Heutiges Thema

- Anmerkungen zu Aufgabenblatt 4
- Anwendung: Eine imperative Sprache zur Robotersteuerung
  ⇝ Beispiel für eine anwendungsspezifische eingebettete Sprache (engl. *domain specific language* (*DSL*))

Literaturhinweis zur heutigen Vorlesung

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

Anm. zu Aufgabenblatt 4 1(13)

Erinnerung: Gegeben war folgende Aufgabe...

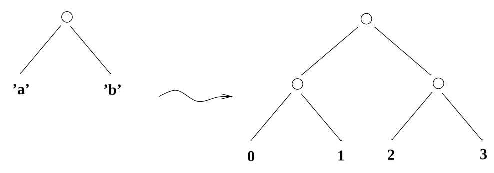
Aufgabe: ...ersetze die Benennungen der Blätter in einem Zeichenbaum durch fortlaufende natürliche Zahlen.

wobei

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


Anm. zu Aufgabenblatt 4 2(13)

Illustration:



Anm. zu Aufgabenblatt 4 3(13)

Erinnerung: ...monadenfreie “Standardlösung”:

```
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`.

Anm. zu Aufgabenblatt 4 4(13)

Erinnerung: ...Lösung mit Zustandsmonade, wobei hier...

- ...neu zur Deutlichkeit Typ- und Konstruktorname der Zustandsmonade ausgezeichnet werden.

Somit betrachten wir

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

...als Instanz der Klasse `Monad`:

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


Anm. zu Aufgabenblatt 4 5(13)

Die ursprüngliche Implementierung ändert sich damit wie folgt:

```
mlabel   :: Tree a -> Tree Integer
mlabel t = let Lab 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))
```


Anm. zu Aufgabenblatt 4 6(13)

Wir hatten behauptet:

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

Anm. zu Aufgabenblatt 4 7(13)

...wovon wir uns durch Nachrechnen überzeugen:

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

   => ( Assoziativitaet von >>= )

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

   => ( Festlegung m >>= k = concat (map k m) )

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

   => ( Definition von map )

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


Anm. zu Aufgabenblatt 4 8(13)

```
   => ( Applikation )

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

   => ( Festlegung m >>= k = concat (map k m) )

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

   => ( Definitionen von map und return)

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


Anm. zu Aufgabenblatt 4 9(13)

```
   => ( Definition von concat )

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

   => ( Definition von map )

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

   => ( Applikation )

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

      ( Definition von >>=, map, return und concat )

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


Anm. zu Aufgabenblatt 4 10(13)

```
   =>
   ...
   =>

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

   => ( Definition von concat )

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

...there was no magic. It was all just functional programming!

Anm. zu Aufgabenblatt 4 11(13)

Wir hatten weiter behauptet:

```
mlabel (Leaf 'c')

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

Leaf 0
```


Anm. zu Aufgabenblatt 4 12(13)

...wovon wir uns ebenfalls durch Nachrechnen überzeugen:

```
mlabel (Leaf 'c')
   => ( Definition mlabel )
let Lab lt = mlab (Leaf 'c')
in snd (lt 0)

  ?? Lab lt = mlab (Leaf 'c')
  ??    => ( Definition mlab )
  ?? do n <- getLabel
  ??    return (Leaf n)
  ??    => ( Zshg. do und >>= )
  ?? getLabel >>= \n -> do return (Leaf n)
  ??    => ( Zshg. do und >>= )
  ?? getLabel >>= \n -> return (Leaf n)
  ??    => ( Definition getLabel )
  ?? Lab (\n -> (n+1,n)) >>= \n -> return (Leaf n)
  ??    => ( Definition >>= )
  ?? Lab $ \s0 ->
  ??          let (s1,a1) = (\n -> (n+1,n)) s0
  ??              Lab lt1 = (\n -> return (Leaf n)) a1
  ??          in lt1 s1
```


Anm. zu Aufgabenblatt 4 13(13)

```
   => ( Einsetzen von lt in snd (lt 0)
snd ( (\s0 ->
          let (s1,a1) = (\n -> (n+1,n)) s0
              Lab lt1 = (\n -> return (Leaf n)) a1
              in lt1 s1 ) 0 )
   => ( Funktionsanwendung )
snd ( let (s1,a1) = (\n -> (n+1,n)) 0
          Lab lt1 = (\n -> return (Leaf n)) a1
      in lt1 s1 )

  ?? (s1,a1) = (0+1,0) ( = (1,0) )
  ?? Lab lt1 = (\n -> return (Leaf n)) 0
  ??    => ( Funktionsanwendung )
  ?? Lab lt1 = return (Leaf 0)
  ??    => ( Definition von return )
  ?? Lab lt1 = Lab (\s -> (s, Leaf 0))

   => ( Einsetzen von s1 und lt1 in (lt1 s1) )
snd ((\s -> (s, Leaf 0)) 1)
   => ( Funktionsanwendung )
snd ( (1, Leaf 0) )
   => ( Definition snd )
Leaf 0
```

...again, there was no magic, just functional programming!

