber nicht so gut für die Daten Modellierung!



BENUTZERDEFINIERTER DATENTYPEN

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

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

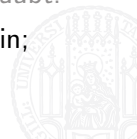
Datentypdeklaration beginnen mit Schlüsselwort `data`, dann ein frischer Name für den neuen Typ, beginnend mit *Großbuchstabe*.

Dann folgt der **Konstruktor**, mit einer Auflistung der Typen seiner Argumente. Konstruktoren kann man als Funktionen betrachten, die Werte des neuen Typs konstruieren:

```
> :type MeinKonstruktor
```

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

- Konstruktoren beginnen immer mit einem Großbuchstaben.
Infix-Konstruktoren beginnend mit `:` sind auch erlaubt.
- Bezeichner von Konstruktoren und Typen dürfen gleich sein; Konstruktoren sind Ausdrücke; Ausdrücke und Typen leben in verschiedenen Welten.



BENUTZERDEFINIERTER DATENTYPEN – BEISPIEL

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

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

KONSTRUKTOREN ALS FUNKTIONEN

```
> let ding1 = MeinKonstruktor 42
```

```
> :type ding1
```

```
ding1 :: Bool -> MeinTyp
```

```
> let ding2 = MeinKonstruktor 69 False
```

```
ding2 :: MeinTyp
```

```
> let ding3 = ding1 True
```

```
ding2 :: MeinTyp
```



KONSTRUKTOREN IN MUSTERVERGLEICHEN

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

Haskell kann mit solchen benutzerdefinierten Datentypen zunächst wenig anfangen; nicht einmal Bildschirmausgabe ist möglich.

Mit Pattern-Matching können die Argumente eines Konstruktors wieder ausgepackt werden, zum Beispiel um eine Funktion zur Umwandlung in einen String zu schreiben:

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

MERKE:

- Konstruktoren mit Argumenten zum Matching immer in runde Klammern fassen. Da `(:)` ein Konstruktor ist, schreiben wir Listen-Patterns auch `(kopf:rumpf)` und *nicht* `[k:r]`
- Reihenfolge der Argumente beachten!



KONSTRUKTOREN IN MUSTERVERGLEICHEN

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

Tipp: GHC kann solche Funktionen automatisch erzeugen. Das `deriving (Show)` hinter der Deklaration erzeugt z.B. Funktion `show :: MeinTyp -> String` zur Bildschirmausgabe. (**Typklassen** `Show`, `Eq`, ... behandeln wir im nächsten Kapitel.)

Umwandlung in einen String zu schreiben:

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

MERKE:

- Konstruktoren mit Argumenten zum Matching immer in runde Klammern fassen. Da `(:)` ein Konstruktor ist, schreiben wir Listen-Patterns auch `(kopf:rumpf)` und *nicht* `[k:r]`
- Reihenfolge der Argumente beachten!



KONSTRUKTOREN IN MUSTERVERGLEICHEN

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

Tipp: GHC kann solche Funktionen automatisch erzeugen. Das `deriving (Show)` hinter der Deklaration erzeugt z.B. Funktion `show :: MeinTyp -> String` zur Bildschirmausgabe. (**Typklassen** `Show`, `Eq`, ... behandeln wir im nächsten Kapitel.)

Umwandlung in einen String zu schreiben:

```
myShow :: MeinTyp -> String
```

```
myShow (MeinKonstruktor i True) = "T(" ++ (show i) ++ ")"
```

Bei Fehlermeldungen wie:

```
No instance for (Show MeinTyp) arising from ...
```

einfach Typdeklaration um `deriving (Show)` erweitern.

- Konstruktoren mit Argumenten zum Matching immer in runde Klammern fassen. Da `(:)` ein Konstruktor ist, schreiben wir Listen-Patterns auch `(kopf:rumpf)` und *nicht* `[k:r]`
- Reihenfolge der Argumente beachten!

