Workshop 4
Haskell

06.02.2024

Jean-Luc Busch

Nicole Enzelsberger

Gregory Hammond

David Miller

Jonathan Seltmann

Ziele

Warum geht es heute um Haskell?

- Einblick in die funktionelle Programmierwelt
- Erlernen der Basics der Programmierung
- Gefühl für Stärken und Schwächen der Sprachen entwickeln
- Mögliche Einsatzgebiete

Agenda

Block 1

Einführung funktionale Programmierung mit Haskell

Block 2

Rekursion und Listenverarbeitung

Block 3

Abstraktion und Typsystem

Block 4

Modules und I/O

Block 5

Memoization, Currying, Funktoren, Monoids, Monads

Installation

Hat alles geklappt?

Installation

Hat alles geklappt?

Falls noch nicht geklappt hat?

- siehe Anleitung im Repo

Alternative:
https://godbolt.org/

Einführung funktionale Programmierung mit Haskell

Block 1 - Jonathan

Funktionale Programmierung

Einführung

- **Definition:** Programmierparadigma, das Funktionen als zentrale Bausteine verwendet.

- Typische Anwendungsfälle:
  - Datenverarbeitung (z. B. Big Data, Analytics)
  - Parallelisierung/Concurrency
  - Transformation von Datenströmen (z. B. Reactive Programming)

Paradigmen

- Don’t mutate data
- Use pure functions
- Use expressions and declarations
- Polymorphism

Don’t mutate data

- **Immutability:** Datenstrukturen sind unveränderlich
  - Änderungen erzeugen neue Kopien
- **Vorteile:**
  - Einfacheres Debugging und Testen
  - Sichere Nebenläufigkeit

Don’t mutate data - Beispiel in Python

Untitled-1 Python

```python
# Imperativ:
array.append(4)

# Funktional:
new_array = array + [4]
```

Use Pure Functions

- **Definition:**
  - Gleiche Eingaben -> Gleicher Output
  - Keine Seiteneffekte (z.B. Logging, globale Variablen)
- **Vorteile:**
  - Vorhersagbares Verhalten
  - Leichtere Wiederverwendbarkeit und Testbarkeit

Use Pure Functions - Beispiel in Python

Untitled-1 Python

```python
# Pure Funktion
def add(a, b):
  return a + b

# Nicht rein: ändert externe Zustände
def add_with_logging(a, b):
  print(f"Adding {a} and {b}")
  return a + b
```

Use Expressions and Declarations

- **Deklarativ vs. Imperativ:**
  - Imperativ: „Wie“ etwas gemacht wird
  - Deklarativ: „Was“ gemacht werden soll
- **Vorteil von Expressions:**
  - Klarheit
  - Geringere Fehleranfälligkeit

Use Expressions and Declarations - Beispiel in Python

Untitled-1 Python

```python
# Imperativ
result = 0
for i in range(10):
  result += i

# Funktional
result = sum(range(10))
```

Polymorphismus

- **Definition:**
  - Funktionen arbeiten mit unterschiedlichen Datentypen, ohne spezifischen Code für jeden Typ zu schreiben.
- **Realisiert durch:**
  - **Higher Order Functions**: Funktionen, die andere Funktionen akzeptieren oder zurückgeben.
  - **Generische Typen**.

Haskell

Funktionale Programmiersprache

- Erster Release: 1990
- Stable Release: 2010

Wichtige Faktoren für Anwendungsgebiete

- Mathematische Basis, Immutabilität
- Transparenz
- Statisches Typsystem
- Deterministische Wirkung
- Eignung für formale Verifikation
- Abstraktion und Modularität

Einsatzgebiete

- Finanzsysteme
- Compilerentwicklung
- Formale Methoden
- Lehre

Kennt ihr Unternehmen, die auf Haskell setzen? Bekannte Programme?