Roboter – Simulation und Steuerung

Die Implementierungsumgebung:

```
module Robot where

import Array
import List
import Monad
import SOEGraphics
import Win32Misc (timeGetTime)
import qualified GraphicsWindows as GW (getEvent)
```

Hinweise:

- `Graphics` ...häufig verwendete Windows-kompatible Graphikbibliothek.
- `SOEGraphics` ...hier verwendete Windows-kompatible Graphikbibliothek.
- *Verfügbarkeit* ...konsultiere die SOE-Leitseite unter `http://haskell.org/soe` bezüglich der Verfügbarkeit der Module `SOEGraphics` und `GraphicsWindows`.

Zentrale Datenstrukturen 1(3)

Parameter der Roboterwelt:

```
type Grid = Array Position [Direction]

type Position = (Int,Int)

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

data Color = Black | Blue | Green | Cyan
         | Red | Magenta | Yellow | White
   deriving (Eq, Ord, Bounded, Enum, Ix, Show, Read)
```


Zentrale Datenstrukturen 2(3)

Roboterparameter/Roboterzustände:

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


Zentrale Datenstrukturen 3(3)

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

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

instance Monad Robot where
  return a         = Robot (\s _ _ -> return(s,a))
  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)
```


Exkurs: Applikation vs. Komposition 1(4)

- Funktionsapplikation

  `($) :: (a -> b) -> a -> b`

- Funktionskomposition

  `(.) :: (a -> b) -> (c -> a) -> (c -> b)`

- Anwendungsbeispiele

```
(fac . fib) 4 => fac (fib 4) => fac 5 => 120
 fac $ fib 4  => fac (fib 4) => fac 5 => 120

Main>:t fac . fib                  -- Typkorrekt!
fac . fib :: Int -> Integer

Main>:t fac $ fib                  -- Nicht typkorrekt!
ERROR - Cannot infer instance
```

Exkurs: Applikation vs. Komposition 2(4)

Beachte:

Die Ausdrücke

```
(fac . fib) 4 => fac (fib 4) => ... => 120

 fac $ fib 4  => fac (fib 4) => ... => 120
```

...sind typkorrekt.

Aber: Die Ausdrücke

```
 fac . fib 4  <=>  fac . (fib 4)

(fac $ fib) 4
```

...sind nicht typkorrekt!

Exkurs: Applikation vs. Komposition 3(4)

Analog:

```
f :: [Int] -> String
f []     = ""
f (x:xs) = 'z' : (f xs)

g :: String -> [Int]
g ""      = []
g (_ : s) = 1 : (g s)
```

Damit sind die Ausdrücke

```
(f . g) "abcd" => f (g "abcd") => f [1,1,1,1] => "zzzz"

 f $ g "abcd"  => f (g "abcd") => f [1,1,1,1] => "zzzz"
```

...typkorrekt!

Wohingegen die Ausdrücke

```
 f . g "abcd"  <=>  f . (g "abcd")

(f $ g) "abcd"
```

...typfehlerhaft sind!

Exkurs: Applikation vs. Komposition 4(4)

Man mache sich klar:

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

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

...ist Anwendungssituation für "$" (Funktionsapplikation), nicht für "." (Funktionskomposition).

Beachte:

- Zur Deutlichkeit sind oben die Bezeichnungen für Typname `Robot` und Typkonstruktor `Rob` explizit verschieden gewählt.
- Es gilt:
  `Rob :: (RobotState -> Grid -> Grid -> IO (RobotState,a)) -> Robot a`

Exkurs: Feldbezeichnersyntax 1(4)

Feldbezeichnersyntax (engl. *field label syntax*):

Vorige Deklaration der Roboterzustände ist äquivalent zu:

```
data RobotState
   = RobotState
      Position
      Direction
      Bool
      Color
      [Position]
      Int
   deriving Show
```

...definiert aber zusätzlich die *Feldbezeichner* (engl. *field labels*): `position`, `facing`, `pen`, `color`, `treasure` und `pocket`.

$\leadsto$ Feldbezeichner erlauben den Zugriff auf Komponenten einer Datenstruktur mittels *Namen* statt Position.

Exkurs: Feldbezeichnersyntax 2(4)

Benutzung von Feldbezeichnern: Erzeugung, Zugriff und Manipulation von Werten des Typs.

Anwendungsbeispiele I ...Erzeugung von Werten

Äquivalent sind...

