Bachelor-Seminar:

Haskell

— Angewandter „abstract nonsense“ —

https://quoteme.github.io/bachelor_seminar_haskell

„Du programmierst nicht Haskell —
Haskell programmiert dich.“

Warum Haskell lernen?

- Funktionale Programmierung lernen
- immutability
  - weniger bugs
  - Parallelität
- arbeiten mit unendlichen Datenstrukturen
- algebraisches Typensystem
- besser Programmieren / Probleme lösen
- high-level mit Performance von low-level

Show Language Network

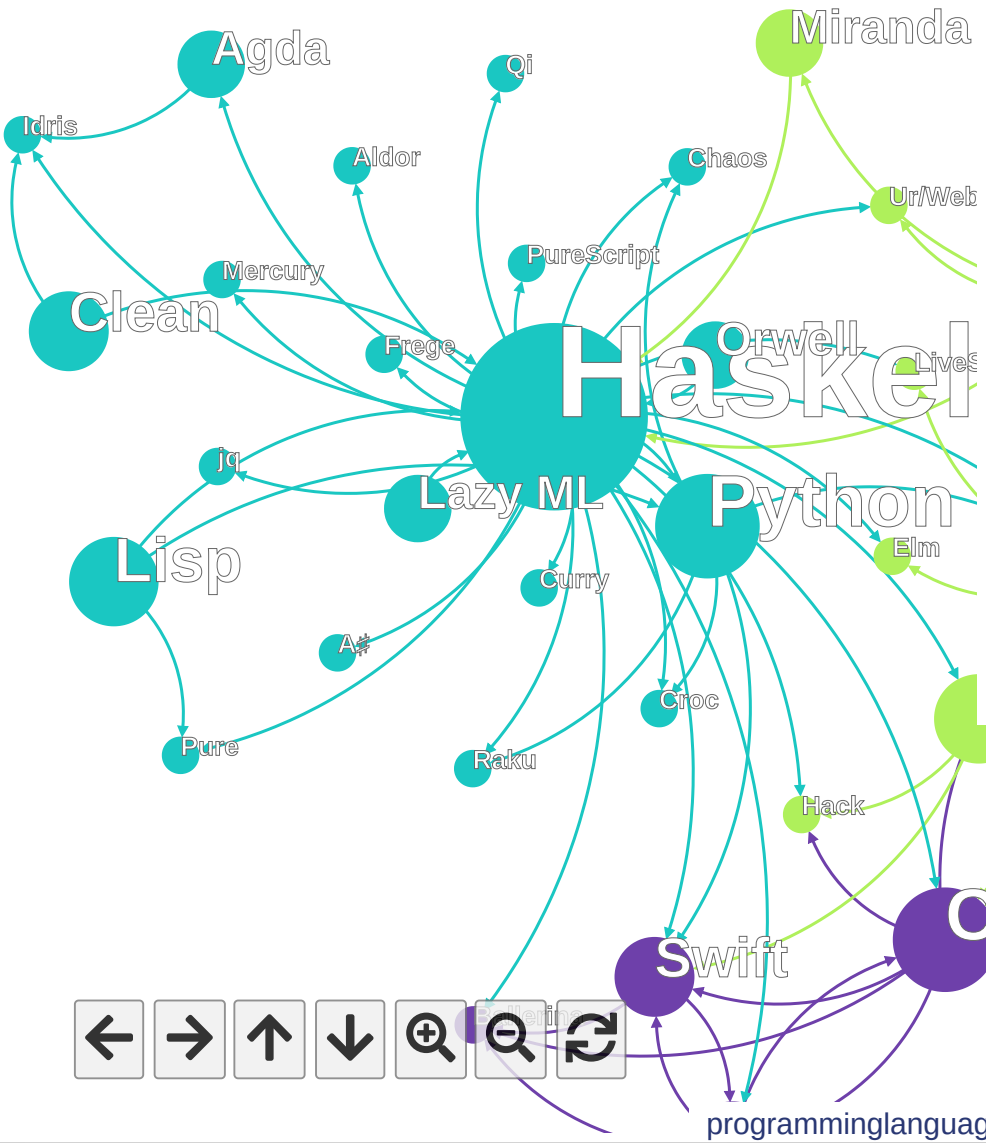
Haskell

This directed network graph shows 32 programming languages connected by 61 edges.

