

PROGRAMMIERUNG

ÜBUNG 1: EINLEITUNG

Eric Kunze

`eric.kunze@mailbox.tu-dresden.de`

TU Dresden, 4. April 2020

Wer bin ich?

- ▶ Eric [Kunze]
- ▶ `eric.kunze@mailbox.tu-dresden.de`
- ▶ Fragen, Wünsche, Vorschläge, ...
gern jederzeit wie auch immer



offizielles Angebot

- ▶ Vorlesungen
 - ▷ Lehrveranstaltungswebsite:
`www.orchid.inf.tu-dresden.de/teaching/2020ss/prog/`
 - ▷ OPAL-Kurs: Link siehe LV-Website
- ▶ Übungen
 - ▷ selbstständige Bearbeitung der Übungen
 - ▷ Abgabe via OPAL an (zufällig) zugewiesenen Tutor
 - ▷ Rückgabe der korrigierten Arbeiten via OPAL

mein (privates) Angebot

- ▶ Übernahme der Korrekturen
- ▶ Vidcasting bzw. Live-Übungen

Slides werden mit Sourcecode auf Github zur Verfügung stehen.

- ▶ `https://github.com/oakonerich/programmierung-ss20`
- ▶ `github.com` → `oakonerich` → `programmierung-ss20`
- ▶ evtl. zusätzliche Materialien (nach Bedarf)
- ▶ **kein Anspruch auf Vollständigkeit & Korrektheit**
- ▶ gefundene Fehler melden - am besten per Request

Einführung in Haskell

Haskell = funktionale Programmiersprache

Wir programmieren nicht *wie* berechnet wird, sondern *was* berechnet wird.

Haskell = funktionale Programmiersprache

Wir programmieren nicht *wie* berechnet wird, sondern *was* berechnet wird.

Mathe: Die Addition $n + m$ können wir auch als Funktion

$$+: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \quad n + m = \begin{cases} n & m = 0 \\ 1 + (n + (m - 1)) & \text{sonst} \end{cases}$$

definieren.

Haskell = funktionale Programmiersprache

Wir programmieren nicht *wie* berechnet wird, sondern *was* berechnet wird.

Mathe: Die Addition $n + m$ können wir auch als Funktion

$$+ : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \quad n + m = \begin{cases} n & m = 0 \\ 1 + (n + (m - 1)) & \text{sonst} \end{cases}$$

definieren.

Informatik: Wir können eine Funktion `add :: Int -> Int -> Int` spezifizieren, die genau die Addition durchführt:

```
1 add :: Int -> Int -> Int
2 add n 0 = n
3 add n m = 1 + add n (m-1)
```


Theorem

Sei B eine Menge, $b \in B$ und $F: B \times \mathbb{N} \rightarrow B$ eine Abbildung. Dann liefert die Vorschrift

$$f(0) := b \tag{1a}$$

$$f(n+1) := F(f(n), n) \quad \forall n \in \mathbb{N} \tag{1b}$$

genau eine Abbildung $f: \mathbb{N} \rightarrow B$.

Theorem

Sei B eine Menge, $b \in B$ und $F: B \times \mathbb{N} \rightarrow B$ eine Abbildung. Dann liefert die Vorschrift

$$f(0) := b \quad (1a)$$

$$f(n+1) := F(f(n), n) \quad \forall n \in \mathbb{N} \quad (1b)$$

genau eine Abbildung $f: \mathbb{N} \rightarrow B$.

Beweis. vollständige Induktion:

(IA) Für $n = 0$ ist $f(0) = b$ eindeutig definiert.

(IS) Angenommen $f(n)$ sei eindeutig definiert. Wegen **(?)** ist dann auch $f(n+1)$ eindeutig definiert.

Man kann dieses Prinzip der Rekursion auf weitere Mengen (unabhängig von den natürlichen Zahlen) erweitern (↗ Formale Systeme).

