

PROGRAMMIERUNG

ÜBUNG 1: EINLEITUNG

Eric Kunze

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

TU Dresden, 13. April 2021

WER BIN ICH?

- ▶ Eric Kunze
- ▶ E-Mail:
`eric.kunze@mailbox.tu-dresden.de`
- ▶ Telegram: @oakoneric
- ▶ Fragen, Wünsche, Vorschläge, ...
gern jederzeit über alle möglichen Wege



EIN WEITERES ONLINE-SEMESTER

- ▶ Vorlesungsvideo und Übungsblatt jeden Freitag im **OPAL-Kurs**
- ▶ Lehrveranstaltungswebsite:
www.orchid.inf.tu-dresden.de/teaching/2021ss/prog/

- ▶ Vorlesungsvideo und Übungsblatt jeden Freitag im **OPAL-Kurs**
- ▶ Lehrveranstaltungswebsite:
www.orchid.inf.tu-dresden.de/teaching/2021ss/prog/
- ▶ **Übungen**
 - ▷ Einschreibung im OPAL
 - ▷ Zugangslink im OPAL-Kurs und meiner Website

- ▶ Vorlesungsvideo und Übungsblatt jeden Freitag im **OPAL-Kurs**
- ▶ Lehrveranstaltungswebsite:
`www.orchid.inf.tu-dresden.de/teaching/2021ss/prog/`

▶ **Übungen**

- ▷ Einschreibung im OPAL
- ▷ Zugangslink im OPAL-Kurs und meiner Website

▶ **meine Website:**

`https://oakonerich.github.io`

- ▷ Vorlesungsmaterialien verlinkt
- ▷ Slides, Handout, Mitschrift/Code
- ▷ Übung als Videoaufzeichnung (passwortgeschützt)
- ▷ Github: `https://github.com/oakonerich/programmierung-ss21`
- ▷ **kein Anspruch auf Vollständigkeit & Korrektheit**

Literatur

- ▶ *Learn You a Haskell For Great Good!*
 - ▷ sehr gut geschrieben, ausführlich und kurzweilig
- ▶ *Real World Haskell*
 - ▷ sehr gut, wesentlich mehr Inhalte als benötigt

beide Bücher als Online-Versionen verfügbar (siehe Website)

Literatur

- ▶ *Learn You a Haskell For Great Good!*
 - ▷ sehr gut geschrieben, ausführlich und kurzweilig
- ▶ *Real World Haskell*
 - ▷ sehr gut, wesentlich mehr Inhalte als benötigt

beide Bücher als Online-Versionen verfügbar (siehe Website)

Altklausuren

- ▶ FTP-Server des iFSR:
`https://ftp.ifsr.de/klausuren/Grundstudium/Programmierung/`
- ▶ VPN-Verbindung notwendig

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: Wir kennen Funktionen bereits aus dem Mathe-Unterricht und den Mathe-Vorlesungen. Zum Beispiel ist

$$f: \mathbb{N} \rightarrow \mathbb{N}$$

$$f(x) = x + 3$$

eine Funktion, die natürliche Zahlen auf natürliche Zahlen abbildet.

Haskell = funktionale Programmiersprache

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

Mathe: Wir kennen Funktionen bereits aus dem Mathe-Unterricht und den Mathe-Vorlesungen. Zum Beispiel ist

$$f: \mathbb{N} \rightarrow \mathbb{N}$$

$$f(x) = x + 3$$

eine Funktion, die natürliche Zahlen auf natürliche Zahlen abbildet.

Haskell: Diese würde in Haskell wie folgt aussehen:

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

EIN WEITERES BEISPIEL

Um zu verdeutlichen, wie ähnlich sich mathematische Funktionen und Haskell-Funktionen sind, betrachten wir folgendes Beispiel. Wir können Funktionen auf ihren Argumenten definieren, d.h.

$$g: \mathbb{N} \rightarrow \mathbb{N}$$

$$g(0) = 1$$

$$g(x) = x^2$$

bzw. in Haskell

```
g :: Int -> Int
g 0 = 1
g x = x * x
```

EIN WEITERES BEISPIEL

Wir können auch deutlich wichtigere und kompliziertere Funktionen programmieren. Zum Beispiel lässt sich die Addition $n + m$ auch als Funktion schreiben.

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

definieren.

EIN WEITERES BEISPIEL

Wir können auch deutlich wichtigere und kompliziertere Funktionen programmieren. Zum Beispiel lässt sich die Addition $n + m$ auch als Funktion schreiben.

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

definieren.

Als Haskell-Funktion sieht das dann so aus:

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

Aufgabe 1

Haskell installieren und compilieren

AUFGABE 1

Gegeben sei eine Haskell-Funktion

```
sum3 :: Int -> Int -> Int -> Int
sum3 x y z = x + y + z
```


AUFGABE 1

Gegeben sei eine Haskell-Funktion

```
sum3 :: Int -> Int -> Int -> Int
sum3 x y z = x + y + z
```

Die entspricht der mathematische Funktion

$$\text{sum3}: \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \quad \text{mit} \quad \text{sum3}(x, y, z) = x + y + z$$

Glasgow Haskell Compiler (ghc(i)) :

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

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`

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

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 `{- ... -}`

Aufgabe 2

Rekursion, Pattern Matching & Conditionals

DAS PRINZIP DER REKURSION

Ein wichtiges Prinzip in der funktionalen Programmierung ist das Prinzip der Rekursion.