Nachteile und Limitierungen

- Hoher Lernaufwand
- Kleine Community
- Kleines Ökosystem
- Keine optimale Performance

Primitive Datentypen und Operationen - Mathe

Untitled-1 Haskell

```haskell
-- Nummern existieren
3 -- 3

-- Mathe funktioniert wie Mathe funktioniert
1 + 1 -- 2
8 - 1 -- 1
10 * 2 -- 20
35 / 4 -- 8.75

-- Integerdivision
35 `div` 4 -- 8
```

Primitive Datentypen und Operationen - Boolean

Untitled-1 Haskell

```haskell
-- Boolsche Werte sind Primitives
True
False

-- Operationen
not True -- False
not False -- True
True && False -- False
True || False -- True
1 == 1 -- True
1 /= 1 -- False
1 < 10 -- True
```

Strings

Untitled-1 Haskell

```haskell
-- Strings und Characters
“Dies ist ein String.”
‘a’ -- character
‘Man kann keine einfachen Codierungszeichen für Strings verwenden.’ -- error!

-- Stringkonkatenierung
“Hello“ ++ “ world!” -- “Hello world!”

-- Ein String ist eine List von Characters
['H', 'e', 'l', 'l', 'o'] -- "Hello"
```

Lists und Ranges

Untitled-1 Haskell

```haskell
-- Jedes Element einer Liste muss den gleichen Typ haben.
-- Diese zwei Listen ist gleich:
[1, 2, 3, 4, 5]
[1..5]

-- Ranges sind Magie
['A'..'F'] -- "ABCDEF"
[0, 2..10] -- [0, 2, 4, 6, 8, 10]
[5..1] -- [] (Haskell inkrementiert standardmäßig)
[5, 4..1] -- [5, 4, 3, 2, 1]
```

Lists

Untitled-1 Haskell

```haskell
-- Listenindexing
[1..10] !! 3 -- 4 (zero-based indexing)

-- Listen konkatenieren
[1..5] ++ [6..10]

-- Vorne anfügen
0:[1..5] -- [0, 1, 2, 3, 4, 5]
```

Weitere Listenoperationen

Untitled-1 Haskell

```haskell
-- Weitere Listenoperationen
head [1..5] -- 1
tail [1..5] -- [2, 3, 4, 5]
init [1..5] -- [1, 2, 3, 4]
last [1..5] -- 5

-- List Comprehensions
[x*2 | x <- [1..5]] -- [2, 4, 6, 8, 10]

-- Mit einem Conditional
[x*2 | x <- [1..5], x*2 > 4] -- [6, 8, 10]
```

Lazy Evaluation

Untitled-1 Haskell

```haskell
-- Unendliche Listen
[1..] -- Alle natürlichen Zahlen
[1..] !! 999 -- 1000 - Elemente 1 - 1000 sind jetzt evaluiert
```

Tuples

Untitled-1 Haskell

```haskell
-- Fixe Länge, unterschiedliche Types
let tuple = (“haskell”, 1)

-- Zugriff auf Elemente eines Paares
fst tuple -- "haskell"
snd tuple -- 1
```

Funktionen

Untitled-1 Haskell

```haskell
-- Einfache Funktion mit zwei Parametern
add a b = a + b
-- oder
let add a b = a + b

-- Funktionsaufruf
add 1 2 -- 3
-- oder
1 `add` 2 -- 3
```

Funktionen - Guards

Untitled-1 Haskell

```haskell
-- Guards: einfaches Branching für Funktionen
fib x
  | x < 2 = 1
  | otherwise = fib (x - 1) + fib (x - 2)
```

Funktionen - Function Composition und Precedence

Untitled-1 Haskell

```haskell
-- Function Composition
foo = (4*) . (10+)
-- 4*(10+5) = 60
foo 5 -- 60

-- Precedence
-- vorher
even (fib 7) -- false
-- gleichwertig
even $ fib 7 -- false
-- mit Composition
even . fib $ 7 -- false
```

Übungen (30 min)

1. Schaltjahr
2. Bob
3. Darts (Bonus, optional)

Rekursion und Listenverarbeitung

Block 2 - David

Warum ist Rekursion in Haskell wichtig?

