## Heutiges Thema

- Von Funktionen höherer Ordnung (Funktionalen) zu Typklassen "höherer Ordnung"
  - Spezielle Typklassen
    - ∗ Functor
    - ∗ Monad
- Anwendung: Eine imperative Sprache zur Robotersteuerung
  ⇝ Beispiel für eine anwendungsspezifische eingebettete Sprache (engl. *domain specific language* (*DSL*))

```
+++ der Einstieg. Weiter beim nächsten Mal ++++++++
```



## Motivation: Roboter und ihre Welt 1(2)

Unsere kleine Roboterwelt...

```
type Grid = Array Position [Direction]

type Position = (Int,Int)

data Direction = North | East | South | West
     deriving (Eq, Show, Enum)

...
```



## Motivation: Roboter und ihre Welt 2(2)

...und ihre Roboter:

```
data RobotState
   = RobotState
       {position :: Position
      , facing :: Direction
      , pen :: Bool
      , color :: Color
      , treasure :: [Position]
      , pocket :: Int
       }
     deriving Show
```

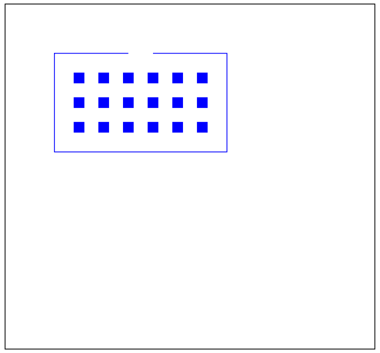


## Ziel: Explorieren und Gestalten

...der kleinen Roboterwelt durch ihre Roboter:

```
drawSquare =           moveToWall =
  do penDown             while (isnt blocked)
     move                do move
     turnRight
     move
     turnRight          getRich =
     move                 while (isnt blocked) $
     turnRight              do move
     move                      checkAndPickCoin
```

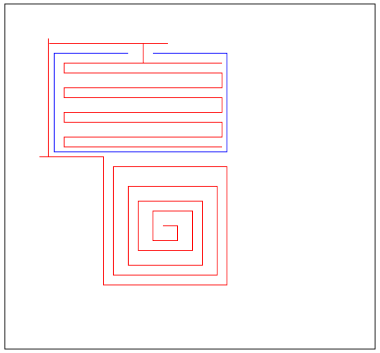
## Illustration: Die Roboterwelt 1(2)



*...die Welt und ihre Schätze.*



## Illustration: Explorieren&Gestalten 2(2)



*...erforschen, Spuren hinterlassen, Schätze einsammeln!*



## Roboter und Monaden

Der Typ `Robot` als Instanz der Klasse `Monad`...

```
newtype Robot a
  = Robot (RobotState -> Grid -> Window -> IO (RobotState,a))

instance Monad Robot where
  Robot sf0 >>= f  = Robot $ \s0 g w -> do
                              (s1,a1) <- sf0 s0 g w
                              let Robot sf1 = f a1
                              (s2,a2) <- sf1 s1 g w
                              return (s2,a2)
  return a         = Robot (\s _ _ -> return(s,a))
```

...deshalb zunächst ein genauerer Blick auf Monaden und relevante Literaturhinweise!



## Literaturhinweise zur heutigen und nächsten Vorlesung


- Kapitel 18 und 19
  Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*, Cambridge University Press, 2000.

- Kapitel 18
  Simon Thompson. *Haskell – The Craft of Functional Programming*, Addison-Wesley, 2nd edition, 1999.

- Kapitel 1, 2 und 10
  Fethi Rabhi, Guy Lapalme. *Algorithms - A Functional Approach*. Addison-Wesley, 1999.

## Rückblick – Funktionale 1(7)

Betrachte folgende Beispiele...

```
-- Fakultaetsfunktion
fact n  | n==0   = 1
        | n>0    = n * fact(n-1)

-- Aufsummieren der n ersten natuerlichen Zahlen
sumNat n | n==0  = 0
         | n>0   = n + sumNat(n-1)

-- Aufsummieren der n ersten Quadratzahlen natuerlicher Zahlen
sumSquNat n | n==0  = 0
            | n>0   = n*n + sumSquNat(n-1)
```



## Rückblick – Funktionale 2(7)

Beobachtung...

- Die Definitionen von `fact`, `sumNat` und `sumSquNat` folgen demselben *Rekursionsschema*.
- Dieses zugrundeliegende gemeinsame Rekursionsschema ist gekennzeichnet durch:
  - Triff eine Festlegung für den Wert der Funktion...
    - ...im *Basisfall*
    - ...im verbleibenden (rekursiven) Fall als *Kombination* des Argumentwerts `n` und des Funktionswerts für `n-1`



## Rückblick – Funktionale 3(7)

Diese Beobachtung legt nahe...

- Obiges Rekursionsschema, gekennzeichnet durch *Basisfall* und *Funktion zur Kombination von Werten*, herauszuziehen (zu abstrahieren) und musterhaft zu realisieren.

Wir erhalten...

- Realisierung des Rekursionsschemas

```
rekPatt base comb n
     | n==0  = base
     | n>0   = comb n (rekPatt base comb(n-1))
```



## Rückblick – Funktionale 4(7)

Unmittelbare Anwendung des Rekursionsschemas...

```
fact n      = rekPatt 1 (*) n

sumNat n    = rekPatt 0 (+) n

sumSquNat n = rekPatt 0 (\x y -> x*x + y) n
```

...bzw. alternativ dazu in nichtargumentbehafteter Ausprägung:

```
fact      = rekPatt 1 (*)

sumNat    = rekPatt 0 (+)

sumSquNat = rekPatt 0 (\x y -> x*x + y)
```

## Rückblick – Funktionale 5(7)

Unmittelbarer Vorteil obigen Vorgehens...

- *Wiederverwendung* und dadurch (hoffentlich)...
  - *kürzerer*, *verlässlicherer*, *wartungsfreundlicherer* Code

Unabdingbar für erfolgreiches Gelingen...

- *Funktionen höherer Ordnung* (bzw. synonym: *Funktionale*)

  *Erinnerung*: Funktionale sind (spezielle) Funktionen, die Funktionen als Argumente erwarten und/oder als Resultat(steil) liefern.



## Rückblick – Funktionale 6(7)

Zusammengefasst und am obigen Beispiel illustriert...

