Algebraische und rekursive Datentypen

Nachteile von Tupeln

Produkttypen: Typen mit mehreren Komponenten

- Personen: Namen und Alter, Autos: Modell und Baujahr

Als Typsynonym: mit Tupeln

```
type People = (String,Int)
jogi :: Car
jogi = ("Joachim Löw",50)

isAdult :: People -> Bool
isAdult (name,age) = (age>=18)
```

```
type Car = (String,Int)
kitt :: People
kitt = ("Firebird TransAm",1982)

isAdult kitt
```

Nachteile:

- Bedeutung von Werten `("Carina",88)` nicht explizit
- Ungewollte Verwendung beliebiger `(String,Int)` Tupel
- Abhilfe: Algebraische Datentypen

Nachteile von Tupeln

Produkttypen: Typen mit mehreren Komponenten

- Personen: Namen und Alter,   Autos: Modell und Baujahr

Als Algebraischer Datentyp: mit Schlüsselwort **data**

```
data People = Person String Int
jogi :: People
jogi = Person "Joachim␣Löw" 50
```

```
data Automobile = Car String Int
kitt :: Automobile
kitt = Car "Firebird␣TransAm" 1982
```

```
isAdult :: People -> Bool
isAdult (Person name age) = (age>=18)
```

Algebraische Datentypen: Definition neuer Typen

- Durch Auflistung aller Konstruktoren, hier:
- `Person :: String -> Int -> People`

  `Car :: String -> Int -> Automobile`

Algebraische Datentypen

Einfachste Datentypen: Aufzählungstypen

```
data Temp = Cold | Hot
data Season = Spring | Summer | Autumn | Winter
```

- Aufzählung aller Werte des neuen Typs
- `Cold`, `Hot`, `Spring` ... 0-stellige Konstruktoren

$\Rightarrow$ Einsatz in Pattern-Matching

Funktionen auf Datentypen: Pattern-Matching

```
weather :: Season -> Temp
weather Spring = Cold
weather Summer = Hot
weather Autumn = Cold
weather Winter = Cold
```

Algebraische Datentypen

Alternativ-Typen: Mehrere Konstruktoren

```
data Shape =   Circle Double                -- Radius
             | Rectangle Double Double      -- Seitenlängen
             | Square Double                -- Seitenlänge
```

Jeder Wert `x :: Shape` ist entweder ein

- Kreis, Rechteck oder Quadrat
- Eigenschaften von `x`: Konstruktorargumente

Beispiel-Werte:

```
unitCircle :: Shape              dinA4 :: Shape
unitCircle = Circle 1.0          dinA4 = Rectangle 210.0 297.0
```

Funktionsdefinition:

```
area :: Shape -> Double
area (Circle r)        = pi*r*r
area (Rectangle a b)   = a*b
area (Square  a)       = a*a
```

Polymorphe Datentypen

Optionale Werte: (vgl.: `null`)

```
data Maybe t = Nothing | Just t
Just True :: Maybe Bool
Nothing   :: Maybe String
```

Summen-Typ:

```
data Either s t = Left s | Right t
Left  42      :: Either Int String
Right "true"  :: Either Int String
```

- Polymorph, parametrisiert durch `s`, `t`

Matrizen (dicht-/dünnbesetzt):

```
data Matrix t =   Dense  [[t]] -- Liste von Zeilen
                | Sparse [(Integer,Integer,t)] t -- Einträge (i,j,v)
                                                 -- und Default-Wert

unit :: Integer -> Matrix Float
unit n = Sparse [(i,i,1.0) | i <- [1..n]] 0

rotation :: Double -> Matrix Double
rotation alpha = Dense [[cos alpha, -sin alpha],
                        [sin alpha,  cos alpha]]
```

Rekursive Datentypen

Stacks sind entweder

1. Der leere Stack `Empty`, oder
2. Ein Stack `Stacked x s`, mit oberstem Element `x` und Rest-Stack `s`

Als Datentypdefinition:

```
data Stack t =  Empty | Stacked t (Stack t)

pop Empty           = error "Empty"      push x s = Stacked x s
pop (Stacked x s) = s

top Empty           = error "Empty"
top (Stacked x s) = x

someStack :: Stack Integer
someStack  = Stacked 3 (Stacked 1 Empty)
```

- rekursiv (im Konstruktor `Stacked`)
- polymorph (paremetrisiert durch `t`)