- Haskell ist eine rein funktionale Sprache
  - Schleifen wie **for** oder **while** gibt es nicht
- Rekursion ersetzt iterative Konstrukte
- Rekursion in Haskell wird durch Pattern Matching und Guards unterstützt

Beispielcode (nur als Illustration) Haskell

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

Pattern Matching

- ermöglicht eine saubere Definition von Basis- und Rekursionsfällen
- Vorteile: Lesbarkeit, Eleganz und Fehlervermeidung

Längenberechnung einer Liste — Haskell

```haskell
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

Listenverarbeitung in Haskell

- zentraler Datentyp in Haskell
- Funktionen auf Listen basieren oft auf Rekursion
- Syntax für Listen: [1, 2, 3] oder 1 : 2 : 3 : []
- Wichtige Operationen:
  - head, tail, null
  - Konstruktion: (:)
  - Rekursive Definition

Summenberechnung einer Liste — Haskell

```haskell
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs
```

Effizienz und Tail-Rekursion

- Vermeidet unnötige Stack-Frames ➔ effizienter Speicherverbrauch
- Rekursiver Aufruf ist die letzte Operation
- Funktion kann als Schleife optimiert werden (Tail Call Optimization)
- Oft mit einem Akkumulator zur Speicherung von Zwischenergebnissen
- Reduziert das Risiko eines Stack Overflow

Praktisches Beispiel - Fakultät

- Vergleich: Naive Rekursion vs. Akkumulator-Ansatz

Naive Rekursion — Haskell

```haskell
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Tail Rekursion — Haskell

```haskell
factorialTail :: Integer -> Integer -> Integer
factorialTail 0 acc = acc
factorialTail n acc = factorialTail (n - 1) (n * acc)

factorial :: Integer -> Integer
factorial n = factorialTail n 1
```

Praktisches Beispiel - Fakultät

Tail Rekursion — Haskell

```haskell
factorialTail :: Integer -> Integer -> Integer
factorialTail 0 acc = acc
factorialTail n acc = factorialTail (n - 1) (n * acc)

factorial :: Integer -> Integer
factorial n = factorialTail n 1
```

Übung 1 & 2 - 15 min

Aufgabe 1: Implementieren Sie die Funktion map rekursiv

Beschreibung: Schreibe eine Funktion, die auf jedes Element einer Liste eine Funktion anwendet

map — Haskell

```haskell
map :: (a -> b) -> [a] -> [b]
```

Zusatzaufgabe 2: Implementieren Sie die Funktion zip rekursiv

Beschreibung: Kombiniere zwei Listen in eine Liste von Paaren

zip — Haskell

```haskell
zip :: [a] -> [b] -> [(a, b)]
```

Übung 3 - 15 min

Aufgabe:
Implementieren Sie fibTail, eine effiziente Version mit Tail-Rekursion

Naive Fibonacci — Haskell

```haskell
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Zusammenfassung

- Rekursion ist ein fundamentaler Bestandteil der Programmierung in Haskell
- Listenverarbeitung in Haskell basiert stark auf rekursiven Ansätzen
- Optimierungen wie Tail-Rekursion können helfen, effizienteren Code zu schreiben

Abstraktion und Typsystem

Block 3 - Nicole

Haskell’s Typsystem: stark + statisch

Statische Typisierung:

- Typen werden beim Kompilieren überprüft, nicht zur Laufzeit

Welche Sprachen sind noch statisch?

- *C, C++, C#, Java, Go, Rust, Kotlin, TypeScript?, ...*

Welche Sprachen sind dynamisch?

- *Python, PHP, Ruby, JavaScript, ...*

Haskell’s Typsystem: stark + statisch

Definition von “stark” / “schwach” ist umstritten!

Starke Typisierung:

- keine implizite Konvertierung (automatische Umwandlung)
- explizite Konvertierung möglich (manuelle Umwandlung)

stark — Haskell

```haskell
5 + "3"; -- Fehler
```

schwach — JavaScript

```javascript
console.log(5 + "3"); // Ergebnis: "53"
```

stark — Python

```python
5 + "3"; # Fehler
```

schwach — PHP

```php
echo 5 + "3"; // Ergebnis: 8
```

Typisierung Übersicht

Sprache	**Statisch?** **(zur Kompilierzeit festgelegt)**	**Stark?** **(keine impliziten Umwandlungen)**	**Erklärung**
Haskell	✅ Ja	✅ Ja	
Rust	✅ Ja	✅ Ja	
Java	✅ Ja	✅ Ja	
C	✅ Ja	❌ Nein	erlaubt implizite Casts (z. B. int -> float)
Python	❌ Nein	✅ Ja	dynamisch, keine impliziten Casts
JavaScript	❌ Nein	❌ Nein	automatische Konvertierungen

Typinferenz

- Haskell erkennt Typen automatisch
- Typen: erster Buchstabe groß

Typinferenz Haskell

