

Deklarative (= fortgeschrittene) Programmierung

Vorlesung WS 2009, WS 2010, SS 2012

Johannes Waldmann, HTWK Leipzig

25. Juni 2012

1 Einleitung

Formen der deklarative Programmierung

- funktionale Programmierung: `foldr (+) 0 [1,2,3]`
`foldr f z l = case l of`
 `[] -> z ; (x:xs) -> f x (foldr f z xs)`
- logische Programmierung: `append(A,B,[1,2,3])`.
`append([],YS,YS)`.
`append([X|XS],YS,[X|ZS]) :- append(XS,YS,ZS)`.
- Constraint-Programmierung
`(set-logic QF_LIA) (set-option :produce-models true)`
`(declare-fun a () Int) (declare-fun b () Int)`
`(assert (and (>= a 5) (<= b 30) (= (+ a b) 20)))`
`(check-sat) (get-value (a b))`

Definition

deklarativ: jedes (Teil-)Programm/Ausdruck hat einen *Wert*
(... und keine weitere (versteckte) *Wirkung*).

Werte können sein:

- “klassische” Daten (Zahlen, Listen, Bäume...)
- Funktionen (Sinus, ...)
- Aktionen (Datei schreiben, ...)

Softwaretechnische Vorteile

... der deklarativen Programmierung

- Beweisbarkeit: Rechnen mit Programmen wie in der Mathematik mit Termen
- Sicherheit: es gibt keine Nebenwirkungen und Wirkungen sieht man bereits am Typ
- Wiederverwendbarkeit: durch Entwurfsmuster (= Funktionen höherer Ordnung)
- Effizienz: durch Programmtransformationen im Compiler,
- Parallelisierbarkeit: durch Nebenwirkungsfreiheit

Beispiel Spezifikation/Test

```
import Test.SmallCheck

append :: forall t . [t] -> [t] -> [t]
append x y = case x of
  [] -> y
  h : t -> h : append t y

associative f =
  \ x y z -> f x (f y z) == f (f x y) z

test1 = smallCheckI
  (associative (append :: [Int]->[Int]->[Int]))
```

Übung: Kommutativität (formulieren und testen)

Beispiel Verifikation

```
app x y = case x of
  [] -> y
  h : t -> h : app t y
```

Beweise

```
app x (app y z) == app (app x y) z
```

Beweismethode: Induktion nach x.

- Induktionsanfang: $x == [] \dots$
- Induktionsschritt: $x == h : t \dots$

Beispiel Parallelisierung

Klassische Implementierung von Mergesort

```
sort :: Ord a => [a] -> [a]
sort [] = [] ; sort [x] = [x]
sort xs = let ( left,right ) = split xs
            sleft  = sort left
            sright = sort right
            in merge sleft sright
```

wird parallelisiert durch *Annotationen*:

```
sleft  = sort left
        `using` rpar `dot` spineList
sright = sort right `using` spineList
```

vgl. <http://thread.gmane.org/gmane.comp.lang.haskell.parallel/181/focus=202>

Deklarative Programmierung in der Lehre

- funktionale Programmierung: diese Vorlesung
- logische Programmierung: in *Angew. Künstl. Intell.*
- Constraint-Programmierung: als Master-Wahlfach

Beziehungen zu weiteren LV: Voraussetzungen

- Bäume, Terme (Alg.+DS, Grundlagen Theor. Inf.)
- Logik (Grundlagen TI, Softwaretechnik)

Anwendungen:

- Softwarepraktikum
- weitere Sprachkonzepte in *Prinzipien v. Programmiersprachen*
- *Programmverifikation* (vorw. f. imperative Programme)

Konzepte und Sprachen

Funktionale Programmierung ist ein *Konzept*. Realisierungen:

- in prozeduralen Sprachen:
 - Unterprogramme als Argumente (in Pascal)
 - Funktionszeiger (in C)
- in OO-Sprachen: Befehlsobjekte
- Multi-Paradigmen-Sprachen:
 - Lambda-Ausdrücke in C#, Scala, Clojure
- funktionale Programmiersprachen (LISP, ML, Haskell)

Die Erkenntnisse sind sprachunabhängig.

- A good programmer can write LISP in any language.
- Learn Haskell and become a better Java programmer.

Gliederung der Vorlesung

- algebraische Datentypen, Pattern Matching, Termersetzungssysteme
- Funktionen (polymorph, höherer Ordnung), Lambda-Kalkül, Rekursionsmuster
- Typklassen zur Steuerung der Polymorphie
- Bedarfsauswertung, unendl. Datenstrukturen
- Programmstrukturierung durch Monaden
- Modelle für Nebenläufigkeit und Parallelität

Softwaretechnische Aspekte

- algebraische Datentypen, Pattern Matching, Termersetzungssysteme
Entwurfsmuster Kompositum, immutable objects,
das Datenmodell von Git
- Funktionen (höherer Ordnung), Lambda-Kalkül, Rekursionsmuster
Lambda-Ausdrücke in C#, Entwurfsmuster Besucher
Codequalität, code smells, Refaktorisierung

- Typklassen zur Steuerung der Polymorphie
Interfaces in Java/C# , automatische Testfallgenerierung
- Bedarfsauswertung, unendl. Datenstrukturen
Iteratoren, Ströme
- Programmstrukturierung durch Monaden
Entwurfsmuster Zustand, LINQ in C#

Organisation der LV

- jede Woche eine Vorlesung, eine Übung
- Hausaufgaben (teilw. autotool)
- Prüfung: Klausur (ohne Hilfsmittel)

Literatur

- Skripte:
 - Deklarative Programmierung WS10 <http://www.imn.htwk-leipzig.de/~waldmann/edu/ws10/dp/folien/main/>
 - Softwaretechnik II SS11 <http://www.imn.htwk-leipzig.de/~waldmann/edu/ss11/st2/folien/main/>
- Entwurfsmuster: <http://www.imn.htwk-leipzig.de/~waldmann/draft/pub/hal4/emu/>
- Maurice Naftalin und Phil Wadler: *Java Generics and Collections*, O'Reilly 2006
- <http://haskell.org/> (Sprache, Werkzeuge, Tutorials), <http://www.realworldhaskell.org/>

Übungen KW11

- im Pool Z423
 - Ubuntu-11.04 booten (nicht 10.10)
 - `export PATH=/usr/local/waldmann/bin:$PATH`
- Beispiele f. deklarative Programmierung
 - funktional: Haskell mit ghci,

- logisch: Prolog mit swipl,
- constraint: mit mathsat, z3
- Haskell-Entwicklungswerkzeuge (eclipsefp, leksah, ..., aber *real programmers* ... <http://xkcd.org/378/>), API-Suchmaschine <http://www.haskell.org/hoogle/>
- Auswertung Klausur Softwaretechnik

2 Daten

Wiederholung: Terme

- (Prädikatenlogik) ein Term t in Signatur Σ ist
 - Funktionssymbol $f \in \Sigma$ der Stelligkeit k mit Argumenten (t_1, \dots, t_k) , die selbst Terme sind.

(wo ist dabei der Induktionsanfang?)
- (Graphentheorie) ein Term ist ein gerichteter, geordneter, markierter Baum
- (Datenstrukturen)
 - Funktionssymbol = Konstruktor, Term = Baum

Algebraische Datentypen

```
data Foo = Foo { bar :: Int, baz :: String }
  deriving Show
```

Bezeichnungen (benannte Notation)

- `data Foo` ist Typname
- `Foo { .. }` ist Konstruktor
- `bar`, `baz` sind Komponenten

```
x :: Foo
x = Foo { bar = 3, baz = "hal" }
```

Bezeichnungen (positionelle Notation)

```
data Foo = Foo Int String
y = Foo 3 "bar"
```

Datentyp mit mehreren Konstruktoren

Beispiel (selbst definiert)

```
data T = A { foo :: Int }
        | B { bar :: String, baz :: Bool }
    deriving Show
```

Beispiele (in Prelude vordefiniert)

```
data Bool = False | True
data Ordering = LT | EQ | GT
```

Rekursive Datentypen

```
data Tree = Leaf {}
          | Branch { left :: Tree
                    , right :: Tree }
```

Übung: Objekte dieses Typs erzeugen
(benannte und positionelle Notation der Konstruktoren)

Polymorphe Typen

Container-Datentypen sollten *generisch polymorph* im Inhaltstyp sein

```
data Tree a = Leaf {}
            | Branch { left :: Tree a, key :: a
                      , right :: Tree a }
data List a = Nil {}
            | Cons { head :: a, tail :: List a }
```

(Kleinbuchstabe = Typvariable, implizit all-quantifiziert)

Übung: Konstruiere Objekte dieser Typen (aus Prelude):

```
Maybe a ; Either a b ; (a, b)
```

Termersetzungssysteme

- Programm $R =$ Menge von Regeln
- Regel = Paar (l, r) von Termen mit Variablen
- Relation \rightarrow_R ist Menge (t, t') mit
 - es existiert Position p in t
 - es existiert Substitution $\sigma : \text{Var}(l) \rightarrow \text{Term}(\Sigma)$
 - es existiert $(l, r) \in R$
 - so daß $t|_p = l\sigma$ und $t' = t[p := r\sigma]$.

Hilfsbegriffe:

- Position: Folge von natürlichen Zahlen
- $t|_p$: der Teilterm von t an Position p
- $t[p := s]$: wie t , aber mit Term s an Position p
- Substitution σ : Abbildung $\text{Var} \rightarrow \text{Term}(\Sigma)$
- $l\sigma$: wie l , aber statt v immer $\sigma(v)$

Funktionale Programme

... sind spezielle Term-Ersetzungssysteme. Beispiel:

Signatur: S einstellig, Z nullstellig, f zweistellig.

Ersetzungssystem $\{f(Z, y) \rightarrow y, f(S(x), y) \rightarrow S(f(x, y))\}$.