```
s1 = RobotState (0,0) East True Green [(2,3),(7,9),(12,42)] 2
        :: RobotState
```

und

```
s2 = RobotState { position = (0,0)
                , facing   = East
                , pen      = True
                , color    = Green
                , treasure = [(2,3),(7,9),(12,42)]
                , pocket   = 2
                } :: RobotState
```


Exkurs: Feldbezeichnersyntax 3(4)

Vorteile der Feldbezeichersyntax...

- Informativer
- Reihenfolge der Felder unerheblich.

  So ist z.B.

```
s3 = RobotState { position = (0,0)
                , pocket   = 2
                , pen      = True
                , color    = Green
                , treasure = [(2,3),(7,9),(12,42)]
                , facing   = East
                } :: RobotState
```

...genauso gut möglich.

Exkurs: Feldbezeichnersyntax 4(4)

Anwendungsbeispiele II ...Selektiver Zugriff auf Feldwerte

```
position s2  =>  (0,0)
treasure s3  =>  [(2,3),(7,9),(12,42)]
color    s3  =>  Green
```

Anwendungsbeispiele III ...Selektive Manipulation von Feldwerten

```
s3 { position = (22,43), pen = False }
      => RobotState { position=(22,43)
                    , facing=East
                    , pen=False
                    , color=Green
                    , treasure=[(2,3),(7,9),(12,42)]
                    , pocket=2
                    } :: RobotState
```

Anwendungsbeispiele IV ...Verwendung in Mustern

```
jump (RobotState { position = (x,y) }) = (x+2,y+1)
```


Exkurs: `newtype`-Datentypdeklaration

Deklarationen der Form

```
newtype Temp = MkTemp Float
```

...sind semantisch äquivalent, aber effizienter implementiert als Deklarationen der Form

```
data Temp = MkTemp Float
```

...bei voller Typsicherheit:

```
newtype Temp   = MkTemp Float
newtype Weight = MkWeight Float

t = MkTemp 5.0 :: Temp
w = MkWeight 5.0 :: Weight

Main> t == w
        => ERROR - Type error in application
```

Beachte: ...`newtype` ist auf einkonstruktorige Datentypen beschränkt.

Implementierung der IRL-Kommandos

IRL... *Imperative Roboter Language*

- Kommandos ohne Graphikbezug:

```
right, left :: Direction -> Direction

right d = toEnum (succ (mod (fromEnum d) 4))
left d  = toEnum (pred (mod (fromEnum d) 4))
```

- Hilfsfunktionen zum Aktualisieren und Abfragen von Zuständen:

```
updateState   :: (RobotState -> RobotState) -> Robot ()
updateState u = Robot (\s _ _ -> return (u s, ()))

queryState :: (RobotState -> a) -> Robot a
queryState q  = Robot (\s _ _ -> return (s, q s))
```


Exkurs: Die Aufzählungsklasse `Enum` 1(2)

...vordefiniert im *Standard Prelude*:

```
class Enum a where
  succ, pred     :: a -> a
  toEnum         :: Int -> a
  fromEnum       :: a -> Int
  enumFrom       :: a -> [a]                -- [n..]
  enumFromThen   :: a -> a -> [a]           -- [n,n'..]
  enumFromTo     :: a -> a -> [a]           -- [n..m]
  enumFromThenTo :: a -> a -> a -> [a]      -- [n,n'..m]

  succ                 = toEnum . (+1) . fromEnum
  pred                 = toEnum . (subtract 1) . fromEnum
  enumFrom x           = map toEnum [fromEnum x..]
  enumFromThen x y     = map toEnum [fromEnum x, fromEnum y..]
  enumFromTo x y       = map toEnum [fromEnum x..fromEnum y]
  enumFromThenTo x y z = map toEnum [fromEnum x, fromEnum y..fromEnum z]

  toEnum, fromEnum =     ...sind in der Regel typabhaengig zu implementieren.
```


Exkurs: Die Aufzählungsklasse `Enum` 2(2)

Für arithmetische Sequenzen gelten folgende Äquivalenzen:

```
enumFrom n             [n..]
enumFromThen n n'      [n,n'..]
enumFromTo n m         [n..m]
enumFromThenTo n n' m  [n,n'..m]
```

Anwendungsbeispiel:

```
data Color = Red | Orange | Yellow | Green | Blue | Indigo | Violet

instance Enum Color where...

