

Reaktive Programmierung  
Vorlesung 1 vom 19.04.2022  
Was ist Reaktive Programmierung?

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

# Organisatorisches

- ▶ Vorlesung: Di 12-14, MZH 1470
- ▶ Übung: Mi 12-14, MZH 1110 (nach Bedarf)
- ▶ Webseite: [www.informatik.uni-bremen.de/~cxl/lehre/rp.ss22](http://www.informatik.uni-bremen.de/~cxl/lehre/rp.ss22)
- ▶ Scheinkriterien:
  - ▶ Voraussichtlich 6 Übungsblätter
  - ▶ Alle bearbeitet, insgesamt 40% (Notenspiegel PI3)
  - ▶ Übungsgruppen 2 – 4 Mitglieder
  - ▶ **Danach:** Fachgespräch **oder** Modulprüfung

# Warum Reaktive Programmierung?

Herkömmliche Sprachen:

- ▶ PHP, JavaScript, Ruby, Python
- ▶ C, C++, Java

Eigenschaften:

- ▶ **Imperativ** und **prozedural**
- ▶ **Sequentiell**

Zugrundeliegendes Paradigma:



# Warum Reaktive Programmierung?

Herkömmliche Sprachen:

- ▶ PHP, JavaScript, Ruby, Python
- ▶ C, C++, Java

Eigenschaften:

- ▶ **Imperativ** und **prozedural**
- ▶ **Sequentiell**

Zugrundeliegendes Paradigma:

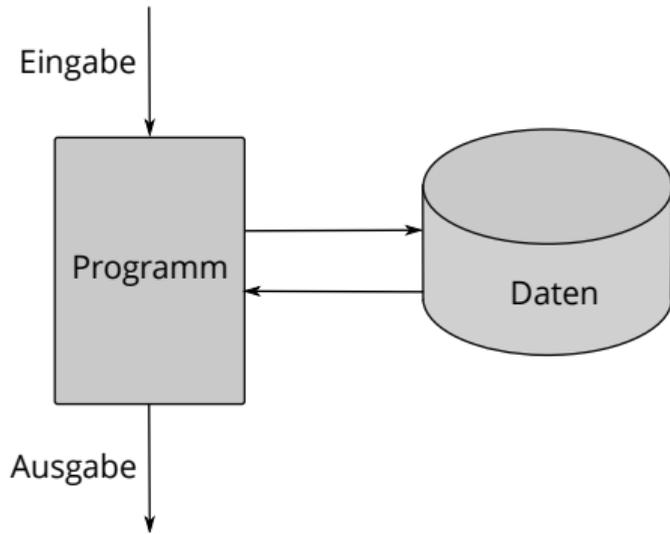


... aber die Welt ändert sich:

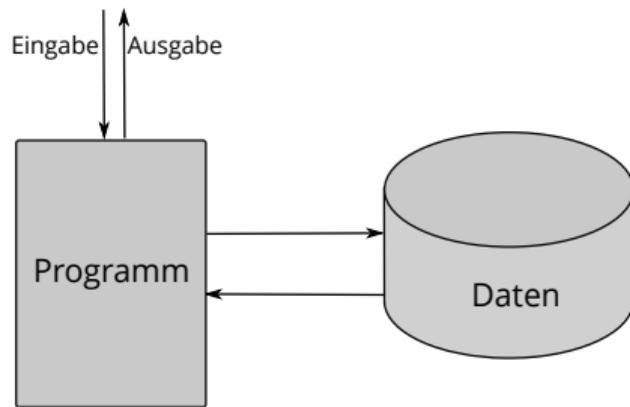


- ▶ Das **Netz** verbindet Rechner
- ▶ Selbst eingebettete Systeme sind vernetzt (Auto: ca. 130 Proz.)
- ▶ Mikroprozessoren sind **mehrkernig**
- ▶ Systeme sind **eingebettet**, **nebenläufig**, **reagieren** auf ihre Umwelt.