BEISPIEL: VERWECHSLUNG VERHINDERN

Drei Datentypen, welche alle lediglich ein Tripel von `Double` sind, jedoch unterschiedlich zu nutzen sind:

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

```
myCircle :: Circle
myCircle = Circle 17.3 17.3 80.0
```

```
hydralisk :: Monster
hydralisk = Monster 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
```



AUFZÄHLUNGEN

Ein Datentyp darf auch mehrere Konstruktoren besitzen:

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

Das `|` liest man als “oder”: Ein Wert des Typs `Bool` wurde entweder mit dem Konstruktor `True` oder dem Konstruktor `False` konstruiert. Die Reihenfolge der Konstruktoren innerhalb der Deklaration ist oft egal. Ausnahme: `deriving (Ord,Enum)`

SPEZIALFALL: Konstruktoren ohne Argumente

Falls alle Konstruktoren eines Datentyps keine Argumente haben, wie hier bei `Bool`, dann bezeichnet man diesen Datentyp auch als **Aufzählung** (engl. **Enumeration**).

Ein Wert eines Aufzählungs-Typen ist immer eine von endlich vielen möglichen Konstanten.



BEISPIEL: AUFZÄHLUNG

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 Konstruktoren
- Funktion `next` zählt einfach einen Wochentag weiter



BEISPIEL: AUFZÄHLUNG

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 Anderen durch 'deriving' generiert:
succ, pred :: Enum a => a -> a
toEnum      :: Enum a => Int -> a
```

- Datentyp `Day` hat 7 Konstruktoren
- Funktion `next` zählt einfach einen Wochentag weiter



KOMBINATION: ALTERNATIVEN & ARGUMENTE

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

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

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

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



KOMBINATION: ALTERNATIVEN & ARGUMENTE

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

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

```
> gesamtPreis meineFruechte
1700
```



KOMBINATION: ALTERNATIVEN & ARGUMENTE

```
data Frucht = 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.

Schlechte Modellierung!

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

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

```
> gesamtPreis meineFrüchte
1700
```



DUPLIKATION VERMEIDEN

SCHLECHT:

```
data Frucht = Apfel (Int,Int)
            | Birne  Int Int
            | Banane Int Int Double
```

Wenn ein Argument mit der gleichen Bedeutung in allen Konstruktoren vorkommt, sollte man den Datentyp meist unterteilen, um Redundanzen im Code zu vermeiden.

BESSER:

```
data Frucht = Frucht Sorte Int Int
data Sorte  = Apfel | Birne | Banane Double
```

```
preis  :: Frucht -> Int
preis (Frucht _ p _) = p
anzahl :: Frucht -> Int
anzahl (Frucht _ _ z) = z
-- gesamtPreis wie zuvor
```



DUPLIKATION VERMEIDEN

SCHLECHT:

```
data Frucht = Apfel (Int,Int)
             | Birne  Int Int
             | Banane Int Int Double
```

Wenn ein Argument mit der gleichen Bedeutung in allen Konstruktoren vorkommt, sollte man den Datentyp meist unterteilen, um Redundanzen im Code zu vermeiden.

BESSER:

```
data Frucht = Frucht Sorte Int Int
data Sorte  = Apfel | Birne | Banane Double
```

```
hatApfel ((Frucht Apfel _ _):_) = True
hatApfel (_:t)                   = hatApfel t
hatApfel []                      = False
```

Dank Verschachtelter Patterns entsteht kein Nachteil.



RECORDS

Eine weitere Alternative bietet die **Record**-Syntax:

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

Jedes Argument eines Konstruktors bekommt einen Namen, diese werden dann auch als **Felder** bezeichnet.

- Namen sind sinnvoll, wenn man viele Argumente hat
- Reihenfolge der Felder in geschweiften Klammern ist egal
- **Projektion** für jedes Feld werden automatisch definiert; hier:

```
preis      :: Frucht -> Int  
anzahl     :: Frucht -> Int  
krümmung   :: Frucht -> Double
```

Warnung: partiell

- Auch Pattern-Matching erlaubt dann Record-Syntax; Patterns mit { } müssen nicht alle Felder matchen

RECORDS

Eine weitere Alternative bietet die **Record**-Syntax:

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

BEISPIEL FÜR PATTERN-MATCHING MIT RECORDS

```
krümmung' :: Frucht -> Double  
krümmung' Banane { krümmung=x }          = x  
krümmung' Birne { anzahl=1, preis=p } = pi^p  
krümmung' _ = 0.0
```

Bemerkung Die automatisch generierte Funktion `krümmung` ist eine partielle Funktion und liefert für `Apfel` und `Birne` einen Fehler.