- Die Untersuchung des Typs von `rekPatt`...

  ```
  rekPatt :: Int -> (Int -> Int -> Int) -> Int
  ```

  zeigt:
  - Die Funktion `rekPatt` ist also ein *Funktional*.

In der Anwendungssituation dieses Beispiels gilt weiter...

| | Wert i. Basisf. (base) | Fkt. z. Kb. v. W. (comb) |
|---|---|---|
| `fact` | 1 | `(*)` |
| `sumNat` | 0 | `(+)` |
| `sumSquNat` | 0 | `\x y -> x*x + y` |



## Rückblick – Funktionale 7(7)

*Komplexere Situation*:

Die Funktion `map` auf...

- Listen

  ```
  mapList :: (a -> b) -> [a] -> [b]
  mapList f []     = []
  mapList f (l:ls) = f l : mapList f ls
  ```

- (Binär-) Bäumen

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

  mapTree :: (a -> b) -> Tree a -> Tree b
  mapTree f (Leaf l)       = Leaf (f l)
  mapTree f (Node l lt rt) = (Node f l)
                               (mapTree f lt) (mapTree f rt)
  ```



## Von Funktionen höherer Ordnung zu Typklassen “höherer Ordnung”

*Beobachtung*:

...offenbar liegt auch den Implementierungen von `mapList` und `mapTree` eine gleiche Struktur zugrunde.

*Naheliegende Frage*:

...lässt sich diese Strukturähnlichkeit ausnutzen und mit einer Funktion *“commonMap”* auskommen?

*Die Antwort*:

Ja, die Klasse `Functor` liefert den Schlüssel dazu...

## Die Klasse Functor

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

*Beachte...*

- Innerhalb der Funktion der Klasse `Functor` Anwendung des Typkonstruktors auf andere Typen, genauer: der Typvariablen `f` auf die Typvariablen `a` und `b`.
- Eine Situation vertraut von der Klasse `Monad`, ansonsten neu.



## Zur Erinnerung: Die Klasse Monad

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a

  m >> k = m >>= \_ -> k    -- vordefiniert
  fail s = error s          -- vordefiniert
```

*Beachte...*

- Innerhalb der Funktionen der Klasse `Monad` Anwendung des Typkonstruktors auf andere Typen, genauer: der Typvariablen `m` auf die Typvariablen `a` und `b`.



## Zum Vergleich: Die Klasse Eq

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y     = not (x==y)
  x == y     = not (x/=y)

data Tree a = Leaf a | Node a (Tree a) (Tree a)

instance Eq a => Eq (Tree a) where
  Leaf a == Leaf b              =  a==b
  Node a l1 r1 == Node b l2 r2  =  a==b && l1==l2 && r1==r2
  _ == _                        =  False
```

*Beachte...*

- In den Funktionen der Klasse `Eq` keine Anwendung von Typkonstruktoren auf Typen oder Typvariablen.
- Gilt analog auch für andere von uns betrachtete Typklassen wie `Ord`, `Num`, `Fractional`, etc.



## Bäume und Listen als Instanzen der Klasse Functor 1(3)

```
instance Functor Tree where
  fmap g (Leaf l)       = Leaf (g l)
  fmap g (Node l rt lt) = (Node (g l)) (fmap g lt) (fmap g rt)


instance Functor [] where
  fmap g []     = []
  fmap g (l:ls) = g l : fmap g ls
```

*Beachte*: ...[] wird in obiger Instanzdeklaration einmal als Typkonstruktor und einmal als Wert eines Listentyps verwendet.

## Bäume und Listen als Instanzen der Klasse Functor 2(3)

Gleichbedeutend und kürzer erhalten wir...

```
instance Functor Tree where
  fmap = mapTree

instance Functor [] where
  fmap = mapList            -- selbst definiertes mapList

instance Functor [] where
  fmap = map                -- vordefiniertes map
```



## Bäume und Listen als Instanzen der Klasse Functor 3(3)

Insgesamt erhalten wir...

- `fmap` der Klasse `functor` ist unser gesuchtes *"common-Map"*.

  ⇝ ...mit vorausgegangenen Deklarationen ist möglich:

```
fmap (*2) (Node 2 (Leaf 3 Leaf 5))
                   =>  (Node 4 (Leaf 6 Leaf 10))

fmap (*2) [2,3,5]  =>  [4,6,10]
```



## (Standard-) Typkonstruktoren

Beispiele einiger (Standard-) Typkonstruktoren sind...

- `(,)`,`(,,)`,`(,,,)`, usw. ...für Tupel (genauer: Tupeltypen)
- `[]` ...für Listen(typen)
- `(->)` ...für Funktionen/Funktionstypen

*Beachte* ...folgende (Typ-) Schreibweisen sind hier äquivalent:

- `(a,b)` und `(,) a b`, `(a,b,c)` und `(,,) a b c`, usw.
- `[a]` und `[] a`
- `f -> g` und `(->) f g`
- Darüberhinaus gilt generell: `T a b` entspricht `((T a) b)` (d.h. Linksassoziativität wie bei Funktionsanwendungen)

## Beispiel

Die Signatur der Funktionen...

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)

list2pair :: [a] -> (a,a)
list2pair (x : (y : _ )) = (x,y)
list2pair (x : _)        = (x,x)
```

...kann auch mit jeder der folgenden Varianten erfolgen:

```
fac :: (->) Int Int

list2pair :: [] a -> (a,a)
list2pair :: [a] -> (,) a a
list2pair :: (->) [a] (a,a)
...
list2pair :: [] a -> (,) a a
list2pair :: (->) ([] a) ((,) a a)
```

...übersichtlicher (oder nur gewohnter?) ist sicherlich die "Standard"-Variante.

# Für Functor einzuhaltende Gesetze

...für Instanzen der Klasse `Functor` wird die Einhaltung folgender *Gesetze* gefordert:

```
fmap id    = id
fmap (g.h) = fmap g . fmap h
```

*Intuitive Bedeutung...*

- Die "Gestalt des Containertyps" bleibt erhalten.
- Containerinhalte werden nicht umgruppiert.

*Beachte...*

- Ob diese Gesetze von einer konkreten Instanz eingehalten sind, z.B. für Bäume und Listen, ist jeweils zu beweisen ($\rightsquigarrow$ *Beweisverpflichtung für den Programmierer!*)



# Zur Einhaltungsforderung von Gesetzen

...ist sinnvoll und (für alle Klassen) üblich.

