

I/O: Warum ist I/O ein Problem?

- Funktionaler Ansatz: An Namen werden feste Werte gebunden

```
val :: Int  
val = 42
```

```
function :: Int -> Int  
function n = val + n
```

I/O: Warum ist I/O ein Problem? (2)

- Eine Möglichkeit: I/O-Operationen einfach so hinzufügen

```
inputInt :: Int
```

```
inputDiff = inputInt - inputInt
```

- Probleme
 - Auswertungsreihenfolge spielt plötzlich eine Rolle
 - Bedeutung von Teilen eines Ausdrucks nicht mehr nur von den Teilen alleine bestimmt

```
funny :: Int -> Int
```

```
funny n = inputInt + n
```

I/O mittels I/O-Aktionen!

- Idee: die komplette I/O besteht aus einzelnen I/O-Aktionen, die in fester Reihenfolge passieren
- in Haskell:
 - Typ `IO a` beschreibt eine I/O-Aktion, die ein Objekt vom Typ `a` zurückliefert
 - eingebaute, primitive I/O-Aktionen: `getLine`, `putStrLn`, ...
 - Mechanismus, um I/O-Aktionen zu Sequentialisieren
 - `main :: IO ()` ist immer eine große I/O-Aktion

I/O mittels I/O-Aktionen: primitive, eingebaute I/O-Aktionen

- `getLine :: IO String`
- `getChar :: IO Char`
- `putStr :: IO ()`
- `return :: a -> IO a`

I/O mittels I/O-Aktionen: Sequentialisieren mittels do

- Beispiel: Zwei Zeilen einlesen und umgekehrt wieder ausgeben:

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

I/O mittels I/O-Aktionen: Sequentialisieren mittels do (2)

- ... oder alternativ ...

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

Überladung: Warum?

- Statt

```
elemBool :: Bool -> [Bool] -> Bool
elemBool x [] = False
elemBool x (y:ys) = (x ==Bool y) || elemBool x ys
```

```
elemInt :: Int -> [Int] -> Int
```

```
...
```

Überladung: Erste Möglichkeit

- Explizit

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

```
... elemGen (==Bool) ...
```

Überladung in Haskell

- mittels Typklassen

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

```
instance Eq Bool where
    True == True = True
```

...

```
instance Eq Int where
```

...

```
elem :: Eq a => a -> [a] -> Bool
```

Andere Typklassen in Haskell

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

```
class Show a where  
    ...
```

```
class Read a where
```