

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 8 vom 21.12.2020: Abstrakte Datentypen

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität
Bremen

Wintersemester 2020/21

Organisatorisches

- ▶ Abgabe des 7. Übungsblattes in Gruppen zu **drei** Studenten.
 - ▶ Bitte **jetzt** eine Gruppe suchen!
- ▶ Klausurtermine:
 - ▶ Klausur: 03.02.2020, 10:00/11:30/15:00
 - ▶ Wiederholungstermin: 21.04.2020, 10:00/11:30/15:00

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

I. Modularisierung und Abstrakte Datentypen

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:

- ① Werte des Typen können nur über die Operationen **erzeugt** werden
- ② Eigenschaften von Werten des Typen werden nur über die Operationen **beobachtet**
- ③ 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** durch **Konstruktoren**
 - ▶ Keine Einschränkungen
 - ▶ Insbesondere keine Gleichheiten der Konstruktoren ($[] \neq x:xs$, $x:ls \neq y:ls$ etc.)
- ▶ ADTs:
 - ▶ Keine ausgezeichneten Konstruktoren
 - ▶ Einschränkungen und Invarianten möglich
 - ▶ Gleichheiten möglich

ADTs vs. Objekte

- ▶ ADTs (z.B. 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
 - ▶ **Vererbungsstruktur** auf Objekten (**Verfeinerung** für ADTs)
 - ▶ Java: interface eigenes Sprachkonstrukt
 - ▶ Java: packages für Sichtbarkeit

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 **rekursiv** sein (*don't try at home*)

Refakturierung im Einkaufsparadies

```

module Shoppe4 where

import Data.Maybe

-- Modellierung der Artikel.

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

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

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

kpreis :: Kaesesorte -> Double
kpreis Gouda = 1450
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 (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 ++ " und: " ++ 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 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 = Elwg [Posten]
  deriving (Eq, Show)

leererWagen :: Einkaufswagen
leererWagen = Elwg []

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

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

kassenbon :: Einkaufswagen -> String
kassenbon ew (Elwg ps) =
  "Bibi's-Aulde-Grocery-Shoppe, n." ++
  "Artikel: " ++ show (concatMap artikel ps) ++
  "Menge: " ++ show (concatMap mnt ps) ++
  "Summe: " ++ formatR 31 (showEuro (kasse ew))

artikel :: Posten -> String
artikel p (Posten a m) =
  formatL 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) ++ "EUR"

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

```

Refakturierung im Einkaufsparadies

```

module Shopper where
import Data.Maybe

-- Modellierung der Artikel.

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

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

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

kpreis :: Kaesesorte -> Double
kpreis Gouda = 1450
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 | Gramm 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) (Gramm g) = Just (round (fromIntegral g * 1000 * kpreis k))
preis Schinken (Gramm g) = Just (div (g * 199) 100)
preis Salami (Gramm g) = Just (div (g * 159) 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 (Gramm g) (Gramm h) = Gramm (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 m) = fromMaybe 0 (preis a m) -- 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 hinein a m [] = [Posten a m]
      hinein a m (Posten al m1:) =
        | a == al = (Posten a (addiere m m1)) : l
        | otherwise = (Posten al m1 : hinein a m l)
  in case preis a m of
    Nothing -> Lager ps
    _ -> Lager (hinein a m ps)

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

leeresWagen :: Einkaufswagen
leeresWagen = EWag []

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

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

kassenbon :: Einkaufswagen -> String
kassenbon ew (EWag ps) =
  "Bob's-Audite-Grocery-Shoppe,n,n" ++
  "Artikel-----Menge-----Preis,n" ++
  "-----" ++
  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)) ++ ",n"

menge :: Menge -> String
menge (Stueck n) = show n ++ "St"
menge (Gramm 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) ++ "EUR"

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

```

Refakturierung im Einkaufsparadies

```

module Shopper where

import Data.Maybe

-- Modellierung der Artikel.

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

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

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

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

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

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

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

type Preis = Maybe Int

preis :: Artikel -> Merg -> 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 (1 + case bio of Bio -> 119; Konv -> 69))
preis _ _ = Nothing

-- Addition von Mengen
addiere :: Merg -> Merg -> Merg
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 Merg
  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 []