Startterm $f(S(S(Z)), S(Z))$.

entsprechendes funktionales Programm:

```
data N = S N | Z
f :: N -> N -> N
f a y = case a of
  Z    -> y
  S x  -> S (f x y)
```

Aufruf: $f (S (S Z)) (S Z)$

Auswertung = Folge von Ersetzungsschritten \rightarrow_R^* Resultat = Normalform (hat keine \rightarrow_R -Nachfolger)

data und case

typisches Vorgehen beim Programmieren einer Funktion

```
f :: T -> ...
```

- Für jeden Konstruktor des Datentyps

```
data T = C1 ...
      | C2 ...
```

- schreibe einen Zweig in der Fallunterscheidung

```
f x = case x of
      C1 ... -> ...
      C2 ... -> ...
```

Rekursive Datentypen und Funktionen

Wenn der Datentyp rekursiv ist, dann auch die Funktion, die ihn verarbeitet:

```
data Tree a = Leaf {}
            | Branch { left :: Tree a, key :: a
                      , right :: Tree a }
leaves :: Tree a -> Int
leaves t = case t of
  Leaf {} -> 1
  Branch {} -> ...
```

Aufgabe: erzeuge und bestimme Blatt-Anzahl für:

- vollständige Binärbäume
- maximal unbalancierte AVL-Bäume

Peano-Zahlen

```
data N = Z | S N
plus :: N -> N -> N
plus x y = case x of
  Z -> y
  S x' -> S (plus x' y)
```

Aufgaben:

- implementiere Multiplikation, Potenz
- beweise die üblichen Eigenschaften (Addition, Multiplikation sind assoziativ, kommutativ, besitzen neutrale Element)

Wiederholung Bäume

```
data Tree a = Leaf {}
            | Branch { left :: Tree a
                      , key :: a
                      , right :: Tree a }
branches :: Tree a -> Int
branches t = case t of
  Leaf {} -> 0
  Branch {} ->
    branches (left t) + 1 + branches (right t)
```

Zusammenhang:

Datentyp	Funktion
zwei Konstruktoren	zwei Zweige
rekursiv (Tree → Tree)	rekursiv (branches → branches)

Listen

eigentlich:

```
data List a = Nil {}
            | Cons { head :: a, tail :: List a }
```

wegen früher häufiger Benutzung verkürzte Schreibweise: List a = [a], Nil = [], Cons = (:)

```
data [a] = []
         | (:) { head :: a, tail :: [a] }
```

Pattern matching für Listen:

```
append :: [a] -> [a] -> [a]
append a b = case a of
  [] -> b
  x : xs -> ...
```

Operationen auf Listen

- append:
 - Definition

– Beweis Assoziativität, neutrales Element

• reverse:

– Definition

– Beweis: `reverse (reverse xs) == xs`

– benutze Funktion mit dieser Spezifikation

```
arev :: [a] -> [a] -> [a]
arev xs ys == append (reverse xs) ys
```

zur effizienteren Implementierung von reverse

3 Algebraische Datentypen in OOP

Strukturmuster: Kompositum

Finde wenigstens sieben (Entwurfs-)Fehler und ihre wahrscheinlichen Auswirkungen...

```
class Geo {
  int type; // Kreis, Quadrat,
  Geo teil1, teil2; // falls Teilobjekte
  int ul, ur, ol, or; // unten links, ...
  void draw () {
    if (type == 0) { ... } // Kreis
    else if (type == 1) { ... } // Quadrat
  }
}
```

Kompositum - Definition

so ist es richtig:

```
interface Geo { }
class Kreis implements Geo {
  double radius;          }
class Neben implements Geo {
  Neben (Geo links, Geo rechts) { .. } }
```

entspricht dem algebraischen Datentyp

```
data Geo
  = Kreis { radius :: Double }
  | Neben { links :: Geo, rechts :: Geo }
```

Merke:

Haskell (Alg. DT)	Typ	Konstruktor
OO (Kompositum)	Interface	Klasse

Kompositum - Beispiel

Gestaltung von zusammengesetzten Layouts:

- Container extends Component
- Container { add (Component c); }

Modellierung als algebraischer Datentyp ist:

```
data Component
  = ...
  | Container [ Component ]
```

Kompositum - Beispiel

```
public class Composite {
  public static void main(String[] args) {
    JFrame f = new JFrame ("Composite");
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container c = new JPanel (new BorderLayout());
    c.add (new JButton ("foo"), BorderLayout.CENTER);
    f.getContentPane().add(c);
    f.pack(); f.setVisible(true);
  }
}
```

Übung: geschachtelte Layouts bauen, vgl. <http://www.imn.htwk-leipzig.de/~waldmann/edu/ws06/informatik/manage/>

Kompositum - Verarbeitung

- Anzahl aller Kreise der Figur (Ü: Summe aller Kreisflächen)

```
interface Geo {  
    int circles ();  
}
```

und Implementierungen in den jeweiligen Klassen.

- vergleiche: in Haskell

```
data Geo = ...  
circles :: Geo -> Int  
circles g = case g of ...
```

- Vorteile/Nachteile?

Binäre Bäume als Komposita

- Knoten sind *innere* (Verzweigung) und *äußere* (Blatt).
- Die richtige Realisierung ist Kompositum

```
interface Tree<K>;  
class Branch<K> implements Tree<K>;  
class Leaf<K> implements Tree<K>;
```

- Schlüssel: in allen Knoten, nur innen, nur außen.

der entsprechende algebraische Datentyp ist:

```
data Tree k = Leaf { ... }  
| Branch { left :: Tree k , ...  
          , right :: Tree k }
```

Übung: Anzahl aller Blätter, Summe aller Schlüssel (Typ?), der größte Schlüssel (Typ?)

Kompositum-Vermeidung

Wenn Blätter keine Schlüssel haben, geht es musterfrei?

```
class Tree<K> {  
    Tree<K> left; K key; Tree<K> right;  
}
```

Der entsprechende algebraische Datentyp ist

```
data Tree k =  
  Tree { left :: Maybe (Tree k)  
        , key :: k  
        , right :: Maybe (Tree k)  
        }
```

erzeugt in Java das Problem, daß ...

Übung: betrachte Implementierung in `java.util.Map<K,V>`

Maybe = Nullable

Algebraischer Datentyp (Haskell):

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

<http://hackage.haskell.org/packages/archive/base/latest/doc/html/Prelude.html#t:Maybe>

In Sprachen mit Verweisen (auf Objekte vom Typ `O`) gibt es häufig auch „Verweis auf kein Objekt“— auch vom Typ `O`. Deswegen *null pointer exceptions*.

Ursache ist Verwechslung von `Maybe a` mit `a`.

Trennung in C#: `Nullable<T>` (für primitive Typen `T`)

<http://msdn.microsoft.com/en-us/library/2cf62fcy.aspx>

Alg. DT und Pattern Matching in Scala

<http://scala-lang.org>

algebraische Datentypen:

```
abstract class Tree[A]  
case class Leaf[A](key: A) extends Tree[A]  
case class Branch[A]  
  (left: Tree[A], right: Tree[A])  
  extends Tree[A]
```

pattern matching:

```
def size[A](t: Tree[A]): Int = t match {  
  case Leaf(k) => 1  
  case Branch(l, r) => size(l) + size(r)  
}
```

beachte: Typparameter in eckigen Klammern

4 Funktionen

Funktionen als Daten

bisher:

```
f :: Int -> Int
f x = 2 * x + 5
```

äquivalent: Lambda-Ausdruck

```
f = \ x -> 2 * x + 5
```

Lambda-Kalkül: Alonzo Church 1936, Henk Barendregt 198*, ...

Funktionsanwendung:

$$(\lambda x \rightarrow A) B = A [x := B]$$

... falls x nicht (frei) in B vorkommt

Der Lambda-Kalkül

... als weiteres Berechnungsmodell,

(vgl. Termersetzungssysteme, Turingmaschine, Random-Access-Maschine)

Syntax: die Menge der Lambda-Terme Λ ist

- jede Variable ist ein Term: $v \in V \Rightarrow v \in \Lambda$
- Funktionsanwendung (Applikation):
 $F \in \Lambda, A \in \Lambda \Rightarrow (FA) \in \Lambda$
- Funktionsdefinition (Abstraktion):
 $v \in V, B \in \Lambda \Rightarrow (\lambda v. B) \in \Lambda$

Semantik: eine Relation \rightarrow_β auf Λ

(vgl. \rightarrow_R für Termersetzungssystem R)

Lambda-Terme: verkürzte Notation

- Applikation als links-assoziativ auffassen, Klammern weglassen:

$$(\dots((FA_1)A_2)\dots A_n) \sim FA_1A_2\dots A_n$$

Beispiel: $((xz)(yz)) \sim xz(yz)$

- geschachtelte Abstraktionen unter ein Lambda schreiben:

$$\lambda x_1.(\lambda x_2. \dots (\lambda x_n.B) \dots) \sim \lambda x_1 x_2 \dots x_n. B$$

Beispiel: $\lambda x. \lambda y. \lambda z. B \sim \lambda x y z. B$

Freie und gebundene Variablen(vorkommen)

- Das Vorkommen von $v \in V$ an Position p in Term t heißt *frei*, wenn „darüber kein $\lambda v. \dots$ steht“
- Def. $\text{fvar}(t)$ = Menge der in t frei vorkommenden Variablen
- Def. $\text{bvar}(t)$ = Menge der in t gebundenen Variablen

Semantik des Lambda-Kalküls

Relation \rightarrow_β auf Λ (ein Reduktionsschritt)

- Reduktion $(\lambda x.B)A \rightarrow_\beta B[x := A]$ (Vorsicht)
Kopie von B , jedes freie Vorkommen von x durch A ersetzt
- ... eines beliebigen Teilterms:
 - $F \rightarrow_\beta F' \Rightarrow (FA) \rightarrow_\beta (F'A)$
 - $A \rightarrow_\beta A' \Rightarrow (FA) \rightarrow_\beta (FA')$
 - $B \rightarrow_\beta B' \Rightarrow \lambda x.B \rightarrow_\beta \lambda x.B'$

$$(\lambda x.(\lambda y. x y x))(y y) \stackrel{?}{\rightarrow}_\beta (\lambda y. y x)[x := (y y)] = \lambda y. y(y y)$$

das freie y wird fälschlich gebunden. Lösung: vorher umbenennen (später (bei Refactoring) genaueres dazu)

Ein- und mehrstellige Funktionen

eine einstellige Funktion zweiter Ordnung:

$$f = \lambda x \rightarrow (\lambda y \rightarrow (x * x + y * y))$$

Anwendung dieser Funktion:

$$(f \ 3) \ 4 = \dots$$

Kurzschreibweisen (Klammern weglassen):

```
f = \ x y -> x * x + y * y ; f 3 4
```

Übung:

```
gegeben t = \ f x -> f (f x)
```

```
bestimme t succ 0, t t succ 0, t t t succ 0, t t t t succ 0, ...
```

Typen

für nicht polymorphe Typen: tatsächlicher Argumenttyp muß mit deklariertem Argumenttyp übereinstimmen:

wenn $f :: A \rightarrow B$ und $x :: A$, dann $(fx) :: B$.

bei polymorphen Typen können der Typ von $f :: A \rightarrow B$ und der Typ von $x :: A'$ Typvariablen enthalten.