Network Info

Two language nodes are linked by an edge if one language influenced the other. Node size is determined by its outdegree, i. e. the number of languages influenced. All edges have the same weight, because the "amount of influence" is unmeasured, although it certainly varies.

Node colors are assigned based on the Louvain method for community detection in networks. Nodes with the same color belong to the same community. Edges get the color of the source node, i. e. the language that influenced the



Haskell

Influenced by

Clean • Lazy ML • Lisp • ML • Miranda • Orwell

Influenced

Agda • Aldor • A♯ • Ballerina • C# • Chaos • Croc • Curry • Elm • F# • Frege • Hack • Idris • LiveScript • Mercury • Pure • PureScript • Python • Qi • Raku • Rust • Scala • Swift • Ur/Web • jq

More info

- Haskell
- wikidata.org/wiki/Q34010


LEARN
A FRAMEWORK
LEARN
A LIBRARY
LEARN A
PROGRAMMING
LANGUAGE
LEARN
MATH
imgflip.com

Was ist Haskell?

Haskell is a computer programming language. In particular, it is a **polymorphically statically typed**, **lazy**, **purely functional language**,

quite different from most other programming languages

— wiki.haskell.org/Introduction

Was heißt „**functional language**“?

- Programme sind (verschachtelte) Funktionen

- Programm ausführen ⇔ Funktion auswerten

```haskell
-- Main.hs
-- Beispiel Hello world Programm

main :: IO ()
main = putStrLn "Hello, World!"
```

```
[haskell@curry] $ ./Main
 Hello, World!
```

```haskell
test :: String -> String
test s = s ++ " Haskell ist Cool!"

-- Hier können wir nun Funktionen komponieren
main :: IO ()
main = putStrLn (test "Hello, World!")
```

```
[haskell@curry] $ ./Main
 Hello, World! Haskell ist Cool!
```

```
(f . g) x = f (g x)
-- (.) :: (b -> c) -> (a -> b) -> a -> c

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

Was heißt „lazy“?

- Nichts berechnen, was nicht gerade gebraucht wird
- Erlaubt u.A., unendliche Datenstrukturen zu definieren

```
r = [1..] -- Analog zu {1,...}
s = take 10 t
t = [":) " | _ <- s] -- Analog zu (x for x in y)
```

$$r = [1, 2, 3, 4, \ldots]$$

$$s = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

$$t = [\text{":)"}, \text{":)"}, \text{":)"}, \text{":)"}, \text{":)"}, \text{":)"}, \text{":)"}, \text{":)"}, \text{":)"}, \text{":)"}]$$

```haskell
p = [t | t <- [1..], isPrime t]
  where
    isPrime :: Integer -> Bool
    isPrime 1 = False
    isPrime n = all (\x -> n `mod` x /= 0) [2..n-1]
```

p = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, …]

- λ-Funktion mit Param. x
- schicke x auf n mod x
- teste ob n mod x ≠ 0

```haskell
p = [t | t <- [1..], isPrime t]
  where
    isPrime :: Integer -> Bool
    isPrime 1 = False
    isPrime n = all (\x -> n `mod` x /= 0) [2..n-1]
```

```haskell
p = [t | t <- [1..], isPrime t]
  where
    isPrime :: Integer -> Bool
    isPrime 1 = False
    --                          Infix
    isPrime n = all (\x -> n `mod` x /= 0) [2..n-1]
```

```haskell
p = [t | t <- [1..], isPrime t]
  where
    isPrime :: Integer -> Bool
    isPrime 1 = False
    --                        Prefix
    isPrime n = all (\x -> mod (n x) /= 0) [2..n-1]
```

```haskell
p = [t | t <- [1..], isPrime t]
  where
    isPrime :: Integer -> Bool
    isPrime 1 = False
    isPrime n = all (\x -> n `mod` x /= 0) [2..n-1]
```

```haskell
p = [t | t <- [1..], isPrime t]
  where
    isPrime :: Integer -> Bool
    isPrime 1 = False
    isPrime n = (\x -> n `mod` x /= 0) `all` [2..n-1]
```