```
ghci> :t (True, "Hello", 'b')

ghci> addThree x y z = x + y + z
ghci> :t addThree
```

(True, “Hello”, ‘b’) :: (Bool, [Char], Char)

addThree :: Num a => a -> a -> a -> a

Typinferenz Haskell

```
ghci> :t 'a'
'a' :: Char

ghci> :t "Hello!"
"Hello!" :: [Char]

ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)

ghci> :t 4 == 5
4 == 5 :: Bool
```

Algebraische Datentypen (ADTs)

Primitive Datentypen:

- Int, Double, Char, Bool,...

Algebraische Datentypen:

- zusammengesezter Datentypen
- Hauptmechanismus zur Typkonstruktion
- Unterscheidung:
  - Produkttyp
  - Summentyp

ADTs: Aufzählungstyp

- “entweder - oder”
- feste Menge an diskreten Werten
- im Prinzip *enums*
  - trotzdem Unterschiede zu enums z. B. in Rust

ADT — Haskell

```haskell
data Color = Red | Blue | Green

say :: Color -> String
say Red = "You are Red!"
say Blue = "You are Blue!"
say Green = "You are Green!"
```

https://www.informatik.uni-bremen.de/agbkb/lehre/pi3/folien/Algebraisch.pdf

ADTs: Summentypen

- Varianten haben zusätzliche Werte
- Datentypen mit mehreren, unterschiedlichen Konstruktoren

type constructor

value constructor

ADT — Haskell

```haskell
data Person = Person String String Int

data Person =
    Student String String Int Double
  | Teacher String String
  | Child String String
deriving (Eq, Show)
```

Typdeklarationen

- auch Funktionen haben einen festen Typen!
- Annotation nicht nötig, aber üblich!
  - ab sofort sollten eure Funktionen immer Typen deklarieren

Typdeklaration — Haskell

```haskell
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

Typvariablen

- vgl. Generics (Java), Templates (C++)
- *head* ist eine polymorphe Funktion

Typvariablen — Haskell

```
ghci> :t head
head :: [a] -> a
```

Polymorphismus

- in C++/Java: ***Subtype-Polymorphismus***
  - Vererbung und Interfaces in objektorientierten Programmiersprachen
  - Base-Klasse definiert Verhalten
  - Sub-Klassen verändern und/oder erweitern Verhalten

- in Haskell: ***Parametric-Polymorphismus***
  - definiere Funktionen oder Datentypen mit generischem Typen
  - können für verschiedene konkrete Typen verwendet werden

Beispiel: Polymorphe Typen *Maybe* und *Either*

Maybe:

Maybe Haskell

```haskell
data Maybe a = Nothing
             | Just a
```

Either:

Maybe Haskell

```haskell
data Either a b = Left a
                | Right b
```

z. B.
– Fehlermeldung (String)
– erfolgreiches Ergebnis

Typklassen

- Gruppen von Typen, die bestimmtes Verhalten definieren
  - vgl. Interfaces in anderen Sprachen
  - nicht mit regulären OOP-Klassen zu verwechseln!

Typklassen — Haskell

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

ghci> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

Beispiele für Typklassen

- Eq
  - Gleicheit von Datentypen prüfen
- Num
  - numerische Werte (Int, Float, Double, ...)
- Ord
  - Vergleichbarkeit (<, >, >=, ...) schaffen
- Show
  - Datentypen können als String ausgegeben werden

Überblick: Typen, Typvariablen, Typklassen

Konzept	**Bedeutung**	**Vergleich**
Primitiver Typ (Int, Bool, String)		
Algebraischer Typ (data … =)	Zusammengesetzter Datentyp	~ Structs, Enums
Typvariable (a)	Platzhalter für beliebigen Typ	~ Generics
Typklasse (Eq, Show, Ord)	Definiert Menge von Operationen, die ein Typ unterstützen muss	~ Interfaces

Pure, deterministische Funktionen

Wiederholung: Grundsätze funktionaler Programmierung:

- Keine Mutation, alles ist *immutable*
- Keine Seiteneffekte
- Determinismus: gleiche Funktion mit gleichem Input -> gleicher Output

Mutation — Python

```
>> a = [1,2,3]
>> b = a.reverse()
>> a
[3,2,1]
>> b
(No output)
```

Mutation — Haskell

```
ghci> a = [1,2,3]
ghci> reverse a
[3,2,1]
ghci> a
[1,2,3]
```

Higher-Order Functions

- nehmen Funktion als Argument an und/oder
- geben Funktion als Ergebnis zurück

- ➜ Funktionen werden als Werte behandelt
- z. B. map

In Konzepten denken!

- “Wholemeal programming”
  - verarbeite Datenstrukturen als Ganzes, statt Element für Element
  - weg von “Indexitis” in C- oder Java-Sprachen

Sum * 3 — Java

```java
int acc = 0

for (int i = 0; i < my_list.length; i++) {
  acc = acc + 3 * my_list[i];
}
```

Sum * 3 — Haskell

```haskell
sum (map (3*) my_list)
```

Übungsaufgabe: Zauberschule (~30 Minuten)

vs.

vs.

Optional: Übungsaufgabe 2: Maybe

Modules und I/O

Block 4 - Jean-Luc

Was haben wir bisher gelernt?

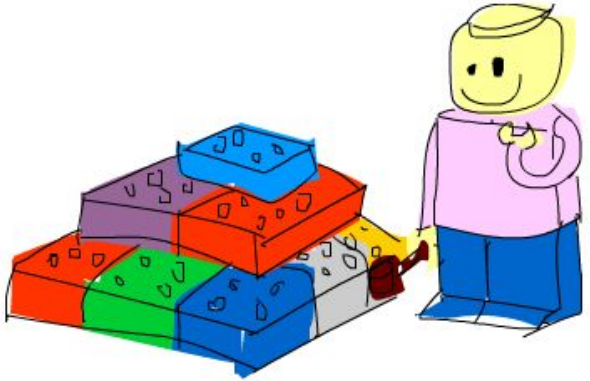
- Typischen Grundlagen einer Programmiersprache
  - Datentypen, Funktionen, Syntax usw.

- wenn wir etwas programmiert haben, dann war alles von scratch

- auf Dauer aber nicht wirklich die Lösung

Modules

- Haskell hat eine weitläufige Sammlung von Modules
- Gute Übersicht findet man unter: https://downloads.haskell.org/ghc/latest/docs/libraries/
- man kann eigene Module/Submodule zu schreiben
- alle bisherigen genutzten Funktionen sind Teil der Standardbibliothek (Prelude) und sind immer geladen.



Importieren Modules

- Syntax in Skript: import <module name>

- Syntax in ghci: :m + <module name>

Beispiel: Data.List

- Module mit allerlei Funktionen für den Umgang mit Listen
- import Data.List
- Kann auch einzelne Funktionen laden

Untitled-1 Haskell

```haskell
--Füge ein Module/Package hinzu
import Data.List
```

Untitled-1 Haskell

```haskell
--nur einzelne Funktionen
import Data.List (sort, concat, find)
sort [4,6,7,8,1]
```

Problem: Name clashes

- Es kann gleichnamige Methoden in verschiedenen Modulen geben
- Haskell weiß bei Aufruf dann nicht, welche genutzt werden sollen
- Beispiel: filter
  - sowohl Data.List als auch Data.Set eine solche Funktion

- Lösung: qualified Import
  - import qualified Data.Set as Set
  - Set.filter

Aufgabe 1 listenverdreh

- Schreibe ein haskell-Funktion welches alle doppelten Elemente aus einer Liste eliminiert, dann sortiert und anschließend rückwärts umstellt

- Nutze das Module Data.List und lade die benötigten Funktionen ein

- Nutze zur Recherche https://downloads.haskell.org/ghc/latest/docs/libraries/

- Datei listenverdrehen.hs

Lösung

Modules

- wichtige Module:
  - Data.List
  - Data.Map
  - Data.Set
  - Data.Char

- grundsätzlich gibt es aber sehr sehr viel Module
- aber eigentlich lebt es davon eigene Module zu schreiben und wiederzuverwenden