Anwendungen algebraischer Datentypen

Anwendungen von Datentypen

Algebraische Datentypen ermöglichen

- Implementierung von Datenstrukturen
- Modellierung problemspezifischer Daten

Einsatz von Pattern-Matching

- erleichtert Umsetzung komplexer Algorithmen
- besonders für baumartige Datentypen

Anwendungsbeispiele:

- Datenstrukturen: Maps, Bäume, Rot-Schwarz-Bäume
- Fehlerbehandlung
- Termersetzungssysteme

Maps

Datenstruktur: Map (auch: Dictionary)

- Repräsentation partieller Abbildungen
- Funktionen zum Auslesen/Einfügen

```
type Map k v = ...

insert :: (Eq k) => k -> v -> Map k v -> Map k v
lookup :: (Eq k) => k -> Map k v -> Maybe v
empty  :: Map k v

noten :: Map String Integer
noten = insert "Lisa" 1 (insert "Max" 3 empty)
```

Auswertungs-Beispiele:

`lookup "Max" noten`	$\Rightarrow^+$	`Just 3`
`lookup "Mark" noten`	$\Rightarrow^+$	`Nothing`
`lookup "Max" (insert "Max" 2 noten)`	$\Rightarrow^+$	`Just 2`

Map-Implementierung: Listen

Einfachste Implementierung: Assoziativlisten

- Listen vom Typ `[(k,v)]`, zusammen mit
- Funktionen `lookup`, `insert`

```
type Map k v = [(k,v)]

empty = []

insert :: (Eq k) => k -> v -> Map k v -> Map k v
insert k v []              = [(k,v)]
insert k v ((k',v'):kvs)
   | k==k'      = (k ,v ):kvs
   | otherwise = (k',v'):(insert k v kvs)

lookup :: (Eq k) => k -> Map k v -> Maybe v
lookup k []            =  Nothing
lookup k ((k',v'):kvs)
   | k == k'    =  Just v'
   | otherwise =  lookup k kvs
```

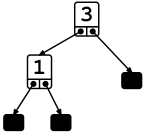
Effiziente Implementierung: Suchbäume

Binäre Bäume

Binäre Bäume: Entweder

1. Ein Blatt `Leaf`, oder
2. Ein Knoten `Node left x right` , mit Teilbäumen `left`, `right` und Knoteneintrag `x`

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

someTree = Node (Node Leaf 1 Leaf) 3 Leaf
```



Größe und Höhe eines binären Baumes

```
size :: Tree t -> Int
size   Leaf = 0
size   (Node left x right) =  (size left) + 1 + (size right)

height :: Tree t -> Int
height Leaf = 0
height (Node left x right) = 1 + (max (height left) (height right))
```

Binäre Bäume

Analog zu Listen: `mapT` über Baum-Elemente

```
mapT :: (s -> t) -> Tree s -> Tree t
mapT f Leaf = Leaf
mapT f (Node left x right) =
    Node (mapT f left) (f x) (mapT f right)
```

Einfache Beispiele:

```
add5 :: Tree Integer -> Tree Integer
add5 = mapT (+5)

fstT :: Tree (s,t) -> Tree s
fstT  = mapT fst
```

Binäre Bäume

Analog zu Listen: `foldT` über Baum-Elemente

```
foldT :: (s -> t -> s -> s) -> s -> Tree t -> s
foldT f i Leaf = i
foldT f i (Node left x right) = f (foldT f i left) x (foldT f i right)
```

Größe und Höhe eines Baumes

```
size   = foldT (\left x right -> left+1+right) 0
height = foldT (\left x right -> 1+(max left right)) 0
```

Binäre Bäume

Analog zu Listen: `foldT` über Baum-Elemente

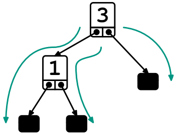
```
foldT :: (s -> t -> s -> s) -> s -> Tree t -> s
foldT f i Leaf = i
foldT f i (Node left x right) = f (foldT f i left) x (foldT f i right)
```

Liste aller Pfade durch einen Baum