[Red..Green]     =>  [Red, Orange, Yellow, Green]
[Red, Yellow..]  =>  [Red, Yellow, Blue, Violet]
fromEnum Blue    =>  4
toEnum 3         =>  Green
```


Weitere IRL-Kommandos 1(5)

- Kommandos zur Roboterausrichtung (-orientierung):

```
turnLeft  :: Robot ()
turnLeft  =  updateState (\s -> s {facing = left (facing s)})

turnRight :: Robot ()
turnRight =  updateState (\s -> s {facing = right (facing s)})

turnTo   :: Direction -> Robot ()
turtTo d =  updateState (\s -> s {facing = d})

direction :: Robot Direction
direction =  queryState facing

blocked :: Robot Bool
blocked =  Robot $ \s g _ ->
             return(s, facing s 'notElem' (g 'at' position s))
```

Weitere IRL-Kommandos 2(5)

- Kommandos zur Roboterbewegung:

```
move :: Robot ()
move =  cond1 (isnt blocked)
          (Robot $ \s _ w -> do
             let newPos = movePos (position s) (facing s)
             graphicsMove w s newPos
             return (s {position = newPos}, ())
          )

movePos :: Position -> Direction -> Position
movePos (x,y) d
        = case d of
            North -> (x,y+1)
            South -> (x,y-1)
            East  -> (x+1,y)
            West  -> (x-1,y)
```


Weitere IRL-Kommandos 3(5)

- Kommandos zur Stiftbenutzung:

```
penUp :: Robot ()
penUp =  updateState (\s -> s {pen = False})

penDown :: Robot ()
penDown =  updateState (\s -> s {pen = True})

setPenColor   :: Color -> Robot ()
setPenColor c =  updateState (\s -> s {color = c})
```


Weitere IRL-Kommandos 4(5)

- Kommandos zum Umgang mit Münzen:

```
onCoin :: Robot Bool
onCoin =  queryState (\s -> position s 'elem' treasure s)

coins :: Robot Int
coins =  queryState pocket
```


Weitere IRL-Kommandos 5(5)

- Weitere Kommandos zum Umgang mit Münzen:

```
pickCoin :: Robot ()
pickCoin =  cond1 onCoin
              (Robot $ \s _ w ->
                 do eraseCoin w (position s)
                    return (s {treasure = position s 'delete' treasure s,
                               pocket = pocket s+1}, ())
              )

dropCoin :: Robot ()
dropCoin =  cond1 (coins >* return 0)
              (Robot $ \s _ w ->
                 do drawCoin w (position s)
                    return (s {treasure = position s : treasure s,
                               pocket = pocket s-1}, ())
              )
```


Roboterlogik und Kontrolle 1(4)

- Roboter-Logik und Kontrollfunktionen:

```
cond       :: Robot Bool -> Robot a -> Robot a -> Robot a
cond p c a =  do pred <- p
                 if pred then c else a

cond1 p c = cond p c (return ())

while     :: Robot Bool -> Robot () -> Robot ()
while p b =  cond1 p (b >> while p b)

(||*)     :: Robot Bool -> Robot Bool -> Robot Bool
b1 ||* b2 =  do p <- b1
                if p then return True
                     else b2
```


Roboterlogik und Kontrolle 2(4)

- Roboter-Logik und Kontrollfunktionen fortgesetzt:

```
(&&*)     :: Robot Bool -> Robot Bool -> Robot Bool
b1 &&* b2 =  do p <- b1
                if p then b2
                     else return False

isnt :: Robot Bool -> Robot Bool
isnt =  liftM not

(>*), (<*) :: Robot Int -> Robot Int -> Robot Bool
(>*)       =  liftM2 (>)
(<*)       =  liftM2 (<)
```


Roboterlogik und Kontrolle 3(4)

Die Funktionale `liftM` und `liftM2` sind (wie auch `liftM3` bis `liftM5`) in der Bibliothek `Monad` definiert:

```
liftM   :: (Monad m) => (a -> b) -> (m a -> m b)
liftM f = \a -> do a' <- a
                   return (f a')

liftM2   :: (Monad m) => (a -> b -> c) -> (m a -> m b -> m c)
liftM2 f =  \a b -> do a' <- a
                       b' <- b
                       return (f a' b')
```


Roboterlogik und Kontrolle 4(4)

Bemerkung:

- Die Abstützung der Implementierungen von `isnt`, `(>*)` und `(<*)` auf `liftM` und `liftM2` erlaubt den Verzicht auf spezielle *lifting* Funktionen.
- Keine Abstützung der Implementierungen von `(||*)` und `(&&*)` auf `liftM2` zur Vermeidung (unnötiger) Striktheit in ihren zweiten Argumenten.

Weitere benötigte Datenstrukturen

```
colors :: Array Int Color
colors = array (0,7) [(0,Black),(1,Blue),(2,Green),(3,Cyan),
                     (4,Red),(5,Magenta),(6,Yellow),(7,White)]
```

wobei (hier zur Erinnerung wiederholt sei)

```
data Color = Black | Blue | Green | Cyan
           | Red | Magenta | Yellow | White
     deriving (Eq, Ord, Bounded, Enum, Ix, Show, Read)
```

Beachte:

- ...`Color` definiert wie in der Bibliothek `Graphics`.
- ...mit obiger data-Deklaration von `Color` wäre auch folgende knappere Definition von `colors` möglich:

```
colors :: Array Int Color
colors = array (0,7) (zip [0..7] [Black..White])
```


Exkurs: Felder in Haskell 1(3)

Es gilt:

- Ein Wert vom Typ `Ix a => Array a b` ist (in Haskell)
  - ...ein Feld mit Elementen vom Typ `b`,
  - ...auf die mit Werten vom Typ `a` als Indizes zugegriffen wird,
  - ...mithilfe der Indexfunktion
    `(!) :: Ix a => Array a b -> a -> b`

Damit gilt speziell:

- Ein Wert des Typs `arr :: Array Int Color` ist ein Feld von Farben, wobei die Farbe an der Position `i :: Int` gegeben ist durch `arr!i`

Exkurs: Felder in Haskell 2(3)

Beachte:

- Felder sind in der Bibliothek `Array` definiert.
- Abgrenzung Listen vs. Felder
  - *Listen*: ...beliebig (potentiell unendlich) lang, dafür sind die Kosten des Zugriffs auf das i-te Element mithilfe von `(!!)` proportional zu i.
  - *Felder*: ...feste, endliche Länge, dafür Kosten des Zugriffs auf das i-te Element mithilfe von `(!)` konstant (also unabhängig von i).

Exkurs: Felder in Haskell 3(3)

Deklaration von Feldern mittels:

```
array :: Ix a => (a,a) -> [(a,b)] -> Array a b
```

Intuition:

- 1. Argument: ...Indexgrenzen (niedrig, hoch)
- 2. Argument: ...Liste von Index/Wert-Paaren (*Feldeinträge* (engl. *array entries*)
  $\leadsto$ Feldinitialisierung

Beispiel:

```
colors :: Array Int Color
colors = array (0,7) [(0,Black),(1,Blue),(2,Green),(3,Cyan),
                     (4,Red),(5,Magenta),(6,Yellow),(7,White)]

colors!3 => Cyan
```


Exkurs: Die Indexklasse `Ix` 1(3)

...vordefiniert im *Standard Prelude*:

```
class (Ord a) => Ix a where
  range   :: (a,a) -> [a]
  index   :: (a,a) -> a -> Int
  inRange :: (a,a) -> a -> Bool
```

- Instanzen der Klasse `Ix` sind insbesondere: `Int`, `Integer`, `Char`, `Bool` und Tupel von `Ix`-Typen.
- Instanzen sind automatisch ableitbar (mittels `deriving`-Klausel) für Aufzählungs- (Enumerations-) und Tupeltypen.

  Intuition: Primitive Typen entsprechen Vektorindizes, Tupeltypen Indizes mehrdimensionaler Felder.

Exkurs: Die Indexklasse `Ix` 2(3)

Beachte:

- Erstes Argument der `Ix`-Klassenfunktionen: ...jeweils ein Wert vom Typ `(a,a)`-Paar.

  Intuition: Typischerweise bezeichnet der Paarwert die *Grenzen* (erster und letzter Index) eines Feldes (eng. array).

  Beispiele:
  - `(0,9)`: Indixgrenzen eines 10-elementigen mit Index 0 beginnenden Vektors mit Indizes vom Typ `Int`.
  - `((1,1),(100,100))`: Indexgrenzen einer 100×100-Matrix mit `(1,1)` als kleinstem Index.

    Anm. zur Notation: `((1,1),(100,100))` "passt besser" zum Typsystem als z.B. `(1:100,1:100)`, da die Typen aller Indexgrenzen mit dem allgemeinen Indexgrenzentyp übereinstimmen.

Exkurs: Die Indexklasse `Ix` 3(3)

- Anwendungsbeispiele für `range`, `index` und `inRange`