Eigenes Modul schreiben

- sehr simpel
- werden in einem separaten File gespeichert
- File muss gleichen Namen wie Module haben
- bei import muss Module im gleichen Verzeichnis wie aufrufendes Programm sein

Eigenes Modul schreiben

```
module Modulename
( Funktionsname 1
, Funktionsname 2
, ...
, Funktionsname n
) where


<Definition Funktion 1>
<Definition Funktion 2>
...
<Definition Funktion n>
```

Module Flächenberechnung

- Wir schreiben ein Modul was den Flächeninhalt von geometrischen Formen berechnen soll
  - Rechteck, Trapez, Kreis, Drache

- Beispiel

Aufgabe 2-ownmodule (10 Minuten)

Schreibe ein eigenes Module, welches bei der Berechnung von Volumen hilft.

Es soll Möglichkeiten geben folgende Körper zu berechnen:

Kugel

Würfel

Pyramide

Zylinder

- implementation in Volumenberechnung.hs
- anschließend modultest.hs ausführen

Submodule

Es wäre möglich auch Submodule zu schreiben

Dafür müssen die jeweiligen Submodule in einem gemeinsam Ordner gespeichert sein. Modulnamen müssen dann Hauptmodul im Namen tragen

Hier: Geometry

-> Geometry.Flaechenberechnung

-> Geometry.Volumenberechnung

Was haben wir bisher gelernt?

- Typischen Grundlagen einer Programmiersprache
  - Datentypen, Funktionen, Syntax usw.
  - jetzt auch Module

- Was haben wir bisher aber noch nicht gemacht?

Was haben wir bisher gelernt?

- Typischen Grundlagen einer Programmiersprache
  - Datentypen, Funktionen, Syntax usw.
  - jetzt auch Module

- Was haben wir bisher aber noch nicht gemacht?
- Simples “Hello world”

“Hello, world!”

1. Skript mit Befehl

   main = putStrLn “hello, world!”

2. Programm kompilieren
   über normale Konsole in das Verzeichnis der Datei gehen und kompilieren
   ghc --make helloworld

3. Programm ausführen
   ./helloworld

- Nutze helloWorld.hs

Warum ist ein “Hello world” hier besonders?

- wir sind in keiner imperativen Programmiersprache
- Haskell ist funktionell

Mantra: Nicht beschreiben was getan werden soll, sondern was etwas ist

- Funktionen “berechnen” etwas. Sie verändern aber nichts.
- Beispiel: Funktion sort Data.List -> Es wird eine komplett neue Liste berechnet

Problem

Wenn eine Funktion nichts verändern kann, wie kann man dann wissen, was sie berechnet hat?

Weil wenn man das Ergebnis darstellen möchte, verändert man ja den Zustand des Ausgabegeräts

Lösung: pure vs. impure

- Trennung zwischen Code ohne Seiteneffekte (pure)…

- und Code mit Seiteneffekten (impure)

I/O - Aktion

Prüfen des Typs von putStrLn

```
ghci> :t putStrLn
```

```
ghci> :t putStrLn “hello, world”
```

I/O - Aktion

Prüfen des Typs von putStrLn

```
ghci> :t putStrLn
```

```
ghci> :t putStrLn “hello, world”
```

Nimmt einen String und gibt eine I/O action zurück

I/O Action

Warum und wann wird das ausgeführt?

> entscheidend ist das Keywort main. Markiert entsprechend so einen impure Block

Natürlich kann man aber auch mit Eingaben umgehen

Eingabe

Untitled-1 Haskell

```haskell
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

Eingabe

Untitled-1 Haskell

```haskell
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

Wichtig ist dabei die zuweisung mit <- . Mit = funktioniert das nicht

I/O-Aktion ist nämlich nichts anderes als eine Box in die etwas hereingelegt wird und dessen Inhalt ausgelesen werden kann. Inhalt kann auch nichts sein

Beispiel

Untitled-1 Haskell

```haskell
--main = do     --impure Code-Abschnitt
    c <- getChar  -- lese ein Char ein
    if c /= ' '  -- gebe ein Char aus sofern etwas da ist
        then do
            putChar c  --
            main
        else return () -- ansonsten springe raus
```

