

# Haskell Done Quick (01)

Lars Pfrenger | 08. November 2024

# 1. Organisatorisches

# Who dis?

@ Lars Pfrenger

✉ lars.pfrenger@uni-ulm.de

🎓 5. Semester Bachelor Software Engineering



# Übung

- 50 % auf 5/7 Blättern zum Bestehen
- Blätter sind eine eigene Prüfungsleistung!

# Tutorium

- Jede Woche in 027/122
- Neues Blatt jede 2. Woche
- Abgabe bitte nicht handschriftlich  
(.pdf, .txt, .hs)
- Nur Code der kompiliert wird bewertet  
→ Nicht funktionierenden Code  
auskommentieren

*-- Einzelner Kommentar*

*{-*

*Kommentar der  
über mehrere  
Zeilen geht :)*

*-}*

# Tutorium

**Woche A:** Lösungen Besprechen + Infos fürs nächste Blatt

**Woche B:** Infos fürs Blatt + Zeit zum Blatt bearbeiten + Fragen stellen

# Tutorium

Stellt Fragen :D

Themenswünsche oder Fragen gerne per Mail an [lars.pfrenger@uni-ulm.de](mailto:lars.pfrenger@uni-ulm.de)

## 2. Haskell



# Deklerativ vs Imperativ

## Deklerativ (z.B. Haskell)

Wir beschreiben, **was** berechnet werden soll.

## Imperativ (z.B. Java)

Wir beschreiben, **wie** etwas berechnet werden soll.

# Deklerativ vs Imperativ

## Deklerativ

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

## Imperativ

```
int firstNumber = 0;
int secondNumber = 1;
int fibonacci = 0;

for (int i = 1; i < N; i++) {
    fibonacci =
        firstNumber + secondNumber;
    firstNumber = secondNumber;
    secondNumber = fibonacci;
}
```

# Laziness

- Haskell ist Lazy
- Umgang mit Unendlichen Listen

```
take 5 [1..]
```

# 3. Blatt 1

# Aufgabe 1a

Ist der Algorithmus deterministisch?

Nein, der Algorithmus ist *nicht deterministisch*, da in Schritt 1 eine zufällige initiale Gästeliste gezogen wird. Außerdem geben die Schritte 2 und 3 nicht an, für welche Gäste die Regeln zuerst angewandt werden sollen.

- "**Schritt 1** Zieh eine zufällige Liste von möglichen Gästen"
- Ein Algorithmus heißt **deterministisch**, wenn die Wirkung und die Reihenfolge der Einzelschritte eindeutig festgelegt ist. <sup>1</sup>

---

<sup>1</sup><https://www.dbs.ifi.lmu.de/Lehre/EIP/WS1415/skript/EiP-04-DatenAlgorithmen-Teil2.pdf>

# Aufgabe 1b

Terminiert der Algorithmus immer?

Nein, es ist möglich, dass der Algorithmus nicht terminiert, wenn die gezogenen initialen Listen immer zu einer ungenügend großen Gästeliste führen.

→ **Terminierend:** Der Algorithmus endet für alle validen Schrittfolgen nach endlich vielen Schritten

# Aufgabe 1c

Ist der Algorithmus partiell korrekt?

Ja, denn Schritt 2 stellt sicher, dass alle Gäste hinzugefügt werden, sodass die Aussagen von Typ 1 erfüllt sind, und Schritt 3 entfernt alle Gäste, sodass Aussagen beider Typen erfüllt bleiben.

→ Ein Algorithmus heißt **partiell korrekt**, wenn für alle gültigen Eingaben (Vorbedingung) das Resultat der Spezifikation (Nachbedingung) des Algorithmus entspricht.<sup>1</sup>

---

<sup>1</sup><https://www.dbs.ifi.lmu.de/Lehre/EIP/WS1415/skript/EiP-04-DatenAlgorithmen-Teil2.pdf>

# Aufgabe 2a

Ist der Algorithmus iterativ oder rekursiv beschrieben?

Rekursiv

$$foo(a, b) = \begin{cases} a & \text{wenn } a = b \\ foo(a, b - a) & \text{wenn } a < b \\ foo(b, a - b) & \text{wenn } a > b \end{cases}$$



## Aufgabe 2b

Berechnen Sie das Ergebnis für die Eingabe...

$$\begin{aligned} \text{foo}(12, 9) &=_{12 > 9} \text{foo}(9, 12 - 9) = \text{foo}(9, 3) \\ &=_{9 > 3} \text{foo}(3, 9 - 3) = \text{foo}(3, 6) \\ &=_{3 < 6} \text{foo}(3, 6 - 3) = \text{foo}(3, 3) \\ &=_{3 = 3} 3 \end{aligned}$$

# Aufgabe 2c

Beschreiben Sie in Ihren Worten, was der Algorithmus berechnet.