```
paths :: Tree t -> [[t]]
paths tree = foldT consAll [[]] tree
   where consAll left x right = map (x:) (left++right)
```

Beispiel:

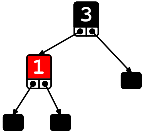
```
   paths (Node (Node Leaf 1 Leaf) 3 Leaf)
⇒+ consAll (foldT consAll [[]] (Node Leaf 1 Leaf) 3
           (foldT consAll [[]] Leaf )
⇒+ consAll (foldT consAll [[]] (Node Leaf 1 Leaf) 3
           [[]]
⇒+ consAll (consAll (foldT consAll [[]] Leaf) 1
                    (foldT consAll [[]] Leaf)   ) 3
           [[]]
⇒+ consAll (consAll [[]] 1 [[]]) 3  [[]]
⇒+ consAll [[1],[1]] 3 [[]] ⇒+ [[3,1],[3,1],[3]]
```



Rot-Schwarz-Bäume

Rot-Schwarz-Bäume:

- Knoten rot oder schwarz
- Blätter schwarz




```
data Color = Red | Black
data RedBlackTree t = Leaf |
                      Node Color (RedBlackTree t) t (RedBlackTree t)
```

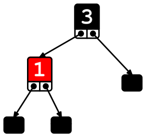
Fold und Map: Farbe beachten

```
fold :: (Color -> s -> t -> s -> s) -> s -> RedBlackTree t -> s
fold f i Leaf = i
fold f i (Node c left x right) = f c (fold f i left) x (fold f i right)

mapRB :: (Color -> s -> t) -> RedBlackTree s -> RedBlackTree t
mapRB f Leaf = Leaf
mapRB f (Node c left x right) =
   Node c (mapRB f left) (f c x) (mapRB f right)
```

Rot-Schwarz-Bäume

Invarianten:

1. Kein roter Knoten hat roten Elternknoten
2. Alle vollständige Pfade haben gleiche Anzahl schwarzer Knoten
3. Baum ist sortiert:
   Elemente linker Teilbaum $\leq$ Knoten-Element
   Elemente rechter Teilbaum $\geq$ Knoten-Element



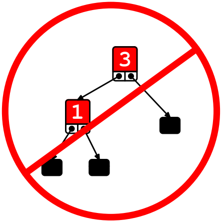
Garantiert: $h \leq 2 \log (n + 1)$

Ermöglicht: Einfügen, Suchen und Löschen in $\mathcal{O} (\log n)$
$h =$ Höhe, $n =$ Größe

Rot-Schwarz-Bäume

Invarianten:

1. Kein roter Knoten hat roten Elternknoten
2. Alle vollständige Pfade haben gleiche Anzahl schwarzer Knoten
3. Baum ist sortiert:
   Elemente linker Teilbaum $\leq$ Knoten-Element
   Elemente rechter Teilbaum $\geq$ Knoten-Element



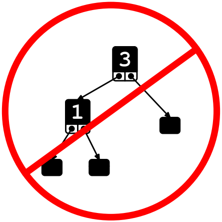
Garantiert: $h \leq 2 \log (n + 1)$

Ermöglicht: Einfügen, Suchen und Löschen in $\mathcal{O}(\log n)$
$h =$ Höhe, $n =$ Größe

Rot-Schwarz-Bäume

Invarianten:

1. Kein roter Knoten hat roten Elternknoten
2. Alle vollständige Pfade haben gleiche Anzahl schwarzer Knoten
3. Baum ist sortiert:
   Elemente linker Teilbaum $\leq$ Knoten-Element
   Elemente rechter Teilbaum $\geq$ Knoten-Element



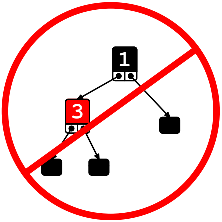
Garantiert: $h \leq 2 \log(n + 1)$

Ermöglicht: Einfügen, Suchen und Löschen in $\mathcal{O}(\log n)$
$h =$ Höhe, $n =$ Größe

Rot-Schwarz-Bäume

Invarianten:

1. Kein roter Knoten hat roten Elternknoten
2. Alle vollständige Pfade haben gleiche Anzahl schwarzer Knoten
3. Baum ist sortiert:
   Elemente linker Teilbaum $\leq$ Knoten-Element
   Elemente rechter Teilbaum $\geq$ Knoten-Element



Garantiert: $h \leq 2 \log(n+1)$

Ermöglicht: Einfügen, Suchen und Löschen in $\mathcal{O}(\log n)$
$h =$ Höhe, $n =$ Größe

Rot-Schwarz Bäume, Implementierung

1. Kein roter Knoten hat roten Elternknoten
2. Alle vollständige Pfade haben gleiche Anzahl schwarzer Knoten
3. Baum ist sortiert

Einfügen in Rot-Schwarz Bäume:

- Ersetze Blatt durch neuen Knoten (Beachte Invariante 3)
- Färbe neue Knoten rot (Invariante 2: ✓)
- Invariante 1: von unten nach oben **rebalancieren**
  - Invarianten 2, 3 aufrechterhalten
  - Invariante 1 höchstens an Wurzel verletzen

