

Funktionale Programmierung - fortgeschrittene Konzepte und Anwendungen

Vorlesung 8 vom 26.11.2019: Abstrakte Datentypen

Till Mossakowski

Otto-von-Guericke Universität Magdeburg

Wintersemester 2019/20

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ **Teil II: Funktionale Programmierung im Großen**
 - ▶ Abstrakte Datentypen
 - ▶ Signaturen und Eigenschaften
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

- ▶ **Abstrakte Datentypen**

- ▶ Allgemeine Einführung

- ▶ Realisierung in Haskell

- ▶ Beispiele

Warum Modularisierung?

▶ Übersichtlichkeit der Module

Lesbarkeit

▶ Getrennte Übersetzung

technische Handhabbarkeit

▶ Verkapselung

konzeptionelle Handhabbarkeit

Abstrakte Datentypen

Definition (Abstrakter Datentyp)

Ein **abstrakter Datentyp** (ADT) besteht aus einem (oder mehreren) **Typen** und **Operationen** darauf, mit folgenden Eigenschaften:

- ▶ Werte des Typen können nur über die bereitgestellten Operationen erzeugt werden
- ▶ Eigenschaften von Werten des Typen werden nur über die bereitgestellten Operationen beobachtet
- ▶ Einhaltung von **Invarianten** über dem Typ kann garantiert werden

Implementation von ADTs in einer Programmiersprache:

- ▶ benötigt Möglichkeit der **Kapselung** (Einschränkung der Sichtbarkeit)
- ▶ bspw. durch **Module** oder **Objekte**

ADTs vs. algebraische Datentypen

- ▶ Algebraische Datentypen
 - ▶ **Frei erzeugt**
 - ▶ Keine Einschränkungen
 - ▶ Insbesondere keine Gleichheiten ($[] \neq x:xs$, $x \neq y \rightarrow x:ls \neq y:ls$ etc.)
- ▶ ADTs:
 - ▶ Einschränkungen und Invarianten möglich
 - ▶ Gleichheiten möglich

ADTs in Haskell: Module

- ▶ Einschränkung der Sichtbarkeit durch **Verkapselung**
- ▶ **Modul**: Kleinste verkapselbare **Einheit**
- ▶ Ein **Modul** umfaßt:
 - ▶ **Definitionen** von Typen, Funktionen, Klassen
 - ▶ **Deklaration** der nach außen **sichtbaren** Definitionen
- ▶ **Gleichzeitig**: Modul $\hat{=}$ Übersetzungseinheit (getrennte Übersetzung)

Module: Syntax

- ▶ Syntax:

```
module Name(Bezeichner) where Rumpf
```

- ▶ Bezeichner können leer sein (dann wird alles exportiert)

- ▶ Bezeichner sind:

- ▶ **Typen**: $T, T(c_1, \dots, c_n), T(\dots)$

- ▶ **Klassen**: $C, C(f_1, \dots, f_n), C(\dots)$

- ▶ Andere Bezeichner: **Werte, Felder, Klassenmethoden**

- ▶ Importierte **Module**: **module** M

- ▶ Typsynonyme und Klasseninstanzen bleiben sichtbar

- ▶ Module können sich **rekursiv** importieren (*don't try at home*)

Refakturierung im Einkaufsparadies

```
module Shop4 where
import Data.Maybe

-- Modellierung der Artikel.
data Apfelsorte = Boskoop | CoaOrange | GrannySmith
  deriving (Eq, Show)

apreis :: Apfelsorte -> Int
apreis Boskoop = 55
apreis CoaOrange = 60
apreis GrannySmith = 50

data Kaesorte = Gouda | Appenzeller
  deriving (Eq, Show)

kpreis :: Kaesorte -> Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270

data Bio = Bio | Konv
  deriving (Eq, Show)

data Artikel =
  Apfel Apfelsorte | Eier
  | Kase Kaesorte | Schinken
  | Salami | Milch Bio
  deriving (Eq, Show)

data Menge = Stueck Int | Gamm Int | Liter Double
  deriving (Eq, Show)

type Preis = Maybe Int

preis :: Artikel -> Menge -> Preis
preis (Apfel a) (Stueck n) = Just (n * apreis a)
preis Eier (Stueck n) = Just (n * 20)
preis (Kase k) (Gamm g) = Just (round (fromIntegral g * 1000 * kpreis k))
preis Schinken (Gamm g) = Just (div (g * 199) 100)
preis Salami (Gamm g) = Just (div (g * 159) 100)
preis (Milch bio) (Liter l) =
  Just (round (l * case bio of Bio -> 119; Konv -> 69))
preis _ _ = Nothing

-- Addition von Mengen
addere :: Menge -> Menge -> Menge
addere (Stueck i) (Stueck j) = Stueck (i + j)
addere (Gamm g) (Gamm h) = Gamm (g + h)
addere (Liter l) (Liter m) = Liter (l + m)
addere m n = error ("addiere: " * show m * ", " * show n)

-- Posten:
data Posten = Posten Artikel Menge
  deriving (Eq, Show)

cent :: Posten -> Int
cent (Posten a m) = fromMaybe 0 (preis a m) -- gibt keinen Laufzeitfehler!

-- Lagerhaltung
data Lager = Lager [Posten]
  deriving (Eq, Show)

leeresLager :: Lager
leeresLager = Lager []
```

```
suche :: Artikel -> Lager -> Maybe Menge
suche a (Lager ps) =
  listToMaybe [ m | Posten la m -> ps, la == a ]

einlagern :: Artikel -> Menge -> Lager -> Lager
einlagern a m (Lager ps) =
  let hinein a m [] = [Posten a m]
      hinein a m (Posten al mt:) |
          | a == al = (Posten a (addiere mt mt)) : |
          | otherwise = (Posten al mt : hinein a m) |
  in case preis a m of
    Nothing -> Lager ps
    _ -> Lager (hinein a m ps)

data Einkaufswagen = Etwag [Posten]
  deriving (Eq, Show)

leeresWagen :: Einkaufswagen
leeresWagen = Etwag []

einkauf :: Artikel -> Menge -> Einkaufswagen -> Einkaufswagen
einkauf a m (Etwag ps)
  | isJust (preis a m) = Etwag (Posten a m ps)
  | otherwise = Etwag ps

kasse :: Einkaufswagen -> Int
kasse (Etwag ps) = sum (map cent ps)

kassenbon :: Einkaufswagen -> String
kassenbon ew@(Etwag ps) =
  "Bob's_Auld's_Grocery_Shoppe'n,n" *
  "Artikel" * concatMap (\(Posten a m) ->
    "-----" * show a * " " * show m * "\n") ps *
  "Summe" * formatR 31 (showEuro (kasse ew))

artikel :: Posten -> String
artikel p@(Posten a m) =
  formatR 20 (show a) * " " *
  formatR 7 (show m) * " " *
  formatR 10 (showEuro (cent p)) * "\n"

menge :: Menge -> String
menge (Stueck n) = show n * "St"
menge (Gamm g) = show g * "G"
menge (Liter l) = show l * "L"

formatR :: Int -> String -> String
formatR n str = take n (str * replicate n ' ')

formatR :: Int -> String -> String
formatR n str =
  take n (replicate (n - length str) ' ' * str)

showEuro :: Int -> String
showEuro i =
  show (div i 100) * "." *
  show (mod (div i 10) 10) *
  show (mod i 10) * "EU"

inventur :: Lager -> Int
inventur (Lager l) = sum (map cent l)
```