# Probleme mit dem herkömmlichen Ansatz

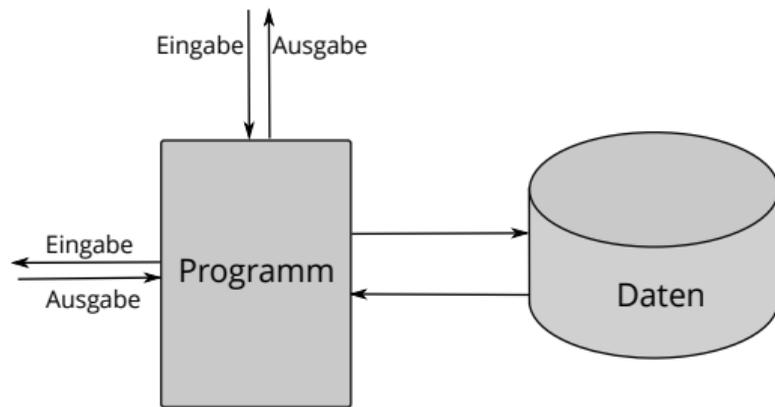


# Probleme mit dem herkömmlichen Ansatz



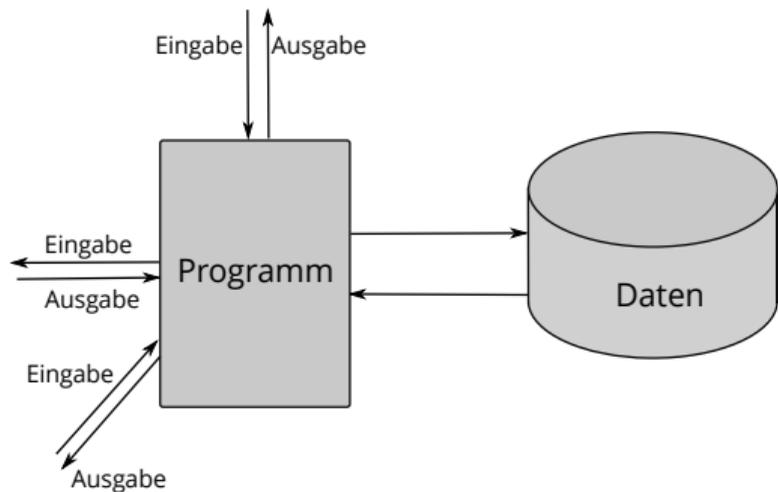
► Problem: **Nebenläufigkeit**

# Probleme mit dem herkömmlichen Ansatz



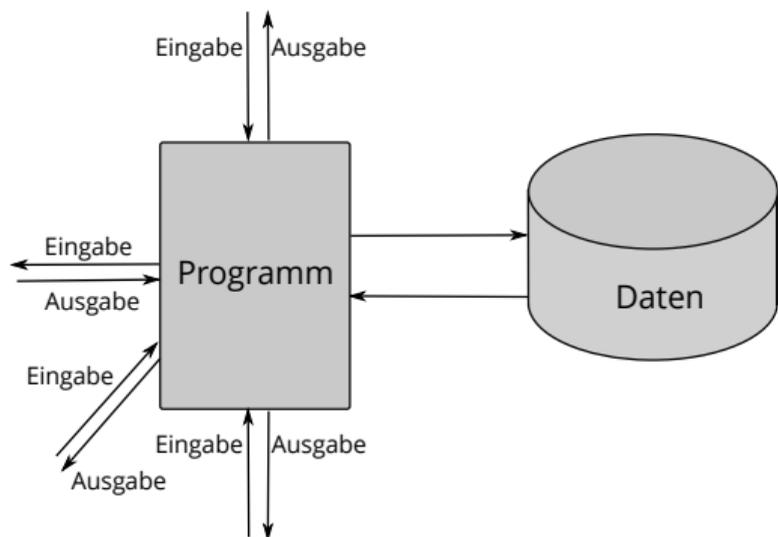
► Problem: **Nebenläufigkeit**

# Probleme mit dem herkömmlichen Ansatz



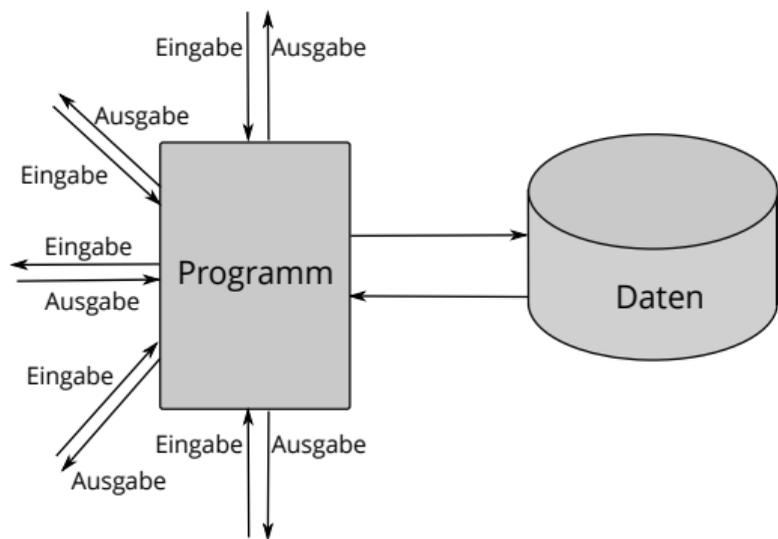
- ▶ Problem: **Nebenläufigkeit**
- ▶ Nebenläufigkeit verursacht **Synchronisationsprobleme**

# Probleme mit dem herkömmlichen Ansatz



- ▶ Problem: **Nebenläufigkeit**
- ▶ Nebenläufigkeit verursacht **Synchronisationsprobleme**
- ▶ Behandlung:
  - ▶ Callbacks (JavaScript, PHP)
  - ▶ Events (Java)
  - ▶ Global Locks (Python, Ruby)
  - ▶ Programmiersprachenkonstrukte: Locks, Semaphoren, Monitore

