TECHNISCHE UNIVERSITÄT DRESDEN

PROGRAMMIERUNG

ÜBUNG 1: EINLEITUNG

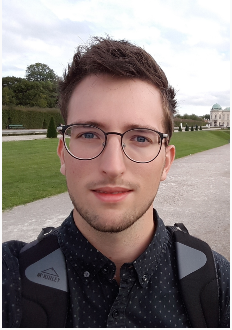
Eric Kunze

eric.kunze@mailbox.tu-dresden.de

TU Dresden, 6. April 2020

Wer bin ich?

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



DAS ETWAS ANDERE SEMESTER

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

LÖSUNGEN

Slides werden mit Sourcecode auf Github zur Verfügung stehen.

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

Einführung in Haskell

$\boxed{\text{Haskell} = \text{funktionale Programmiersprache}}$

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

$\boxed{\text{Haskell} = \text{funktionale Programmiersprache}}$

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

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

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

definieren.

$\boxed{\text{Haskell} = \text{funktionale Programmiersprache}}$

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

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

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

definieren.

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

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

EIN WEITERES BEISPIEL

Um die Analogie zwischen Funktionen in Mathe und Funktionen in Haskell noch einmal zu verdeutlichen, betrachten wir die Funktion

$$f : \mathbb{N} \to \mathbb{N}$$
$$f(x) = x + 3$$

Diese würde in Haskell wie folgt aussehen:

```haskell
f :: Int -> Int
f x = x + 3
```

EIN WEITERES BEISPIEL

Um die Analogie zwischen Funktionen in Mathe und Funktionen in Haskell noch einmal zu verdeutlichen, betrachten wir die Funktion

$$f : \mathbb{N} \to \mathbb{N}$$
$$f(x) = x + 3$$

Diese würde in Haskell wie folgt aussehen:

```
f :: Int -> Int
f x = x + 3
```

Außerdem kennen wir bereits die Variante Funktionen auf ihren Argumenten zu definieren, z.B.

$$g : \mathbb{N} \to \mathbb{N}$$
$$g(0) = 1$$
$$g(x) = x^2$$

bzw. in Haskell

```
g :: Int -> Int
g 0 = 1
g x = x^2
```

Theorem

Sei B eine Menge, $b \in B$ und $F\colon B \times \mathbb{N} \to B$ 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 $f : \mathbb{N} \to B$.

DAS PRINZIP DER REKURSION

Theorem

Sei B eine Menge, $b \in B$ und $F\colon B \times \mathbb{N} \to B$ 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 $f : \mathbb{N} \to B$.

Beweis. vollständige Induktion:

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

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

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

HASKELL INSTALLIEREN UND COMPILIEREN

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

- einzeilige Kommentare mit `--`
- mehrzeilige Kommentare mit `{- ... -}`

GRUNDLEGENDE STRUKTUREN – PATTERN MATCHING

Mit Pattern Matching kann man prüfen, ob Funktionsargumente eine bestimmte Form aufweisen. Zum Beispiel:

- Der Aufruf `func 3 0` passt auf das Pattern `func x 0`, aber nicht auf das Pattern `func 0 x`.

Damit kann man verschiedene Fälle in einfacher Form nacheinander abgreifen, z.B. Basis- und Rekursionsfall. Vergleiche dazu auch das Beispiel mit der `add`-Funktion:

- Der Aufruf `add 5 0` matched mit Zeile 2, also berechnen wir `add 5 0 = 5`.
- Der Aufruf `add 5 1` matched nicht auf Zeile 2, also probieren wir Zeile 3. Das matched mit `n = 5` und `m = 1` und wir berechnen

```
add 5 1 = 1 + add 5 0
        = 1 + 5
        = 6
```

Beachte, dass dabei von oben nach unten getestet wird!