Abhilfe mit eigener Definition wie hier z.B. `krümmung'`

BESSERE MODELLIERUNG WÄRE AUCH HIER WIEDER:

```
data Frucht = Frucht {sorte::Sorte, preis,anzahl::Int}  
data Sorte  = Apfel | Birne | Banane Double
```



RECORDS

Eine weitere Alternative bietet die **Record**-Syntax:

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

BEISPIEL FÜR PATTERN-MATCHING MIT RECORDS

```
krümmung' :: Frucht -> Double  
krümmung' Banane { krümmung=x }          = x  
krümmung' Birne { anzahl=1, preis=p } = pi^p  
krümmung' _ = 0.0
```

Bemerkung Die automatisch generierte Funktion `krümmung` ist eine partielle Funktion und liefert für `Apfel` und `Birne` einen Fehler.

Abhilfe mit eigener Definition wie hier z.B. `krümmung'`

BESSERE MODELLIERUNG WÄRE AUCH HIER WIEDER:

```
data Frucht = Frucht {sorte::Sorte, preis,anzahl::Int}  
data Sorte  = Apfel | Birne | Banane Double
```



RECORDS

Eine weitere Alternative bietet die **Record**-Syntax:

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

Datentypen können *leicht nachträglich* zu Records gemacht werden, da die Record-Syntax überall optional ist.

Werden mal keine geschweiften Klammern und Feldnamen verwendet, gilt die Reihenfolge in der Definition wie üblich.

Obige Datentypdeklaration in Record-Syntax erlaubt also auch alles, welches folgende gewöhnliche Deklaration ermöglichen würde:

```
data Frucht = Apfel  Int Int  
              | Birne  Int Int  
              | Banane Int Int Double
```



RECORDS

Eine weitere Alternative bietet die **Record**-Syntax:

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

Datentypen können *leicht nachträglich* zu Records gemacht werden, da die Record-Syntax überall optional ist.

Werden mal keine geschweiften Klammern und Feldnamen

Wenn man nachträglich noch weitere Felder hinzufügt, kompiliert der Code mit einigen Warnungen weiterhin.

Unspezifizierte Felder werden automatisch mit **undefined** initialisiert:

Obige Datentypdeklaration in Record-Syntax erlaubt also auch alles, welches folgende gewöhnliche Deklaration ermöglichen würde:
`Apfel {preis=1}` entspricht `Apfel {preis=1, anzahl=undefined}`

`data Frucht = Apfel Int Int`

`| Birne Int Int`

`| Banane Int Int Double`

Da die Auswertung von **undefined** die Ausführung sofort abbricht, sollte man diese Warnungen bearbeiten!

RECORDS

Eine weitere Alternative bietet die **Record**-Syntax:

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

Datentypen können in Haskell mit der Record-Syntax definiert werden, da die Record-Syntax über optionalen Zuckersyntax verfügt. Werden in einem Programm keine geschweiften Klammern und Feldnamen verwendet, gilt die Reihenfolge in der Definition wie oben.

HINWEIS: Record-Syntax ist eine rein syntaktische Erweiterung für uns Menschen. Records können nicht mehr oder weniger wie gewöhnliche Datentypdeklarationen.

Obige Datentypdeklaration in Record-Syntax erlaubt also auch alles, welches folgende gewöhnliche Deklaration ermöglichen würde:

```
data Frucht = Apfel  Int Int  
              | Birne  Int Int  
              | Banane Int Int Double
```



RECORD PSEUDOUUPDATE

```
data Frucht = Frucht {sorte::Sorte, preis,anzahl::Int}  
    deriving Show  
data Sorte  = Apfel | Birne | Banane Double  
    deriving Show
```

Die Record-Syntax erlaubt auch **Pseudoupdates**.