$\rightsquigarrow$ ...zur Illustration betrachte die Klasse `Eq` mit ihrem Gleichheitstest:

```
newtype Tuple = Tpl (Int, Int)

instance Eq Tuple where
  Tpl (x1,x2) == Tpl (y1,y2) = (x1==y1) || (x2==y2)
            _ == _           = False
```

...wäre irritierende Implementierung des Gleichheitstests auf Tupeln wegen (für Gleichheit gemeinhin unterstellter, aber hier) fehlender Transitivität. Betrachte:

```
Tpl (2,3) == Tpl (2,5)  =>  True
Tpl (2,5) == Tpl (3,5)  =>  True

Tpl (2,3) == Tpl (3,5)  =>  False   (also nicht transitiv!)
```



# Die Typklasse Monad

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a

  m >> k = m >>= \_ -> k     -- vordefiniert
  fail s = error s           -- vordefiniert
```

*Intuitiv...*

- ...beschreiben Aktionen mit Seiteneffekten
- ...erlauben Berechnungsreihenfolgen festzulegen
- ...unterstützen dadurch einen imperativ-artigen Programmierstil (ohne das funktionale Paradigma zu brechen)



# Zu den Funktionen der Klasse Monad

*Intuitiv...*

- (>>=): ...der Sequenzoperator (gelesen als *then* (nach Simon Thompson) bzw. als *bind* (nach Paul Hudak)).
- `return`: ...Resultatrückgabe ohne Seiteneffekt
- (>>): ...von (>>=) abgeleiteter Sequenzoperator (gelesen als *sequence* (nach Paul Hudak)).
- `fail`: ...Fehlerbehandlung

## Segensreiche notationelle Alternative

...die do-Notation: abkürzende Schreibweise monadischer Operatoren!

Es gilt:

```
              do e      =>  e

      do e1;e2;...;en      =>  e1 >> do e2;...;en

  do let decllist;e2;...;en =>  let decllist in do e2;...;en

  do pat <- e1;e2;...;en    =>  let ok pat = do e2;...;en
                                    ok _   = fail "..."
                                in e1 >>= ok
```

...und als Spezialfall der "Musterregel" "do pat <-...":

```
   do x <- e1;e2;...;en     =>  e1 >>= \x -> do e2;...;en
```



## Ein Beispiel 1(2)

Die Ausgabesequenz

```
do writeFile "testFile.txt" "Hello File System!"
   putStr "Hello World!"
```

...ist äquivalent zu:

```
writeFile "testFile.txt" "Hello File System!" >>
putStr "Hello World!"
```



## Ein Beispiel 2(2)

Anmerkung zur Funktionalität...

Aus

```
(>>) :: Monad m => m a -> m b -> m b
```

und

```
writeFile "testFile.txt" "Hello File System!" :: IO ()
putStr "Hello World!"                          :: IO ()
```

...lässt sich für den Fall des Beispiels schließen: `m = IO`, `a = ()` und `b = ()`. Und somit insgesamt:

```
(>>) :: IO () -> IO () -> IO ()
```



## Beispiel zu lokalen Deklarationen...

...innerhalb von do-Konstrukten.

Die Formulierung

```
reverse2lines :: IO ()
reverse2lines
   = do line1 <- getLine
        line2 <- getLine
        let rev1 = reverse line1
        let rev2 = reverse line2
        putStrLn rev2
        putStrLn rev1
```

ist äquivalent zu:

```
reverse2lines :: IO ()
reverse2lines
   = do line1 <- getLine
        line2 <- getLine
        putStrLn (reverse line2)
        putStrLn (reverse line1)
```

## Für Monad einzuhaltende Gesetze 1(3)

```
return a >>= f              =  f a
c >>= return                =  c
c >>= (\x -> (f x) >>= g)   =  (c >>= f) >>= g
```

*Intuitiv...*

- *1. Gesetz*: ...`return` reicht den Wert an die nächste Aktion weiter, ohne weiteren Effekt.
- *2. Gesetz*: ...ein unmittelbar zurückgegebenes Resultat einer Aktion kann gleich von dieser Aktion zurückgegeben werden.
- *3. Gesetz*: ...Assoziativität des Sequenzoperators (`>>=`).



## Für Monad einzuhaltende Gesetze 2(3)

Alternativ in `do`-Notation:

```
do x <- return a; f x  =  f x
do x <- c; return x    =  c
do x <- c; y <- f x; g y  =  do y <- (do x <- c; f x); g y
```

Zum einfacheren Vergleich noch einmal wiederholt:

```
return a >>= f              =  f a
c >>= return                =  c
c >>= (\x -> (f x) >>= g)   =  (c >>= f) >>= g
```



## Für Monad einzuhaltende Gesetze 3(3)

Offensichtlicher ist die Assoziativitätseigenschaft für (`>>`):

```
c1 >> (c2 >> c3)  =  (c1 >> c2) >> c3
```

Für Instanzen von beiden Klassen `Monad` und `Functor` wird zusätzlich gefordert:

```
fmap g xs  =  xs >>= return . g

           =  do x <- xs; return (g x)
```



## Weitere Instanzen von Monad: Maybe

`Maybe` vordefiniert als Instanz der Klassen `Monad` und `Functor`:

```
data Maybe a = Nothing | Just a deriving (Eq, Ord, Read, Show)

instance Monad Maybe where
  Just x >>= k  = k x
  Nothing >>= k = Nothing
  return        = Just
  fail s        = Nothing

instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

*Beachte*: ...für die Typen der Monadenoperatoren im Falle `Maybe` gilt:

```
(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
return :: a -> Maybe a
```

## Anwendung der Monade Maybe 1(4)

Betrachte `g (f x)` mit

```
f :: Int -> Int
g :: Int -> Int
x :: Int
```

...unter der Annahme, dass die Berechnungen von `f` und `g` fehlschlagen können.

Zur Lösung folgende Einbettung in Typ `Maybe` möglich:

```
case (f x) of
  Nothing -> Nothing
  Just y  -> case (g y) of
               Nothing -> Nothing
               Just z  -> z
```

...möglich, aber "unbequem".