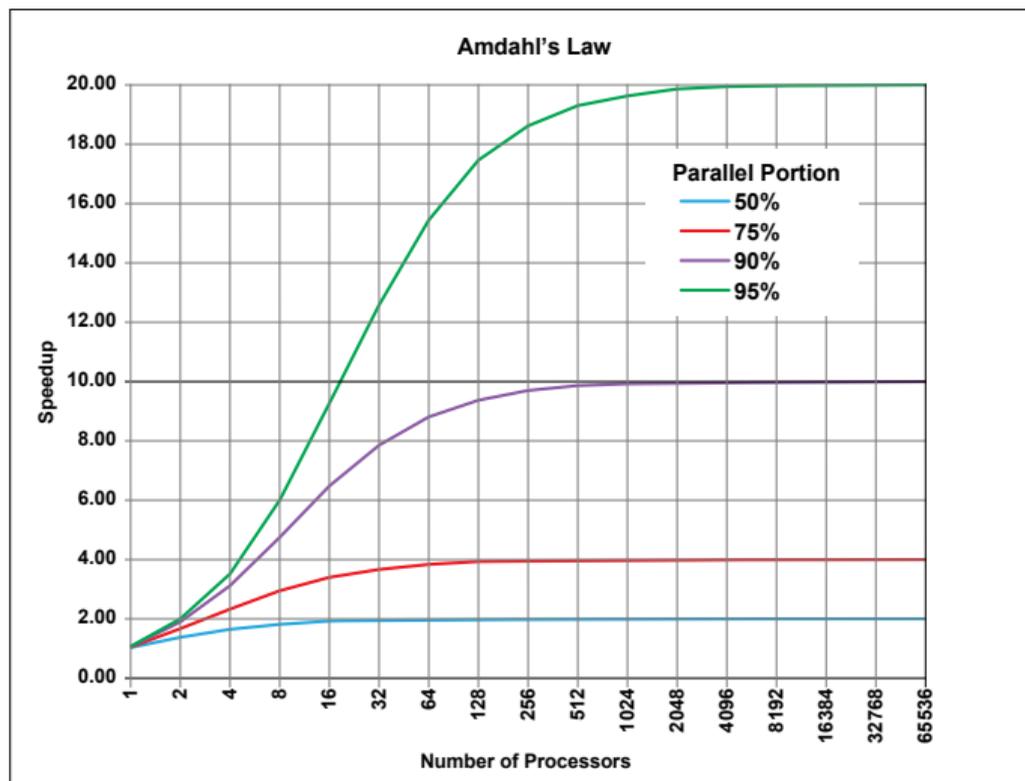
# Probleme mit dem herkömmlichen Ansatz



- ▶ Problem: **Nebenläufigkeit**
- ▶ Nebenläufigkeit verursacht **Synchronisationsprobleme**
- ▶ Behandlung:
  - ▶ Callbacks (JavaScript, PHP)
  - ▶ Events (Java)
  - ▶ Global Locks (Python, Ruby)
  - ▶ Programmiersprachenkonstrukte: Locks, Semaphoren, Monitore

# Amdahl's Law

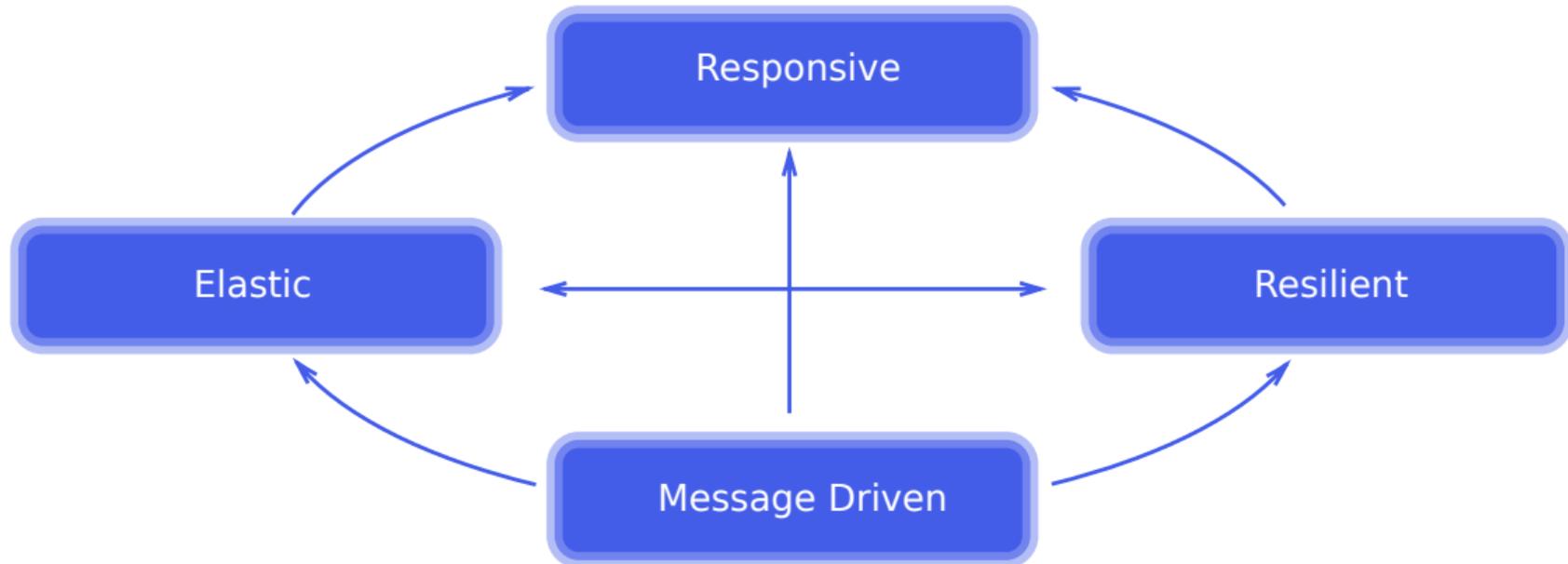
“The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20× as shown in the diagram, no matter how many processors are used.”