Es werden natürlich Kopien erstellt, denn bestehende Werte werden in einer funktionalen Welt nie verändert!

```
> let f1 = Frucht {sorte=Apfel, anzahl=3, preis=2}  
> let f2 = f1 { sorte  = Birne }  
> let f3 = f1 { anzahl = anzahl f1 - 1 }  
> f2  
Frucht {sorte = Birne, preis = 2, anzahl = 3}  
> f3  
Frucht {sorte = Apfel, preis = 2, anzahl = 2}  
> f1  
Frucht {sorte = Apfel, preis = 2, anzahl = 3}
```



RECORD DEFAULTS

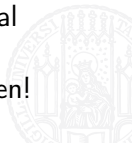
```
data Frucht = Frucht {sorte::Sorte, preis,anzahl::Int}  
    deriving Show  
data Sorte  = Apfel | Birne | Banane Double  
    deriving Show
```

Die Record-Pseudoupdate-Notation erlaubt auch leicht, sinnvolle Standardwerte festzulegen:

```
standardApfel = Frucht {sorte=Apfel, preis=2, Anzahl=1}  
deluxeApfel   = standardApfel { preis = 7 }
```

Wenn man konsequent alle Werte des Typs `Frucht` mit Hilfe von `standardApfel` generiert, dann hat man es eventuell leichter, wenn man später weitere Felder hinzufügt — denn man muss erst Mal nur die Definition von `standardApfel` bearbeiten.

Nachteil: Keine Warnungen, welche alle relevanten Stellen zeigen!



REKURSIVE DATENTYPEN

Typdeklarationen dürfen auch (wechselseitig) rekursiv sein.

BEISPIEL: **Listen** als Beispiel selbst definiert

Konstruktor **ListKnoten** hat ein rekursives Argument **IntList**

```
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

Typdeklarationen dürfen auch (wechselseitig) rekursiv sein.

BEISPIEL: **Listen** als Beispiel selbst definiert

Konstruktor **ListKnoten** hat ein rekursives Argument **IntList**

```
data IntList = LeereListe | ListKnoten Int IntList
```

```
myList : > :type LeereListe
myList = IntList LeereListe
> :type ListKnoten
mySum :: Int -> IntList -> IntList
mySum Le mit
mySum (L > :type []
[a]
> mySum : > :type (:)
3 a -> [a] -> [a]
```



REKURSIVE DATENTYPEN

Typdeklarationen dürfen auch (wechselseitig) rekursiv sein.

BEISPIEL: **Listen** als Beispiel selbst definiert

Konstruktor **ListKnoten** hat ein rekursives Argument **IntList**

```
data IntList = LeereListe | ListKnoten Int IntList
```

```
myList : > :type LeereListe
myList = IntList LeereListe
        > :type ListKnoten
mySum :: Int -> IntList -> IntList
mySum Le mit
mySum (L > :type []
        [Int]
> mySum : > :type (:)
3        Int -> [Int] -> [Int]
```



REKURSIVE DATENTYPEN

BEISPIEL: Binärbäume

```
data Baum = Blatt | Ast Baum Char Baum
```

```
> :type Ast
```

```
Ast :: Baum -> Char -> Baum -> Baum
```

```
myBaum :: Baum
```

```
myBaum = Ast (Ast Blatt 'a' Blatt)
```

```
        'T' (Ast (Ast Blatt 'z' Blatt)
```

```
              'n' (Ast Blatt '!' Blatt))
```



TERMINOLOGIE

- **Wurzel**-knoten 'T'
- **linker Teilbaum** von **Knoten** 'n' enthält nur Knoten 'z'
- Knoten 'a', 'z', '!' werden oft auch schon als **Blätter** bezeichnet



REKURSIVE DATENTYPEN

BEISPIEL: Binärbäume

```
data Baum = Blatt | Ast Baum Char Baum
```

```
> :type Ast
```

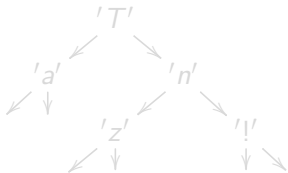
```
Ast :: Baum -> Char -> Baum -> Baum
```

```
myBaum :: Baum
```

```
myBaum = Ast (Ast Blatt 'a' Blatt)
```

```
          'T' (Ast (Ast Blatt 'z' Blatt)
```

```
                'n' (Ast Blatt '!' Blatt))
```



TERMINOLOGIE

- **Wurzel**-knoten 'T'
- **linker Teilbaum** von **Knoten** 'n'
enthält nur Knoten 'z'
- Knoten 'a', 'z', '!' werden oft
auch schon als **Blätter** bezeichnet

REKURSIVE DATENTYPEN

BEISPIEL: Binärbäume