```haskell
ins :: (Ord t) => t -> RedBlackTree t -> RedBlackTree t
ins x (Leaf) = Node Red Leaf x Leaf
ins x (Node c left y right)
   | (x<=y)      = let left'  = ins x left  in  balance (Node c left' y right )
   | otherwise   = let right' = ins x right in  balance (Node c left  y right')
```

Rot-Schwarz Bäume, Implementierung

1. Kein roter Knoten hat roten Elternknoten
2. Alle vollständige Pfade haben gleiche Anzahl schwarzer Knoten
3. Baum ist sortiert

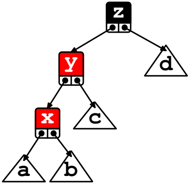
Einfügen in Rot-Schwarz Bäume:

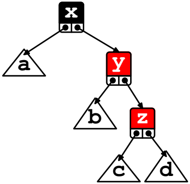
- Ersetze Blatt durch neuen Knoten (Beachte Invariante 3)
- Färbe neue Knoten rot (Invariante 2: ✓)
- Invariante 1: von unten nach oben **rebalancieren**
  - Invarianten 2, 3 aufrechterhalten
  - Invariante 1 höchstens an Wurzel verletzen
  - Schließlich: Wurzel schwarz färben

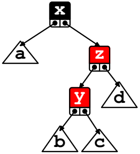
```
ins :: (Ord t) => t -> RedBlackTree t -> RedBlackTree t
ins x (Leaf) = Node Red Leaf x Leaf
ins x (Node c left y right)
   | (x<=y)      = let left'  = ins x left  in  balance (Node c left' y right )
   | otherwise   = let right' = ins x right in  balance (Node c left  y right')


insert x tree = makeBlack (ins x tree)
  where makeBlack (Node c a y b) = Node Black a y b
```

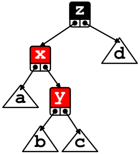
Rot-Schwarz Bäume - Balancierung

Invarianten-Verletzung: mögliche Konfigurationen



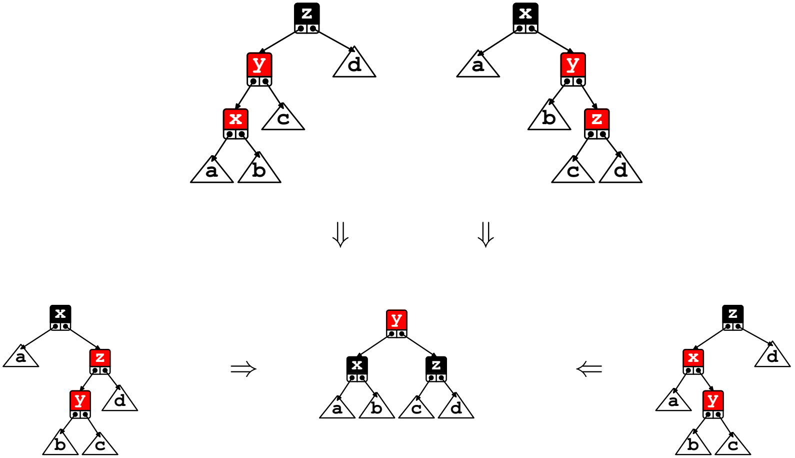




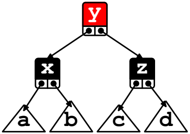


Rot-Schwarz Bäume - Balancierung

Invarianten-Verletzung: Behebung durch Rebalancierung



Rot Schwarz Bäume - Balancierung

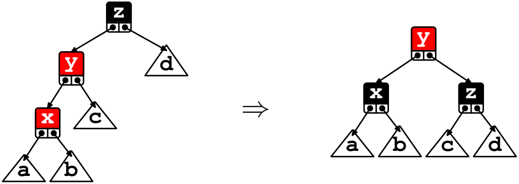


Implementierung: <u>direkt</u> mit Pattern-Matching

- Zielmuster
- Muster der Invariantenverletzung