```

Artikel

Posten

```

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

einlagern :: Artikel -> Merg -> 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 = Etwag [Posten]
  deriving (Eq, Show)

leeresWagen :: Einkaufswagen
leeresWagen = Etwag []

einkaufen :: Artikel -> Merg -> Einkaufswagen -> Einkaufswagen
einkaufen 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 Auldre Grocery Shoppe, n" ++
  "Artikel " ++ show (length ps) ++
  " Menge " ++ show (sum (map (length) ps)) ++
  " Summe " ++ show (round (fromIntegral (sum (map (length) ps) * 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 :: Merg -> 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) ++ "EUR"

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

```

Refakturierung im Einkaufsparadies

```

module Shopped where

import Data.Maybe

-- Modellierung der Artikel.

data ApfelSorte = Boskoop | CoxOrange | GrannySmith
    deriving (Eq, Show)

apreis :: ApfelSorte -> Int
apreis Boskoop = 55
apreis CoxOrange = 60
apreis GrannySmith = 50

data KasseSorte = Gaudi | Appenzeller
    deriving (Eq, Show)

kpreis :: KasseSorte -> Double
kpreis Gaudi = 1450
kpreis Appenzeller = 2270

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

data Artikel =
    | Apfel ApfelSorte
    | Kasse KasseSorte
    | Salami
    deriving (Eq, Show)

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

type Preis = Maybe Int

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

-- Addition von Mengen
addiere :: Menge -> Menge -> Menge
addiere (Stück i) (Stück j) = Stück (i + j)
addiere (Gamm g) (Gamm h) = Gamm (g + h)
addiere (Liter i) (Liter j) = Liter (i + j)
addiere m1 n = error ("addiere: " <= show m1 <= ", " <= show n)

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

cant :: Posten -> Int
cant (Posten a m) = fromIntegral 0 (preis a m) -- gibt keinen Laubstiftfehler!

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

leeresLager :: Lager
leeresLager = Lager []

```

```

zuuche :: Artikel => Lager-> Maybe Menge
zuuche a (Lager ps) =
  listMMaybe [m] $ Posten la m-> ps, la == a ]

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

data Einkaufswagen = Blog (Posten)
  deriving (Eq, Show)

leererWagen :: Einkaufswagen
leererWagen = Blog []

einkauf :: Artikel-> Menge-> Einkaufswagen-> Einkaufswagen
einkauf a m (Blog ps) =
  | lg ist (preis a mt) = Blog (Posten a m : ps)
  | otherwise = Blog ps

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

kassenton :: Einkaufswagen-> String
kassenton ew(Blog ps) =
  "Bob's_Aside_Grocery_Shopp(n,m)*\n"
  ^ "Artikel _____ Menge _____ Preis(n)*\n"
  ^ "-----\n"
  ^ concatMap artikel ps
  ^ "-----\n"
  ^ "Summe" ^ formatR 31 (showEuro (kasse ew))

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

menge :: Menge-> String
menge (Stueck n) = show n ^ " St."
menge (Gramm 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) ^ ".EUR"

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

```

Lager

Artikel

Posten

Lager

Refakturierung im Einkaufsparadies

module Shoppet where

import Data.Maybe

--- Modellierung der Artikel.

data Apfelorte = Boskop | CoxOrange | GrannySmith
deriving (Eq, Show)

apreis :: Apfelorte → Int
apreis Boskop = 55
apreis CoxOrange = 60
apreis GrannySmith = 50

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

kpreis :: Kasesorte → Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270

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

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

data Menge = Stueck Int | Gramm 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) (Gramm g) = Just (round (fromIntegral g * 1000 * kpreis k))
preis (Schinken (Gramm g)) = Just (div (g * 199) 100)
preis (Salami (Gramm 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
addiere :: Menge → Menge → Menge
addiere (Stueck i) (Stueck j) = Stueck (i + j)
addiere (Gramm g) (Gramm h) = Gramm (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 m) = fromMaybe 0 (preis a m) --- 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) =
listOfMaybe [m | Posten la m ← ps, la == a]

einlagern :: Artikel → Menge → Lager → Lager
einlagern a m (Lager ps) =
let hineln a m l = (Posten a m)
hineln a m (Posten al mt:l)
| a == al = (Posten a (addiere mt:l))
| otherwise = (Posten al mt: hineln a m l)
in case preis a m of
Nothing → Lager ps
_ → Lager (hineln 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_Auldre_Grocery_Shoppe,l,n,n" <+
"Artikel.....Menge.....Preis,n" <+
concatMap artikel ps <+
"Summe" <+ formatR 31 (showEuro (kasse ew))

artikel :: Posten → String
artikel pg (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 (Gramm 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 =