Beispiele für I/O-Funktion

Gibt jede Menge Funktionen:

putStrLn

putStr

getChar

sequence: rs <- sequence [getLine, getLine, getLine]

print

getContent (für Files wichtig)

Aufgabe 4-oche

Schreibe ein Programm welches eine Eingabe in der Konsole einlesen kann und rückwärts wieder ausspuckt. Falls eine leere Eingabe passiert soll, das Programm enden

```
reverseWords :: String -> String

reverseWords = unwords . map reverse . words
```

- siehe rueckwartsschreiber.hs

Aufgabe

Untitled-1 Haskell

```haskell
reverseWords :: String -> String
--reverseWords = unwords . map reverse . words
--obige Zeile ist die kurzschreibweise mit der Punktnotation
reverseWords message = unwords (map reverse (words message))
main = do
  line <- getLine
  if null line
      then return ()
      else do
          putStrLn $ reverseWords line
          main
```

Memoization, Currying, Funktoren, Monoids, Monads

Block 5 - Greg

Funktionen mit mehreren Argumenten

Unter der Haube, gibt es in Haskell nichts als unäre anonyme Funktionen. In Haskell werden anonyme Funktionen mit dem Backslash Operator definiert.

Funktionen - Annonyme Funktionen — Haskell

```haskell
(\x -> x + 1) 4
```

Funktionen mit mehreren Argumenten

Wie also kann eine Funktion mehrere Argumente haben? Haskell nutzt hierfür eine Technik namens **currying**: (dazu aber später mehr)

Funktionen - Annonyme Funktionen — Haskell

```haskell
f :: Int -> Int -> Int -> Int
f x y z = x + y + z

-- intern verarbeitet als
f 3
(f 3) 17
((f 3) 17) 8
```

Funktionskomposition

Es wird in Haskell als guter Stil angesehen Funktionen zusammenzusetzen und entsprechenden Namen zu geben. Diesen Prozess nennt man Funktionskomposition. Haskell bietet hierfür den . Operator. Dieser funktioniert genau wie in der Mathematik:

$$(f \circ g)(x) = f(g(x))$$

Funktionen - Komposition — Haskell

```
(f . g) x  ==  f (g x)
```

Currying und partielle Anwendung

Currying ist eine Technik welche benutzt wird um eine Funktion mit mehreren Argumenten als eine Sequenz von Funktionen zu verarbeiten.

Currying — Haskell

```haskell
add :: Int -> Int -> Int
add x y = x + y

add5 :: Int -> Int
add5 = add 5  -- Partial Application

result = add5 10  -- Ergibt 15
```

Memoization

Memoization ist eine Technik zur Optimierung rekursiver Funktionen, indem bereits berechnete Ergebnisse gespeichert und wiederverwendet werden

Memoization — Haskell

```haskell
import qualified Data.Map as Map

memoFib :: Integer -> Integer
memoFib = (map fib [0..] !!)
  where fib 0 = 0
        fib 1 = 1
        fib n = memoFib (n-1) + memoFib (n-2)
```

Funktoren

Konzept, das es erlaubt, eine Funktion auf Werte innerhalb eines kontextbehafteten Typs (z. B. Listen, Maybe, Either) anzuwenden, ohne den Kontext zu verändern.

- durch Typeclass Functor definiert
- Typ ist Funktor, wenn er die Funktion fmap unterstützt

```
class Functor f where

    fmap :: (a -> b) -> f a -> f b
```

- (a->b) - Funktion, die Wert vom Typ a in b umwandelt
- f a - Wert innerhalb eines kontextbehafteten Typs f
- f b - Ergebnis der Anwendung der Funktion auf enthaltenen Wert

Beispiel Funktor:

1. Listen als Funktor
   a. fmap (*2) [1,2,3] – Ergebnis: [2,4,6]

2. Maybe als Funktor
   a. fmap (+1) (Just 5) – Just 6
   b. fmap (+1) Nothing – Nothing

3. Either als Funktor
   a. fmap (*2) (Right 3) – Ergebnis: Right 6
   b. fmap (*2) (Left “Fehler”) – Left “Fehler”

4. Set?
   a. kein Funktor!
   b. Elemente in Set sind geordnet, Transformation könnte das ändern

Was sind Monoids?