Dann müssen A und A' nicht übereinstimmen, sondern nur *unifizierbar* sein (eine gemeinsame Instanz besitzen).

$\sigma := \text{mgu}(A, A')$ (allgemeinster Unifikator)

allgemeinster Typ von (fx) ist dann $B\sigma$.

Typ von x wird dadurch spezialisiert auf $A'\sigma$

Bestimme allgemeinsten Typ von $t = \lambda fx.f(fx)$, von (tt) .

Rekursion über Bäume (Beispiele)

```
data Tree a = Leaf
  | Branch { left :: Tree a, key :: a, right :: Tree a }
```

```
summe :: Tree Int -> Int
```

```
summe t = case t of
```

```
  Leaf {} -> 0
```

```
  Branch {} ->
```

```
    summe (left t) + key t + summe (right t)
```

```
preorder :: Tree a -> [a]
```

```
preorder t = case t of
```

```
  Leaf {} -> []
```

```
  Branch {} ->
```

```
    key t : preorder (left t) ++ preorder (right t)
```

Rekursion über Bäume (Schema)

```
f :: Tree a -> b
f t = case t of
  Leaf {} -> ...
  Branch {} ->
    ... (f (left t)) (key t) (f (right t))
```

dieses Schema *ist* eine Funktion höherer Ordnung:

```
fold :: ( ... ) -> ( ... ) -> ( Tree a -> b )
fold leaf branch = \ t -> case t of
  Leaf {} -> leaf
  Branch {} ->
    branch (fold leaf branch (left t))
            (key t) (fold leaf branch (right t))
summe = fold 0 ( \ l k r -> l + k + r )
```

Rekursion über Listen

```
and :: [ Bool ] -> Bool
and xs = case xs of
  [] -> True ; x : xs' -> x && and xs'
length :: [ a ] -> Int
length xs = case xs of
  [] -> 0 ; x : xs' -> 1 + length xs'

fold :: b -> ( a -> b -> b ) -> [a] -> b
fold nil cons xs = case xs of
  [] -> nil
  x : xs' -> cons x ( fold nil cons xs' )
and = fold True (&&)
length = fold 0 ( \ x y -> 1 + y )
```

Rekursionsmuster (Prinzip)

jeden Konstruktor durch eine passende Funktion ersetzen.

```
data List a = Nil | Cons a (List a)
fold ( nil :: b ) ( cons :: a -> b -> b )
  :: List a -> b
```

Rekursionsmuster instantiieren = (Konstruktor-)Symbole interpretieren (durch Funktionen) = eine Algebra angeben.

```
length = fold 0 ( \ _ l -> l + 1 )
reverse = fold [] ( \ x ys ->          )
```

Rekursion über Listen (Übung)

das vordefinierte Rekursionsschema über Listen ist:

```
foldr :: (a -> b -> b) -> b -> ([a] -> b)
```

```
length = foldr ( \ x y -> 1 + y ) 0
```

Beachte:

- Argument-Reihenfolge (erst cons, dann nil)
- foldr nicht mit foldl verwechseln (foldr ist das „richtige“)

Aufgaben:

- append, reverse, concat, inits, tails mit foldr (d. h., ohne Rekursion)

Weitere Beispiele für Folds

```
data Tree a
  = Leaf { key :: a }
  | Branch { left :: Tree a, right :: Tree a }
```

```
fold :: ...
```

- Anzahl der Blätter
- Anzahl der Verzweigungsknoten
- Summe der Schlüssel
- die Tiefe des Baumes
- der größte Schlüssel

Strukturerhaltende Folds

elementweise Operation:

Argument und Resultat haben gleiche Struktur, aber (mglw.) verschiedene Elemente:

```
map :: (a -> b) -> (Tree a -> Tree b)
map f = fold Leaf
      ( \ l k r -> Branch l (f k) r )
```

```
map :: (a -> b) -> ([a] -> [b])
map f = foldr ( \ x ys -> f x : ys ) []
```

```
map length [ "foo", "bar" ] = [ 3, 3 ]
```

Ü: Unterschiede zw. `map reverse` und `reverse`

Programmtransformationen

Komposition von Funktionen:

```
(f . g) = \ x -> f (g x)
```

Ü: Typ von `(.)`. Bem: Notation leider falschherum.

Satz: (wenn `map` „richtig“ definiert ist, gilt:)

- `map id == id`
- `map (f . g) == map f . map g`

Anwendung: Einsparung von Zwischen-Strukturen.

Programmtransformationen (II)

Satz: (wenn `map` „richtig“ definiert ist, gilt:)

- `foldr nil cons . map f == foldr`
- desgl. für Bäume

5 Objektorientierte Rekursionsmuster

Plan

- algebraischer Datentyp = Kompositum
(Typ \Rightarrow Interface, Konstruktor \Rightarrow Klasse)
- Rekursionsschema = Besucher (Visitor)
(Realisierung der Fallunterscheidung)

(Zum Vergleich von Java- und Haskell-Programmierung)

sagte bereits Albert Einstein: *Das Holzhacken ist deswegen so beliebt, weil man den Erfolg sofort sieht.*

Wiederholung Rekursionsschema

fold anwenden: jeden Konstruktor d. Funktion ersetzen

- Konstruktor \Rightarrow Schema-Argument
- ... mit gleicher Stelligkeit
- Rekursion im Typ \Rightarrow Anwendung auf Schema-Resultat

```
data List a = Nil | Cons a (List a)
  Nil      :: List a
  Cons    :: a -> List a -> List a
fold     :: b -> (a -> b -> b) -> List a -> b
fold nil cons l = case l of
  Nil -> nil
  Cons x xs -> cons x (fold nil cons xs)
length = fold 0 ( \ x y -> 1 + y )
```

Wiederholung: Kompositum

Haskell: algebraischer Datentyp

```
data List a = Nil | Cons a (List a)
  Nil      :: List a
  Cons    :: a -> List a -> List a
```

Java: Kompositum

```

interface List<A> { }
class Nil implements List<A> { }
class Cons<A> implements List<A> {
    A head; List<A> tail;
}

```

Scala: case class

Übung Kompositum

implementieren/testen

(nicht `java.util.List`, sondern der selbstgebaute Typ)

```

public class Main {
    static List<Integer> range(from,to) { ..
    }
    static List<Integer> make(Integer ... xs) {
    }
    public static void main(String[] args) {
        List<Integer> l = Main.range(0,10);
        List<Integer> m = Main.make(3,1,4,1,5,9);
    }
}

```

Kompositum und Visitor

Definition eines Besucher-Objektes (für Rekursionsmuster mit Resultattyp `R` über `List<A>`)
entspricht einem Tupel von Funktionen

```

interface Visitor<A,R> {
    R nil();
    R cons(A elem, R result); }

```

Empfangen eines Besuchers: durch jeden Teilnehmer des Kompositums

```

interface List<A> { ..
    <R> R receive (Visitor<A,R> v); }

```

- Implementierung
- Anwendung (length, sum)

Aufgabe: Besucher für Listen

schreibe Methoden für

- Produkt
- Minimum, Maximum
- Wert als Binärzahl, Bsp: $(1, 1, 0, 1) \implies 13$

Aufgabe: Besucher für Bäume

```
data Tree k = Leaf { key  :: k }
             | Branch { left :: Tree k
                       , right :: Tree k }
```

- das entsprechende Kompositum
(alle Attribute `final`, Konstruktoren, `toString()`)
- Methode zum Erzeugen von Testdaten
`static Tree<Integer> full (int h);`
- das passende Besucher-Muster
- Anwendung: Zählen von Blättern usw.

6 Unveränderliche Objekte

Überblick

- alle Attribute aller Objekte sind unveränderlich (`final`)
- anstatt Objekt zu ändern, konstruiert man ein neues

Eigenschaften des Programmierstils:

- vereinfacht Formulierung und Beweis von Objekteigenschaften
- parallelisierbar (keine updates, keine *data races*)
<http://fpcomplete.com/the-downfall-of-imperative-programming/>
- Persistenz (Verfügbarkeit früherer Versionen)
- Belastung des Garbage Collectors (... dafür ist er da)

Beispiel: Einfügen in Baum

- destruktiv:

```
interface Tree<K> { void insert (K key); }
Tree<String> t = ... ;
t.insert ("foo");
```

- persistent (Java):

```
interface Tree<K> { Tree<K> insert (K key); }
Tree<String> t = ... ;
Tree<String> u = t.insert ("foo");
```

- persistent (Haskell):

```
insert :: Tree k -> k -> Tree k
```

Beispiel: (unbalancierter) Suchbaum

```
data Tree k = Leaf
            | Branch (Tree k) k (Tree k)
insert :: Ord k => k -> Tree k -> Tree k
insert k t = case t of ...
```

Diskussion:

- Ord k entspricht K implements Comparable<K>, genaueres später (Haskell-Typklassen)
- wie teuer ist die Persistenz?
(wieviel Müll entsteht bei einem insert?)