```haskell
-- :t all
all :: Foldable t => (a -> Bool) -> t a -> Bool
```

```haskell
erstePrimzahl = head p
alleAndderen = tail p

ersteZahlUeber8000 :: [Integer] -> Integer
ersteZahlUeber8000 (x:xs) = if x > 8000 then x else ersteZahlUeber8

elementAnStelle10 :: [a] -> a
elementAnStelle10 x = x !! 10
```

haskell wiki: How to work on lists

Was heißt „polymorphically typed“?

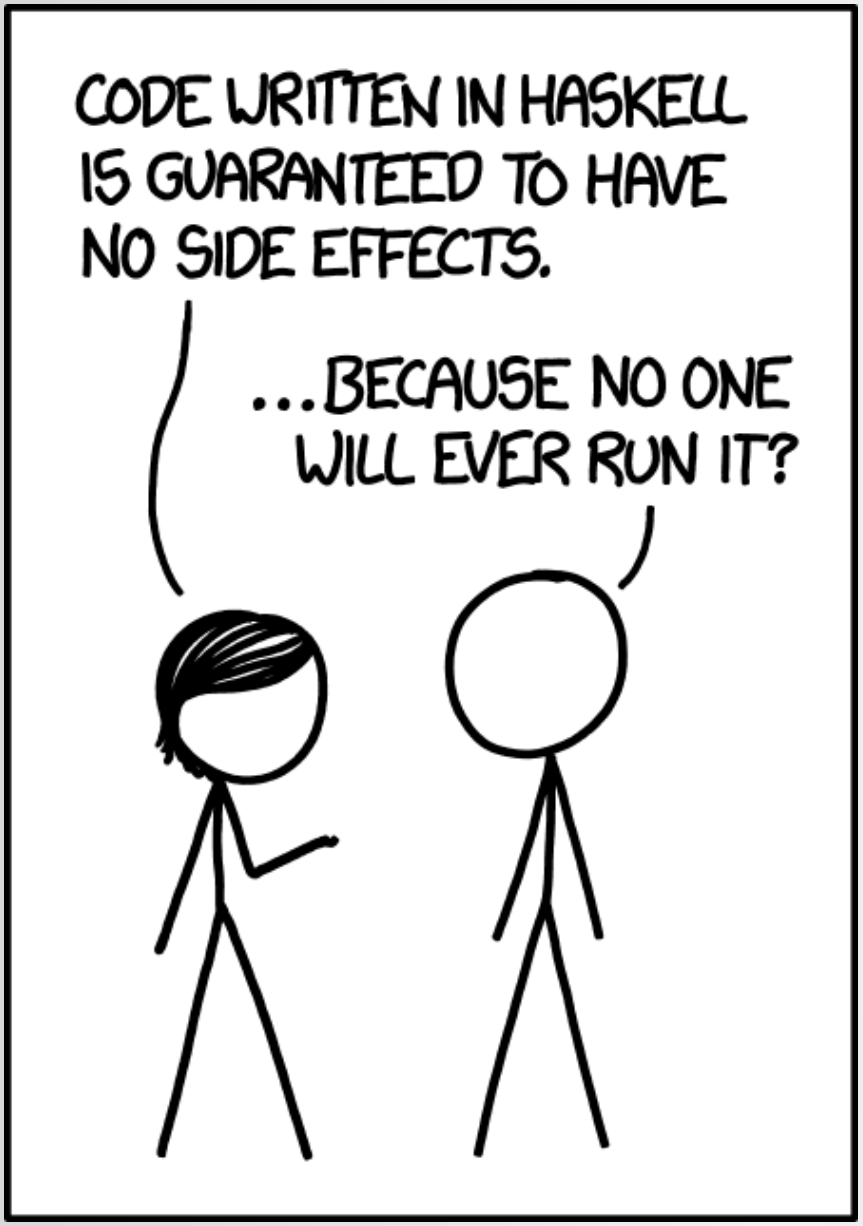
- Ein Wert: mehrere Typen
- Viele verschiedene Arten in Haskell:
  - Parametrisch
  - Ad-hoc, ...