Refakturierung im Einkaufsparadies

module Shoppe! where

import Data.Maybe

--- Modellierung der Artikel.

data Apfelsorte = Boskoop | CoiOrange | GrannySmith
deriving (Eq, Show)

apreis :: Apfelsorte -> Int
apreis Boskoop = 55
apreis CoiOrange = 60
apreis GrannySmith = 50

data Kaesesorte = Gouda | Appenzeller
deriving (Eq, Show)

kpreis :: Kaesesorte -> Double
kpreis Gouda = 1400
kpreis Appenzeller = 2270

data Bio = Bio | Konv
deriving (Eq, Show)

data Artikel =
 Apfel Apfelsorte | Eier
 | Kasee Kaesesorte | Schinken
 | Salami | Milch Bio
deriving (Eq, Show)

data Menge = Stueck Int | Gamm Int | Liter Double
deriving (Eq, Show)

type Preis = Maybe Int

preis :: Artikel -> Menge -> Preis
preis (Apfel a) (Stueck n) = Just (n * apreis a)
preis Eier (Stueck n) = Just (n * 20)
preis (Kasee k) (Gamm g) = Just (round (fromIntegral g * 1000 * kpreis k))
preis Schinken (Gamm g) = Just (div (g * 199) 100)
preis Salami (Gamm g) = Just (div (g * 159) 100)
preis (Milch bio) (Liter l) =
 Just (round (l * case bio of Bio -> 119; Konv -> 69))
preis _ _ = Nothing

--- Addition von Mengen
addere :: Menge -> Menge -> Menge
addere (Stueck i) (Stueck j) = Stueck (i + j)
addere (Gamm g) (Gamm h) = Gamm (g + h)
addere (Liter l) (Liter m) = Liter (l + m)
addere m n = error ("addiere: " * show m * ", " * show n)

--- Posten:

data Posten = Posten Artikel Menge
deriving (Eq, Show)

cent :: Posten -> Int
cent (Posten a n) = fromMaybe 0 (preis a n) -- gibt keinen Laufzeitfehler!

--- Lagerhaltung

data Lager = Lager [Posten]
deriving (Eq, Show)

leeresLager :: Lager
leeresLager = Lager []

Artikel

suche :: Artikel -> Lager -> Maybe Menge

suche a (Lager ps) =
 listToMaybe [m | Posten la m -> ps, la == a]

einlagern :: Artikel -> Menge -> Lager -> Lager

einlagern a m (Lager ps) =
 let hinen a m [] = [Posten a m]
 hinen a m (Posten al mt : _)
 | a == al = (Posten a (addiere mt mt)) :
 | otherwise = (Posten al mt : hinen a m)
 in case preis a m of
 Nothing -> Lager ps
 _ -> Lager (hinen a m ps)

data Einkaufswagen = Etwag [Posten]
deriving (Eq, Show)

leeresWagen :: Einkaufswagen
leeresWagen = Etwag []

einkauf :: Artikel -> Menge -> Einkaufswagen -> Einkaufswagen

einkauf a m (Etwag ps)
 | isJust (preis a m) = Etwag (Posten a m ps)
 | otherwise = Etwag ps

kasse :: Einkaufswagen -> Int
kasse (Etwag ps) = sum (map cent ps)

kassenbon :: Einkaufswagen -> String

kassenbon ew@(Etwag ps) =
 "Bob's_Audio_Grocery_Shopp@n" *
 "Artikel_____Menge_____Preis/n" *
 "-----\n" *
 concatMap artikel ps *
 "-----\n" *
 "Summe" * formatR 33 (showEuro (kasse ew))

artikel :: Posten -> String

artikel p@(Posten a m) =
 formatR 20 (show a) *
 formatR 7 (menge m) *
 formatR 10 (showEuro (cent p)) * "\n"

menge :: Menge -> String

menge (Stueck n) = show n * "St"
menge (Gamm g) = show g * "G"
menge (Liter l) = show l * "L"

format :: Int -> String -> String
format n str = take n (str * replicate n ' ')

formatR :: Int -> String -> String
formatR n str =
 take n (replicate (n - length str) ' ' * str)

showEuro :: Int -> String

showEuro i =
 show (div i 100) * "." *
 show (mod (div i 10) 10) *
 show (mod i 10) * "_EU"

inventur :: Lager -> Int

inventur (Lager l) = sum (map cent l)