- ▶ **Rekursionsfall**

- ▶ **Basisfall**

Ein wichtiges Prinzip in der funktionalen Programmierung ist das Prinzip der Rekursion.

► Rekursionsfall

- ▷ Reduktion eines großen Problems auf ein kleineres Problem
- ▷ `Int`-Funktionen: Reduktion von n auf $n - 1$
- ▷ Liste: Reduktion durch Abspaltung eines Listenelements

► Basisfall

DAS PRINZIP DER REKURSION

Ein wichtiges Prinzip in der funktionalen Programmierung ist das Prinzip der Rekursion.

► Rekursionsfall

- ▷ Reduktion eines großen Problems auf ein kleineres Problem
- ▷ `Int`-Funktionen: Reduktion von n auf $n - 1$
- ▷ Liste: Reduktion durch Abspaltung eines Listenelements

► Basisfall

- ▷ kleinste Probleme - einfach zu lösen
- ▷ rechte Seite deterministisch
- ▷ `Int`-Funktionen: rechte Seite hängt nicht von n ab
- ▷ Liste: leere Liste

PATTERN MATCHING

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

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

$$\begin{aligned}\text{add } 5 \ 1 &= 1 + \text{add } 5 \ 0 \\ &= 1 + 5 \\ &= 6\end{aligned}$$

Beachte, dass dabei von oben nach unten getestet wird!

AUFGABE 2A – FAKULTÄTSFUNKTION

Fakultät

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

AUFGABE 2A – FAKULTÄTSFUNKTION

Fakultät

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

Rekursionsvorschrift: $n \rightsquigarrow n - 1$

AUFGABE 2A – FAKULTÄTSFUNKTION

Fakultät

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

Rekursionsvorschrift: $n \rightsquigarrow n - 1$

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

- ▶ links: $n!$ hängt von n ab
- ▶ rechts: $(n - 1)!$ hängt nur von $n - 1$ ab

AUFGABE 2A – FAKULTÄTSFUNKTION

Fakultät

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

Rekursionsvorschrift: $n \rightsquigarrow n - 1$

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

- ▶ links: $n!$ hängt von n ab
- ▶ rechts: $(n - 1)!$ hängt nur von $n - 1$ ab

Um die Rekursion vollständig zu definieren, benötigen wir einen *Basisfall*.
Wann können wir also die Rekursion der Fakultät abbrechen?

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

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

AUFGABE 2A – LÖSUNG

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

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

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.

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

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


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.

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

```
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.

AUFGABE 2B — SUMMIERTE FAKULTÄTEN (1)

$$f(n, m) = \sum_{i=n}^m i!$$

AUFGABE 2B — SUMMIERTE FAKULTÄTEN (1)

$$f(n, m) = \sum_{i=n}^m i!$$

- ▶ Rekursionsfall:

$$f(n, m) = \sum_{i=n}^m i! = \mathbf{m!} + \sum_{i=n}^{\mathbf{m-1}} i! = m! + f(n, m - 1)$$

- ▶ Basisfall: $f(n, m) = 0$ für $n > m$

AUFGABE 2B — SUMMIERTE FAKULTÄTEN (1)

$$f(n, m) = \sum_{i=n}^m i!$$

- ▶ Rekursionsfall:

$$f(n, m) = \sum_{i=n}^m i! = m! + \sum_{i=n}^{m-1} i! = m! + f(n, m-1)$$

- ▶ Basisfall: $f(n, m) = 0$ für $n > m$

Lösung:

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

AUFGABE 2B — SUMMIERTE FAKULTÄTEN (2)

$$f(n, m) = \sum_{i=n}^m i!$$

AUFGABE 2B — SUMMIERTE FAKULTÄTEN (2)

$$f(n, m) = \sum_{i=n}^m i!$$

- ▶ Rekursionsfall:

$$f(n, m) = \sum_{i=n}^m i! = \mathbf{n!} + \sum_{i=n+1}^m i! = n! + f(n+1, m)$$

- ▶ Basisfall: $f(n, m) = 0$ für $n > m$

AUFGABE 2B — SUMMIERTE FAKULTÄTEN (2)

$$f(n, m) = \sum_{i=n}^m i!$$

- ▶ Rekursionsfall:

$$f(n, m) = \sum_{i=n}^m i! = n! + \sum_{i=n+1}^m i! = n! + f(n+1, m)$$

- ▶ Basisfall: $f(n, m) = 0$ für $n > m$

Lösung:

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

Aufgabe 3

AUFGABE 3 – FIBONACCI-ZAHLEN

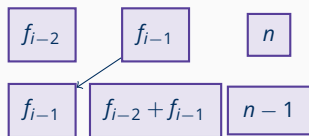
$$f_n := \begin{cases} 1 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ f_{n-1} + f_{n-2} & \text{sonst} \end{cases}$$

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.

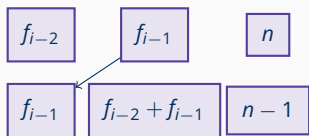


AUFGABE 3 – FIBONACCI-ZAHLEN

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

⇒ Rekursionsvorschrift schon gegeben.

Verfahren ohne Rekursion.



Explizite Formel.

$$f_n = \frac{\phi^n - \left(-\frac{1}{\phi}\right)^n}{\sqrt{5}} \text{ mit } \phi = \frac{1 + \sqrt{5}}{2}$$

AUFGABE 3 – LÖSUNG

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