```haskell
-- parametrisch
-- funktioniert für jede Funktion * -> *
id :: a -> a
map :: (a -> b) -> [a] -> [b]
[] :: [a]

-- ad-hoc
-- eigene implementation für jeden Typ
(==) :: Eq a => a -> a -> Bool
elem :: (Eq a) => a -> [a] -> Bool
```

Was heißt „pure“?

- Funktionen haben keine Seiteneffekte (keine Mutation, Variablen, I/O)
- Funktionen geben das gleiche Ergebnis für die gleichen Eingaben zurück


CODE WRITTEN IN HASKELL IS GUARANTEED TO HAVE NO SIDE EFFECTS.
...BECAUSE NO ONE WILL EVER RUN IT?

```
#!/usr/bin/env python3
def unrein(x):
    x = 1

if __name__ == "__main__":
    x = 0
    unrein(x)
    print(x)
```

```haskell
module Main where

main :: IO ()
main = putStrLn "Hello, World!" -- unrein?
```

```
[haskell@curry] $ ./Main
                  Hello, World!
```

„Haskell is [...] quite different from most other programming languages“

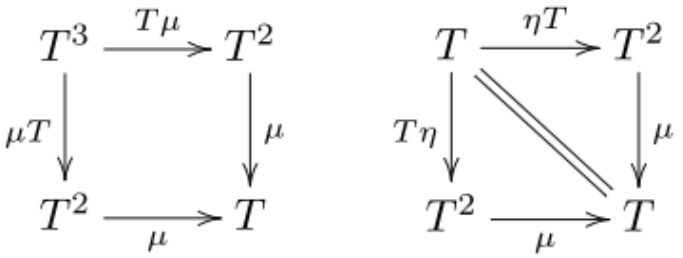
```haskell
module Main where