Refakturierung im Einkaufsparadies

```
module Shoppe where
```

```
import Data.Maybe
```

```
-- Modellierung der Artikel.
```

```
data Apfelsorte = Boskoop | CoOrange | GrannySmith
  deriving (Eq, Show)
```

```
apreis :: Apfelsorte -> Int
apreis Boskoop = 55
apreis CoOrange = 60
apreis GrannySmith = 50
```

```
data Kaesesorte = Gouda | Appenzeller
  deriving (Eq, Show)
```

```
kpreis :: Kaesesorte -> Double
kpreis Gouda = 1400
kpreis Appenzeller = 2270
```

```
data Bio = Bio | Konv
  deriving (Eq, Show)
```

```
data Artikel =
  Apfel Apfelsorte | Eier
  | Kasse Kaesesorte | Schinken
  | Salami | Milch Bio
  deriving (Eq, Show)
```

```
data Menge = Stueck Int | Gamm Int | Liter Double
  deriving (Eq, Show)
```

```
type Preis = Maybe Int
```

```
preis :: Artikel -> Menge -> Preis
preis (Apfel a) (Stueck n) = Just (n * apreis a)
preis Eier (Stueck n) = Just (n * 20)
preis (Kasse k) (Gamm g) = Just (round (fromIntegral g * 1000 * kpreis k))
preis Schinken (Gamm g) = Just (div (g * 199) 100)
preis Salami (Gamm g) = Just (div (g * 199) 100)
preis (Milch bio) (Liter l) =
  Just (round (1 + case bio of Bio -> 119; Konv -> 69))
preis _ _ = Nothing
```

```
-- Addition von Mengen
addiere :: Menge -> Menge -> Menge
addiere (Stueck i) (Stueck j) = Stueck (i + j)
addiere (Gamm g) (Gamm h) = Gamm (g + h)
addiere (Liter l) (Liter m) = Liter (l + m)
addiere m n = error ("addiere: " * show m * ", " * show n)
```

```
-- Posten:
```

```
data Posten = Posten Artikel Menge
  deriving (Eq, Show)
```

```
cent :: Posten -> Int
cent (Posten a n) = fromMaybe 0 (preis a n) -- gibt keinen Laufzeitfehler!
```

```
-- Lagerhaltung
```

```
data Lager = Lager [Posten]
  deriving (Eq, Show)
```

```
leeresLager :: Lager
leeresLager = Lager []
```

Artikel

Posten

```
suche :: Artikel -> Lager -> Maybe Menge
```

```
suche a (Lager ps) =
  listToMaybe [m | Posten la m -> ps, la == a]
```

```
einlagern :: Artikel -> Menge -> Lager -> Lager
```

```
einlagern a m (Lager ps) =
  let hinen a m [] = [Posten a m]
      hinen a m (Posten al m1) | a == al = [Posten a (addiere m m1)]
                              | otherwise = [Posten al m1; hinen a m1]
  in case preis a m of
    Nothing -> Lager ps
    _ -> Lager (hinen a m ps)
```

```
data Einkaufswagen = Bagg [Posten]
  deriving (Eq, Show)
```

```
leeresWagen :: Einkaufswagen
```

```
leeresWagen = Bagg []
```

```
einkauf :: Artikel -> Menge -> Einkaufswagen -> Einkaufswagen
```

```
einkauf a m (Bagg ps)
  | isJust (preis a m) = Bagg (Posten a m ps)
  | otherwise = Bagg ps
```

```
kasse :: Einkaufswagen -> Int
kasse (Bagg ps) = sum (map cent ps)
```

```
kassenbon :: Einkaufswagen -> String
```

```
kassenbon w@(Bagg ps) =
  "Bob's_Auld's_Grocery_Shoppe\n" *
  "Artikel_____Menge_____Preis\n" *
  "-----\n" *
  concatMap artikel ps *
  "-----\n" *
  "Summe: " * formatR 33 (showEuro (kasse w))
```

```
artikel :: Posten -> String
```

```
artikel p@(Posten a m) =
  formatR 20 (show a) *
  formatR 7 (show m) *
  formatR 10 (showEuro (cent p)) * "\n"
```

```
menge :: Menge -> String
```

```
menge (Stueck n) = show n * ",St"
menge (Gamm g) = show g * ",G"
menge (Liter l) = show l * ",L"
```

```
formatL :: Int -> String -> String
formatL n str = take n (str * replicate n ' ')
```

```
formatR :: Int -> String -> String
formatR n str =
  take n (replicate (n - length str) ' ' * str)
```

```
showEuro :: Int -> String
```

```
showEuro i =
  show (div i 100) * "." *
  show (mod (div i 10) 10) *
  show (mod i 10) * "_EU"
```

```
inventur :: Lager -> Int
```

```
inventur (Lager l) = sum (map cent l)
```

Refakturierung im Einkaufsparadies

module Shoppel where

import Data.Maybe

-- Modellierung der Artikel.

```
data Apfelsorte = BioOrange | CoaOrange | GrannySmith
  deriving (Eq, Show)
```

```
apreis :: Apfelsorte -> Int
apreis BioOrange = 55
apreis CoaOrange = 60
apreis GrannySmith = 50
```

```
data Kaeasesorte = Gouda | Appenzeller
  deriving (Eq, Show)
```

```
kpreis :: Kaeasesorte -> Double
kpreis Gouda = 1400
kpreis Appenzeller = 2270
```

```
data Bio = Bio | Korv
  deriving (Eq, Show)
```

```
data Artikel =
  Apfel Apfelsorte | Eier
  | Kase Kaeasesorte | Schinken
  | Salami | Milch Bio
  deriving (Eq, Show)
```

```
data Menge = Stueck Int | Gamm Int | Liter Double
  deriving (Eq, Show)
```

type Preis = Maybe Int

```
preis :: Artikel -> Menge -> Preis
preis (Apfel a) (Stueck n) = Just (n * apreis a)
preis Eier (Stueck n) = Just (n * 20)
preis (Kase k) (Gamm g) = Just (round (fromIntegral g * 1000 * kpreis k))
preis Schinken (Gamm g) = Just (div (g * 190) 100)
preis Salami (Gamm g) = Just (div (g * 150) 100)
preis (Milch bio) (Liter l) =
  Just (round (l * case bio of Bio -> 110; Korv -> 60))
preis _ _ = Nothing
```

```
-- Addition von Mengen
addere :: Menge -> Menge -> Menge
addere (Stueck i) (Stueck j) = Stueck (i + j)
addere (Gamm g) (Gamm h) = Gamm (g + h)
addere (Liter l) (Liter m) = Liter (l + m)
addere m n = error ("addiere: " * show m * ", und, " * show n)
```

-- Posten:

```
data Posten = Posten Artikel Menge
  deriving (Eq, Show)
```

```
cant :: Posten -> Int
cant (Posten a n) = fromMaybe 0 (preis a n) -- gibt keinen Laufzeitfehler!
```

-- Lagerhaltung

```
data Lager = Lager [Posten]
  deriving (Eq, Show)
```

```
leeresLager :: Lager
leeresLager = Lager []
```