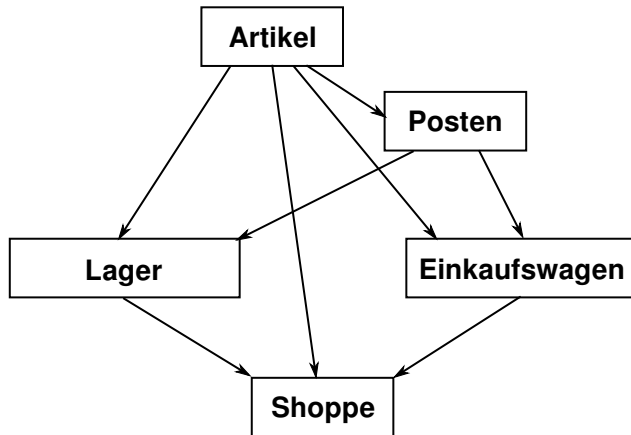
show (div i 100) <+ "." <+
show (mod (div i 10) 10) <+
show (mod i 10) <+ ",EU"

inventur :: Lager → Int
inventur (Lager l) = sum (map cent l)

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 Menge
              deriving (Eq, Show)
```

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

```
artikel :: Posten → Artikel
artikel (Posten a _) = a
```

- Konstruktor wird **nicht** exportiert
- 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 a (Lager l) = M.lookup a l
```

```
liste (Lager m) = M.toList m
```

```
inventur = sum ∘ map (fromJust ∘ uncurry preis) ∘ liste
```

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

Refakturierung im Einkaufsparadies IV: Einkaufswagen

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

- ▶ ADT durch **Verkapselung**:

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

- ▶ Ein Typsynonymym würde exportiert
- ▶ **Invariante**: Korrekte Menge zu Artikel im Einkaufswagen

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

- ▶ Nutzt dazu ADT `Posten`

Refakturierung im Einkaufsparadies V: Hauptmodul

```
module Shoppe where

import Artikel
import Lager
import Einkaufswagen
```

► Nutzt andere Module

```
w0= leererWagen
w1= einkauf (Apfel Boskoop) (Stueck 3) w0
w2= einkauf Schinken (Gramm 50) w1
w3= einkauf (Milch Bio) (Liter 1) w2
w4= einkauf Schinken (Gramm 50) w3
```

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

Beispiel

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

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	

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)	

Beispiel

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

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

Beispiel

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

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

Beispiel

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

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

Beispiel

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

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

Beispiel

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

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

Beispiel

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

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

Beispiel

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

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

Beispiel

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

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

Beispiel

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

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

Beispiel

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

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

Beispiel

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

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

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

Was zum Nachdenken

Übung 8.1: Import

Warum schreibt man

```
import Prelude hiding (repeat)
```

und was bewirkt das? (Hinweis: `Prelude` ist das Modul der vordefinierten Funktionen.)

Was zum Nachdenken

Übung 8.1: Import

Warum schreibt man

```
import Prelude hiding (repeat)
```

und was bewirkt das? (Hinweis: `Prelude` ist das Modul der vordefinierten Funktionen.)

Lösung: Die Import-Anweisung import alle vordefinierten Funktionen **bis auf** `repeat`.
Dadurch können wir `repeat` selber (anders) definieren.

II. Schnittstelle vs. Implementation

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 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**

Übung 8.2: Die Map als Assoziativliste

```
data Map  $\alpha$   $\beta$  = Map [( $\alpha$ ,  $\beta$ )]  
insert :: Ord  $\alpha$   $\Rightarrow$   $\alpha \rightarrow \beta \rightarrow$  Map  $\alpha$   $\beta \rightarrow$  Map  $\alpha$   $\beta$   
insert a b m = (a,b):m
```

Was ist der Nachteil dieser einfachen Implementation?

Übung 8.2: Die Map als Assoziativliste

```
data Map  $\alpha$   $\beta$  = Map [( $\alpha$ ,  $\beta$ )]  
insert :: Ord  $\alpha$   $\Rightarrow$   $\alpha \rightarrow \beta \rightarrow$  Map  $\alpha$   $\beta \rightarrow$  Map  $\alpha$   $\beta$   
insert a b m = (a,b):m
```

Was ist der Nachteil dieser einfachen Implementation?

Lösung: Erzeugt ein Speicherleck — überschriebene Elemente bleiben in der Liste. Besser: beim Einfügen alte Elemente entfernen

```
insert :: Ord  $\alpha$   $\Rightarrow$   $\alpha \rightarrow \beta \rightarrow$  Map  $\alpha$   $\beta \rightarrow$  Map  $\alpha$   $\beta$   
insert a b xs = (a, b): filter ((a  $\neq$  ).fst) xs
```

Nicht sehr effizient. Besser: **Map** als **sortierte** Liste.