main :: IO () -- ← Monade
main = putStrLn "Hello, World!"
```

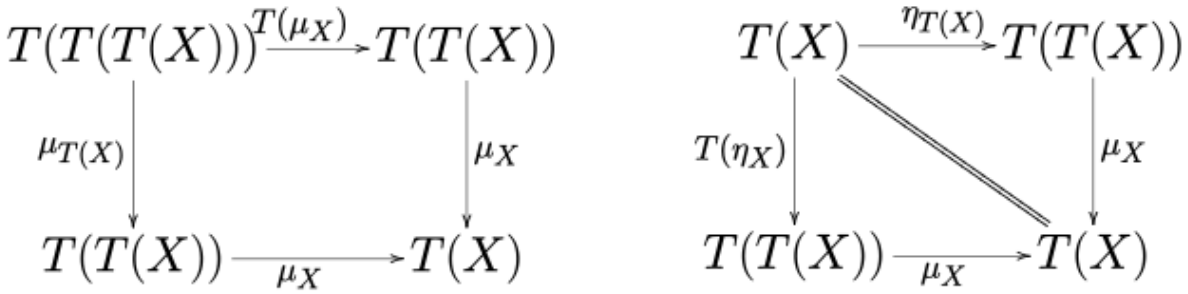
Throughout this article C denotes a category. A *monad* on C consists of an endofunctor $T: C \to C$ together with two natural transformations: $\eta: 1_C \to T$ (where 1_C denotes the identity functor on C) and $\mu: T^2 \to T$ (where T^2 is the functor $T \circ T$ from C to C). These are required to fulfill the following conditions (sometimes called coherence conditions):

- $\mu \circ T\mu = \mu \circ \mu T$ (as natural transformations $T^3 \to T$); here $T\mu$ and μT are formed by "horizontal composition"
- $\mu \circ T\eta = \mu \circ \eta T = 1_T$ (as natural transformations $T \to T$; here 1_T denotes the identity transformation from T to T).

We can rewrite these conditions using the following commutative diagrams:



See the article on natural transformations for the explanation of the notations $T\mu$ and μT, or see below the commutative diagrams not using these notions:

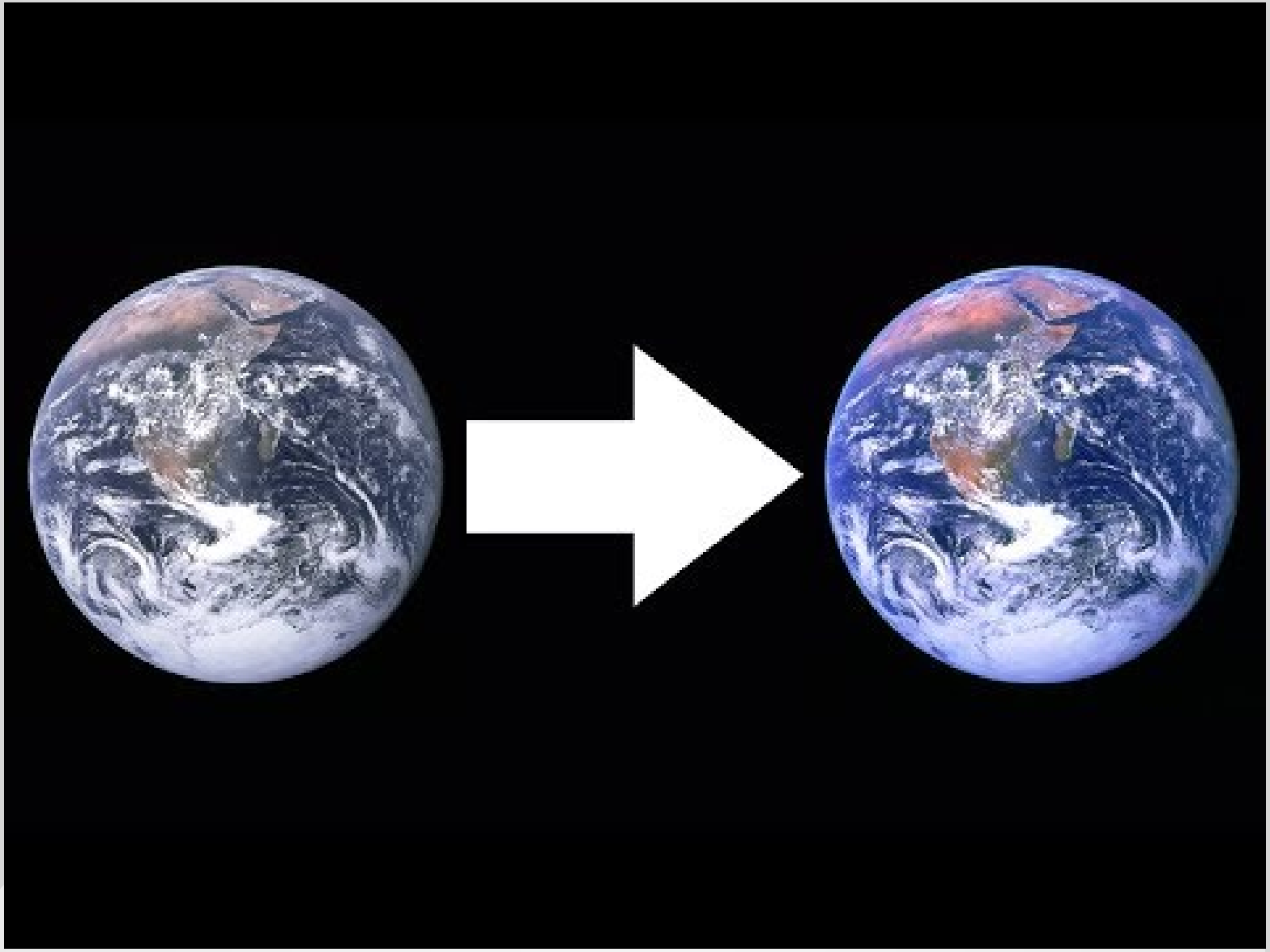


```
-- :t putStrLn
putStrLn :: String -> IO ()
-- :t "Hello, World!"
"Hello, World!" :: String
```

$$\text{putStrLn} : \text{String} \longrightarrow \text{IO}\ ()$$

$$\text{putStrLn}(\text{"Hello, World!"}) = ???$$

putStrLn : String ⟶ IO ()




trapd in IO monad
plz help

```haskell
-- Haskell logo
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

👉 Weiteres zu Monoiden und verbotene coole Tricks

Was heißt „statically typed“?