GRUNDLEGENDE STRUKTUREN – CONDITIONALS

Um Bedingungen zu testen, gibt es die Möglichkeit auf `if-then-else` zu verzichten und sogenannte **guards** mit **pipes** zu verwenden. Das sieht dann wieder so aus, wie eine geschweifte Klammer in mathematischen Fallunterscheidungen.

$$h(x) = \begin{cases} x^2 & \text{für } x < 0 \\ 0.5 \cdot x & \text{für } x \geq 0 \end{cases}$$

```
h :: Int -> Int
h x
  | x < 0  = x^2
  | x <= 0 = 0.5 * x
```

GRUNDLEGENDE STRUKTUREN – CONDITIONALS

Um Bedingungen zu testen, gibt es die Möglichkeit auf `if-then-else` zu verzichten und sogenannte **guards** mit **pipes** zu verwenden. Das sieht dann wieder so aus, wie eine geschweifte Klammer in mathematischen Fallunterscheidungen.

$$h(x) = \begin{cases} x^2 & \text{für } x < 0 \\ 0.5 \cdot x & \text{für } \cancel{x \geq 0} \; \textbf{sonst} \end{cases}$$

```haskell
h :: Int -> Int
h x
  | x < 0     = x^2
  | otherwise = 0.5 * x
```

Wie auch in Mathe sollte man bei gegensätzlichen Bedingungen ein „sonst“ bzw. `otherwise` verwenden.

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

HASKELL & LISTEN

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

```
[ x1 , x2 , x3 , x4 , x5] = x1 : [x2 , x3 , x4 , x5]
```

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

```
[ x1 , x2 , x3 , x4 , x5] = x1 : [x2 , x3 , x4 , x5]
```

Verkettungsoperator " `++` " Verkettung zweier Listen gleichen Typs

```
[x1 , x2 ] ++ [x3 , x4 , x5] = [x1 , x2 , x3 , x4 , x5]
```

ZEICHEN & ZEICHENKETTEN

Zeichen

- Datentyp `Char`
- Eingabe in einfachen Anführungszeichen
- z.B. `'a'`, `'e'`, `'3'`

Zeichenketten

- Datentyp `String = [Char]`
- Eingabe in doppelten Anführungszeichen
- z.B. `"hallo"`, `"welt"`
- Konkatenation von Zeichenketten:

```
"hallo " ++ "welt" = "hallo welt"
```

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:

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

AUFGABE 2 – FAKULTÄTSFUNKTION

Fakultät

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

Daraus bilden wir nun eine Rekursionsvorschrift:

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

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

$$0! = 1 \qquad 1! = 1 \qquad 2! = 2 \qquad \cdots$$

$\Rightarrow$ Welcher Basisfall ist sinnvoll? $\quad 0! = 1$

AUFGABE 2 – LÖSUNG

```haskell
-- Aufgabe 2(a)
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)

-- Aufgabe 2(b)
sumFacs :: Int -> Int -> Int
sumFacs n m
    | n > m = 0
    | otherwise = fac n + sumFacs (n+1) m
```

Hinweis: In der Musterlösung werden noch `undefined`-Fälle angegeben. Das ist für uns erst einmal optional, aber natürlich schöner.

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

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

$\Rightarrow$ Rekursionsvorschrift schon gegeben.

Verfahren ohne Rekursion.

$$\boxed{f_{i-2}} \quad \boxed{f_{i-1}} \quad \boxed{n}$$

$$\boxed{f_{i-1}} \quad \boxed{f_{i-2} + f_{i-1}} \quad \boxed{n-1}$$

AUFGABE 3 – FIBONACCI-ZAHLEN

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

$\Rightarrow$ Rekursionsvorschrift schon gegeben.

Verfahren ohne Rekursion.

$$\boxed{f_{i-2}} \quad \boxed{f_{i-1}} \quad \boxed{n}$$

$$\boxed{f_{i-1}} \quad \boxed{f_{i-2} + f_{i-1}} \quad \boxed{n-1}$$

Explizite Formel.

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

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

AUFGABE 3 – LÖSUNG

```
-- Aufgabe 3
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

fib' :: Int -> Int
fib' n = fib_help 1 1 n

fib_help :: Int -> Int -> Int -> Int
fib_help x _ 0 = x
fib_help x y n = fib_help y (x+y) (n-1)
```