Beispiel: Sortieren mit Suchbäumen

```
data Tree k = Leaf
            | Branch (Tree k) k (Tree k)

insert :: Ord k => k -> Tree k -> Tree k

build :: Ord k => [k] -> Tree k
build = foldr ... ...

sort :: Ord k => [k] -> [k]
sort xs = ... ( ... xs )
```

Persistente Objekte in Git

<http://git-scm.com/>

- *Distributed* development.
- Strong support for *non-linear* development.
(Branching and merging are fast and easy.)
- Efficient handling of *large* projects.
(z. B. Linux-Kernel, <http://kernel.org/>)
- Toolkit design.
- Cryptographic authentication of history.

Objekt-Versionierung in Git

- Objekt-Typen:
 - Datei (blob),
 - Verzeichnis (tree), mit Verweisen auf blobs und trees
 - Commit, mit Verweisen auf tree und commits (Vorgänger)
- ```
git cat-file [-t|-p] <hash> git ls-tree [-t|-p] <hash>
```
- Objekte sind *unveränderlich* und durch SHA1-Hash (160 bit = 40 Hex-Zeichen) identifiziert

- statt Überschreiben: neue Objekte anlegen
- jeder Zustand ist durch Commit-Hash (weltweit) eindeutig beschrieben und kann wiederhergestellt werden

## 7 Polymorphie/Typklassen

### Einleitung

```
reverse [1,2,3,4] = [4,3,2,1]
reverse "foobar" = "raboof"
reverse :: [a] -> [a]
```

reverse ist polymorph

```
sort [5,1,4,3] = [1,3,4,5]
sort "foobar" = "abfoor"
```

```
sort :: [a] -> [a] -- ??
sort [sin,cos,log] = ??
```

sort ist *eingeschränkt polymorph*

### Der Typ von sort

zur Erinnerung: sort = inorder . foldr insert Leaf mit

```
insert x t = case t of
 Branch {} -> if x < key t then ...
```

Für alle a, die für die es eine Vergleichs-Funktion gibt, hat sort den Typ [a] -> [a].

```
sort :: Ord a => [a] -> [a]
```

Hier ist Ord eine *Typklasse*, so definiert:

```
class Ord a where
 compare :: a -> a -> Ordering
data Ordering = LT | EQ | GT
```

vgl. Java:

```
interface Comparable<T> { int compareTo (T o); }
```

## Instanzen

Typen können Instanzen von *Typklassen* sein.

(OO-Sprech: Klassen implementieren Interfaces)

Für vordefinierte Typen sind auch die meisten sinnvollen Instanzen vordefiniert

```
instance Ord Int ; instance Ord Char ; ...
```

weiter Instanzen kann man selbst deklarieren:

```
data Student = Student { vorname :: String
 , nachname :: String
 , matrikel :: Int
 }
instance Ord Student where
 compare s t =
 compare (matrikel s) (matrikel t)
```

## Typen und Typklassen

In Haskell sind diese drei Dinge *unabhängig*

1. Deklaration einer Typklasse (= Deklaration von abstrakten Methoden) `class C where { m :: ... }`
2. Deklaration eines Typs (= Sammlung von Konstruktoren und konkreten Methoden)  
`data T = ...`
3. Instanz-Deklaration (= Implementierung der abstrakten Methoden) `instance C T where { m = ... }`

In Java sind 2 und 3 nur *gemeinsam* möglich `class T implements C { ... }`

## Wörterbücher

Haskell-Typklassen/Constraints...

```
class C a where m :: a -> a -> Foo
```

```
f :: C a => a -> Int
```

```
f x = m x x + 5
```

...sind Abkürzungen für Wörterbücher:

```
data C a = C { m :: a -> a -> Foo }
```

```
f :: C a -> a -> Int
```

```
f dict x = (m dict) x x + 5
```

Für jedes Constraint setzt der Compiler ein Wörterbuch ein.

## Wörterbücher (II)

```
instance C Bar where m x y = ...
```

```
dict_C_Bar :: C Bar
dict_C_Bar = C { m = \ x y -> ... }
```

An der aufrufenden Stelle ist das Wörterbuch *statisch* bekannt (hängt nur vom Typ ab).

```
b :: Bar ; ... f b ...
 ==> ... f dict_C_bar b ...
```

## Vergleich Polymorphie

- Haskell-Typklassen:  
statische Polymorphie,  
Wörterbuch ist zusätzliches Argument der Funktion
- OO-Programmierung:  
dynamische Polymorphie,  
Wörterbuch ist im Argument-Objekt enthalten.  
(OO-Wörterbuch = Methodentabelle der Klasse)

## Klassen-Hierarchien

Typklassen können in Beziehung stehen.  
Ord ist tatsächlich „abgeleitet“ von Eq:

```
class Eq a where
 (==) :: a -> a -> Bool
```

```
class Eq a => Ord a where
 (<) :: a -> a -> Bool
```

Ord ist Typklasse mit Typconstraint (Eq)  
also muß man erst die Eq-Instanz deklarieren, dann die Ord-Instanz.  
Jedes Ord-Wörterbuch hat ein Eq-Wörterbuch.

## Die Klasse Show

```
class Show a where
 show :: a -> String
```

vgl. Java: toString()

Die Interpreter Ghci/Hugs geben bei Eingab `exp` (normalerweise) `show exp` aus.

Man sollte (u. a. deswegen) für jeden selbst deklarierten Datentyp eine Show-Instanz schreiben.

...oder schreiben lassen: deriving Show

## Generische Instanzen (I)

```
class Eq a where
 (==) :: a -> a -> Bool
```

Vergleichen von Listen (elementweise)

*wenn* a in Eq, *dann* [a] in Eq:

```
instance Eq a => Eq [a] where
 l == r = case (l,r) of
 ([], []) -> True
 (x : xs, y : ys)
 -> (x == y) && (xs == ys)
 (_, _) -> False
```

Übung: wie sieht `instance Ord a => Ord [a]` aus? (lexikografischer Vergleich)

## Generische Instanzen (II)

```
class Show a where
 show :: a -> String
instance Show a => Show [a] where
 show [] = "[]"
 show xs = brackets
 $ concat
 $ intersperse ", "
 $ map show xs
show 1 = "1"
show [1,2,3] = "[1,2,3]"
```

## Benutzung von Typklassen bei Smallcheck

Colin Runciman, Matthew Naylor, Fredrik Lindblad:

*SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values*

- Testen von universellen Eigenschaften ( $\forall a \in A : \forall b \in B : pab$ )
- automatische Generierung der Testdaten ...
- ... aus dem Typ von  $p$
- ... mittels generischer Instanzen

<http://hackage.haskell.org/package/smallcheck>

## Smallcheck—Beispiel

```
import Test.SmallCheck

assoc op = \ a b c ->
 op a (op b c) == op (op a b) c

main = smallCheck 3
 (assoc ((++) :: [Bool] -> [Bool] -> [Bool]))
```

Übung: Kommutativität

## Typgesteuertes Generieren von Werten

```
class Testable t where ...

test :: Testable t => t -> Depth -> [TestCase]

instance Testable Bool where ...

instance (Serial a, Testable b)
 => Testable (a -> b) where ...

test (\ (x::Int) (y::Int) -> x+y == y+x)
```

## Generieren der Größe nach

```
class Serial a where
 -- | series d : alle Objekte mit Tiefe d
 series :: Int -> [a]
```

jedes Objekt hat endliche Tiefe, zu jeder Tiefe nur endliche viele Objekte  
Die „Tiefe“ von Objekten:

- algebraischer Datentyp: maximale Konstruktortiefe
- Tupel: maximale Komponententiefe
- ganze Zahl  $n$ : absoluter Wert  $|n|$
- Gleitkommazahl  $m \cdot 2^e$ : Tiefe von  $(m, e)$

## Kombinatoren für Folgen

```
type Series a = Int -> [a]

(\/) :: Series a -> Series a -> Series a
s1 \/ s2 = \ d -> s1 d ++ s2 d
(><) :: Series a -> Series b -> Series (a,b)
s1 >< s2 = \ d ->
 do x1 <- s1 d; x2 <- s2 d; return (x1, x2)

cons0 :: a -> Series a
cons1 :: Serial a
 => (a -> b) -> Series b
cons2 :: (Serial a, Serial b)
 => (a -> b -> c) -> Series c
```

## Anwendung I: Generierung von Bäumen

Variante A (explizite Implementierung)

```
data Tree a = Leaf | Branch { left :: Tree a
 , key :: a , right :: Tree a }
instance Serial a => Serial (Tree a) where
 series = cons0 Leaf \/ cons3 Branch
```

### Variante B (automatische Implementierung)

```
{-# LANGUAGE DeriveGeneric #-}
import Test.SmallCheck
import GHC.Generics
data Tree a = Leaf | Branch { left :: Tree a
 , key :: a , right :: Tree a }
 deriving Generics
instance Serial a => Serial (Tree a)
```

### Anwendung II: geordnete Bäume

```
inorder :: Tree a -> [a]

ordered :: Ord a => Tree a -> Tree a
ordered t =
 relabel t $ Data.List.sort $ inorder t
relabel :: Tree a -> [b] -> Tree b

data Ordered a = Ordered (Tree a)
instance (Ord a, Serial a)
=> Serial (Ordered a) where
 series = \ d -> map ordered $ series d

test (\ (Ordered t :: Ordered Int) -> ...)
```

### Weitere Werkzeuge zur Testfallgenerierung

Haskell (typgesteuert, statisch)

- Smallcheck (Objekte der Größe nach)
- Lazy-Smallcheck (bedarfswise)
- Quickcheck (zufällige Objekte)

OO-Sprachen (typgesteuert, dynamisch—runtime reflection)

- JCheck <http://www.jcheck.org/tutorial/>  
@RunWith(org.jcheck.runners.JCheckRunner.class)  
class SimpleTest {  
 @Test public void m(int i, int j) { ...
- ähnlich für weitere Sprachen, <https://github.com/rickynils/scalacheck>