Artikel

Posten

Lager

```
suche :: Artikel -> Lager -> Maybe Menge
suche a (Lager ps) =
  listToMaybe [ m | Posten la m -> ps, la == a ]
```

```
einlagern :: Artikel -> Menge -> Lager -> Lager
einlagern a m (Lager ps) =
  let: hinein a m [] = [Posten a m]
      hinein a m (Posten al m1) |
        a == al = (Posten a (addiere m m1))
        | otherwise = (Posten al m1 : hinein a m1)
  in case preis a m of
    Nothing -> Lager ps
    _ -> Lager (hinein a m ps)
```

```
data Einkaufswagen = Bagg [Posten]
  deriving (Eq, Show)
```

```
leeresWagen :: Einkaufswagen
leeresWagen = Bagg []
```

```
einkauf :: Artikel -> Menge -> Einkaufswagen -> Einkaufswagen
einkauf a m (Bagg ps) =
  | isJust (preis a m) = Bagg (Posten a m : ps)
  | otherwise = Bagg ps
```

```
kasse :: Einkaufswagen -> Int
kasse (Bagg ps) = sum (mp cent ps)
```

```
kassenbon :: Einkaufswagen -> String
kassenbon w@(Bagg ps) =
  "Bob's_Auld's_Grocery_Shoppark\n" *
  "Artikel: _____Menge: _____Preis:\n" *
  "_____|\n" *
  concatMap artikel ps *
  "Summe: " * formatR 31 (showEuro (kasse w))
```

```
artikel :: Posten -> String
artikel p@(Posten a n) =
  formatR 20 (show a) *
  formatR 7 (mergen n) *
  formatR 10 (showEuro (cant p)) * "\n"
```

```
menge :: Menge -> String
menge (Stueck n) = show n * "St"
menge (Gamm g) = show g * "G"
menge (Liter l) = show l * "L"
```

```
formatL :: Int -> String -> String
formatL n str = take n (str * replicate n ' ')
```

```
formatR :: Int -> String -> String
formatR n str =
  take n (replicate (n - length str) ' ' * str)
```

```
showEuro :: Int -> String
showEuro i =
  show (div i 100) * "." *
  show (mod (div i 10) 10) *
  show (mod i 10) * "EUR"
```

```
inventur :: Lager -> Int
inventur (Lager l) = sum (mp cent l)
```

Lager

Refakturierung im Einkaufsparadies

module Shoppel where

import Data.Maybe

-- Modifizierung der Artikel.

```
data Apfelsorte = Biocoop | CoaOrange | GrannySmith
  deriving (Eq, Show)
```

```
apreis :: Apfelsorte -> Int
apreis Biocoop = 55
apreis CoaOrange = 60
apreis GrannySmith = 50
```

```
data Kaeasesorte = Gouda | Appenzeller
  deriving (Eq, Show)
```

```
kpreis :: Kaeasesorte -> Double
kpreis Gouda = 1400
kpreis Appenzeller = 2270
```

```
data Bio = Bio | Korv
  deriving (Eq, Show)
```

```
data Artikel =
  Apfel Apfelsorte | Eier
  | Kase Kaeasesorte | Schinken
  | Salami | Milch Bio
  deriving (Eq, Show)
```

```
data Menge = Stueck Int | Gamm Int | Liter Double
  deriving (Eq, Show)
```

type Preis = Maybe Int

```
preis :: Artikel -> Menge -> Preis
preis (Apfel a) (Stueck n) = Just (n * apreis a)
preis Eier (Stueck n) = Just (n * 20)
preis (Kase k) (Gamm g) = Just (round (fromIntegral g * 1000 * kpreis k))
preis Schinken (Gamm g) = Just (div (g * 199) 100)
preis Salami (Gamm g) = Just (div (g * 199) 100)
preis (Milch bio) (Liter l) =
  Just (round (l * case bio of Bio -> 119; Korv -> 69))
preis _ _ = Nothing
```

```
-- Addition von Mengen
addiere :: Menge -> Menge -> Menge
addiere (Stueck i) (Stueck j) = Stueck (i + j)
addiere (Gamm g) (Gamm h) = Gamm (g + h)
addiere (Liter l) (Liter m) = Liter (l + m)
addiere m n = error ("addiere: " * show m * ", und, " * show n)
```

-- Posten:

```
data Posten = Posten Artikel Menge
  deriving (Eq, Show)
```

```
cent :: Posten -> Int
cent (Posten a n) = fromMaybe 0 (preis a n) -- gibt keinen Laufzeitfehler!
```

-- Lagerhaltung

```
data Lager = Lager [Posten]
  deriving (Eq, Show)
```

```
leeresLager :: Lager
leeresLager = Lager []
```

Artikel

Posten

Lager

```
suche :: Artikel -> Lager -> Maybe Menge
suche a (Lager ps) =
  listToMaybe [ m | Posten la m -> ps, la == a ]
```

```
einlagern :: Artikel -> Menge -> Lager -> Lager
einlagern a m (Lager ps) =
  let: hinein a m [] = (Posten a m)
      hinein a m (Posten al m1) |
        a == al = (Posten a (addiere m m1))
        | otherwise = (Posten al m1; hinein a m1)
  in case preis a m of
    Nothing -> Lager ps
    _ -> Lager (hinein a m ps)
```

```
data Einkaufswagen = Bueag [Posten]
  deriving (Eq, Show)
```

```
leeresWagen :: Einkaufswagen
leeresWagen = Bueag []
```

```
einkauf :: Artikel -> Menge -> Einkaufswagen -> Einkaufswagen
einkauf a m (Bueag ps)
  | isJust (preis a m) = Bueag (Posten a m ps)
  | otherwise = Bueag ps
```

```
kasse :: Einkaufswagen -> Int
kasse (Bueag ps) = sum (map cent ps)
```

```
kassenbon :: Einkaufswagen -> String
kassenbon ew@(Bueag ps) =
  "Bob's_Auflue_Grocery_Shopping_rn" *
  "Artikel: _____Preis:rn" *
  "-----rn" *
  concatMap artikel ps *
  "Summe: " * formatR 31 (showEuro (kasse ew))
```

```
artikel :: Posten -> String
artikel p@(Posten a m) =
  formatR 20 (show a) *
  formatR 7 (menge m) *
  formatR 10 (showEuro (cent p)) * "rn"
```

```
menge :: Menge -> String
menge (Stueck n) = show n * "St"
menge (Gamm g) = show g * "G"
menge (Liter l) = show l * "L"
```

```
formatl :: Int -> String -> String
formatl n str = take n (str * replicate n ' ')
```

```
formatR :: Int -> String -> String
formatR n str =
  take n (replicate (n - length str) ' ' * str)
```