```
-- range liefert die Indizes innerhalb der Argumentgrenzen
range (0,4)         => [0,1,2,3,4]
range ((0,0),(1,2)) => [(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)]

-- index liefert bezogen auf 0 die Position eines Index
index (1,9) 2             => 1
index ((0,0),(1,2)) (1,1) => 4

-- inRange ueberprueft, ob ein Index innerhalb der Grenzen liegt
inRange (1,9) 42              => False
inRange ((0,0),(1,2)) (1,0) => (inRange (0,1) 1) && (inRange (0,2) 0)
                            => True && True
                            => True
```


Die Roboterwelt: Ein Gitter

Die Welt, ein Gitter vom Typ `Array`:

```
type Grid = Array Position [Direction]
```

Auf deren Punkte der Zugriff möglich ist mittels:

```
at :: Grid -> Position -> [Direction]
at =  (!)
```

Deren Größe und äußere Grenzen festgelegt sind durch:

```
size :: Int
size =  20
```

Mit Zentrum `(0,0)` und den Ecken `(size,size)`, `((-size),size)`, `((-size),(-size))` und `(size,(-size))`.

Die initiale Welt g0 1(2)

...begrenzt durch die vier Außenwände (ohne weitere Wände im Inneren).

Innere Punkte von g0 gegeben durch:

```
interior = [North, South, East, West]
```

Gitterpunkte auf dem Rand (Nordgrenze (border), Nordostecke (corner), etc.) gegeben durch:

```
nb  = [South, East, West]      -- nb: north border
sb  = [North, East, West]
eb  = [North, South, West]
wb  = [North, South, East]     -- wb: west border
nwc = [South, East]            -- nwc: north west corner
nec = [South, West]
swc = [North, East]
sec = [North, West]            -- sec: south east corner
```


Die initiale Welt g0 2(2)

Damit möglich:

...Enumeration der *echt inneren* und der *Gitterrandpunkte* mittels Listenkomprehension:

```
g0 :: Grid
g0 =  array ((-size, -size), (size, size))
        ([((i, size), nb) | i <- r ] ++
        ([((i, -size), sb) | i <- r ] ++
        ([((size, i), eb) | i <- r ] ++
        ([((-size, i), wb) | i <- r ] ++
        ([((size, i), eb) | i <- r ] ++
        ([((i,j), interior) | i <- r, j <- r ] ++
        ([((size, size), nec), ((size, -size), sec),
          ((-size, size), nwc), ((-size, -size), swc)])
      where r = [1-size..size-1]
```


g1 als Erweiterung der Welt g0 1(3)

...z.B. durch Hinzufügen von Wänden mithilfe folgender Array-Bibliotheksfunktion:

```
(//) :: Ix a => Array a b -> [(a,b)] -> Array a b
```

Anwendungsbeispiel von (//):

- ...vertauschen der Zuordnungen von "schwarz" und "weiß" in colors:

```
colors//[(0,White),(7,Black)]

   =>

array (0,7) [(0,White),(1,Blue),(2,Green),
             (3,Cyan),(4,Red),(5,Magenta),
             (6,Yellow),(7,Black)] :: Array Integer Color
```


g1 als Erweiterung der Welt g0 2(3)

...Hilfsfunktionen zum Aufbau zusätzlicher Wände:

```
mkHorWall, mkVerWall ::
          Int -> Int -> Int -> [(Position,[Direction])]

-- Aufbau West/Ost-ausgerichteter Waende (von (x1,y) nach (x2,y))
mkHorWall x1 x2 y
   = [((x,y), nb) | x <- [x1..x2]] ++
     [((x,y+1), sb) | x <- [x1..x2]]