- Werte haben immer einen Typen (typed)
- Alle Typen jedes Wertes sind beim Kompilieren bekannt (statically)

```haskell
module Types where

s :: String
s = "Hallo :)"

i :: Int
i = 5

f :: Float
f = 3.141592

todouble :: Float -> Double
todouble x = realToFrac x

proj :: a -> String
proj etwas = "😎"
```

```haskell
module Datatypes where

data Liste a = LeereListe | Listenelement (Liste a)  a

data Arbeiter = Angestellter
              | Manager
              | Hilfspersonal
```

```haskell
data Liste a = LeereListe | Listenelement (Liste a)  a

data Arbeiter = Angestellter
              | Manager
              | Hilfspersonal

listeErzeugen :: Arbeiter -> Liste Arbeiter
listeErzeugen Angestellter  = Listenelement LeereListe A
listeErzeugen Manager       = Listenelement LeereListe M
listeErzeugen Hilfspersonal = Listenelement LeereListe H

elementHinzufuegen :: Liste Arbeiter
                   -> Arbeiter
                   -> Liste Arbeiter
elementHinzufuegen x y = Listenelement x y
```

Wir erkennen, dass

```
data Liste a = LeereListe | Listenelement (Liste a)  a
```

eine Monoidstruktur hat!

Monoidstruktur:

- Es gibt eine binäre Operation (Listen Konkatenieren) (Magmastruktur)
- Die Operation ist Assoziativ (Halbgruppenstruktur)
- Es gibt ein Neutrales Element (leere Liste) (Monoidstruktur)

```haskell
instance Semigroup (Liste a) where
  (<>) LeereListe LeereListe = LeereListe
  (<>) LeereListe v = v
  (<>) v LeereListe = v
  (<>) v (Listenelement w z) = Listenelement (v <> w) z

instance Monoid (Liste a) where
  mempty = LeereListe
```

Jetzt gehört unser Listendatentyp zur Klasse aller Typen, welche eine Monoidstruktur haben!

Triviale Instanzen können automatisch abgeleitet werden

```haskell
data Liste a = LeereListe | Listenelement (Liste a)  a
  deriving (Show)

data Arbeiter = Angestellter
              | Manager
              | Hilfspersonal
                deriving (Show)
```

```haskell
show ( Listenelement LeereListe "hallo" )
-- Ausgabe: Listenelement LeereListe "hallo" :: String
```

Und warum so „umständlich“?

Wir können nun **alle** Funktionen, welche für Monoide geschrieben wurden verwenden

Es gibt eine kanonische Monoidstruktur auf Abbildungen in Monoide.

```
f :: a -> Liste a
f x = Listenelement LeereListe x
g :: a -> Liste a
g x =  Listenelement LeereListe x
    <> Listenelement LeereListe  x

f ":)" -- Wird zu ...
Listenelement LeereListe ":)"
g ":)" -- Wird zu ...
Listenelement (Listenelement LeereListe ":)") ":)"
(f <> g) ":)" -- Wird zu ...
Listenelement (Listenelement (Listenelement LeereListe ":)") ":)") ":)"
```

```haskell
module ListBeispiel where

import Data.List
import Data.Ord

test :: [String] -> [String]
-- Allgemeiner: test :: (Foldable t, Ord (t a)) => [t a] -> [t a]
test = sortBy (comparing length <> compare)
```

```haskell
test = sortBy (comparing length <> compare)

test ["hallo", "wie", "geht", "es", "dir", "?"] -- Wird zu...
-- => ["?","es","dir","wie","geht","hallo"]
```

```haskell
test = sortBy (comparing length <> compare)
     -- Umformung
     = sortBy (comparing length) <> sortBy compare
```

```
-- Hilfe zu den Typen
sortBy    :: (a -> a -> Ordering) -> [a] -> [a]       -- Documentation
comparing :: Ord a => (b -> a) -> b -> b -> Ordering -- Documentation
length    :: Foldable t => t a -> Int               -- Documentation
compare   :: Ord a => a -> a -> Ordering            -- Documentation

-- Typen von Kompositionen
comparing length :: Foldable t => t a -> t a -> Ordering
```

- Hat Haskell eurer Meinung nach eine Zukunft?
- Haskell gut/schlecht?
- Würdet ihr Haskell probieren/verwenden?
- Mathematische Konstrukte statt Objekte?

Die Aufgaben finden Sie hier!