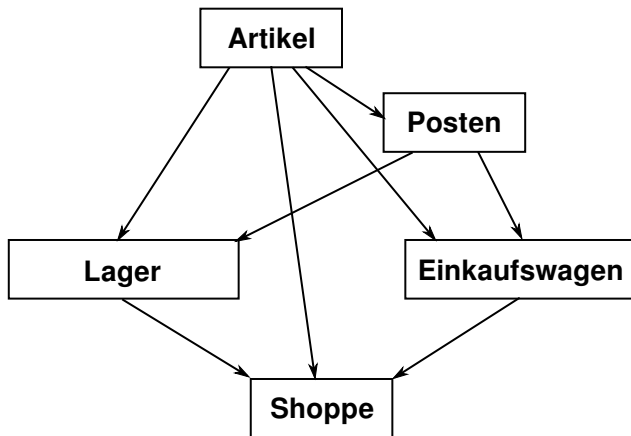
```
showEuro :: Int -> String
showEuro i =
  show (div i 100) * "." *
  show (mod (div i 10) 10) *
  show (mod i 10) * "EUR"
```

```
inventur :: Lager -> Int
inventur (Lager l) = sum (map cent l)
```

Lager

Einkaufswagen

Refakturierung im Einkaufsparadies: Modularchitektur



Refakturierung im Einkaufsparadies I: Artikel

- ▶ Es wird **alles** exportiert
- ▶ Reine Datenmodellierung

```
module Artikel where
```

```
data Apfelsorte = Boskoop | CoxOrange | GrannySmith  
apreis :: Apfelsorte → Int
```

```
data Kaesesorte = Gouda | Appenzeller  
kpreis :: Kaesesorte → Double
```

```
data Menge = Stueck Int | Gramm Int | Liter Double  
addiere :: Menge → Menge → Menge
```

Refakturierung im Einkaufsparadies II: Posten

- ▶ Implementiert ADT Posten:

```
data Posten = Posten { artikel :: Artikel  
                      , menge   :: Menge  
                      }
```

```
module Posten(  
  Posten,  
  artikel,  
  menge,  
  posten,  
  cent,  
  hinzu) where
```

- ▶ Konstruktor wird **nicht** exportiert
- ▶ Garantierte Invariante:
 - ▶ Posten hat immer die korrekte Menge zu Artikel

```
posten :: Artikel → Menge → Maybe Posten  
posten a m =  
  case preis a m of  
    Just _ → Just (Posten a m)  
    Nothing → Nothing
```


Refakturierung im Einkaufsparadies III: Lager

```
module Lager(  
  Lager,  
  leeresLager,  
  einlagern,  
  suche,  
  liste,  
  inventur  
) where  
import Artikel  
import Posten
```

- ▶ Implementiert ADT Lager

```
data Lager
```

- ▶ Signatur der exportierten Funktionen:

```
leeresLager :: Lager
```

```
einlagern :: Artikel → Menge → Lager → Lager
```

```
suche :: Artikel → Lager → Maybe Menge
```

```
liste :: Lager → [(Artikel, Menge)]
```

```
inventur :: Lager → Int
```

- ▶ Garantierte **Invariante**:
 - ▶ Lager enthält keine doppelten Artikel

Refakturierung IV: Einkaufswagen

```
module Einkaufswagen(  
  Einkaufswagen,  
  leererWagen,  
  einkauf,  
  kasse,  
  kassenbon  
) where
```

- ▶ Implementiert ADT Einkaufswagen

```
data Einkaufswagen = Ekwg [Posten]
```

- ▶ Garantierte Invariante:

- ▶ Korrekte Menge zu Artikel im Einkaufswagen

```
einkauf :: Artikel → Menge  
                → Einkaufswagen  
                → Einkaufswagen  
einkauf a m (Ekwg ps) =  
  case posten a m of  
    Just p → Ekwg (p: ps)  
    Nothing → Ekwg ps
```

- ▶ Nutzt dazu ADT Posten

Refakturierung im Einkaufsparadies V: Hauptmodul

► Nutzt andere Module

```
module Shoppe where
```

```
import Artikel
```

```
import Lager
```

```
import Einkaufswagen
```

```
l0= leeresLager
```

```
l1= einlagern (Apfel Boskoop) (Stueck 1) l0
```

```
l2= einlagern Schinken (Gramm 50) l1
```

```
l3= einlagern (Milch Bio) (Liter 6) l2
```

```
l4= einlagern (Apfel Boskoop) (Stueck 4) l3
```

```
l5= einlagern (Milch Bio) (Liter 4) l4
```

```
l6= einlagern Schinken (Gramm 50) l5
```

Benutzung von ADTs

- ▶ **Operationen** und **Typen** müssen **importiert** werden
- ▶ Möglichkeiten des Imports:
 - ▶ **Alles** importieren
 - ▶ **Nur bestimmte** Operationen und Typen importieren
 - ▶ Bestimmte Typen und Operationen **nicht** importieren

Importe in Haskell

- ▶ Syntax:

```
import [ qualified ] M [ as N ] [ hiding ] [( Bezeichner )]
```

- ▶ *Bezeichner* geben an, **was** importiert werden soll:
 - ▶ Ohne Bezeichner wird **alles** importiert
 - ▶ Mit **hiding** werden Bezeichner **nicht** importiert
- ▶ Für jeden exportierten Bezeichner *f* aus *M* wird importiert
 - ▶ *f* und **qualifizierter** Bezeichner *M.f*
 - ▶ **qualified**: **nur qualifizierter** Bezeichner *M.f*
 - ▶ Umbenennung bei Import mit *as* (dann *N.f*)
 - ▶ Klasseninstanzen und Typsynonyme werden immer importiert
- ▶ Alle Importe stehen immer am **Anfang** des Moduls

Beispiel

module M(a, b) **where** . . .