-- Aufbau Nord/Sued-ausgerichteter Waende (von (x,y1) nach (x,y2))
mkVerWall y1 y2 x
   = [((x,y), eb) | y <- [y1..y2]] ++
     [((x+1,y), wb) | y <- [y1..y2]]
```


g1 als Erweiterung der Welt g0 3(3)

Aus g0 wird g1 durch...

- Aufbau einer West/Ost-ausgerichteten Wand von (-5,10) nach (5,10):

```
g1 :: Grid
g1 =  g0//mkHorWall (-5) 5 10
```


g2 als Erweiterung der Welt g0 1(3)

...Funktion zur Aufbau eines "Zimmers":

```
mkBox :: Position -> Position -> [(Position, [Direction])]
mkBox (x1, y1) (x2, y2)
      = mkHorWall (x1+1) x2 y1 ++
        mkHorWall (x1+1) x2 y2 ++
        mkVerWall (y1+1) y2 x1 ++
        mkVerWall (y1+1) y2 x2
```

Achtung: Obige Funktion kreiert zwei Feldeinträge für jede der vier inneren Ecken. Konsequenz in Haskell: Der Wert dieser Einträge ist undefiniert.

Elegante Abhilfe:

...mittels der Array-Bibliotheksfunktion accum:

```
accum :: (Ix a) => (b -> c -> b) -> Array a b -> [(a,c)] -> Array a b
```


g2 als Erweiterung der Welt g0 2(3)

Zur Funktion accum:

```
accum :: (Ix a) => (b -> c -> b) -> Array a b -> [(a,c)] -> Array a b
```

- ...entspricht i.w. der Funktion (//).
- ...im Falle duplizierter Einträge wird die Funktion des ersten Arguments zur Konfliktauflösung herangezogen.
- ...in unserem Beispiel ist dafür die List-Bibliotheksfunktion intersect geeignet:

  Beispiel:

  ```
  [South, East, West] 'intersect' [North, South, West]
      => [South, West]
  ```

  ...was einer Nordostecke entspricht (vgl. frühere Folie).

g2 als Erweiterung der Welt g0 3(3)

Einbau eines Zimmers (mit (-10,5) als unterem linken und (-5,10) als oberem rechten Eckpunkt):

$\leadsto$ ...mittels accum und intersect.

Aus g0 wird g2:

```
g2 :: Grid
g2 =  accum intersect g0 (mkBox (-15,8) (2,17))
```

g3 als Erweiterung der Welt g2

Einbau einer Tür (in die Mitte der oberen Zimmerwand):

⇝ ...mittels `accum` und `union`.

Aus g2 wird g3:

```
g3 :: Grid
g3 =  accum union g2 [((-7,17), interior), ((-7,18), interior)]
```


Zur Animation: Robotergraphik 1(3)

Animation...

⇝ ...mittels inkrementeller Aktualisierung der Welt.

Dazu wird benutzt:

Die Funktion

```
drawLine :: Window -> Color -> Point -> Point -> IO ()
drawLine w c p1 p2
        = drawInWindowNow w (withColor c (line p1 p2))
```

...die sich auf die `Graphics`-Bibliotheksfunktion `drawInWindowNow` abstützt.

Zur Animation: Robotergraphik 2(3)

Die inkrementelle Aktualisierung muss gewährleisten:

⇝ ...Interferenzfreiheit der Graphikaktionen.

Dazu folgende Annahmen für das Weitere:

1. Gitterpunktabstände betragen 10 Pixel
2. Wände verlaufen genau in der Mitte zwischen Gitterpunkten
3. Roboterstiftlinien gehen von Gitterpunkt zu Gitterpunkt
4. Münzen als gelbe Kreise links oberhalb von Gitterpunkten dargestellt
5. Wegnehmen von Münzen durch Übermalen der gelben durch schwarze Kreise

Zur Animation: Robotergraphik 3(3)

Folgende Werte für die "Toplevel"-Konstanten gewährleisten die Interferenzfreiheit:

```
d          :: Int
d          =  5       -- halber Abstand zwischen Gitterpunkten

wc, cc     :: Color
wc         =  Green   -- Farbe der Waende
cc         =  Yellow  -- Farbe der Muenzen

xWin, yWin :: Int
xWin       =  600
yWin       =  500
```


Die Animation in Aktion 1(5)

"Putting it all together"...

Fortschrittskontrolle durch Warten auf Leertastenbestätigung:

```
spaceWait   :: Window -> IO ()
spaceWait w = do k <- getKey w
                 if k==' ' then return ()
                           else spaceWait w
```


Die Animation in Aktion 2(5)

"Laufenlassen" eines IRL-Programms...