## Übung Typklassen und Smallcheck

- definiere: Liste ist monoton steigend

```
increasing :: Ord a => [a] -> Bool
```

(a) explizite Rekursion, (b) mit zipWith

- teste mit Test.SmallCheck, ob jede Liste monoton ist
- Einfügen und Suchen in unbal. Suchbaum (mit Tests):

```
insert :: Ord a => a -> Tree a -> Tree a
contains :: Ord a => ...
```

- schreibe als fold: inorder :: Tree a -> [a]
- damit sortieren und Tests dafür
- instance Show a => Show (Tree a) als fold
- implementiere den lexikografischen Vergleich von Listen: instance Ord a => Ord (List a)

## 8 Bedarfs-Auswertung

### Motivation: Datenströme

Folge von Daten:

- erzeugen (producer)
- transformieren
- verarbeiten (consumer)

aus softwaretechnischen Gründen diese drei Aspekte im Programmtext trennen,  
aus Effizienzgründen in der Ausführung verschränken (bedarfsgesteuerter Transfor-  
mation/Erzeugung)

### Bedarfs-Auswertung, Beispiele

- Unix: Prozesskopplung durch Pipes

```
cat foo.text | tr ' ' '\n' | wc -l
```

- OO: Iterator-Muster

```
Sequence.Range(0,10).Select(n => n*n).Sum()
```

Realisierung: Co-Routinen (simulierte Nebenläufigkeit)

- FP: lazy evaluation

```
let nats = natsFrom 0 where
 natsFrom n = n : natsFrom (n+1)
sum $ map (\ n -> n*n) $ take 10 nats
```

Realisierung: Termersetzung  $\Rightarrow$  Graphersetzung, innermost-Strategie  $\Rightarrow$  outermost

### Bedarfsauswertung in Haskell

*jeder Funktionsaufruf ist lazy:*

- kehrt *sofort* zurück
- Resultat ist *thunk*
- thunk wird erst bei Bedarf ausgewertet
- Bedarf entsteht durch Pattern Matching

```
data N = Z | S N
positive :: N -> Bool
positive n = case n of
 Z -> False ; S {} -> True
x = S (error "err")
positive x
```

### Strictness

zu jedem Typ  $T$  betrachte  $T_{\perp} = \{\perp\} \cup T$   
Funktion  $f$  heißt *strikt*, wenn  $f(\perp) = \perp$ .  
in Haskell:

- Konstruktoren (Cons,...) sind nicht strikt,
- Destruktoren (head, tail,...) sind strikt.

für Fkt. mit mehreren Argumenten: betrachte Striktheit in jedem Argument einzeln.

Striktheit bekannt  $\Rightarrow$  Compiler kann effizienteren Code erzeugen (frühe Argumentauswertung)

## Bedarfsauswertung in Scala

```
object L {
 def F (x : Int) : Int = {
 println ("F", x) ; x*x
 }
 def main (args : Array[String]) {
 lazy val a = F(3);
 println ("here")
 println (a);
 } }
```

<http://www.scala-lang.org/>

## Primzahlen

```
primes :: [Int]
primes = sieve $ enumFrom 2

enumFrom :: Int -> [Int]
enumFrom n = n : enumFrom (n+1)

sieve :: [Int] -> [Int]
sieve (x : xs) = x : ...
```

## Rekursive Stream-Definitionen

```
naturals = 0 : map succ naturals

fibonacci = 0
 : 1
 : zipWith (+) fibonacci (tail fibonacci)

bin = False
 : True
 : concat (map (\ x -> [x, not x])
 (tail bin))
```

### Die Thue-Morse-Folge

$t := \lim_{n \rightarrow \infty} \tau^n(0)$  für  $\tau : 0 \mapsto 01, 1 \mapsto 10$   
 $t = 0110100110010110\dots$

$t$  ist kubikfrei

Abstandsfolge  $v := 210201210120\dots$   
ist auch Fixpunkt eines Morphismus

$v$  ist quadratfrei

### Traversieren

```
data Tree a = Branch (Tree a) (Tree a)
 | Leaf a
fold :: ...
largest :: Ord a => Tree a -> a
replace_all_by :: a -> Tree a -> Tree a
replace_all_by_largest
 :: Ord a => Tree a -> Tree a
```

die offensichtliche Implementierung

```
replace_all_by_largest t =
 let l = largest t
 in replace_all_by l t
```

durchquert den Baum zweimal.  
Eine Durchquerung reicht aus!

## 9 OO-Simulation v. Bedarfsauswertung

### Motivation (Wdhlg.)

Unix:

```
cat stream.tex | tr -c -d aeuiio | wc -m
```

Haskell:

```
sum $ take 10 $ map (\ x -> x^3) $ naturals
```

C#:

```
Enumerable.Range(0,10).Select(x=>x*x*x).Sum();
```

- logische Trennung: Produzent → Transformator(en) → Konsument
- wegen Speichereffizienz: verschränkte Auswertung.
- gibt es bei *lazy* Datenstrukturen geschenkt, wird ansonsten durch Iterator (Enumerator) simuliert.

### Iterator (Java)

```
interface Iterator<E> {
 boolean hasNext(); // liefert Status
 E next(); // schaltet weiter
}
interface Iterable<E> {
 Iterator<E> iterator();
}
```

typische Verwendung:

```
Iterator<E> it = c.iterator();
while (it.hasNext()) {
 E x = it.next (); ...
}
```

Abkürzung: `for (E x : c) { ... }`

### Beispiele Iterator

- ein Iterator (bzw. Iterable), der/das die Folge der Quadrate natürlicher Zahlen liefert
- Transformation eines Iterators (map)
- Zusammenfügen zweier Iteratoren (merge)
- Anwendungen: Hamming-Folge, Mergesort

## Beispiel Iterator Java

```
Iterable<Integer> nats = new Iterable<Integer>() {
 public Iterator<Integer> iterator() {
 return new Iterator<Integer>() {
 int s = 0;
 public Integer next() {
 int res = s ; s++; return res;
 }
 public boolean hasNext() { return true; }
 };
 }
};
for (int x : nats) { System.out.println(x); }
```

**Aufgabe: implementiere (und benutze) eine Methode**

```
static Iterable<Integer> range(int start, int count)
```

soll count Zahlen ab start liefern

## Enumerator (C#)

```
interface IEnumerator<E> {
 E Current; // Status
 bool MoveNext (); // Nebenwirkung
}
interface IEnumerable<E> {
 IEnumerator<E> GetEnumerator();
}
```

typische Benutzung: ...

Abkürzung: foreach (E x in c) { ... }

## Iteratoren mit yield

```
using System.Collections.Generic;

IEnumerable<int> Range (int lo, int hi) {
 for (int x = lo; x < hi ; x++) {
```

```

 yield return x;
 }
 yield break;
}

```

## Aufgaben Iterator C#

```

IEnumerable<int> Nats () {
 for (int s = 0; true; s++) {
 yield return s;
 }
}

```

Implementiere „das merge aus mergesort“(Spezifikation?)

```

static IEnumerable<E> Merge<E>
 (IEnumerable<E> xs, IEnumerable<E> ys)
 where E : IComparable<E>

```

zunächst für unendliche Ströme, Test: Merge (Nats () .Select (x=>x\*x) , Nats () .Select (x=>x\*x))  
(benötigt using System.Linq und Assembly System.Core)

Dann auch für endliche Ströme, Test: Merge(new int [] {1,3,4}, new int [] {2,7,8})

Dann Mergesort

```

static IEnumerable<E> Sort<E> (IEnumerable<E> xs)
 where E : IComparable<E> {
 if (xs.Count() <= 1) {
 return xs;
 } else { // zwei Zeilen folgen
 ...
 }
}

```

Test: Sort(new int [] { 3,1,4,1,5,9})

## Streams in C#: funktional, Linq

Funktional

```

IEnumerable.Range(0,10).Select(x => x^3).Sum();

```

Typ von Select? Implementierung?

Linq-Schreibweise:

```
(from x in new Range(0,10) select x*x*x).Sum();
```

Beachte: SQL-select „vom Kopf auf die Füße gestellt“.

## 10 Fkt. höherer Ord. für Streams

### Motivation

- Verarbeitung von Datenströmen,
- durch modulare Programme,  
zusammengesetzt aus elementaren Strom-Operationen
- angenehme Nebenwirkung (1):  
(einige) elementare Operationen sind parallelisierbar
- angenehme Nebenwirkung (2):  
externe Datenbank als Datenquelle, Verarbeitung mit Syntax und Semantik (Typsystem) der Gastsprache

### Strom-Operationen

- erzeugen (produzieren):
  - `Enumerable.Range(int start, int count)`
  - eigene Instanzen von `IEnumerable`
- transformieren:
  - elementweise: `Select`
  - gesamt: `Take`, `Drop`, `Where`
- verbrauchen (konsumieren):
  - `Aggregate`
  - Spezialfälle: `All`, `Any`, `Sum`, `Count`

### **Strom-Transformationen (1)**

elementweise (unter Beibehaltung der Struktur)

Vorbild:

```
map :: (a -> b) -> [a] -> [b]
```

Realisierung in C#:

```
IEnumerable Select<A,B>
 (this IEnumerable <A> source,
 Func<A,B> selector);
```

Rechenregeln für map:

```
map f [] = ...
```

```
map f (x : xs) = ...
```

```
map f (map g xs) = ...
```

### **Strom-Transformationen (2)**

Änderung der Struktur, Beibehaltung der Elemente

Vorbild:

```
take :: Int -> [a] -> [a]
```

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

```
filter :: (a -> Bool) -> [a] -> [a]
```

Realisierung:

Take, Drop, Where

Übung: takeWhile, dropWhile, ...

### **Strom-Transformationen (3)**

neue Struktur, neue Elemente

Vorbild:

```
(>>=) :: [a] -> (a -> [b]) -> [b]
```

Realisierung:

SelectMany

Rechenregel (Beispiel):

```
map f xs = xs >>= ...
```

Übung:

Definition des Operators >>= durch

```
(s >>= t) = \ x -> (s x >>= t)
```

Typ von >>=? Assoziativität? neutrale Elemente?