```
fin a b c d x y z = Node Red (Node Black a x b) y (Node Black c z d)

balance :: RedBlackTree t -> RedBlackTree t
balance (Node Black (Node Red (Node Red a x b) y c) z d) = fin a b c d x y z
balance (Node Black a x (Node Red b y (Node Red c z d))) = fin a b c d x y z
balance (Node Black (Node Red a x (Node Red b y c)) z d) = fin a b c d x y z
balance (Node Black a x (Node Red (Node Red b y c) z d)) = fin a b c d x y z
balance tree = tree
```

Rot Schwarz Bäume - Balancierung



Implementierung: direkt mit Pattern-Matching

- Zielmuster
- Muster der Invariantenverletzung

```
fin a b c d x y z = Node Red (Node Black a x b) y (Node Black c z d)

balance :: RedBlackTree t -> RedBlackTree t
balance (Node Black (Node Red (Node Red a x b) y c) z d) = fin a b c d x y z
balance (Node Black a x (Node Red b y (Node Red c z d))) = fin a b c d x y z
balance (Node Black (Node Red a x (Node Red b y c)) z d) = fin a b c d x y z
balance (Node Black a x (Node Red (Node Red b y c) z d)) = fin a b c d x y z
balance tree = tree
```

Termersetzung

Termersetzung: Allgemeines Berechnungsmodell

- Wiederholte Anwendung von Termersetzungsregeln auf Startterm
- Bis keine Ersetzungsregel mehr anwendbar $\Rightarrow$Ergebnisterm

Ersetzungsregeln zur Berechnung der konjunktiven Normalform

$$
\begin{array}{lcl}
\neg\neg A & \rightarrow_1 & A \\
\neg(A \wedge B) & \rightarrow_2 & \neg A \vee \neg B \\
\neg(A \vee B) & \rightarrow_3 & \neg A \wedge \neg B \\
(A \wedge B) \vee C & \rightarrow_4 & (A \vee C) \wedge (B \vee C) \\
A \vee (B \wedge C) & \rightarrow_5 & (A \vee B) \wedge (A \vee C)
\end{array}
$$

Beispiel-Berechnung:

$$
\begin{array}{lr}
(A \rightarrow B) \rightarrow C \equiv & \neg(\neg A \vee B) \vee C \\
\rightarrow_3 & (\neg\neg A \wedge \neg B) \vee C \\
\rightarrow_4 & (\neg\neg A \vee C) \wedge (\neg B \vee C) \\
\rightarrow_1 & (A \vee C) \wedge (\neg B \vee C)
\end{array}
$$

Boolsche Ausdrücke

Minimale Syntax: $bool ::= var \mid \neg bool \mid bool \wedge bool \mid bool \vee bool$

Als Datentyp: Idee: abstrakter Syntaxbaum

```
data BExp = | Var String | Not BExp  | And BExp BExp | Or BExp BExp
```

Abgeleitete Terme: $bool \rightarrow bool$, true, false

```
implies a b = (Not a) `Or` b
true  = (Not (Var "A")) `Or`  (Var "A")
false = (Not (Var "A")) `And` (Var "A")
```

Beispielterm:

$(A \rightarrow B) \rightarrow C$ $\simeq$ `((Var "A") `implies` (Var "B")) `implies` (Var "C")`
$\Rightarrow^+$ `Or (Not (Or (Not (Var "A")) (Var "B"))) (Var "C")`

Ersetzungsregeln

Ersetzungsregeln zur Berechnung der konjunktiven Normalform

$$
\begin{array}{lll}
\neg\neg A & \rightarrow_1 & A \\
\neg(A \wedge B) & \rightarrow_2 & \neg A \vee \neg B \\
\neg(A \vee B) & \rightarrow_3 & \neg A \wedge \neg B \\
(A \wedge B) \vee C & \rightarrow_4 & (A \vee C) \wedge (B \vee C) \\
A \vee (B \wedge C) & \rightarrow_5 & (A \vee B) \wedge (A \vee C)
\end{array}
$$

Ersetzungsregeln in Haskell

```
rewrite :: BExp -> BExp
rewrite (Not (Not a))          = a
rewrite (Not (a `And` b))      = (Not a) `Or`  (Not b)
rewrite (Not (a `Or`  b))      = (Not a) `And` (Not b)
rewrite ((a `And` b) `Or`  c)  = ( a `Or` c) `And` (b `Or` c)
rewrite ( a `Or` (b  `And` c)) = ( a `Or` b) `And` (a `Or` c)
```

Ersetzungsregeln

Ersetzungsregeln zur Berechnung der konjunktiven Normalform

$$
\begin{array}{lll}
\neg\neg A & \rightarrow_1 & A \\
\neg(A \wedge B) & \rightarrow_2 & \neg A \vee \neg B \\
\neg(A \vee B) & \rightarrow_3 & \neg A \wedge \neg B \\
(A \wedge B) \vee C & \rightarrow_4 & (A \vee C) \wedge (B \vee C) \\
A \vee (B \wedge C) & \rightarrow_5 & (A \vee B) \wedge (A \vee C)
\end{array}
$$

Ersetzungsregeln in Haskell