Map als sortierte Assoziativliste

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

- ▶ Invariante: Liste ist in der ersten Komponente aufsteigend sortiert
- ▶ `lookup` ist vordefiniert; beim Einfügen auch überschreiben;

```
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
```

- ▶ ... 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)

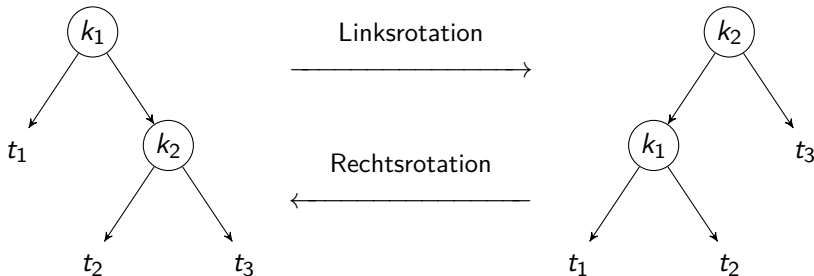
Implementation

- ▶ Balanciertheit ist **Invariante**
- ▶ Nach Einfügen oder Löschen: Balanciertheit wiederherstellen
- ▶ Dabei drei Fälle:
 - 1 Linker Unterbaum größer $size(l) > w \cdot size(r)$
 - 2 Rechter Unterbaum größer $size(r) > w \cdot \dots \cdot size(l)$
 - 3 Keiner größer — Baum balanciert

Balanciertheit durch Einfache Rotation

- ▶ Sei der rechte Unterbaum größer
- ▶ Zwei Unterfälle:
 - 1 Linkes Enkelkind t_2 größer
 - 2 Rechtes Enkelkind t_3 größer

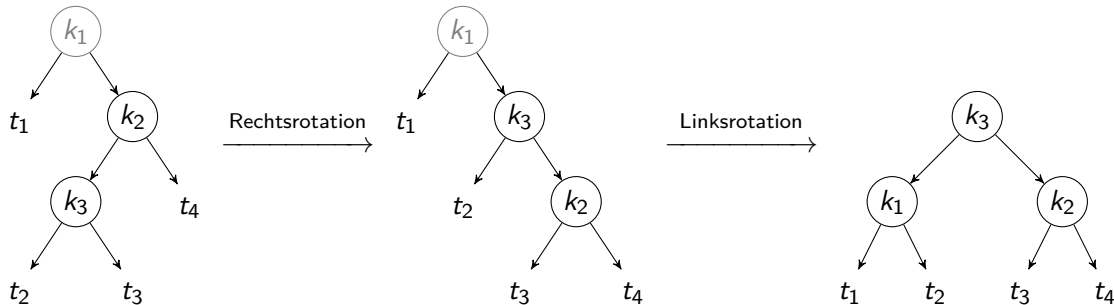
- ▶ Einfache **Linksrotation** heilt (2)
- ▶ Ansonsten: **Doppelrotation** reduziert (1) zu (2)



Balanciertheit durch Doppelrotation

Falls linkes Enkelkind um Faktor α größer als rechtes:

- ▶ Nach einer einfachen Rechtsrotation des Unterbaumes ist rechtes Enkelkind größer
- ▶ Danach Linksrotation des gesamten Baumes



Implementation in Haskell

► Der Datentyp

```
data Map  $\alpha$   $\beta$  = Empty  
              | Node  $\alpha$   $\beta$  Int (Map  $\alpha$   $\beta$ ) (Map  $\alpha$   $\beta$ )  
              deriving Eq
```

► Parameter:

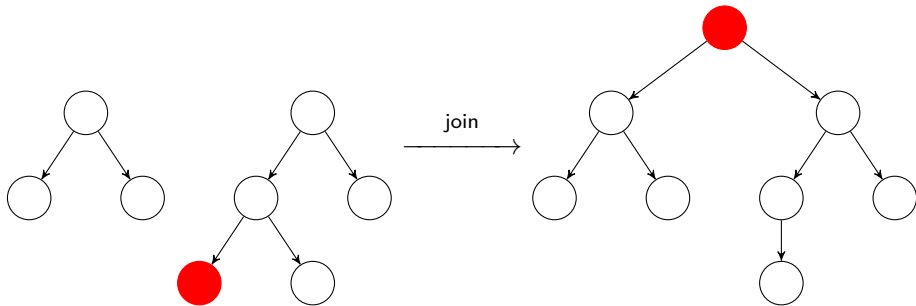
- `weight` Gewichtungsfaktor w (für Einfachrotation)
- `ratio` Gewichtungsfaktor α (für Doppelrotation)
- Hilfskonstruktor `node`, setzt Größe (`l`, `r` balanciert)
- Selektor `size` für Größe des Baumes (0 für `Empty`)