## Anwendung der Monade Maybe 2(4)

Einfacher wird es unter Ausnutzung von `Maybe`'s Mitgliedschaft in `Monad`:

```
f x >>= \y ->
g y >>= \z ->
return z
```

bzw. vermöge der `do`-Notation...

```
do y <- f x
   z <- g y
   return z
```

...die "lästige" Fehlerabfrage ist "versteckt" in der Monade.



## Anwendung der Monade Maybe 3(4)

Auch folgende Vereinfachung ist möglich...

```
 f x >>= \y ->
 g y >>= \z ->
 return z
   => { (\z -> return z) z' entspricht return z'}
 f x >>= \y ->
 g y >>= return
   => { 2. Monadengesetz fuer return }
f x >>= \y ->
 g y
  => { (\y -> g y) y' entspricht g y' }
f x >>= g
```

Aus `g (f x)` wurde somit `f x >>= g`



## Anwendung der Monade Maybe 4(4)

Eine weitere gute Option...

```
composeM :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)
(g 'composeM' f) x = f x >>= g
```

Unter Ausnutzung dieser Option erhalten wir:

...aus `(g . f) x` wird `(g 'composeM' f) x`

*Beachte*: ...in beiden Varianten bleibt die ursprüngliche Schreibweise nahezu 1-zu-1 erhalten.

## Kleine Beispiele: Teil I 1(4)

Eine konkrete Ausprägung des vorbeschriebenen Szenarios...

```
double :: Int -> Int
double n
  | (n>=0)  =  2*n

triple :: Int -> Int
triple n
  | (n>=0)  =  3*n

-- Gewuenscht:  (triple . double) n = triple (double n)

err_double :: Int -> Maybe Int
err_double n
  | (n>=0)    = Just (double n)
  | otherwise = Nothing

err_triple :: Int -> Maybe Int
err_triple n
  | (n>=0)    = Just (triple n)
  | otherwise = Nothing
```



## Kleine Beispiele: Teil I 2(4)

Betrachte folgende Testfälle:

```
test1  n = err_double n >>= \x -> err_triple x >>= \y -> return y

test1a n = err_double n >>= \x -> err_triple x >>= return

test1b n = do x <- err_double n
              y <- err_triple x
              return y
```

Wir erhalten:

```
test1  5 => Just 30      -- test1  :: Int -> Maybe Int
test1a 5 => Just 30      -- test1a :: Int -> Maybe Int
test1b 5 => Just 30      -- test1b :: Int -> Maybe Int
```



## Kleine Beispiele: Teil I 3(4)

```
test2 n = err_double n >>= \x -> err_triple x >>= \y -> return x

test3 n = err_double n >>= \x -> err_triple x >>= \y -> return (x,y)

test4 n = do x <- err_double n
             y <- err_triple x
             return [y,x]

test5 n = err_double n >>= \x -> err_triple x >>= \y -> return
                                     [(x,y,x),(y,y,x)]

test6 n = err_double n >>= \x -> err_triple x >>= \y -> return
                                     [(x,y,x+y),(fac x,x-y,x*y)]

test7 n = err_double n >>= \x -> err_triple x >>= \y -> return
                                     ((x,y,x+y),(x-y,x*y))

test8 n = err_double n >>= \x -> err_triple x >>= \y -> return 'c'
```



## Kleine Beispiele: Teil I 4(4)

Damit erhalten wir...

```
test2 :: Int -> Maybe Int
test3 :: Int -> Maybe (Int,Int)
test4 :: Int -> Maybe [Int]
test5 :: Int -> Maybe [(Int,Int,Int)]
test6 :: Int -> Maybe [(Int,Int,Int)]
test7 :: Int -> Maybe ((Int,Int,Int),(Int,Int))
test8 :: Int -> Maybe Char
```

...und

```
test2 5 => Just 10
test3 5 => Just (10,30)
test4 5 => Just [30,10]
test5 5 => Just [(10,30,10),(30,30,10)]
test6 5 => Just [(10,30,40),(3628800,-20,300)]
test7 5 => Just ((10,30,40),(-20,300))
test8 5 => Just 'c'
```

## Kleine Beispiele: Teil II 1(4)

*Beachte*: ...nur für `m` liegt die Entscheidung fest, nicht für `a`!

```
mkChr :: Int -> Char
mkChr 0 = 'a'
mkChr 1 = 'b'

mkStr :: Char -> String
mkStr 'a' = "Hello!"
mkStr 'b' = "Good Bye!"

-- Gewuenscht:  (mkStr . mkChar) n = mkStr (mkChar n)

err_mkChr :: Int -> Maybe Char
err_mkChr n
  | ((n==0) || (n==1)) = Just (mkChr n)
  | otherwise          = Nothing

err_mkStr :: Char -> Maybe String
err_mkStr c
  | ((c=='a') || (c=='b')) = Just (mkStr c)
  | otherwise              = Nothing
```



## Kleine Beispiele: Teil II 2(2)

Betrachte zusätzlich...

```
goFloat :: Int -> Float
goFloat n
  | n < 7     = -3.14
  | otherwise = 3.14
```

Dann erhalten wir für...

```
test n = do x <- err_mkChr n
            y <- err_mkStr x
            return (goFloat (length y))
```

...folgendes:

```
test :: Int -> Maybe Float

test 0 = Just (-3.14)
test 1 = Just 3.14
test 2 = Nothing
```



## Kleine Beispiele: Teil III 1(2)

Ein letztes Beispiel: "komplexere" Typen von Anfang an.

```
doubleP :: Int -> (Int,Int)
doubleP n
  | (n>=0)  =  (n,2*n)

tripleP :: Int -> (Int,Int)
tripleP n
  | (n>=0)  =  (n,3*n)

-- Gewuenscht:  (tripleP . fst . doubleP) n = tripleP (fst (doubleP n))

err_doubleP :: Int -> Maybe (Int,Int)
err_doubleP n
  | (n>=0)    = Just (doubleP n)
  | otherwise = Nothing

err_tripleP :: Int -> Maybe (Int,Int)
err_tripleP n
  | (n>=0)    = Just (tripleP n)
  | otherwise = Nothing
```



## Kleine Beispiele: Teil III 2(2)

Betrachte als Tests...

```
test n = do x <- err_doubleP n
            y <- err_tripleP (fst x)
            return ([(x,y)],snd x)

test1 n = err_doubleP n >>= \x -> err_tripleP (fst x) >>=
                                  \y -> return ([(x,y)],snd x)
```

...wofür wir erhalten:

```
test :: Int -> Maybe ([((Int,Int),(Int,Int))],Int)

test 5    => Just ([((5,10),(5,15))],10)
test (-3) => Nothing
```

...was sich in genau gleicher Weise auch für `test1` ergibt.