```
data Baum = Blatt | Ast Baum Char Baum
```

```
> :type Ast
```

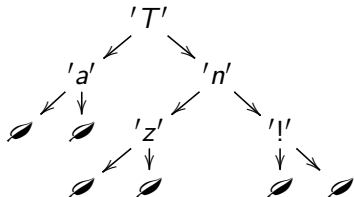
```
Ast :: Baum -> Char -> Baum -> Baum
```

```
myBaum :: Baum
```

```
myBaum = Ast (Ast Blatt 'a' Blatt)
```

```
          'T' (Ast (Ast Blatt 'z' Blatt)
```

```
                'n' (Ast Blatt '!' Blatt))
```



TERMINOLOGIE

- **Wurzel**-knoten 'T'
- **linker Teilbaum** von **Knoten** 'n' enthält nur Knoten 'z'
- Knoten 'a', 'z', '!' werden oft auch schon als **Blätter** bezeichnet

REKURSIVE DATENTYPEN

BEISPIEL: Binärbäume

```
data Baum = Blatt | Ast Baum Char Baum
```

```
> :type Ast
```

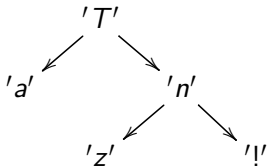
```
Ast :: Baum -> Char -> Baum -> Baum
```

```
myBaum :: Baum
```

```
myBaum = Ast (Ast Blatt 'a' Blatt)
```

```
          'T' (Ast (Ast Blatt 'z' Blatt)
```

```
                'n' (Ast Blatt '!' Blatt))
```



TERMINOLOGIE

- **Wurzel**-knoten 'T'
- **linker Teilbaum** von **Knoten** 'n' enthält nur Knoten 'z'
- Knoten 'a', 'z', '!' werden oft auch schon als **Blätter** bezeichnet

REKURSIVE DATENTYPEN

BEISPIEL: Binärbäume

```
data Baum = Blatt | Ast Baum Char Baum
```

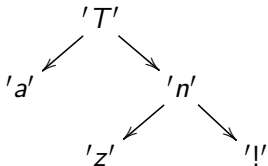
> Datentyp `Baum` enthält nur eine **Beschriftung** mit Typ `Char`
As innerhalb der Ast-Knoten.

Ein gebräuchliche Alternative ist es, die Beschriftung nur an den
my Blättern zu haben, aber nicht an den Knoten, also etwa:

my

```
data Baum' = Blatt' Char | Ast' Baum' Baum'
```

Dies hängt vom Anwendungsfall ab.



TERMINOLOGIE

- **Wurzel**-knoten `'T'`
- **linker Teilbaum** von **Knoten** `'n'` enthält nur Knoten `'z'`
- Knoten `'a'`, `'z'`, `'!'` werden oft auch schon als **Blätter** bezeichnet

REKURSIVE DATENTYPEN

BEISPIEL: Binärbäume

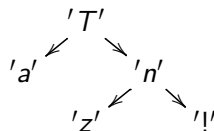
```
data Baum = Blatt | Ast Baum Char Baum
```

```
myBaum :: Baum
```

```
myBaum = Ast (Ast Blatt 'a' Blatt)
```

```
      'T' (Ast (Ast Blatt 'z' Blatt)
```

```
            'n' (Ast Blatt '!' Blatt))
```



```
dfCollect :: Baum -> String
```

```
dfCollect Blatt = ""
```

```
dfCollect (Ast 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

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

- Reihenfolge der Definition ist egal
- Ganz normale Verwendung

Bemerkung Mann könnte hier auch nur einen Datentyp mit 3 Alternativen definieren, doch dann könnte man keine Funktionen schreiben, welche nur gezielt Verzeichnisse bearbeiten.