Quelle: Wikipedia

# The Reactive Manifesto

► <http://www.reactivemanifesto.org/>

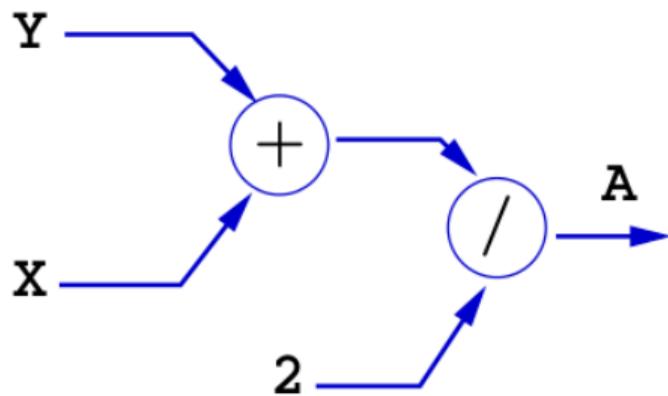


# Was ist Reaktive Programmierung?

- ▶ Imperative Programmierung: Zustandsübergang
- ▶ Prozedural und OO: Verkapselter Zustand
- ▶ Funktionale Programmierung: Abbildung (mathematische Funktion)
- ▶ **Reaktive** Programmierung:
  - ① **Datenabhängigkeit**
  - ② **Reaktiv** = funktional + nebenläufig

# Datenflusssprachen (data flow languages)

- ▶ Frühe Sprachen: VAL, SISAL, ID, LUCID (1980/1990)
- ▶ Heutige Sprachen: Esterel, Lustre (Gérard Berry, Verimag)
  - ▶ Keine **Zuweisungen**, sondern **Datenfluss**
  - ▶ **Synchron**: alle Aktionen ohne Zeitverzug
  - ▶ Verwendung in der Luftfahrtindustrie (Airbus)



```
node Average(X, Y : int)
returns (A : int);
let
    A = (X + Y) / 2 ;
tel
```

# Struktur der VL

- ▶ **Kernkonzepte** in Scala und Haskell:
  - ▶ Nebenläufigkeit: Futures, Aktoren, Reaktive Ströme
  - ▶ FFP: Bidirektionale und Meta-Programmierung, FRP, sexy types
  - ▶ Robustheit: Eventual Consistency, Entwurfsmuster
- ▶ Bilingualer **Übungsbetrieb** und **Vorlesung**
  - ▶ Kein Scala-Programmierkurs
  - ▶ Erlernen von Scala ist nützlicher Seiteneffekt

# Fahrplan

- ▶ Einführung
- ▶ Monaden und Monadentransformer
- ▶ Nebenläufigkeit: Futures and Promises
- ▶ Aktoren: Grundlagen & Implementierung
- ▶ Bidirektionale Programmierung
- ▶ Meta-Programmierung
- ▶ Reaktive Ströme I
- ▶ Reaktive Ströme II
- ▶ Funktional-Reaktive Programmierung
- ▶ Software Transactional Memory
- ▶ Eventual Consistency
- ▶ CRDTs
- ▶ Robustheit, Entwurfsmuster und Theorie der Nebenläufigkeit, Abschluss
- ▶ Reaktive Programmierung in der Praxis