# Listen als Monade: Die Monade []

Der Listenkonstrukor als Instanz der Klasse `Monad`...

```
instance Monad [] where
  m >>= k  = concat (map k m)
  return x = [x]
  fail x   = []
```

...mit `concat` aus dem Standard Prelude:

```
concat     :: [[a]] -> [a]
concat lss = foldr (++) [] lss
```

*Beachte*: ...für diese Instanz von `Monad` ergibt sich für die Typen der monadischen Operatoren:

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



# Anwendung der Monade [] 1(4)

...Listenkomprehension kann als Spielart monadischer Syntax aufgefasst werden.

Sowohl

```
[(x,y) | x <- [1,2,3], y <- [4,5,6] ]
```

als auch

```
do x <- [1,2,3]
   y <- [4,5,6]
   return (x,y)
```

gleichbedeutend mit

```
[1,2,3] >>= \x -> [4,5,6] >>= \y -> return (x,y)
```

liefern

```
[(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]
```



# Anwendung der Monade [] 2(4)

Generell gilt:

```
do x <- xs; return (f x)
```

...ist äquivalent zu

```
[f x | x <- xs]
```

*Beachte*:

- Die Äquivalenz von `[f x | x <- xs]` und `map f xs` liefert (für die Instanz [] von `Functor`) eine gute Motivation für das Gesetz:

```
fmap f xs = do x <- xs; return (f x)
```

- `Maybe` als Monade verhält sich ähnlich einer "abgeschnittenen" Liste: `Nothing` entspricht `[]`, `Just x` entspricht `[x]`.



# Anwendung der Monade [] 3(4)

Welche Ausgaben liefern folgende Sequenzen?

```
do x <- [1,2]; y <- [4,5,6]; return x
        => ...

do x <- [1,2]; y <- [4,5,6]; return y
        => ...

do x <- [1,2,3]; y <- [4,5,6]; return [x]
        => ...

do x <- [1,2,3]; y <- [4,5,6]; return [y]
        => ...

do x <- [1,2]; y <- [4,5,6]; return (x,y)
        => ...

do x <- [1,2,3]; y <- [4,5,6,42]; return x
        => ...

do x <- [1,2,3]; y <- [4,5,6,42]; return y
        => ...
```

## Anwendung der Monade [] 4(4)

Hier die Antwort:

```
do x <- [1,2]; y <- [4,5,6]; return x
        => [1,1,1,2,2,2]

do x <- [1,2]; y <- [4,5,6]; return y
        => [4,5,6,4,5,6]

do x <- [1,2,3]; y <- [4,5,6]; return [x]
        => [[1],[1],[1],[2],[2],[2],[3],[3],[3]]

do x <- [1,2,3]; y <- [4,5,6]; return [y]
        => [[4],[5],[6],[4],[5],[6],[4],[5],[6]]

do x <- [1,2]; y <- [4,5,6]; return (x,y)
        => [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6)]

do x <- [1,2,3]; y <- [4,5,6,42]; return x
        => [1,1,1,1,2,2,2,2,3,3,3,3]

do x <- [1,2,3]; y <- [4,5,6,42]; return y
        => [4,5,6,42,4,5,6,42,4,5,6,42]
```



## Vergleich Listenkomprehension 1(3)

Betrachte folgende Listenkomprehensionsausdrücke zum Vergleich:

```
[ (x,y) | x <- [1,2,3], y <- [4,5,6]]
      => ...

[ x | x <- [1,2], y <- [4,5,6]]
      => ...

[ y | x <- [1,2], y <- [4,5,6]]
      => ...

[ [x] | x <- [1,2,3], y <- [4,5,6]]
      => ...

[ [y] | x <- [1,2,3], y <- [4,5,6]]
      => ...

[ (x,y)| x <- [1,2], y <- [4,5,6]]
      => [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6)]

[ x | x <- [1,2,3], y <- [4,5,6,42]]
      => ...

[ y | x <- [1,2,3], y <- [4,5,6,42]]
      => ...
```

## Vergleich Listenkomprehension 2(3)

Wie für die entsprechenden `do`-Ausdrücke erhalten wir:

```
[ (x,y) | x <- [1,2,3], y <- [4,5,6]]
      => [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]

[ x | x <- [1,2], y <- [4,5,6]]
      => [1,1,1,2,2,2]

[ y | x <- [1,2], y <- [4,5,6]]
      => [4,5,6,4,5,6]

[ [x] | x <- [1,2,3], y <- [4,5,6]]
      => [[1],[1],[1],[2],[2],[2],[3],[3],[3]]

[ [y] | x <- [1,2,3], y <- [4,5,6]]
      => [[4],[5],[6],[4],[5],[6],[4],[5],[6]]

[ (x,y)| x <- [1,2], y <- [4,5,6]]
      => [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6)]

[ x | x <- [1,2,3], y <- [4,5,6,42]]
      => [1,1,1,1,2,2,2,2,3,3,3,3]

[ y | x <- [1,2,3], y <- [4,5,6,42]]
      => [4,5,6,42,4,5,6,42,4,5,6,42]
```

## Vergleich Listenkomprehension 3(3)

Listenkomprehensionen lassen sich mithilfe von `map`, `filter` und `concat` ausdrücken:

Es gilt:

```
[ x | x <- xs ]                = xs

[ f x | x <- xs ]              = map f xs

[ e | x <- xs, p x, ... ]      = [ e | x <- filter p xs, ... ]

[ e | x <- xs, y <- ys, ... ] = concat [ [ e | y <- ys, ... ] | x <- xs ]
```

...mit `concat` aus dem Standard Prelude:

```
concat     :: [[a]] -> [a]
concat lss = foldr (++) [] lss
```

## Beispiel

Durch Nachrechnen erhalten wir:

```
[ x | x <- [1,2], y <- [4,5,6] ]
    => concat [ [ x | y<-[4,5,6] | x<-[1,2] ]
    => concat [ [1,1,1], [2,2,2] ]
    => [1,1,1,2,2,2]

[ y | x <- [1,2], y <- [4,5,6] ]
    => concat [ [ y | y<-[4,5,6] | x<-[1,2] ]
    => concat [ [4,5,6], [4,5,6] ]
    => [4,5,6,4,5,6]
```