Der Algorithmus berechnet den größten gemeinsamen Teiler zweier Zahlen.

- Euklidischer Algorithmus
- Der Algorithmus nutzt aus, dass sich der größte gemeinsame Teiler zweier Zahlen nicht ändert, wenn man die kleinere von der größeren abzieht.<sup>1</sup>

---

<sup>1</sup>[https://de.wikipedia.org/wiki/Euklidischer\\_Algorithmus](https://de.wikipedia.org/wiki/Euklidischer_Algorithmus)

# Aufgabe 3a

Ist der Algorithmus iterativ oder rekursiv beschrieben?

Iterativ

- Schleife
- Keine Funktionen, die sich selbst aufrufen

## Aufgabe 3b

Gegeben die Eingabe  $n = 3$ . Führen Sie den Algorithmus zeilenweise aus und geben Sie alle Zwischenzustände an. Geben Sie auch den Rückgabewert an.

$$\begin{aligned} \langle 1 \mid \emptyset \rangle &\mapsto \langle 2 \mid n = 3 \rangle \mapsto \langle 3 \mid n = 3, a = 1 \rangle \\ &\mapsto \langle 4 \mid n = 3, a = 1, b = 1 \rangle \mapsto \langle 5 \mid n = 3, a = 1, b = 1 \rangle \\ &\mapsto \langle 6 \mid n = 3, a = 1, b = 1, a_{alt} = 1 \rangle \mapsto \langle 7 \mid n = 3, a = 1, b = 1, a_{alt} = 1 \rangle \\ &\mapsto \langle 8 \mid n = 3, a = 1, b = 2, a_{alt} = 1 \rangle \mapsto \langle 4 \mid n = 2, a = 1, b = 2, a_{alt} = 1 \rangle \\ &\dots \\ &\mapsto \langle 9 \mid n = 0, a = 3, b = 5, a_{alt} = 2 \rangle \mapsto 3 \end{aligned}$$

→ Aufgabe komplett Lesen: Geben Sie auch den Rückgabewert an.

# Aufgabe 3c

Beschreiben Sie in Ihren Worten, was der Algorithmus berechnet.

Der Algorithmus berechnet eine Folge von  $n$  Zahlen, beginnend mit 1, 1, bei der ein Element jeweils die Summe der beiden Element davor ist. Diese Folge wird auch die Fibonacci-Folge genannt.

# 4. Blatt 2

# Funktionsdekleration

```
foo :: Int -> Double -> Double
```

# Funktionsdekleration

```
foo :: Int -> Double -> Double
```




Parameter



# Funktionsdekleration


`foo :: Int -> Double -> Double`



Parameter      Return Type

# Funktionsdekleration

`foo :: Int -> Double -> Double`



`foo x y = fromIntegral x * y`

# Funktionsdekleration

Beispiel: Wir wollen nur Tupel vom Typ `(Int, Int)` erlauben

```
fst :: (Int, Int) -> Int
```

```
fst (a, b) = a
```

# Tupel

(1, 5, "Hallo")

- Fixe Anzahl an Elementen
- Verschiedene Typen erlaubt

# Tupel

```
(1, 5, "Hallo") :: (Num, Num, String)
```

- Fixe Anzahl an Elementen
- Verschiedene Typen erlaubt

# Tupel

```
fst (x, y) = x -- Erstes Element  
snd (x, y) = y -- Zweites Element
```

# Listen

```
1 : [2,3]           = [1,2,3]           -- Element einfügen
[1,2,3] ++ [4,5,6] = [1,2,3,4,5,6]       -- Listen konkatenieren
head [1,2,3]        = 1                 -- 1. Element
last [1,2,3]        = 3                 -- Letzte Element
tail [1,2,3]        = [2,3]            -- Alle außer das 1.
init [1,2,3]        = [1,2]            -- Alle außer das letzte
```

# Typevariablen

```
fst :: (a, b) -> a
```



# Klassen Constraints

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

# ghci

```
:t <Identifier>  -- Typ Signatur ausgeben lassen  
:l <filename.hs> -- Datei laden  
:r              -- Datei neu laden
```

# 5. Ende

# Ende

Danke für Eure Aufmerksamkeit