### Strom-Verbraucher

„Vernichtung“ der Struktur

(d. h. kann danach zur Garbage Collection, wenn keine weiteren Verweise existieren)

Vorbild:

```
fold :: b -> (a -> b -> b) -> [a] -> b
```

in der Version „von links“

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

Realisierung:

```
B Aggregate<A, B>
```

```
(this IEnumerable<A> source,
 ... seed, ... func)
```

### Zusammenfassung: C#(Linq) (Semantik)

| C# (Linq)  | Haskell |
|------------|---------|
| Select     | map     |
| SelectMany | >>=     |
| Where      | filter  |
| Aggregate  | foldl   |

mehr zu Linq: <http://msdn.microsoft.com/en-us/library/bb336768>

## Linq-Syntax (type-safe SQL)

```
var stream = from c in cars
 where c.colour == Colour.Red
 select c.wheels;
```

wird vom Compiler übersetzt in

```
var stream = cars
 .Where (c => c.colour == Colour.Red)
 .Select (c.wheels);
```

Beachte:

- das Schlüsselwort ist `from`
- Typinferenz (mit `var`)
- Kompilation: `dmcs Foo.cs -r:System.Core`

Übung: Ausdrücke mit mehreren `from`, usw.

## Linq und Parallelität

... das ist ganz einfach: anstatt

```
var s = Enumerable.Range(1, 20000)
 .Select(f).Sum() ;
```

schreibe

```
var s = Enumerable.Range(1, 20000)
 .AsParallel()
 .Select(f).Sum() ;
```

Dadurch werden

- Elemente parallel verarbeitet (`.Select(f)`)
- Resultate parallel zusammengefaßt (`.Sum()`)

vgl. <http://msdn.microsoft.com/en-us/library/dd460688.aspx>

## Iterierte assoziative Operationen

Prinzip: wenn  $\oplus$  assoziativ, dann sollte man

$$x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_8 \oplus$$

so auswerten:

$$((x_1 \oplus x_2) \oplus (x_3 \oplus x_4)) \oplus ((x_5 \oplus x_6) \oplus (x_7 \oplus x_8))$$

Beispiel: carry-lookahead-Addierer

(die assoziative Operation ist die Verkettung der Weitergabefunktionen des Carry)

- beweise Assoziativität von ++
- welches ist die assoziative Operation für „(parallele) maximale Präfix-Summe“?

## Map/Reduce-Algorithmen

map\_reduce

```
:: ((ki, vi) -> [(ko, vm)]) -- ^ map
-> ((ko, [vm]) -> [vo]) -- ^ reduce
-> [(ki, vi)] -- ^ eingabe
-> [(ko, vo)] -- ^ ausgabe
```

Beispiel (word count)

```
ki = Dateiname, vi = Dateiinhalte
ko = Wort, vm = vo = Anzahl
```

- parallele Berechnung von map
- parallele Berechnung von reduce
- verteiltes Dateisystem für Ein- und Ausgabe

## Map-Reduce: Literatur

- Jeffrey Dean and Sanjay Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004. <http://labs.google.com/papers/mapreduce.html>
- Ralf Lämmel: *Google's MapReduce programming model - Revisited*, Science of Computer Programming - SCP, vol. 70, no. 1, pp. 1-30, 2008 <http://www.systems.ethz.ch/education/past-courses/hs08/map-reduce/reading/mapreduce-progmodel-scp08.pdf>

# 11 Serialisierung, Persistenz

## Motivation

Die meisten Daten leben länger als ein Programmlauf, vgl.

- Akten (Papier), Archiv, ...
- Bearbeitung/Ergänzung einer Akte

Akten (Daten) in maschinenlesbarer Form:

- Lochkarten (US-Volkszählung 1890)
- Magnetbänder, Festplatten

Programmtexte sprechen nur über Daten während des Programmlaufes.

## Typisierung von Daten

von untypisiert bis statisch typisiert:

- Zeichenketten (String), Bytefolgen
- XML-Baum (DOM) ohne Schema
- Objekt eines bestimmten Typs, mit bestimmten Attributen

beachte:

- statische Typisierung ist das anzustrebende Ziel  
(Typprüfung durch Compiler)
- wird das nicht erkannt oder nicht erreicht, dann:  
Typprüfung durch Programm, zur Laufzeit (unsicher, ineffizient)

## Ansätze

- Programm bestimmt Form der Daten  
externe Repräsentation (DB-Schema) wird aus interner Repräsentation (Typ, Klassen) abgeleitet (automatisch, unsichtbar)
- Programm verarbeitet vorhandene Daten  
interne Repräsentation (Typen) wird aus externer Repr. (DB-Schema) abgeleitet
- Programm läuft (scheinbar) immer  
Application Server verwaltet Softwarekomponenten und Datenkomponenten

## Enterprise Java Beans

Klasse als Entity Bean (vereinfacht):

```
import javax.persistence.*;
@Entity public class C {
 @Id int id;
 String foo;
 double bar;
}
```

Application Server (z. B. JBoss) verwaltet diese Beans, Datenbankschema kann autom. generiert werden.

JSR 220: Enterprise JavaBeans™ 3.0 <http://www.jcp.org/en/jsr/detail?id=220>

## DB-Anfragen in Java EE

```
public List findWithName(String name) {
return em.createQuery(
 "SELECT c FROM Customer c WHERE c.name LIKE :custName")
 .setParameter("custName", name)
 .setMaxResults(10).getResultList(); }
```

<http://docs.oracle.com/javaee/5/tutorial/doc/bnbqw.html#bnbrg>  
beachte: Query ist hier String, aber gemeint ist: Funktion ( $\lambda$  custName  $\rightarrow$  ...)  
Nachteile (vgl. auch <http://xkcd.com/327/>)

- drei Namensbereiche
- keine statische Typisierung
- keine Syntaxprüfung

## Noch mehr Sprachen: HTML, Javascript

<http://weblogs.java.net/blog/driscoll/archive/2009/09/26/ajax-tag-even>

```
<h:body> <h:form id="form">
Echo test: <h:outputText id="out" value="#{listenBean.hello}"/>
String Length: <h:outputText id="count" value="#{listenBean.length}"/>
<h:inputText id="in" value="#{listenBean.hello}" autocomplete="off">
<f:ajax event="keyup" render="out count eventcount"
listener="#{listenBean.update}"/></h:inputText>
```

grundsätzliche Probleme werden dadurch noch verstärkt:

- jede Sprache: eigene Abstraktionen, eigenes Typsystem;
- es gibt keine übergeordneten Mechanismen dafür;
- (anscheinend) z. T. für Benutzer entworfen, die nicht wissen, was eine Abstraktion und was ein Typ ist

### **LINQ und SQLmetal (1)**

<http://msdn.microsoft.com/en-us/library/bb386987.aspx>  
generiert C#-Typdeklaration aus DB-Schema

```
sqlmetal /namespace:nwind /provider:Sqlite
 '/conn:Data Source=Northwind.db3' /code:nwind.cs
```

Objekte können dann statisch typisiert verarbeitet werden.

### **LINQ und SQLmetal (2)**

Datenbankverbindung herstellen:

```
using System; using System.Data.Linq;
using System.Linq; using Mono.Data.Sqlite;
using nwind;
var conn = new SqliteConnection
 ("DbLinqProvider=Sqlite; Data Source=Northwind.db3");
Main db = new Main (conn);
```

Datenquelle benutzen:

```
var pens = from p in db.Customers
 select new { p.City, p.Address} ;
foreach (var pen in pens) { Console.WriteLine (pen); }
```

beachte LINQ-Notation (from, select)  
und Verwendung von anonymen Typen (new) (für Tupel)

## 12 Refactoring (KW23)

### Definition

Martin Fowler: *Refactoring: Improving the Design of Existing Code*, A.-W. 1999, <http://www.refactoring.com/>

Def: Software so ändern, daß sich

- externes Verhalten nicht ändert,
- interne Struktur verbessert.

siehe auch William C. Wake: *Refactoring Workbook*, A.-W. 2004 <http://www.xp123.com/rwb/> und Stefan Buchholz: *Refactoring (Seminarvortrag)* <http://www.imn.htwk-leipzig.de/~waldmann/edu/ss05/se/talk/sbuchhol/>

### Refactoring: Herkunft

Kent Beck: *Extreme Programming*, Addison-Wesley 2000:

- Paar-Programmierung (zwei Leute, ein Rechner)
- test driven: erst Test schreiben, dann Programm implementieren
- Design nicht fixiert, sondern flexibel

Grundlagen: semantikerhaltende Transformationen

- von Daten (Mengenlehre)
- von Unterprogrammen (Lambda-Kalkül)

### Refactoring anwenden

- mancher Code „riecht“ (schlecht)  
(Liste von *smells*)
- er (oder anderer) muß geändert werden  
(Liste von *refactorings*, Werkzeugunterstützung)
- Änderungen (vorher!) durch Tests absichern  
(JUnit)

## Refaktorisierungen

- Abstraktionen einführen:  
neue Schnittstelle, Klasse (Entwurfsmuster!)  
Methode, (temp.) Variable
- Abstraktionen ändern:  
Attribut/Methode bewegen (in andere Klasse)

## Guter und schlechter Code

- clarity and simplicity are of paramount importance
- the user of a module should never be surprised by its behaviour
- modules should be as small as possible but not smaller
- code should be reused rather than copied
- dependencies between modules should be minimal
- errors should be detected as soon as possible, ideally at compile time

(Joshua Bloch: Effective Java <http://java.sun.com/docs/books/effective/>)

## Für wen schreibt man Code?

Donald Knuth 1993, vgl. <http://tex.loria.fr/historique/interviews/knuth-clb1993.html>:

- Programming is: telling a *human* what a computer should do.

Donald Knuth 1974, vgl. [http://en.wikiquote.org/wiki/Donald\\_Knuth](http://en.wikiquote.org/wiki/Donald_Knuth):

- Premature optimization is the root of all evil.