# I. Rückblick Haskell

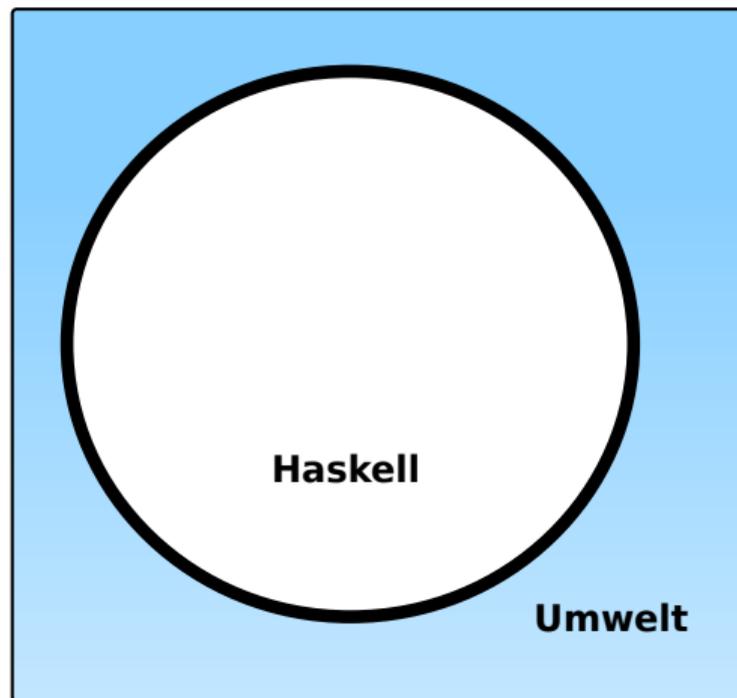
# Rückblick Haskell

- ▶ Definition von Funktionen:
  - ▶ lokale Definitionen mit `let` und `where`
  - ▶ Fallunterscheidung und guarded equations
  - ▶ Abseitsregel
  - ▶ Funktionen höherer Ordnung
- ▶ Typen:
  - ▶ Basisdatentypen: `Int`, `Integer`, `Rational`, `Double`, `Char`, `Bool`
  - ▶ Strukturierte Datentypen: `[ $\alpha$ ]`, `( $\alpha$ ,  $\beta$ )`
  - ▶ Algebraische Datentypen: `data Maybe  $\alpha$  = Just  $\alpha$  | Nothing`

# Rückblick Haskell

- ▶ Nichtstriktheit und verzögerte Auswertung
- ▶ Strukturierung:
  - ▶ Abstrakte Datentypen
  - ▶ Module
  - ▶ Typklassen

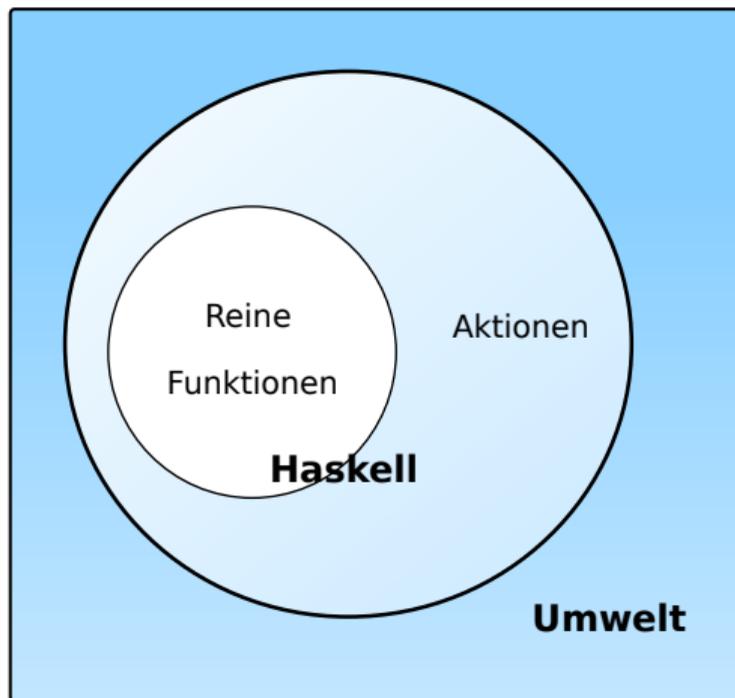
# Ein- und Ausgabe in Haskell



## Problem:

- ▶ Funktionen mit Seiteneffekten nicht referentiell transparent.
- ▶ `readString :: ... → String ??`

# Ein- und Ausgabe in Haskell



## Problem:

- ▶ Funktionen mit Seiteneffekten nicht referentiell transparent.
- ▶ `readString :: ... → String ??`

## Lösung:

- ▶ Seiteneffekte am Typ erkennbar
- ▶ **Aktionen** können **nur** mit **Aktionen** komponiert werden
- ▶ „einmal Aktion, immer Aktion“

# Aktionen als abstrakter Datentyp

- ▶ ADT mit Operationen **Komposition** und **Lifting**

- ▶ Signatur:

```
type IO  $\alpha$ 
```

```
( $\gg$ ) :: IO  $\alpha \rightarrow (\alpha \rightarrow \text{IO } \beta) \rightarrow \text{IO } \beta$ 
```

```
return ::  $\alpha \rightarrow \text{IO } \alpha$ 
```

- ▶ Plus **elementare** Operationen (lesen, schreiben etc)