## Einige weitere Beispiele 1(4)

```
do x <- [1,2]; y <- x ++ [4,5,6]; return (x,y)
        => ...

do x <- [1,2]; y <- [x] ++ [4,5,6]; return (x,y)
        => ...

do x <- [1,2]; y <- x:[4,5,6]; return (x,y)
        => ...
```



## Einige weitere Beispiele 2(4)

```
do x <- [1,2]; y <- x ++ [4,5,6]; return (x,y)
        => Typfehler!

do x <- [1,2]; y <- [x] ++ [4,5,6]; return (x,y)
        => "ok"

do x <- [1,2]; y <- x:[4,5,6]; return (x,y)
        => "ok"
```



## Einige weitere Beispiele 3(4)

```
do x <- [1,2]; y <- [x] ++ [4,5,6]; return x
       => ...

do x <- [1,2]; y <- [x] ++ [4,5,6]; return y
       => ...

do x <- [1,2]; y <- [x] ++ [4,5,6]; return (x,y)
       => ...

do x <- [1,2]; y <- x:[4,5,6]; return y
       => ...

do x <- [1,2]; y <- x:(x:[4,5,6]); return y
       => ...

do x <- [1,2]; y <- x:(x:[4,5,6]); return (x,y)
       => ...

[1,2,3] >>= \x -> [4,5]++[x]++[6] >>= \y -> return (x,y)
       => ...
```

## Einige weitere Beispiele 4(4)

```
do x <- [1,2]; y <- [x] ++ [4,5,6]; return x
        => [1,1,1,1,2,2,2,2]

do x <- [1,2]; y <- [x] ++ [4,5,6]; return y
        => [1,4,5,6,2,4,5,6]

do x <- [1,2]; y <- [x] ++ [4,5,6]; return (x,y)
        => [(1,1),(1,4),(1,5),(1,6),(2,2),(2,4),(2,5),(2,6)]

do x <- [1,2]; y <- x:[4,5,6]; return y
        => [1,4,5,6,2,4,5,6]

do x <- [1,2]; y <- x:(x:[4,5,6]); return y
        => [1,1,4,5,6,2,2,4,5,6]

do x <- [1,2]; y <- x:(x:[4,5,6]); return (x,y)
        => [(1,1),(1,1),(1,4),(1,5),(1,6),(2,2),(2,2),(2,4),(2,5),(2,6)]

[1,2,3] >>= \x -> [4,5]++[x]++[6] >>= \y -> return (x,y)
 =>[(1,4),(1,5),(1,1),(1,6),(2,4),(2,5),(2,2),(2,6),(3,4),(3,5),(3,3),(3,6)]
```



## Übungsaufgabe

Überzeugen Sie sich durch Nachrechnen:

```
[1,2,3] >>= \x -> [4,5,6] >>= \y -> return (x,y)

   => ... ( Definitionen von >>=, map, concat, return, ... )

[(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]
```

*...there is no magic. It is all just functional programming!*



## Zustandsbehaftete Monaden

...Zustandsmonaden (engl. *State Monads*) erlauben *zustandsbehaftete*, *imperative* Berechnungen zu simulieren.

Generelles Muster:

```
data SM a = SM (S -> (S,a))

instance Monad SM where
  return a
     = SM (\s -> (s,a))
  SM sm0 >>= fsm1
     = SM $ \s0 ->
         let (s1,a1) = sm0 s0
             SM sm1  = fsm1 a1
             (s2,a2) = sm1 s1
         in (s2,a2)
```



## Der Operator $

...der im Standard Prelude definierte Operator $:

```
($)   :: (a -> b) -> a -> b
f $ x = f x
```

...entspricht dem (Funktions-) Applikationsoperator.

Der Operator $ ist nützlich, um Klammern einzusparen:

```
f (g x)
```

...ist äquivalent zu

```
f $ g x
```

## Anwendung von Monaden: 1. Beispiel

*Aufgabe*: ...ersetze die Benennungen der Blätter in einem Zeichenbaum durch fortlaufende natürliche Zahlen.

*Gegeben*:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
  deriving Show
```



## 1. Beispiel: Zur Illustration 1(2)

Angesetzt auf

```
Branch (Leaf 'a') (Leaf 'b')
```

...soll `test` mit

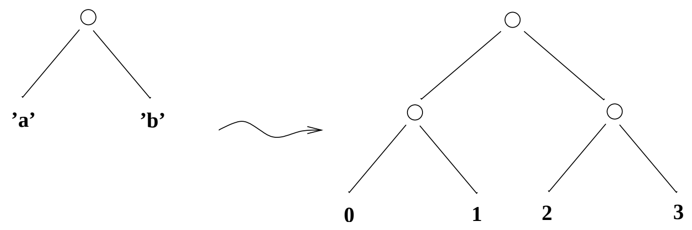
```
test = let t = Branch (Leaf 'a') (Leaf 'b')
       in label (Branch t t)
```

...übereinstimmen mit

```
Branch (Branch (Leaf 0) (Leaf 1))
       (Branch (Leaf 2) (Leaf 3))
```



## 1. Beispiel: Zur Illustration 2(2)






## Erste Lösungsvariante: Monadenfrei

...Standardlösung, ohne Monaden:

```
label   :: Tree a -> Tree Integer
label t = snd (lab t 0)

lab :: Tree a -> Integer -> (Integer, Tree Integer)
lab (Leaf a) n
    = (n+1, Leaf n)
lab (Branch t1 t2) n
    = let (n1,t1') = lab t1 n
          (n2,t2') = lab t2 n1
      in (n2, Branch t1' t2')
```

...einfach, aber "umständliches" Durchreichen des Werts von `n` durch die Inkarnationen von `lab`.

## Zweite Lösungsvariante: Monadisch 1(3)

...mit Zustandsmonade:

```
newtype Label a = Label (Integer -> (Integer,a))
```

...“passt” auf das Muster `SM` der Zustandsmonade.

Wir deklarieren:

```
instance Monad Label where
  return a
    = Label (\s -> (s,a))
  Label lt0 >>= flt1
    = Label $ \s0 ->
        let (s1,a1)   = lt0 s0
            Label lt1 = flt1 a1
        in lt1 s1
```

*Bem.*: Der $-Operator in der Definition von `(>>=)` kann entfallen, wenn im Gegenzug der Ausdruck `\s0 -> let ... in lt1 s1` geklammert wird.

## Zweite Lösungsvariante: Monadisch 2(3)

...womit die Umbennung monadisch wie folgt möglich wird:

```
mlabel   :: Tree a -> Tree Integer
mlabel t = let Label lt = mlab t
           in snd (lt 0)

mlab :: Tree a -> Label (Tree Integer)
mlab (Leaf a)
     = do n <- getLabel
          return (Leaf n)
mlab (Branch t1 t2)
     = do t1' <- mlab t1
          t2' <- mlab t2
          return (Branch t1' t2')

getLabel :: Label Integer
getLabel = Label (\n -> (n+1,n))
```



## Zweite Lösungsvariante: Monadisch 3(3)

Wir erhalten:

`mtest` mit

```
mtest = let t = Branch (Leaf 'a') (Leaf 'b')
        in mlabel (Branch t t)
```

In anderen Worten, `mlabel` angesetzt auf

```
Branch (Leaf 'a') (Leaf 'b')
```

...stimmt wie gewünscht überein mit:

```
Branch (Branch (Leaf 0) (Leaf 1))
       (Branch (Leaf 2) (Leaf 3))
```



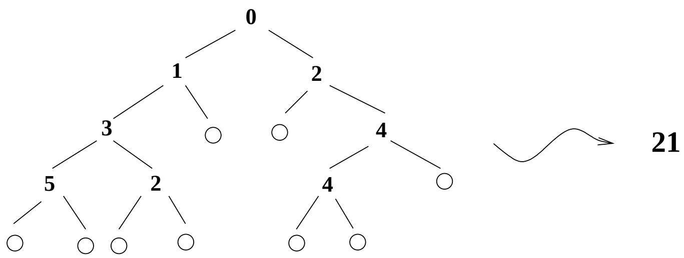
## Anwendung von Monaden: 2. Beispiel

*Aufgabe*: ...summiere die Benennungen der Knoten in einem Ganzzahlbaum.

*Gegeben*:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
  deriving Show
```

## 2. Beispiel: Zur Illustration






## Erste Lösungsvariante: Monadenfrei

...Standardlösung, ohne Monaden:

```
sTree :: Tree Int -> Int
sTree Nil             = 0
sTree (Node n t1 t2) = n + sTree t1 + sTree t2
```



## Zweite Lösungsvariante: Monadisch 1(3)

...mit Monade (keine Zustandsmonade!):

```
sumTree :: Tree Int -> St Int
sumTree Nil            = return 0
sumTree (Node n t1 t2) = do num <- return n
                            s1  <- sumTree t1
                            s2  <- sumTree t2
                            return (num + s1 + s2)
```

...mit `St` geeignet als Instanz von `Monad` vorausgesetzt.



## Zweite Lösungsvariante: Monadisch 2(3)

Tatsächlich reicht in diesem Beispiel die Wahl der Identitätsmonade `Id` für `St`.

Mit dieser Wahl erhalten wir insgesamt:

```
data Id a = Id a

instance Monad Id where
  return         = Id
  (>>=) (Id x) f = f x
```

mit

```
sumTree :: Tree Int -> Id Int
```

(statt ursprünglich `sumTree :: Tree Int -> St Int`).

# Zweite Lösungsvariante: Monadisch 3(3)

Mittels einer Extraktionsfunktion

```
extract :: Id a -> a
extract (Id x) = x
```

...erhalten wir schließlich eine Funktion des Typs `Tree Int -> Int`:

```
extract . sumTree :: Tree Int -> Int
```

*Bsp.*:

```
(extract . sumTree) (Node 5 (Node 3 Nil Nil) (Node 7 Nil Nil))
        =>
 extract (sumTree (Node 5 (Node 3 Nil Nil) (Node 7 Nil Nil)))
        =>
 extract (Id 15)
        =>
 15
```



# Ein Vergleich imperative/monadische Lösung

...offenbart die große Ähnlichkeit der monadischen mit einer imperativen Lösung:

```
Imperativ                    Monadisch


num := n;                    do num <- return n
s1  := sumTree t1;              s1  <- sumTree t1
s2  := sumTree t2;              s2  <- sumTree t2
return (num + s1 + s2);         return (num + s1 + s2)
```



# Anwendung von Monaden: 3. Beispiel

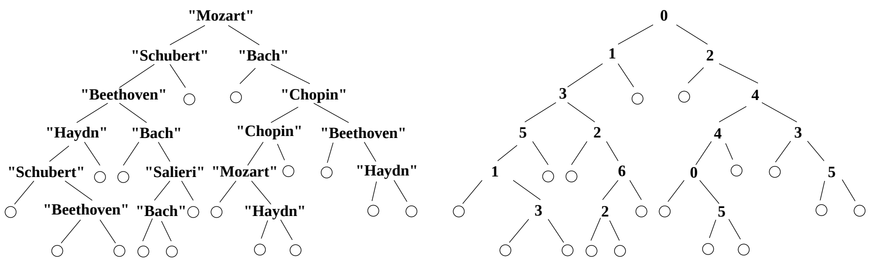
*Aufgabe*: ...ersetze gleiche Benennungen in einem Zeichenreihenbaum durch gleiche natürliche Zahlen.

*Gegeben*:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
  deriving Show
```



# 3. Beispiel: Zur Illustration



## Monadische Lösung 1(5)

Finales Ziel: ...eine Funktion `numTree` mit

```
numTree :: Eq a => Tree a -> Tree Int
```

Unterwegs zu dieser Lösung mithilfe monadischer Programmierung (mit Zustandsmonade)...