## Code Smell # 1: Duplicated Code

jede Idee sollte an *genau einer* Stelle im Code formuliert werden:  
Code wird dadurch

- leichter verständlich
- leichter änderbar

Verdoppelter Quelltext (copy-paste) führt immer zu Wartungsproblemen.

## **Duplicated Code → Schablonen**

duplizierter Code wird verhindert/entfernt durch

- *Schablonen* (beschreiben das Gemeinsame)
- mit *Parametern* (beschreiben die Unterschiede).

Beispiel dafür:

- Unterprogramm (Parameter: Daten, Resultat: Programm)
- polymorphe Klasse (Parameter: Typen, Resultat: Typ)
- Unterprogramm höherer Ordnung (Parameter: Programm, Resultat: Programm)

## **Plan**

(für restl. Vorlesungen)

- code smells und Refactoring für Klassen
- ... für Methoden, Anweisungen
- Leistungsmessungen und -verbesserungen
- Zusammenfassung

## **Klassen-Entwurf**

- benutze Klassen! (sonst: primitive obsession)
- ordne Attribute und Methoden richtig zu  
(Refactoring: move method, usw.)
- dokumentiere Invarianten für Objekte, Kontrakte für Methoden
- stelle Beziehungen zwischen Klassen durch Interfaces dar  
(... Entwurfsmuster)

## Primitive Daten (*primitive obsession*)

Symptome: Benutzung von `int`, `float`, `String`...

Ursachen:

- fehlende Klasse:  
z. B. `String` → `FilePath`, `Email`, `URI` ...
- schlecht implementiertes Fliegengewicht  
z. B. `int i` bedeutet `x[i]`
- simulierter Attributname:  
z. B. `Map<String, String> m; m.get("foo");`

Behebung: Klassen benutzen, Array durch Objekt ersetzen  
(z. B. `class M { String foo; ...}`)

## Verwendung von Daten: Datenklumpen

Fehler: Klumpen von Daten wird immer gemeinsam benutzt

```
String infile_base; String infile_ext;
String outfile_base; String outfile_ext;
```

```
static boolean is_writable
 (String base, String ext);
```

Indikator: ähnliche, schematische Attributnamen

Lösung: Klasse definieren

```
class File
 { String base; String extension; }
static boolean is_writable (File f);
```

Ü: vgl. mit `java.nio.file.Path`

## Datenklumpen—Beispiel

Beispiel für Datenklumpen und -Vermeidung:

```
java.awt
```

```
Rectangle(int x, int y, int width, int height)
Rectangle(Point p, Dimension d)
```

Vergleichen Sie die Lesbarkeit/Sicherheit von:

```
new Rectangle (20, 40, 50, 10);
new Rectangle (new Point (20, 40)
 , new Dimension (50, 10));
```

Vergleichen Sie:

```
java.awt.Graphics: drawRectangle(int,int,int,int)
java.awt.Graphics2D: draw (Shape);
class Rectangle implements Shape;
```

### Verwendung von Daten: Data Class

Fehler:

Klasse mit Attributen, aber ohne Methoden.

```
class File { String base; String ext; }
```

Lösung:

finde typische Verwendung der Attribute in Client-Klassen, (Bsp: `f.base + "/" + f.ext`)  
schreibe entsprechende Methode, versteckte Attribute (und deren Setter/Getter)

```
class File { ...
 String toString () { ... }
}
```

### Mehrfachverzweigungen

Symptom: switch wird verwendet

```
class C {
 int tag; int FOO = 0;
 void foo () {
 switch (this.tag) {
 case FOO: { .. }
 case 3: { .. }
 }
 }
}
```

Ursache: Objekte der Klasse sind nicht ähnlich genug

Abhilfe: Kompositum-Muster

```
interface C { void foo (); }
class Foo implements C { void foo () { .. } }
class Bar implements C { void foo () { .. } }
```

## Das Fabrik-Muster

```
interface I {}
class C implements I {}
class D implements I {}
```

Bei Konstruktion von I-Objekten muß ein konkreter Klassenname benutzt werden.

Wie schaltet man zwischen C- und D-Erzeugung um?

Benutze Fabrik-Objekt: Implementierung von

```
interface F { I create () }
```

## null-Objekte

Symptom: null (in Java) bzw. 0 (in C++) bezeichnet ein besonderes Objekt einer Klasse, z. B. den leeren Baum oder die leere Zeichenkette

Ursache: man wollte Platz sparen oder „Kompositum“ vermeiden.

Nachteil: null bzw. \*0 haben keine Methoden.

Abhilfe: ein extra Null-Objekt deklarieren, das wirklich zu der Klasse gehört.

## Typsichere Aufzählungen

Definition (einfach)

```
public enum Figur { Bauer, Turm, König }
```

Definition mit Attribut (aus JLS)

```
public enum Coin {
 PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
 Coin(int value) { this.value = value; }
 private final int value;
 public int value() { return value; }
}
```

Definition mit Methode:

```
public enum Figur {
 Bauer { int wert () { return 1; } },
 Turm { int wert () { return 5; } },
 König { int wert () { return 1000; } };
 abstract int wert ();
}
```

Benutzung:

```
Figur f = Figur.Bauer;
Figur g = Figur.valueOf("Turm");
for (Figur h : Figur.values()) {
 System.out.println (h + ":" + h.wert());
}
```

## 13 Refactoring (KW24)

### Vererbung bricht Kapselung

(Implementierungs-Vererbung: schlecht, Schnittstellen-Vererbung: gut.)

Problem: `class C extends B`  $\Rightarrow$   
*C* hängt ab von Implementations-Details von *B*.

$\Rightarrow$  wenn Implementierung von *B* unbekannt, dann korrekte Implementierung von *C* nicht möglich.

$\Rightarrow$  Wenn man Implementierung von *B* ändert, kann *C* kaputtgehen.

Beispiel: `class CHS<E> extends HashSet<E>`, Methoden `add` und `addAll`,  
nach: Bloch: Effective Java, Abschnitt 14 (Favor composition over inheritance)

### Vererbung bricht Kapselung

Bloch, Effective Java, Abschnitt 15:

- design and document for inheritance...

API-Beschreibung muß Teile der Implementierung dokumentieren (welche Methoden rufen sich gegenseitig auf), damit man diese sicher überschreiben kann.

- ... or else prohibit it.

– am einfachsten: `final class C { ... }`

– mglw.: `class C { private C () { ... } ... }`

statt Vererbung: benutze Komposition (Wrapper) und dann Delegation.

Übung: `Counting (Hash) Set<E>` mittels Wrapper

## **Immutability**

(Joshua Bloch: Effective Java, Abschnitt 13: Favor Immutability) — immutable = unveränderlich

Beispiele: String, Integer, BigInteger

- keine Set-Methoden
- keine überschreibbaren Methoden
- alle Attribute final

leichter zu entwerfen, zu implementieren, zu benutzen.

## **Immutability**

- immutable Objekte können mehrfach benutzt werden (sharing).  
(statt Konstruktor: statische Fabrikmethode oder Fabrikobjekt. Suche Beispiele in Java-Bibliothek)
- auch die Attribute der immutable Objekte können nachgenutzt werden (keine Kopie nötig)  
(Beispiel: negate für BigInteger)
- immutable Objekte sind sehr gute Attribute anderer Objekte:  
weil sie sich nicht ändern, kann man die Invariante des Objektes leicht garantieren

## **Zustandsänderungen**

Programmzustand ist immer implizit (d. h. unsichtbar).

⇒ jede Zustandsänderung (eines Attributes eines Objektes, einer Variablen in einem Block) erschwert

- Spezifikation, Tests, Korrektheitsbeweis,
- Lesbarkeit, Nachnutzung.

Code smells:

- Variable wird deklariert, aber nicht initialisiert (Refactoring: Variable später deklarieren)
- Konstruktor, der Attribute nicht initialisiert (d. h., der die Klasseninvariante nicht garantiert)

### **Code smell: Temporäre Attribute**

Symptom: viele `if (null == foo)`

Ursache: Attribut hat nur während bestimmter Programmteile einen sinnvollen Wert

Abhilfe: das ist kein Attribut, sondern eine temporäre Variable.

### **Code-Größe und Komplexität**

Motto: was der Mensch nicht *auf einmal* überblicken/verstehen kann, versteht er *gar nicht*.

Folgerung: jede Sinn-Einheit (z. B. Implementierung einer Methode, Schnittstelle einer Klasse) muß auf eine Bildschirmseite passen

Code smells:

- Methode hat zu lange Argumentliste
- Klasse enthält zuviele Attribute
- Klasse enthält zuviele Methoden
- Methode enthält zuviele Anweisungen (Zeilen)
- Anweisung ist zu lang (enthält zu große Ausdrücke)

### **Benannte Abstraktionen**

überlangen Code in überschaubare Bestandteile zerlegen:

- Abstraktionen (Konstante, Methode, Klasse, Schnittstelle) einführen ... und dafür *passende Namen* vergeben.

Code smell: Name drückt Absicht nicht aus.

Symptome:

- besteht aus nur 1 ... 2 Zeichen, enthält keine Vokale
- numerierte Namen (`panel1`, `panel2`, `\dots`)
- unübliche Abkürzungen, irreführende Namen

Behebung: umbenennen, so daß Absicht deutlicher wird. (Dazu muß diese dem Programmierer selbst klar sein!)

Werkzeugunterstützung!

## Name enthält Typ

Symptome:

- Methodename enthält Typ des Arguments oder Resultats

```
class Library { addBook(Book b); }
```

- Attribut- oder Variablenname bezeichnet Typ (sog. Ungarische Notation) z. B. `char ** ppcFoo`  
<http://ootips.org/hungarian-notation.html>
- (grundsätzlich) Name bezeichnet Implementierung statt Bedeutung

Namenskonventionen: schlecht, statische Typprüfung: gut.