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	<i>(nothing)</i>
import M(a)	a, M.a
import qualified M	M.a, M.b
import qualified M()	<i>(nothing)</i>
import qualified M(a)	M.a
import M hiding ()	a, b, M.a, M.b
import M hiding (a)	b, M.b
import qualified M hiding ()	M.a, M.b
import qualified M hiding (a)	M.b
import M as B	a, b, B.a, B.b
import M as B(a)	a, B.a
import qualified M as B	B.a, B.b

Quelle: Haskell98-Report, Sect. 5.3.4

Ein typisches Beispiel

- ▶ Modul implementiert Funktion, die auch importiert wird
- ▶ Umbenennung nicht immer praktisch
- ▶ Qualifizierter Import führt zu **langen** Bezeichnern
- ▶ Einkaufswagen implementiert Funktionen `artikel` und `menge`, die auch aus `Posten` importiert werden:

```
import Posten hiding (artikel, menge)
import qualified Posten as P(artikel, menge)
```

```
artikel p =
  formatL 20 (show (P.artikel p)) ++
  formatR 7  (menge (P.menge p)) ++
  formatR 10 (showEuro (cent p)) ++ "\n"
```

Schnittstelle vs. Implementation

- ▶ Gleiche **Schnittstelle** kann unterschiedliche **Implementationen** haben

- ▶ Beispiel: (endliche) Abbildungen

Endliche Abbildungen

- ▶ Viel gebraucht, oft in Abwandlungen (Hashtables, Sets, Arrays)
- ▶ Abstrakter Datentyp für **endliche Abbildungen**:

- ▶ Datentyp

```
data Map  $\alpha$   $\beta$ 
```

- ▶ Leere Abbildung:

```
empty :: Map  $\alpha$   $\beta$ 
```

- ▶ Abbildung auslesen:

```
lookup :: Ord  $\alpha$   $\Rightarrow$   $\alpha$   $\rightarrow$  Map  $\alpha$   $\beta$   $\rightarrow$  Maybe  $\beta$ 
```

- ▶ Abbildung ändern:

```
insert :: Ord  $\alpha$   $\Rightarrow$   $\alpha$   $\rightarrow$   $\beta$   $\rightarrow$  Map  $\alpha$   $\beta$   $\rightarrow$  Map  $\alpha$   $\beta$ 
```

- ▶ Abbildung löschen:

```
delete :: Ord  $\alpha$   $\Rightarrow$   $\alpha$   $\rightarrow$  Map  $\alpha$   $\beta$   $\rightarrow$  Map  $\alpha$   $\beta$ 
```

Eine naheliegende Implementation

- ▶ Modellierung als Haskell-Funktion:

```
data Map  $\alpha$   $\beta$  = Map ( $\alpha \rightarrow$  Maybe  $\beta$ )
```

- ▶ Damit einfaches lookup, insert, delete:

```
empty = Map ( $\lambda x \rightarrow$  Nothing)
```

```
lookup a (Map s) = s a
```

```
insert a b (Map s) =  
  Map ( $\lambda x \rightarrow$  if  $x == a$  then Just b else s x)
```

```
delete a (Map s) =  
  Map ( $\lambda x \rightarrow$  if  $x == a$  then Nothing else s x)
```

- ▶ Instanzen von Eq, Show **nicht möglich**
- ▶ **Speicherleck**: überschriebene Zellen werden nicht freigegeben

Endliche Abbildungen: Anwendungsbeispiel

- ▶ Lager als endliche Abbildung:

```
data Lager = Lager (M.Map Artikel Menge)
```

- ▶ Artikel suchen:

```
suche :: Artikel → Lager → Maybe Menge  
suche a (Lager l) = M.lookup a l
```

- ▶ Ins Lager hinzufügen:

```
einlagern :: Artikel → Menge → Lager → Lager  
einlagern a m (Lager l) =  
  case posten a m of  
    Just _ → case M.lookup a l of  
      Just q → Lager (M.insert a (addiere m q) l)  
      Nothing → Lager (M.insert a m l)  
    Nothing → Lager l
```

- ▶ Für Inventur fehlt Möglichkeit zur **Iteration**
- ▶ Daher: Map als **Assoziativliste**

Map als sortierte Assoziativliste

```
data Map  $\alpha$   $\beta$  = Map { toList :: [( $\alpha$ ,  $\beta$ )] }
```

- ▶ Einfache Implementierung:

```
insert :: Ord  $\alpha$   $\Rightarrow$   $\alpha$   $\rightarrow$   $\beta$   $\rightarrow$  Map  $\alpha$   $\beta$   $\rightarrow$  Map  $\alpha$   $\beta$ 
insert a v (Map s) = Map (insert' s) where
  insert' [] = [(a, v)]
  insert' s0@((b, w):s) | a > b = (b, w): insert' s
                        | a == b = (a, v): s
                        | a < b = (a, v): s0
```

- ▶ Zusatzfunktionalität:
 - ▶ Iteration (Selektor toList)
 - ▶ Instanzen von Eq und Show (abgeleitet)
- ▶ ... ist aber **ineffizient** (Zugriff/Löschen in $\mathcal{O}(n)$)
- ▶ Deshalb: **balancierte Bäume**

AVL-Bäume und Balancierte Bäume

AVL-Bäume

Ein Baum ist **ausgeglichen**, wenn

- ▶ alle Unterbäume ausgeglichen sind, und
- ▶ der Höhenunterschied zwischen zwei Unterbäumen höchstens eins beträgt.

Balancierte Bäume

Ein Baum ist **balanciert**, wenn

- ▶ alle Unterbäume balanciert sind, und
- ▶ für den linken und rechten Unterbaum l , r gilt:

$$size(l) \leq w \cdot size(r) \quad (1)$$

$$size(r) \leq w \cdot size(l) \quad (2)$$

w — **Gewichtung** (Parameter des Algorithmus)

Implementation von balancierten Bäumen

- ▶ Der Datentyp

```
data Tree  $\alpha$  = Null  
      | Node Int (Tree  $\alpha$ )  $\alpha$  (Tree  $\alpha$ )  
      deriving Eq
```

- ▶ Gewichtung (Parameter des Algorithmus):

```
weight :: Int
```

- ▶ Hilfskonstruktor, setzt Größe (l, r balanciert)

```
node :: Tree  $\alpha \rightarrow \alpha \rightarrow$  Tree  $\alpha \rightarrow$  Tree  $\alpha$ 
```

- ▶ Selektor: Größe des Baumes (0 für Null)

```
size :: Tree  $\alpha \rightarrow$  Int
```

Implementation von balancierten Bäumen

- ▶ Hilfskonstruktor, balanciert ggf. neu aus:

```
mkNode :: Tree  $\alpha$   $\rightarrow$   $\alpha$   $\rightarrow$  Tree  $\alpha$   $\rightarrow$  Tree  $\alpha$ 
```

- ▶ Voraussetzungen:

- ▶ l, r balanciert

- ▶ Gesamtbaum "fast" balanciert:

$$size(l) - 1 \leq w \cdot size(r) \tag{3}$$

$$size(r) - 1 \leq w \cdot size(l) \tag{4}$$

- ▶ Wird beim Löschen und Einfügen benutzt

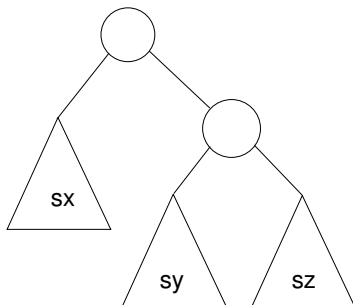
Balance sicherstellen

► Problem:

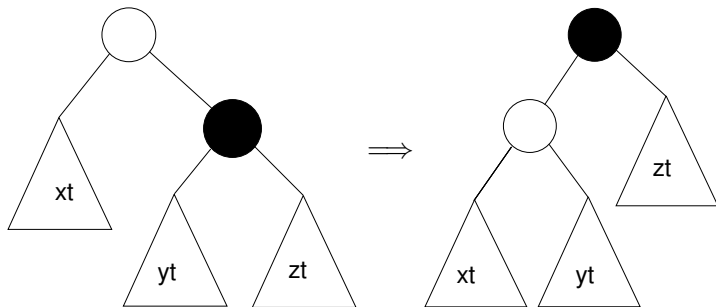
Nach Löschen oder Einfügen zu großes Ungewicht

► Lösung:

Rotieren der Unterbäume

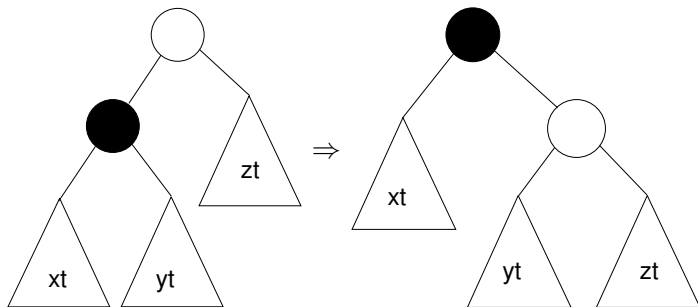


Linksrotation



$\text{rotl} :: \text{Tree } \alpha \rightarrow \text{Tree } \alpha$
 $\text{rotl } (\text{Node } _ \text{xt } y \ (\text{Node } _ \text{yt } x \ \text{zt})) =$
 $\text{node } (\text{node } \text{xt } y \ \text{yt}) \ x \ \text{zt}$

Rechtsrotation

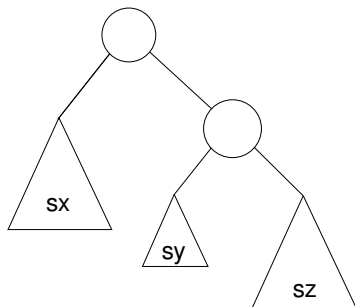


$\text{rotr} :: \text{Tree } \alpha \rightarrow \text{Tree } \alpha$

$\text{rotr } (\text{Node } _ (\text{Node } _ \text{ut } y \text{ vt}) \text{ x } \text{rt}) =$
 $\text{node ut y } (\text{node vt x } \text{rt})$

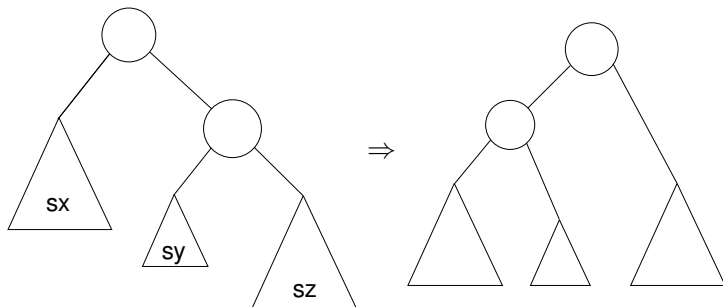
Balanciertheit sicherstellen

- ▶ Fall 1: Äußerer Unterbaum zu groß



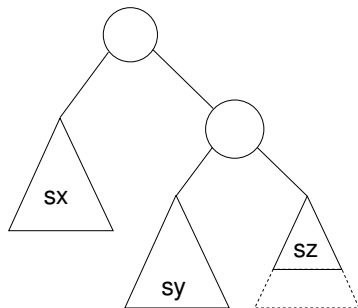
Balanciertheit sicherstellen

- ▶ Fall 1: Äußerer Unterbaum zu groß
- ▶ Lösung: Linksrotation



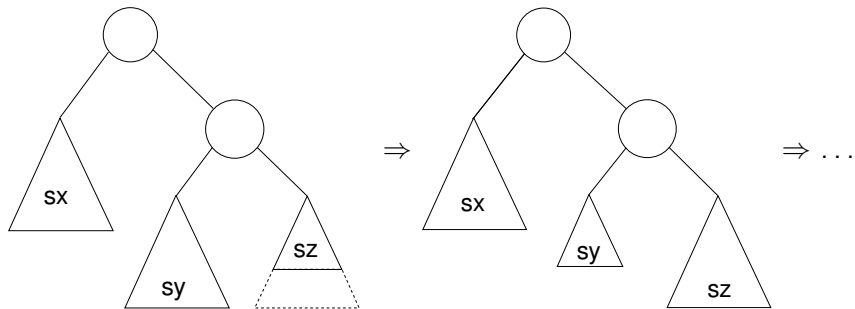
Balanciertheit sicherstellen

- ▶ Fall 2: Innerer Unterbaum zu groß oder gleich groß



Balanciertheit sicherstellen

- ▶ Fall 2: Innerer Unterbaum zu groß oder gleich groß
- ▶ Reduktion auf vorherigen Fall durch Rechtsrotation des Unterbaumes



Balance sicherstellen

- ▶ Hilfsfunktion: **Balance** eines Baumes

```
bias :: Tree  $\alpha$   $\rightarrow$  Ordering
bias Null = EQ
bias (Node _ lt _ rt) = compare (size lt) (size rt)
```

- ▶ Zu implementieren: mkNode lt y rt
 - ▶ Voraussetzung: lt, rt balanciert
 - ▶ Konstruiert neuen balancierten Baum mit Knoten y
- ▶ Fallunterscheidung:
 - ▶ rt zu groß, zwei Unterfälle:
 - ▶ Linker Unterbaum von rt kleiner (Fall 1): bias rt = LT
 - ▶ Linker Unterbaum von rt größer/gleich groß (Fall 2): bias rt = EQ, bias rt = GT
 - ▶ lt zu groß, zwei Unterfälle (symmetrisch).