Glasgow Haskell Compiler (ghc(i)) :

<https://www.haskell.org/ghc/>

- ▶ **Terminal:** `ghci <modulname>`
- ▶ **Module laden:** `:load <modulname>` oder `:l`
- ▶ **Module neu laden:** `:reload` oder `:r`
- ▶ **Hilfe:** `:?` oder `:help`
- ▶ **Interpreter verlassen:** `:quit` oder `:q`

- ▶ `:type <exp>` — Typ des Ausdrucks `<exp>` bestimmen
- ▶ `:info <fkt>` — kurze Dokumentation für `<fkt>`
- ▶ `:browse` — alle geladenen Funktionen anzeigen

Listen Wenn a ein Typ ist, dann bezeichnet $[a]$ den Typ
"Liste mit Elementen vom Typ a ", insbesondere haben
alle Elemente einer Liste den gleichen Typ

Listen Wenn a ein Typ ist, dann bezeichnet $[a]$ den Typ
"Liste mit Elementen vom Typ a ", insbesondere haben
alle Elemente einer Liste den gleichen Typ

cons-Operator " : " Trennung von head und tail einer Liste

$[x_1, x_2, x_3, x_4, x_5] = x_1 : [x_2, x_3, x_4, x_5]$

Listen Wenn a ein Typ ist, dann bezeichnet $[a]$ den Typ
"Liste mit Elementen vom Typ a ", insbesondere haben
alle Elemente einer Liste den gleichen Typ

cons-Operator " : " Trennung von head und tail einer Liste

$[x_1, x_2, x_3, x_4, x_5] = x_1 : [x_2, x_3, x_4, x_5]$

Verkettungsoperator " ++ " Verkettung zweier Listen
gleichen Typs

$[x_1, x_2] ++ [x_3, x_4, x_5] = [x_1, x_2, x_3, x_4, x_5]$

Übungsblatt 1

AUFGABE 2 – FAKULTÄTSFUNKTION

Fakultät

$$n! = \prod_{i=1}^n i$$

AUFGABE 2 – FAKULTÄTSFUNKTION

Fakultät

$$n! = \prod_{i=1}^n i$$

Daraus bilden wir nun eine Rekursionsvorschrift:

$$n! = \mathbf{n} \cdot \prod_{i=1}^{n-1} i = n \cdot (n-1)!$$

AUFGABE 2 – FAKULTÄTSFUNKTION

Fakultät

$$n! = \prod_{i=1}^n i$$

Daraus bilden wir nun eine Rekursionsvorschrift:

$$n! = \mathbf{n} \cdot \prod_{i=1}^{n-1} i = n \cdot (n-1)!$$

Um die Rekursion vollständig zu definieren, benötigen wir einen (oder i.a. mehrere) *Basisfall*. Wann können wir also die Rekursion der Fakultät abbrechen?

$$0! = 1 \quad 1! = 1 \quad 2! = 2 \quad \dots \quad (2)$$

⇒ Welcher Basisfall ist sinnvoll? $0! = 1$

AUFGABE 3 – FIBONACCI-ZAHLEN

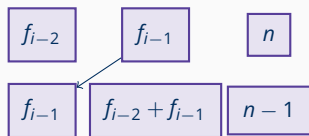
$$f_n := \begin{cases} 1 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ f_{n-1} + f_{n-2} & \text{sonst} \end{cases}$$

AUFGABE 3 – FIBONACCI-ZAHLEN

$$f_n := \begin{cases} 1 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ f_{n-1} + f_{n-2} & \text{sonst} \end{cases}$$

⇒ Rekursionsvorschrift schon gegeben.

Verfahren ohne Rekursion.

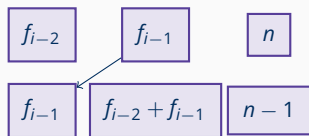


AUFGABE 3 – FIBONACCI-ZAHLEN

$$f_n := \begin{cases} 1 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ f_{n-1} + f_{n-2} & \text{sonst} \end{cases}$$

⇒ Rekursionsvorschrift schon gegeben.

Verfahren ohne Rekursion.



Explizite Formel.

$$f_n = \frac{\phi^n - \left(-\frac{1}{\phi}\right)^n}{\sqrt{5}}$$

mit $\phi = \frac{1+\sqrt{5}}{2}$