TYPPARAMETER – TYPDEKLARATIONEN MIT “LOCH”

Datentypen können **Typvariablen** als Parameter verwenden:

BEISPIEL: Listen

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

```
iList :: List Int
```

```
iList = Cons 1 (Cons 2 (Leer))
```

```
iSum :: List Int -> Int
```

```
iSum Leer = 0
```

```
iSum (Cons h t) = h + iSum t
```

```
type IList = 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 – TYPDEKLARATIONEN MIT “LOCH”

Datentypen können **Typvariablen** als Parameter verwenden

BEISPIEL: Listen

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

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

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

Im Gegensatz zu `iSum :: List Int -> Int` kann die Funktion `myLength :: List a -> Int` mit Listen umgehen, die einen beliebigen Typ in sich tragen.

Solche Funktionen, bei denen Typvariablen in der Signatur auftauchen, 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 kein Ergebnis liefern kann (bzw. fehlschlagen kann).

BEISPIEL

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

```
getKrümmung :: Frucht -> 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 standardWert Nothing = standardWert
```

```
fromMaybe _ (Just x) = x
```

```
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 komplett abubrechen.

`Either` ist der Typ für disjunkte Vereinigungen:

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

BEISPIEL:

$$\{\diamond, \heartsuit\} \uplus \{\clubsuit, \spadesuit, \heartsuit\} = \{(1, \diamond), (1, \heartsuit), (2, \clubsuit), (2, \spadesuit), (2, \heartsuit)\}$$

Für endliche Mengen gilt:

$$|A_1 \uplus A_2| = |A_1| + |A_2|$$

Man spricht daher auch von “Summentypen”

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 immer mit Großbuchstabe
- optionale Typparameter
- optionale Alternativen
- Typnamen müssen mit einem Großbuchstaben beginnen
- Typparameter müssen mit einem kleinen Buchstaben beginnen
- Typparameter werden in Klammern angegeben



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 immer mit Großbuchstabe
- optionale Typparameter
- optionale Alternativen lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Typen oder Typparameter als Argumente
- 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 immer mit Großbuchstabe
- optionale Typparameter
- optionale Alternativen lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Typen oder Typparameter als Argumente
- 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 immer mit Großbuchstabe
- **optionale Typparameter**
- optionale Alternativen lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Typen oder Typparameter als Argumente
- 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 immer mit Großbuchstabe
- optionale Typparameter
- **optionale Alternativen** lies | als “oder”
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Typen oder Typparameter als Argumente
- 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 immer mit Großbuchstabe
- optionale Typparameter
- optionale Alternativen lies | als “oder”
- **frischer Konstruktor – muss mit Großbuchstaben beginnen**
- beliebige Typen oder Typparameter als Argumente
- 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 immer mit Großbuchstabe
- optionale Typparameter
- optionale Alternativen lies | als “oder”
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Typen oder Typparameter als Argumente
- 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 immer mit Großbuchstabe
- optionale Typparameter
- optionale Alternativen lies | als “oder”
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Typen oder Typparameter als Argumente
- **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 immer mit Großbuchstabe
- optionale Typparameter
- optionale Alternativen lies | als “oder”
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Typen oder Typparameter als Argumente
- 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
- **Polymorphe Datentypen** verallgemeinern mit Typparametern
- Datentypen können (wechselseitig) rekursiv definiert werden
- **Konstruktoren** kann man als Funktionen betrachten, deren Funktionsanwendung mit Pattern-Matching quasi wieder rückgängig gemacht werden kann! (konstruierter Wert bleibt)
- **Records** erlauben Benennung der Konstruktorargumente
- Typdeklarationen:
 - `data` Deklaration wirklich neuer Typen
 - `type` Typabkürzungen nur für Lesbarkeit für Menschen



INFORMELLE BESCHREIBUNG

Informell könnte man sagen: Konstruktoren packen Päckchen mit ihren Argumenten und kleben ein Etikett mit Ihren Namen drauf. Pattern Matching packt Päckchen aus, wenn das Etikett stimmt.

Beispiel:

```
data S    = SL1 Double | SL2 Int Int | SL3      deriving Show
data T a = TL1 a | TL2 a (T a)      deriving Show
```

```
unpacker :: S -> T Double
unpacker (SL1 d  ) = TL1 d
unpacker (SL3     ) = TL1 0.0
unpacker (SL2 x y) = TL2 xd (TL1 yd)
  where xd = fromIntegral x
        yd = fromIntegral y
```

Funktion `unpacker` erwartet Päckchen „nach Art“ des Typs `S`. Es wird das Etikett inspiziert. Wenn das Etikett `SL1` ist, wird der Inhalt ausgepackt und in ein Päckchen mit Etikett `TL1` umverpackt. Dieses Päckchen ist dann „nach Art“ des Typs `T Double`.