Monoids sind eine algebraische Struktur in der Mathematik und Programmierung

Sie bestehen aus:

- Einer assoziativen binären Operation (<>)
- Einem neutralen Element (identity)

Monoids ermöglichen generische Algorithmen und Abstraktion über wiederholbare Muster

(a <> b) <> c = a <> (b <> c) -- Assoziativität

a <> mempty = mempty <> a = a -- Neutrales Element

Monoids - Beispiel

Beispiel — Haskell

```haskell
-- Addition und 0 als Monoid
instance Monoid Int where
  mempty = 0
  mappend = (+)

-- Listen als Monoid
instance Monoid [a] where
  mempty = []
  mappend = (++)
```

Warum benötigt man Monoids?

Monoids helfen uns als Haskellprogrammierer, Werte strukturiert (regelbasiert) zu kombinieren. Sie ermöglichen also ein flexibles Zusammenführen von Daten und sind essentiell für reduzierende Berechnungen, z.B:

- Summenbildung
- Listenverknüpfung
- parallele Berechnungen

Monoids - Beispiel Summenbildung

Monoids - Aufsummieren Haskell

```haskell
import Data.Monoid

summe :: [Int] -> Int
summe xs = getSum $ foldMap Sum xs

main = print $ summer [1,2,3,4,5] -- Ausgabe: 15
```

Monoids - Beispiel Listenverknüpfung

Monoids - Listen verknüpfen — Haskell

```haskell
log1 = ["System gestartet"]
log2 = ["Fehler: Verbindung verloren"]
log3 = ["Neustart durchgeführt"]

combinedLog = log1 <> log2 <> log3

main = print combinedLog
-- Ausgabe: ["System gestartet","Fehler: Verbindung verloren","Neustart
durchgeführt"]
```

Monoids - Beispiel Parallele Berechnung

Monoids - Parallele Programmierung — Haskell

```haskell
import Data.Monoid

produkt :: [Int] -> Int
produkt xs = getProduct $ foldMap Product xs

main = print $ produkt [2,3,4]
-- Ausgabe: 24 (2 * 3 * 4)
```

Was sind Monaden?

Monads sind ein Designmuster in der funktionalen Programmierung

Sie erweitern den Kontext eines Wertes (z. B. Fehlerbehandlung, IO, Listen)

Monad Gesetze — Plaintext

```
return a >>= f ≡ f a                          -- Linke Identität
m >>= return ≡ m                              -- Rechte Identität
(m >>= f) >>= g ≡ m >>= (\x -> f x >>= g)     -- Assoziativität
```

Monad Gesetze — Haskell

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

Wozu braucht man Monads?

Monads helfen uns gewollte Nebeneffekte auszulösen. Was waren Nebeneffekte?

Monoids - Nebeneffekte — Java

```java
public int add(int a, int b){
  System.out.println(a + " wurde zu " + b + " hinzugefügt");
  return a+b;
}
```

Fertige Monads - Maybe

Das Maybe Monad, hilft uns mit fehlenden Operationen umzugehen.
Es entsteht also ein binärer Zustand. Es ist entweder ein Wert vorhanden oder nicht:

Just x ➜ Ein Wert existiert

Nothing ➜ Es existiert kein Wert

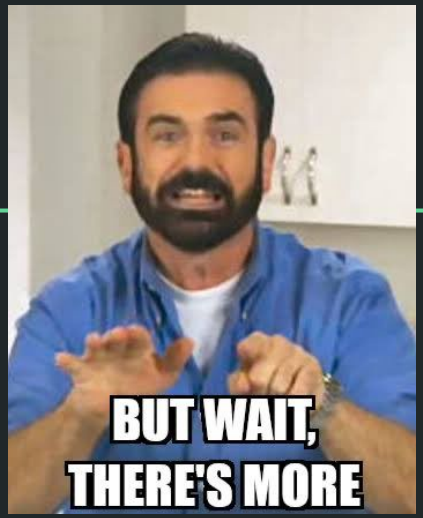
Fertige Mondas - Either

Either erlaubt es uns mit Fehlerhaften zuständen umzugehen:

Left e ➜ Ein Fehler

Right a ➜ Ein Erfolgreicher Wert

Vielen Dank!



Quiz

https://partici.fi/21586784