## Refaktorisierung von Ausdrücken

- code smells: ein langer Ausdruck, mehrfach der gleiche Ausdruck (z. B. ein Zahl- oder String-Literal)

refactoring: Konstante einführen

- *One man's constant is another man's variable.*

(Alan Perlis, 1982, <http://www.cs.yale.edu/quotes.html>)

- code smell: mehrere ähnliche Ausdrücke

refactoring: Unterprogramm (Funktion) einführen

(Funktion = Unterprogramm, das einen Wert liefert)

## Refaktorisierung durch Funktionen

Gegeben: (Code smell: duplizierter/ähnlicher Code)

```
{ int a = ... ; int b = ... ;
 int x = a * 13 + b; int y = a * 15 + b; }
```

Mögliche Refaktorisierungen:

- lokale Funktion (C#) (mit einem Parameter)
- globale Funktion (Java) (mit einem Parameter)?  
(welches Problem entsteht?)
- globale Funktion (Java), die dieses Problem vermeidet

Beobachtung: in Sprachen ohne lokale Unterprogramme werden solche Abstraktionen zu schwerfällig.

vgl. <http://openjdk.java.net/projects/lambda/>

## Refaktorisierung durch Prozeduren

(Prozedur = Unterprogramm, das den Programmzustand ändert)

- gleiche Betrachtung (lokal, global, Hilfsvariablen) wie für Funktionen
- erschwert durch Nebenwirkungen auf lokale Variablen

Eclipse:

- Extract method (mit Bezug auf 1, 2 lokale Variablen)
- Change local variable to field

Übung: Zusammenhang zwischen Code Smell *Kommentar* und Unterprogrammen

## Richtig refaktorisieren

- immer erst die Spezifikation (die Tests) schreiben
- Code kritisch lesen (eigenen, fremden), eine Nase für Anrührigkeiten entwickeln (und für perfekten Code).
- jede Faktorisierung hat ein Inverses.  
(neue Methode deklarieren ↔ Methode inline expandieren)  
entscheiden, welche Richtung stimmt!
- Werkzeug-Unterstützung erlernen

## Aufgaben zu Refactoring (I)

- Code Smell Cheat Sheet (Joshua Kerievsky): <http://industriallogic.com/papers/smellstorefactorings.pdf>
- Smell-Beispiele <http://www.imn.htwk-leipzig.de/~waldmann/edu/ss05/case/rwb/> (aus Refactoring Workbook von William C. Wake <http://www.xp123.com/rwb/>)  
ch6-properties, ch6-template, ch14-ttt

## Aufgaben zu Refactoring (II)

Refactoring-Unterstützung in Eclipse:

```
package simple;

public class Cube {
 static void main (String [] argv) {
 System.out.println (3.0 + " " + 6 * 3.0 * 3.0);
 System.out.println (5.5 + " " + 6 * 5.5 * 5.5);
 }
}
```

extract local variable, extract method, add parameter, ...

## Aufgaben zu Refactoring (II)

- Eclipse → Refactor → Extract Interface
- “Create Factory”
- Finde Beispiel für “Use Supertype”

# 14 Testklausur KW 25

## Übersicht

- hier: 6 Aufgaben je 10 ... 20 min
- Prüfung (wahrscheinlich): 4 Aufgaben je 30 min
- Lösung allein und ohne Unterlagen
- Korrektur durch Kommilitonen
- Bewertung: je Aufgabe ca. 3 Teilaufgaben,  
je Teilaufgabe: richtig (1P), vielleicht (0.5P), falsch (0P);  
gesamt 18 P, Zensurengrenzen (%): 40, 60, 80, 95
- Auswertung in Übung

## Datentypen

Ergänzen Sie die Typdeklarationen:  
die Menge der Wahrheitswerte:

```
data Bool = False | ...
```

die Menge der Listen mit Elementtyp a:

```
data List a = Nil | Cons ...
```

Gegeben sind weiterhin:

```
data Foo = Bar
data Maybe a = Nothing | Just a
data Pair a b =
 Pair { first :: a, second :: b }
```

Geben Sie jeweils alle Elemente dieser Typen an:

```
Maybe Bool ; Pair Bool (Pair Foo Bool)
```

Geben Sie die Pair a b entsprechende Java-Klasse an.

## Pattern Matching

Programmieren Sie das logische Und:

```
und :: Bool -> Bool -> Bool
und x y = case x of
 ...
```

Geben Sie eine möglichst einfache Funktion f an (Typ und Implementierung), die diese Tests erfüllt:

```
f (Just 3) (Just 4) == Just 7
f (Just 5) Nothing == Nothing
f Nothing (Just 6) == Just 6
```

Im Quelltext sollen keine Zahlen-Literale vorkommen.

## Rekursionsmuster

Zu dem Datentyp für nichtleere binäre Bäume

```
data Tree k = Leaf k
 | Node (Tree k) k (Tree k)
```

gehört ein Rekursionsmuster (`fold`). Geben Sie dessen Typ an.

Geben Sie Typ und Bedeutung dieser Funktion an:

```
fold (\ k -> if k then 1 else 0)
 (\ x k y -> x + y)
```

Schreiben Sie mittels `fold` die Funktion, die einen Baum spiegelt. Test:

```
sp (Node (Leaf 1) 2 (Leaf 3))
 == Node (Leaf 3) 2 (Leaf 1)
```

## (Un)veränderliche Objekte

Diese Klasse benutzt veränderliche Objekte:

```
class Clock {
 int clock
 void tick () { this.clock++; }
}
```

Schreiben Sie die Klasse um, so daß ihre Objekte unveränderlich sind.

Wie muß dabei das folgende Anwendungsprogramm geändert werden?

```
Clock c = new Clock(); c.tick();
```

Nennen Sie Vorteile und Nachteile der Verwendung unveränderlicher Objekte.

## Datenströme

In C# kann man diese Notation für Schleifen verwenden:

```
IEnumerable<E> stream = ... ;
foreach (E x in stream) { ... }
```

- Übersetzen Sie das in eine Schleife mit expliziter Benutzung eines Enumerators.
- Geben Sie die Deklarationen der dabei benutzten Methoden an.
- Schreiben Sie einen impliziten Enumerator (d. h. nicht diesen Methoden implementieren, sondern `yield` benutzen) für die Folge 0, 1, 4, 9, ... der Quadratzahlen.

## LINQ

- Bestimmen Sie den Wert von

```
new int[] {1, 2, 3, 4}.Aggregate(0, (x, y) => y - x)
```

- Übersetzen Sie in einen Aufruf von Select:

```
from x in Enumerable.Range(1, 3) select x*x;
```

- (Zusatz) Übersetzen Sie mittels geeigneter Methoden:

```
(from x in Enumerable.Range(1, 5)
from y in Enumerable.Range(1, x)
where 0 == x%y select 1).Sum();
```

- Geben Sie die Typen von Aggregate, Select, SelectMany an
- Wie lassen sich solche Programme parallelisieren?

## 15 Verfrühte Optimierung ...

### ...ist die Quelle allen Übels

So ist es richtig:

1. passende Datenstrukturen und Algorithmen festlegen ...
2. ... und korrekt implementieren,
3. Ressourcenverbrauch messen,
4. *nur bei nachgewiesenem Bedarf* Implementierung ändern, um Ressourcenverbrauch zu verringern.

und jede andere Reihenfolge ist falsch, sinnlos oder riskant.

### Sprüche zur Optimierung

(so zitiert in J. Bloch: Effective Java)

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity. – W. A. Wulf

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. – Donald E. Knuth

We follow two rules in the matter of optimization:

- Rule 1. Don't do it.
- Rule 2 (for experts only). Don't do it yet – that is, not until you have a perfectly clear and unoptimized solution.

– M.A. Jackson

### **Rekursion ist teuer? Falsch!**

Welches Programm ist schneller?

```
int gcd (int x, int y) { // Rekursion:
 if (0==y) return x else return gcd(y,x%y);
}
```

```
int gcd (int x, int y) { // Schleife:
 while (0!=y) {int h = x%y ; x = y; y = h;}
 return x;
}
```

Antwort: keines, gcc erzeugt identischen Assemblercode.

Das funktioniert immer für *Endrekursion* (= die letzte Aktion eines Unterprogramms ist der rekursive Aufruf), diese kann durch Sprung ersetzt werden.

### **Java ist langsam? Falsch!**

```
static int gcd (int x, int y) {
 if (0==y) return x; else return gcd(y,x%y);
}
```

Testtreiber:  $10^8$  Aufrufe, Laufzeit:

- C/gcc: 6.6 s
- Java: 7.1 s
- C#/Mono: 7.9 s

## Array-Index-Prüfungen sind teuer? Falsch!

James Gosling:

One of the magics of modern compilers is that they're able to "theorem-prove away" potential all [array] subscript checks. . . .

You might do a little bit of checking on the outside of the loop, but inside the loop, it just screams.

[The VM] had a crew of really bright people working on it for a decade, a lot of PhD compiler jockeys.

Quelle: Biancuzzi und Warden: Masterminds of Programming, O'Reilly, 2009

Codebeispiel: <http://www.imn.htwk-leipzig.de/~waldmann/talk/12/osmop/> (Folie 8 ff)

## Lokale Variablen sind teuer? Falsch!

Welches Programm braucht weniger Zeit oder Platz?

1) Variable `h` ist „global“:

```
int s = 0; int h;
for (int i = 0; i<n; i++) {
 h = i*i; s += h;
}
```

2) Variable `h` ist lokal:

```
int s = 0;
for (int i = 0; i<n; i++) {
 int h = i*i; s += h;
}
```

Antwort: keines, `javac` erzeugt identischen Bytecode.

## Weitere Diskussionen

z. B. Effizienz von Haskell-Code

vgl. <http://article.gmane.org/gmane.comp.lang.haskell.cafe/99002>

selbst wenn dort noch ein Faktor 2 (zu C/Java) zu sehen ist ...

- Laufzeiteffizienz ist *ein* Kriterium,

- ein anderes ist, was man durch Beschäftigung mit Sprache/Paradigma *lernt*.
- Antwort: Abstraktionen.