Konstruktion eines ausgeglichenen Baumes

- ▶ Voraussetzung: lt, rt balanciert

```
mkNode :: Tree  $\alpha$   $\rightarrow$   $\alpha$   $\rightarrow$  Tree  $\alpha$   $\rightarrow$  Tree  $\alpha$ 
```

```
mkNode lt x rt
```

```
| ls + rs < 2 = node lt x rt
```

```
| weight* ls < rs =
```

```
  if bias rt == LT then rotl (node lt x rt)
```

```
  else rotl (node lt x (rotr rt))
```

```
| ls > weight* rs =
```

```
  if bias lt == GT then rotr (node lt x rt)
```

```
  else rotr (node (rotl lt) x rt)
```

```
| otherwise = node lt x rt where
```

```
  ls = size lt; rs = size rt
```


Balancierte Bäume als Maps

- ▶ Endliche Abbildung: Bäume mit (key, value) Paaren

- ▶ lookup' liest Element aus:

```
lookup' :: Ord  $\alpha$   $\Rightarrow$   $\alpha \rightarrow$  Tree ( $\alpha$ ,  $\beta$ )  $\rightarrow$  Maybe  $\beta$ 
```

- ▶ insert' fügt neues Element ein:

```
insert' :: Ord  $\alpha$   $\Rightarrow$   $\alpha \rightarrow$   $\beta \rightarrow$  Tree ( $\alpha$ ,  $\beta$ )  $\rightarrow$  Tree ( $\alpha$ ,  $\beta$ )
```

```
insert' k v Null = node Null (k, v) Null
```

```
insert' k v (Node n | a@(kn, _) r)
```

```
  | k < kn = mkNode (insert' k v l) a r
```

```
  | k == kn = Node n | (k, v) r
```

```
  | k > kn = mkNode l a (insert' k v r)
```

- ▶ remove' löscht ein Element

- ▶ Benötigt Hilfsfunktion join :: Tree $\alpha \rightarrow$ Tree $\alpha \rightarrow$ Tree α

Zusammenfassung Balancierte Bäume

- ▶ Verkapselung des Datentypen:

```
data Map  $\alpha$   $\beta$  = Map { tree :: Tree ( $\alpha$ ,  $\beta$ ) }
```

```
insert :: Ord  $\alpha$   $\Rightarrow$   $\alpha \rightarrow \beta \rightarrow$  Map  $\alpha$   $\beta \rightarrow$  Map  $\alpha$   $\beta$   
insert k v (Map t) = Map (insert ' k v t)
```

- ▶ Auslesen, einfügen und löschen: logarithmischer Aufwand ($\mathcal{O}(\log n)$)
- ▶ Fold: linearer Aufwand ($\mathcal{O}(n)$)
- ▶ Guter durchschnittlichen Aufwand
- ▶ Auch in der Haskell-Bücherei: `Data.Map` (mit vielen weiteren Funktionen)

Benchmarking: Setup

- ▶ Wie **schnell** sind die Implementationen **wirklich**?
- ▶ Benchmarking: nicht trivial
 - ▶ Verzögerte Auswertung und optimierender Compiler
 - ▶ Messen wir das **richtige**?
 - ▶ Benchmarking-Tool: Criterion
- ▶ Setup: Map Int String mit 50000 zufälligen Einträgen erzeugen
- ▶ Darin:
 - ▶ Einmal zufällig lesen (lookup), schreiben (insert), löschen (delete)
 - ▶ Sequenz aus fünfmal löschen und schreiben, zweihundertmal lesen (mixed)

Benchmarking: Resultate

	create	lookup	insert	delete	mixed
(1)	358.4 ms 7.483 ms	2.66 ms 134.70 μ s	10.75 ns 159.70 ps	10.90 ns 170.90 ps	1.83 ms 111.80 μ s
(2)	6.20 s 37.59 ms	11.57 μ s 351.3 ns	133.8 μ s 2.36 μ s	148.40 μ s 1.99 μ s	5.67 ms 128.10 μ s
(3)	470.00 ms 2.69 ms	265.10 ns 4.54 ns	138.90 μ s 2.35 μ s	137.60 μ s 3.00 μ s	2.18 ms 81.68 μ s
(4)	392.7 ms 5.02 ms	189.2 ns 13.41 ns	135.7 μ s 2.00 μ s	134.50 μ s 3.10 μ s	2.08 ms 80.22 μ s

(1) MapFun, (2) MapList, (3) MapWeighted, (4) Data.Map.Lazy
Einträge: durchschnittl. Ausführungszeit, Standardabweichung

Defizite von Haskell's Modulsystem

- ▶ Signatur ist nur **implizit**
 - ▶ Exportliste enthält nur Bezeichner
 - ▶ Wünschenswert: Signatur an der Exportliste annotierbar, oder Signaturen in separater Datei
 - ▶ In Java: **Interfaces**
- ▶ Klasseninstanzen werden **immer** exportiert.
- ▶ Kein **Paket-System**

ADTs vs. Objekte

- ▶ ADTs (Haskell): **Typ** plus **Operationen**
- ▶ Objekte (z.B. Java): **Interface**, **Methoden**.
- ▶ **Gemeinsamkeiten:**
 - ▶ Verkapselung (information hiding) der Implementation
- ▶ **Unterschiede:**
 - ▶ Objekte haben **internen Zustand**, ADTs sind **referentiell transparent**;
 - ▶ Objekte haben **Konstruktoren**, ADTs nicht (Konstruktoren nicht unterscheidbar)
 - ▶ **Vererbungsstruktur** auf Objekten (**Verfeinerung** für ADTs)
 - ▶ Java: `interface` eigenes Sprachkonstrukt
 - ▶ Java: `packages` für Sichtbarkeit

Zusammenfassung

- ▶ **Abstrakte Datentypen** (ADTs):
 - ▶ Besteht aus **Typen** und **Operationen** darauf
- ▶ Realisierung in Haskell durch **Module**
- ▶ Beispieldatentypen: endliche Abbildungen
- ▶ Nächste Vorlesung: ADTs durch **Eigenschaften** spezifizieren