# Elementare Aktionen

- ▶ Zeile von `stdin` lesen:

```
getLine  :: IO String
```

- ▶ Zeichenkette auf `stdout` ausgeben:

```
putStr   :: String → IO ()
```

- ▶ Zeichenkette mit Zeilenvorschub ausgeben:

```
putStrLn :: String → IO ()
```

# Die do-Notation

- ▶ Syntaktischer Zucker für IO:

```
echo =  
  getLine  
  >>= λs → putStrLn s  
  >> echo
```



```
echo = do  
  s ← getLine  
  putStrLn s  
  echo
```

- ▶ Rechts sind `>>=`, `>>` implizit.
- ▶ Es gilt die **Abseitsregel**.
- ▶ **Einrückung** der ersten Anweisung nach `do` bestimmt Abseits.

# II. Zustandsabhängige Berechnungen

# Funktionen mit Zustand

- ▶ Idee: Seiteneffekt **explizit** machen
- ▶ Funktion  $f : A \rightarrow B$  mit Seiteneffekt in **Zustand**  $S$ :

$$\begin{aligned} f &: A \times S \rightarrow B \times S \\ &\cong \\ f &: A \rightarrow S \rightarrow B \times S \end{aligned}$$

- ▶ Datentyp:  $S \rightarrow B \times S$
- ▶ Komposition: Funktionskomposition und **uncurry**

```
curry    :: (( $\alpha$ ,  $\beta$ )  $\rightarrow$   $\gamma$ )  $\rightarrow$   $\alpha \rightarrow \beta \rightarrow \gamma$ 
uncurry  :: ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow$  ( $\alpha$ ,  $\beta$ )  $\rightarrow$   $\gamma$ 
```

## In Haskell: Zustände **explizit**

- ▶ **Zustandstransformer:** Berechnung mit Seiteneffekt in Typ  $\sigma$  (polymorph über  $\alpha$ )

```
type State  $\sigma$   $\alpha$  =  $\sigma \rightarrow (\alpha, \sigma)$ 
```

- ▶ Komposition zweier solcher Berechnungen:

```
comp :: State  $\sigma$   $\alpha \rightarrow (\alpha \rightarrow$  State  $\sigma$   $\beta) \rightarrow$  State  $\sigma$   $\beta$   
comp f g = uncurry g  $\circ$  f
```

- ▶ Trivialer Zustand:

```
lift ::  $\alpha \rightarrow$  State  $\sigma$   $\alpha$   
lift = curry id
```

- ▶ Lifting von Funktionen:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$  State  $\sigma$   $\alpha \rightarrow$  State  $\sigma$   $\beta$   
map f g = ( $\lambda(a, s) \rightarrow (f a, s)$ )  $\circ$  g
```

# Zugriff auf den Zustand

## ► Zustand lesen:

```
get :: ( $\sigma \rightarrow \alpha$ )  $\rightarrow$  State  $\sigma$   $\alpha$   
get f s = (f s, s)
```

## ► Zustand setzen:

```
set :: ( $\sigma \rightarrow \sigma$ )  $\rightarrow$  State  $\sigma$  ()  
set g s = ((), g s)
```

## Einfaches Beispiel

- ▶ Zähler als Zustand:

```
type WithCounter  $\alpha$  = State Int  $\alpha$ 
```

- ▶ Beispiel: Funktion, die in Kleinbuchstaben konvertiert und **zählt**

```
cntToL :: String  $\rightarrow$  WithCounter String
cntToL [] = lift ""
cntToL (x:xs)
  | isUpper x = cntToL xs 'comp'
                 $\lambda$ ys  $\rightarrow$  set (+1) 'comp'
                 $\lambda$ ()  $\rightarrow$  lift (toLower x: ys)
  | otherwise = cntToL xs 'comp'  $\lambda$ ys  $\rightarrow$  lift (x: ys)
```

- ▶ Hauptfunktion (verkapselt State):

```
cntToLower :: String  $\rightarrow$  (String, Int)
cntToLower s = cntToL s 0
```

# III. Monaden

# Monaden als Berechnungsmuster

- ▶ In `cntToL` werden zustandsabhängige Berechnungen verkettet.
- ▶ So ähnlich wie bei Aktionen!

State:

```
type State  $\sigma$   $\alpha$ 
```

```
comp :: State  $\sigma$   $\alpha$   $\rightarrow$   
      ( $\alpha \rightarrow$  State  $\sigma$   $\beta$ )  $\rightarrow$   
      State  $\sigma$   $\beta$ 
```

```
lift ::  $\alpha \rightarrow$  State  $\sigma$   $\alpha$ 
```

```
map  :: ( $\alpha \rightarrow$   $\beta$ )  $\rightarrow$  State  $\sigma$   $\alpha \rightarrow$   
      State  $\sigma$   $\beta$ 
```

Aktionen:

```
type IO  $\alpha$ 
```

```
( $\gg$ ) :: IO  $\alpha$   $\rightarrow$   
      ( $\alpha \rightarrow$  IO  $\beta$ )  $\rightarrow$   
      IO  $\beta$ 
```