Hauptfunktion

- ▶ `balance k x l r` konstruiert balancierten Baum
 - ▶ `l`, `r` sind balanciert und höchstens um einen Knoten unbalanciert
 - ▶ Vier Fälle:
 - ① Beide Bäume zusammen höchstens einen Knoten \rightarrow keine Rotation
 - ② $w \cdot \text{size}(l) < \text{size}(r)$: \rightarrow Linksrotation
 - ③ $\text{size}(l) > w \cdot \text{size}(r)$: \rightarrow Rechtsrotation
 - ④ Ansonsten: keine Rotation
- ▶ `balanceL k x l r` rotiert nach links. Sei r_l und r_r rechter und linker Unterbaum von `r`:
 - ① $\text{size}(r_l) < \alpha \cdot \text{size}(r_r)$, dann einfache Linksrotation
 - ② $\text{size}(r_l) \geq \alpha \cdot \text{size}(r_r)$ dann Doppelrotation (Rechtsrotation `r`, dann Linksrotation)

Hilfsfunktion join beim Löschen

- ▶ Zwei balancierte Bäume zusammenfügen (nachdem Wurzel gelöscht wurde)
- ▶ Linkester Knoten des rechten Unterbaumes wird neue Wurzel
- ▶ Mit `balance` wieder ausbalancieren



Was zum Selbermachen

Übung 8.3: Use the Source, Luke!

Ladet euch von der Webseite der Veranstaltung die Quellen für die 8. Vorlesung herunter, und öffnet die Datei `MapTree.hs`.

Vergleicht die Haskell-Implementation mit den Beschreibung der Folien.

Welche der Funktionen `lookup`, `insert`, `delete` könnte man als `fold` realisieren?

Was zum Selbermachen

Übung 8.3: Use the Source, Luke!

Ladet euch von der Webseite der Veranstaltung die Quellen für die 8. Vorlesung herunter, und öffnet die Datei `MapTree.hs`.

Vergleicht die Haskell-Implementation mit den Beschreibung der Folien.

Welche der Funktionen `lookup`, `insert`, `delete` könnte man als `fold` realisieren?

Lösung: `lookup` läßt sich falten:

```
lookup' k = fold ( $\lambda$ ak ax l r  $\rightarrow$  if k == ak then Just ax  
                  else maybe r Just l) Nothing
```

Ist aber nicht so effizient (linear statt logarithmisch), weil es immer erst links, dann rechts sucht.

Zusammenfassung Balancierte Bäume

- ▶ Auslesen, einfügen und löschen: logarithmischer Aufwand ($\mathcal{O}(\log n)$)
- ▶ Fold: linearer Aufwand ($\mathcal{O}(n)$)
- ▶ Guten durchschnittlicher Aufwand
- ▶ Auch in der Haskell-Bücherei: [Data.Map](#) (schwer optimiert, 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
MapFun	333,3 ms 13,58 ms	1,634 ms 52,25 μ s	11,27 ns 130,8 ps	11,20 ns 120,3 ps	1,659 ms 79,22 μ s
MapList	5,629 s 168,7 ms	32,70 μ s 9,625 μ s	96,12 μ s 1,294 μ s	101,4 μ s 18,47 μ s	6,182 ms 2,059 μ s
MapTree	383,9 ms 19,62 ms	404,1 ns 135,3 ns	119,4 μ s 13,18 μ s	117,1 μ s 42,82 μ s	2,803 ms 521,5 μ s
Data.Map.Lazy	473,0 ms 44,97 ms	221,6 ns 59,58 ns	104,7 μ s 49,66 μ s	112,7 μ s 11,39 μ s	2,396 ms 278,8 μ s

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**

Zusammenfassung

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

A photograph of a snowy forest. In the foreground, there are two large evergreen trees with snow-laden branches. The ground is covered in a thick layer of snow. In the background, a person wearing a red jacket is visible, standing in the snow. The overall atmosphere is quiet and wintry.

Frohe Weihnachten und einen Guten Rutsch!