```
rewrite :: BExp -> BExp
rewrite (Not (Not a))         = a
rewrite (Not (a `And` b))     = (Not a) `Or`  (Not b)
rewrite (Not (a `Or`  b))     = (Not a) `And` (Not b)
rewrite ((a `And` b) `Or`  c) = ( a `Or` c) `And` (b `Or` c)
rewrite ( a `Or` (b  `And` c)) = ( a `Or` b) `And` (a `Or` c)
```

Anwendung auf Teilterme:

```
rewrite (a `And` b) = (rewrite a) `And` (rewrite b)
rewrite (a `Or`  b) = (rewrite a) `Or`  (rewrite b)
rewrite (Not a)     = (Not (rewrite a))
```

Termersetzung

Termersetzung:

- Wiederholte Anwendung von Termersetzungsregeln
- Bis keine Ersetzungsregel mehr anwendbar

Umsetzung in Haskell: als Fixpunkt-Berechung

- Wenn `a` zu keinem Ersetzungs-Muster passt: `rewrite a = a`
- `a` heißt Fixpunkt von `rewrite`

Default-Regel:

```
rewrite a = a
```

Fixpunktberechnung:

```
fix :: (Eq t) => (t -> t) -> t -> t
fix f x
  | (f x == x) = x
  | otherwise  = fix f (f x)
```

KNF Berechnung:

```
cnf :: BExp -> BExp
cnf = fix rewrite
```

Programmfehler

Viele Funktionen sind partiell

- Kein (sinnvoller) Wert für manche Argumente
- Umsetzung in Haskell?

Variante 1: irgendeinen Wert zurückgeben

- Welchen Wert bei polymorphen Funktionen?
- Fehler für Aufrufer nicht erkennbar

⇒ Folgeberechnungen fehlerhaft

```
hd :: [t] -> t
hd (x:xs) = x
hd []     = ???
```

```
f :: Int -> Int
f x
 |(x/=0)     = ...(... `div` x)...
 |otherwise = 0
```

- Versteckt Fehler, nicht immer anwendbar

Programmfehler

Viele Funktionen sind partiell

- Kein (sinnvoller) Wert für manche Argumente
- Umsetzung in Haskell?

Variante 1': irgendeinen Wert zurückgeben

- Wert für ungültige Argumente: vom Aufrufer
- Fehler für Aufrufer nicht erkennbar

$\Rightarrow$ Folgeberechnungen fehlerhaft

```
hd :: t -> [t] -> t
hd onErr (x:xs) = x
hd onErr []     = onErr
```

```
f :: Int -> Int
f onErr x
 |(x/=0)     = ...(... `div` x)...
 |otherwise = onErr
```

- Versteckt Fehler, umständlich

Programmfehler

Viele Funktionen sind partiell

- Kein (sinnvoller) Wert für manche Argumente
- Umsetzung in Haskell?

Variante 2: Laufzeitfehler erzeugen

- Programmabbruch bei ungültigem Argument
- Fehlervermeidung durch Aufrufer

```
hd :: [t] -> t
hd (x:xs) = x
hd []     = error "Nil"
```

```
f :: Int -> Int
f x
 |(x/=0)     = ...(... `div` x)...
 |otherwise = error "Zero"
```

- Geeignet, falls Laufzeitfehler leicht durch Aufrufer auszuschließen

Programmfehler

Viele Funktionen sind partiell

- Kein (sinnvoller) Wert für manche Argumente
- Umsetzung in Haskell?

Variante 3: Fehler-Typ `Maybe`

- Rückgabewert vom Typ `Maybe t`
- Ungültiges Argument: `Nothing`, sonst: `Just y`
- Fehlerbehandlung durch Aufrufer

```
hd :: [t] -> Maybe t
hd (x:xs) = Just x
hd []     = Nothing
```

```
f :: Int -> Maybe Int
f x
 |(x/=0)     = Just ..(..`div` x)..
 |otherwise  = Nothing
```

- Immer wenn Gültigkeit nicht leicht durch Aufrufer festzustellen