```
runRobot :: Robot () -> RobotState -> Grid -> IO ()
runRobot (Robot sf) s g
        = runGraphics $
          do w <- openWindowEx "Robot World" (Just (0,0))
                    (Just (xWin, yWin)) drawGraphic (Just 10)
             drawGrid w g
             drawCoins w s
             spaceWait w
             sf s g w
             spaceClose w
```


Die Animation in Aktion 3(5)

`runRobot` bewirkt...

- ...öffnet ein Fenster
- ...zeichnet ein Gitter
- ...zeichnet die Münzen
- ...wartet auf Leertastenbestätigung des Benutzers
- ...führt dann das Programm mit Anfangszustand `s` und Gitter `g` aus

Die Animation in Aktion 4(5)

Festlegen einer geeigneten Startkonfiguration...

```
s0 :: RobotState
s0 =  RobotState {position = (0,0)
                 , pen = False
                 , color = Blue
                 , facing = North
                 , treasure = tr
                 , pocket = 0
                  }

tr :: [Position]
tr =  [(x,y) | x <- [-13,-11..1], y <- [9,11..15]]
```

...platziert alle Münzen innerhalb des Zimmers von Gitter `g3`.

Die Animation in Aktion 5(5)

Zu guter Letzt...

```
main = runRobot spiral s0 g0
```

...führt das *"Spiralen"*-Beispiel aus
(vgl. Folie 6, Vorlesungsteil 5).

Was noch fehlt... 1(5)

...sind eine Reihe von Hilfsfunktionen.

Zum Zeichnen des Weltengitters...

```
drawGrid :: Window -> Grid -> IO ()
drawGrid w wld
    = let (low@(xMin,yMin),hi@(xMax,yMax)) = bounds wld
          (x1,y1)                          = trans low
          (x2,y2)                          = trans hi
      in do drawLine w wc (x1-d,y1+d) (x1-d,y2-d)
            drawLine w wc (x1-d,y1+d) (x1+d,y2+d)
            sequence_ [drawPos w (trans (x,y)) (wld 'at' (x,y))
                      | x <- [xMin..xMax], y <- [yMin..yMax]]
```


Was noch fehlt... 2(5)

```
drawPos :: Window -> Point -> [Direction] -> IO ()
drawPos x (x,y) ds
    = do if North 'notElem' ds
            then drawLine w wc (x-d,y-d) (x+d,y-d)
            else return ()
         if East 'notElem' ds
            then  drawLine w wc (x+d,y-d) (x+d,y+d)
            else return ()
```


Was noch fehlt... 3(5)

Zum Ablegen und Wegnehmen von Münzen...

```
drawCoins      :: Window -> RobotState -> IO ()
drawCoins w s  = mapM_ (drawCoin w) (treasure s)

drawCoin       :: Window -> Position -> IO ()
drawCoin w p   = let (x,y) = trans p
                   in drawInWindowNow w
                      (withColor cc (ellipse (x-5,y-1) (x-1,y-5)))

eraseCoin      :: Window -> Position -> IO ()
eraseCoin w p  = let (x,y) = trans p
                   in drawInWindowNow w
                      (withColor Black (ellipse (x-5,y-1) (x-1,y-5)))
```


Was noch fehlt... 4(5)

```
graphicsMove :: Window -> RobotState -> Position -> IO ()
graphicsMove w s newPos
   = do if pen s
          then drawLine w (color s) (trans (position s))
                                    (trans newPos)
          else return ()
        getWindowTick w
```


Was noch fehlt... 5(5)

```
trans       :: Position -> Point
trans (x,y) = (div xWin 2+2*d*x, div yWin 2-2*d*y)

getWindowTick :: Window -> IO ()
-- ''wartet auf den naechsten Zeitimpuls''

bounds :: Ix a => Array a b -> (a,a)
-- aus der Array-Bibliothek
-- liefert die Grenzen des Feldarguments
```


Vorschau auf die nächsten Vorlesungstermine...

- Di, 21.06.2005, 14 - 16 Uhr, CompLang-Bibliothek
- Di, 28.06.2005, 14 - 16 Uhr, CompLang-Bibliothek (ggf.)