```
return ::  $\alpha \rightarrow$  IO  $\alpha$ 
```

```
fmap  :: ( $\alpha \rightarrow$   $\beta$ )  $\rightarrow$  IO  $\alpha \rightarrow$   
      IO  $\beta$ 
```

Berechnungsmuster: **Monade**

# Monaden als Berechnungsmuster

Eine Monade ist:

- ▶ **philosophisch**: metaphysisches Konzept (Leibnitz)
- ▶ **mathematisch**: durch Operationen und Gleichungen definierte, verallgemeinerte algebraische Theorie (MacLane, Kelly)
- ▶ als **Berechnungsmuster**: **verknüpfbare** Berechnungen mit einem **Ergebnis** (Moggi)
- ▶ in **Haskell**: durch mehrere Typklassen definierte Operationen mit **Eigenschaften** (Wadler)

# Monaden in Haskell

- ▶ Aktion auf Funktionen:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

`fmap` bewahrt Identität und Komposition:

```
fmap id == id
fmap (f ∘ g) == fmap f ∘ fmap g
```

- ▶ Die Eigenschaften **sollten** gelten, können aber nicht überprüft werden.
  - ▶ Standard: *“Instances of Functor should satisfy the following laws.”*

# Monaden in Haskell

## ► Applicative:

```
class Functor f => Applicative f where
  pure  :: a -> f a
  <*>  :: f (a -> b) -> f a -> f b
```

Eigenschaften: links-neutralität, bewahrt Komposition, Homomorphismus:

```
      pure id <*> v == v
pure (o) <*> u <*> v <*> w == u <*> (v <*> w)
      pure f <*> pure x == pure (f x)
      u <*> pure y == pure ($ y) <*> u
```

# Monaden in Haskell

- ▶ Verkettung ( $\gg=$ ) und Lifting (`return`):

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

$\gg=$  ist assoziativ und `return` das neutrale Element:

```
return a >>= k == k a
m >>= return == m
m >>= (\x -> k x >>= h) == (m >>= k) >>= h
```

- ▶ Auch diese Eigenschaften können nicht geprüft werden.
- ▶ Den syntaktischen Zucker (`do`-Notation) gibt's umsonst dazu.

# Monaden mit Möglichkeiten

- ▶ Alternativen:

```
class Applicative f => Alternative f where
  empty :: f a
  <|>   :: f a -> f a -> f a
```

- ▶ Monaden mit Alternative (e.g. List):

```
class (Alternative m, Monad m) => MonadPlus m where
  mzero :: m a
  mzero = empty
  mplus  :: m a -> m a -> m a
  mplus = (<|>)
```

- ▶ Gleichungen: `mzero` Identität für `mplus` und `>>=`, `mplus` assoziativ.

# Beispiele für Monaden

- ▶ Zustandstransformer: `Reader`, `Writer`, `State`
- ▶ Fehler und Ausnahmen: `Maybe`, `Either`
- ▶ Mehrdeutige Berechnungen: `List`, `Set`

# Die Reader-Monade

- ▶ Aus dem Zustand wird nur gelesen:

```
data Reader  $\sigma$   $\alpha$  = R {run ::  $\sigma \rightarrow \alpha$ }
```

- ▶ Instanzen:

```
instance Functor (Reader  $\sigma$ ) where  
  fmap f (R g) = R (f. g)
```

```
instance Monad (Reader  $\sigma$ ) where  
  return a = R (const a)  
  R f  $\gg$ = g = R $  $\lambda s \rightarrow$  run (g (f s)) s
```

- ▶ Nur eine elementare Operation:

```
get :: ( $\sigma \rightarrow \alpha$ )  $\rightarrow$  Reader  $\sigma$   $\alpha$   
get f = R $  $\lambda s \rightarrow$  f s
```

# Fehler und Ausnahmen

- ▶ Maybe als Monade:

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

```
instance Monad Maybe where
  Just a >>= g = g a
  Nothing >>= g = Nothing
  return = Just
```

- ▶ Ähnlich mit `Either`
- ▶ Berechnungsmodell: **Ausnahmen** (Fehler)
  - ▶  $f :: \alpha \rightarrow \text{Maybe } \beta$  ist Berechnung mit möglichem Fehler
  - ▶ Fehlerfreie Berechnungen werden verkettet
  - ▶ Fehler (`Nothing` oder `Left x`) werden propagiert

# Mehrdeutigkeit

- ▶ `List` als Monade:
- ▶ Können wir so nicht hinschreiben, Syntax vordefiniert

```
instance Functor [α] where
  fmap = map
```

```
instance Monad [α] where
  a : as >>= g = g a ++ (as >>= g)
  [] >>= g = []
  return a = [a]
```

- ▶ Berechnungsmodell: Mehrdeutigkeit
- ▶  $f :: \alpha \rightarrow [\beta]$  ist Berechnung mit **mehreren** möglichen Ergebnissen
- ▶ Verkettung: Anwendung der folgenden Funktion auf **jedes** Ergebnis (`concatMap`)

# Beispiel

► Berechnung aller Permutationen einer Liste:

① Ein Element überall in eine Liste einfügen:

```
ins ::  $\alpha \rightarrow [\alpha] \rightarrow [[\alpha]]$   
ins x [] = return [x]  
ins x (y:ys) = [x:y:ys] ++ do  
  is ← ins x ys  
  return $ y:is
```

② Damit Permutationen (rekursiv):

```
perms ::  $[\alpha] \rightarrow [[\alpha]]$   
perms [] = return []  
perms (x:xs) = do  
  ps ← perms xs  
  is ← ins x ps  
  return is
```

# Der Listenmonade in der Listenkomprehension

- ▶ Berechnung aller Permutationen einer Liste:

- 1 Ein Element überall in eine Liste einfügen:

```
ins' ::  $\alpha \rightarrow [\alpha] \rightarrow [[\alpha]]$   
ins' x [] = [[x]]  
ins' x (y:ys) = [x:y:ys] ++ map (y :) (ins' x ys)
```

- 2 Damit Permutationen (rekursiv):

```
perms' ::  $[\alpha] \rightarrow [[\alpha]]$   
perms' [] = [[]]  
perms' (x:xs) = [is | ps ← perms' xs, is ← ins' x ps ]
```

- ▶ Listenkomprehension  $\cong$  Listenmonade

# Zum Nachdenken

Listen sind mehrdeutige Berechnungen mit einer **Reihenfolge**.

- ① Was bedeutet das?
- ② Wie können wir das ändern?
- ③ Wie implementieren wir das?

# IV. IO ist keine Magie

# Referenzen in Haskell

- ▶ Zustand als **finite map** von Referenzen auf Werte

```
data Mem  $\alpha$  = Mem { fresh :: Int, mem    :: Map Int  $\alpha$  }
```

```
type Stateful  $\alpha$   $\beta$  = State (Mem  $\alpha$ )  $\beta$ 
```

- ▶ Ungetypt (`SimpleRefs`)
- ▶ Typ der Werte ist Typparameter des Zustands

```
readRef :: Ref  $\rightarrow$  Stateful  $\alpha$   $\alpha$   
writeRef :: Ref  $\rightarrow$   $\alpha$   $\rightarrow$  Stateful  $\alpha$  ()
```

# Zum Nachdenken

- ▶ Wie können wir Referenzen **typisieren**?
- ▶ Referenzen `Ref  $\alpha$`  verweist auf Werte vom Typ  $\alpha$
- ▶ Was ist das Problem?

# Zum Nachdenken

- ▶ Wie können wir Referenzen **typisieren**?
  - ▶ Referenzen `Ref  $\alpha$`  verweist auf Werte vom Typ  $\alpha$
- ▶ Was ist das Problem? Der Typ von `Mem`:

```
data Mem  $\alpha$  = Mem { fresh :: Int, mem    :: Map Int  $\alpha$  }
```

- ▶ Getypt (`Refs`): nutzt **dynamische Typen** (`Dynamic`)

```
data Mem = Mem { fresh :: Int, mem    :: Map Int Dynamic }
```

```
readRef :: Typeable  $\alpha$   $\Rightarrow$  Ref  $\alpha$   $\rightarrow$  Stateful  $\alpha$   
writeRef :: Typeable  $\alpha$   $\Rightarrow$  Ref  $\alpha$   $\rightarrow$   $\alpha$   $\rightarrow$  Stateful ()
```

# Implizite vs. explizite Zustände

- ▶ Wie funktioniert jetzt IO?
- ▶ Nachteil von **State**: Zustand ist **explizit**
  - ▶ Kann **dupliziert** werden
- ▶ Daher: Zustand **implizit** machen
  - ▶ Datentyp **verkapseln** (kein **run**)
  - ▶ Zugriff auf **State** nur über elementare Operationen

# Aktionen als Zustandstransformationen

- ▶ **Idee:** Aktionen sind Transformationen auf Systemzustand  $S$
- ▶  $S$  beinhaltet
  - ▶ Speicher als Abbildung  $A \rightarrow V$  (Adressen  $A$ , Werte  $V$ )
  - ▶ Zustand des Dateisystems
  - ▶ Zustand des Zufallsgenerators
- ▶ In Haskell: Typ `RealWorld`
  - ▶ “Virtueller” Typ, Zugriff nur über elementare Operationen
  - ▶ Entscheidend nur Reihenfolge der Aktionen

# Zusammenfassung

- ▶ War das jetzt **reaktiv**?
- ▶ Haskell ist **funktional**
- ▶ Für eine reaktive Sprache fehlt **Nebenläufigkeit**
- ▶ Nächstes Mal:
  - ▶ Monaden **komponieren** — Monadentransformer
- ▶ Danach: Nebenläufigkeit in Haskell und Scala