```
numberTree :: Eq a => Tree a -> State a (Tree Int)
numberTree Nil
  = return Nil
numberTree (Node x t1 t2)
  = do num <- numberNode x
       nt1 <- numberTree t1
       nt2 <- numberTree t2
       return (Node num nt1 nt2)
```



## Monadische Lösung 2(5)

Speicherung der Paare

(<Zeichenreihe>,<Anzahl Vorkommen>)

in einer Tabelle:

```
type Table a = [a]
```

*Speziell*: Die Tabelle

```
[True, False]
```

gibt an, dass `True` mit `0` und `False` mit `1` assoziiert ist.



## Monadische Lösung 3(5)

Festlegung von Typ und Monadeninstanz:

(*Intuition*: "Wert vom Typ b: Resultat. Änderung der Tabelle: Seiteneffekt)

```
data State a b = State (Table a -> (Table a, b))

instance Monad (State a) where
(State st) >>= f
   = State (\tab -> let
                    (newTab,y)    = st tab
                    (State trans) = f y
                    in
                    trans newTab)
 return x = State (\tab -> (tab,x))
```



## Monadische Lösung 4(5)

Festlegung der noch fehlenden Funktion `numberNode`:

```
numberNode :: Eq a => a -> State a Int
numberNode x = State (nNode x)

nNode :: Eq a => a -> (Table a -> (Table a, Int))
nNode x table
  | elem x table = (table,      lookup x table)
  | otherwise    = (table++[x], length table)
-- nNode liefert die Position von x in der Tabelle:
-- mittels lookup, falls vorhanden; mittels length
-- nach Anhaengen von x an die Tabelle anderenfalls

lookup :: Eq a => a -> Table a -> Int
lookup ... (to do)
```

## Monadische Lösung 5(5)

Insgesamt erhalten wir für `exampleTree :: Eq a => Tree a`:

```
numberTree exampleTree :: State a (Tree Int)
```

Mithilfe einer Extraktionsfunktion erhalten wir nun schließlich die gesuchte Implementierung der Funktion `numTree` mit der Funktionalität `numTree :: Eq a => Tree a -> Tree Int` (vgl. Folie 80):

```
extract :: State a b -> b
extract (State st) = snd (st [])

numTree :: Eq a => Tree a -> Tree Int
numTree = extract . numberTree
```



## Die Monade MonadPlus 1(3)

...für Instanzen von `Monad` mit *Null-Element* und *Plus-Operator*:

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```



## Die Monade MonadPlus 2(3)

Folgende Gesetze sind von Instanzen der Klasse `MonadPlus` einzuhalten:

...vom Null-Element:

```
m >>= (\x -> mzero) = mzero
mzero >>= m         = mzero
```

...vom Plus-Operator:

```
m 'mplus' mzero = m
mzero 'mplus' m = m
```

*...wie immer: Beweisverpflichtung für den Programmierer!*



## Instanzen von MonadPlus 3(3)

`Maybe` und `[]` als Instanzen von `MonadPlus`...

```
instance MonadPlus Maybe where
  mzero             = Nothing
  Nothing 'mplus' ys = ys
  xs 'mplus' ys      = xs

instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

*Beachte*:

- Listenkonkatenation `(++)` ist Spezialfall von `mplus`
- `IO` keine Instanz von `MonadPlus` (wg. fehlendes Null-Elements)

# Ergänzungen

...nützliche monadische Hilfsfunktionen:

```
sequence   :: Monad m => [m a] -> m [a]
sequence   =  foldr mcons (return [])
                where mcons p q = do l  <- p
                                     ls <- q
                                     return (l:ls)

sequence_  :: Monad m => [m a] -> m ()
sequence_  =  foldr (>>) (return ())

mapM       :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as  =  sequence (map f as)

mapM_      :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as =  sequence_ (map f as)

(=<<)      :: Monad m => (a -> m b) -> m a -> m b
f =<< x    =  x >>= f
```



# Zur Erinnerung ans letzte Semester: Konstruktorklassen vs. Typklassen

Grundsätzlich ähnliche Konzepte, wobei...

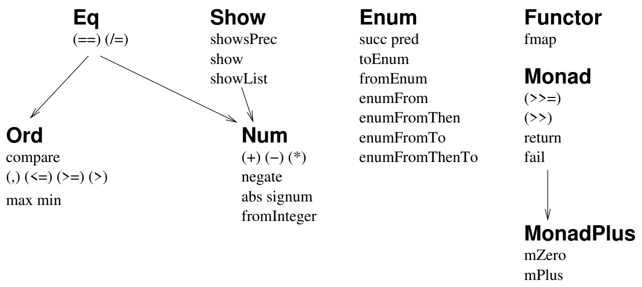
- *Konstruktorklassen* (`Functor`, `Monad`, `MonadPlus`)
  - ...haben Typkonstruktoren (`Tree`,`[]`,`(,)`,`...`) als Elemente
- *Typklassen* (`Eq a`, `Ord a`, `Num a`,`...`)
  - ...haben Typen (`Tree a`,`[a]`,`(a,a)`,`...`) als Elemente

...wobei Typkonstruktoren

- ...Funktionen sind, die aus gegebenen Typen neue Typen erzeugen
  (z.B.: Tupelkonstruktor `( )`, Listenkonstruktor `[ ]`, Funktionskonstruktor `->`, selbstdefinierte Typkonstruktoren (`Tree`, `Graph`,`...`),
  ...und insbesondere auch: Ein-/Ausgabe `IO`)

# `Functor` & `Monad` als spezielle Typklassen

`Functor` & `Monad` im Kontext ausgewählter Typklassen, auf ihnen vordefinierter Funktionen und ihrer Vererbungsbeziehungen...

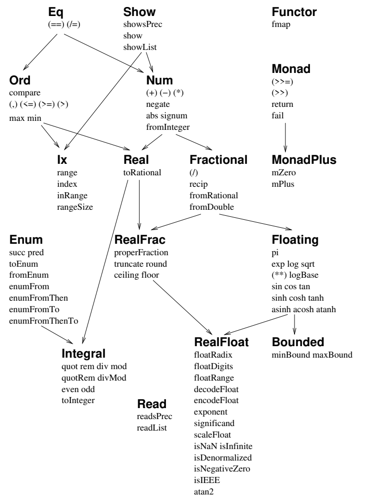



Fethi Rabhi, Guy Lapalme. *Algorithms.* Addison–Wesley, 1999, Figure 2.4, p.46



# Vordefinierte Typklassen im Überblick

Typklassen in Haskell, ein Überblick:




Fethi Rabhi, Guy Lapalme. *Algorithms.* Addison–Wesley, 1999, Figure 2.4, p.46

## Vorschau auf die nächsten Vorlesungstermine...

- Di, 31.05.2005, 14 - 16 Uhr, CompLang-Bibliothek (NEU!!!)
- Di, 07.06.2005, 14 - 16 Uhr, CompLang-Bibliothek