VERGLEICH DER NOTATION

Vergleich der Notation zwischen dem eingebauten Listentyp `[a]` und einem äquivalenten, selbst-definierten Typ `List a`, den wir verwenden könnten, wenn es keine eingebauten Listen gäbe:

```
data [a]    = []    | (:)  a      [a]
data List a = Leer  | Cons a (List a)
```

```
f1 :: [a] -> ...           f2 :: List a -> ...
f1 []      = ...           f2 Leer      = ...
f1 (h:t)   = ...           f3 (Cons h t) = ...
```

- `List a` ist ein rekursiver Datentyp, da es einen Konstruktor gibt, der ein Argument dieses Typs fordert.
- `List a` ist ein polymorpher Datentyp, da es einen Typparameter gibt. Dieser erlaubt es uns, Listen über beliebige andere Typen als Inhalt zu bilden.



KLAMMERUNG IM PATTERN MATCHING

Im Pattern-Matching Konstruktoren mit Argumenten immer klammern;
Reihenfolge der Argumente ist zu beachten! Ausnahme: Records

```
data Maybe a    = Nothing | Just a
data Either a b = Left a   | Right b
```

```
foo :: Maybe Int -> Either Bool String -> String
foo Just x Left y  = "NOT OK"  -- Syntaxfehler
foo (Just x) (Left y) = "OK"
foo Nothing (Right _) = "OK"
```

Konstruktor ohne Argumente müssen im Pattern-Matching nicht geklammert werden.



KLAMMERUNG BEI MUSTERVERGLEICH MIT LISTEN

Schreibweise `[1,2,3]` ist erlaubte Kurzform für `1:2:3:[]`

`(:)` ist ein Infix-Konstruktor;

Konstruktoren mit Argumenten sind im Pattern zu klammern!

```
bar :: [a] -> String
bar []           = "Leere Liste"
bar [x]          = "Genau 1 Element"
bar [x,y]        = "Genau 2 Elemente"
bar [x,y,z]      = "Genau 3 Elemente"
bar (x:xs)       = "Mindestens 1 Element"
bar (x:(y:zs))   = "Mindestens 2 Elemente"
bar (x:y:zs)     = "Mindestens 2 Elemente" --eine () reicht
bar (x:y:z:ls)   = "Mindestens 3 Elemente"
```

Typen der Variablen im Beispiel:

```
x,y,z      :: a      -- ein Element
xs,zs,ls   :: [a]    -- eine Liste
```

Listenbezeichner oft mit englischem Plural-s: ein `x`, mehrere `xs`



PATTERNS VOR WÄCHTER

TIPP: Pattern Matching bevorzugen vor Wächtern

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

```
fooOK :: Maybe a -> Int
```

```
fooOK Nothing = 0
```

```
fooOK (Just x) = 1
```

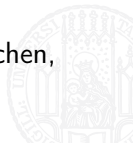
```
fooBAD :: Eq a => Maybe a -> Int
```

```
fooBAD x | x == Nothing = 0
```

```
          | otherwise    = 1
```

Anwendung von `(==)` erzwingt Einschränkung auf Typklasse `Eq`:
für Gleichheit muss der komplette Wert betrachtet werden!

Bei Pattern-Matching wird dagegen nur der Konstruktor verglichen,
aber nicht notwendigerweise der gesamte Wert!



NEWTYPE

Für Datentypen mit *genau einen* Konstruktor der *genau ein* Argument hat, bietet sich **newtype** an:

```
newtype Circle = Circle (Double, Double, Double)
newtype Point3D = Point3D (Double, Double, Double)
newtype 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 Typ und Newtyp.
- Nicht so strikt wie äquivalente Datentypdeklaration Kapitel 11
- Eigene Klassen Instanzen möglich; Kapitel 5
oder Übernahme von Instanzen des ursprünglichen Typs
mit Erweiterung `GeneralizedNewtypeDeriving`

