TECHNISCHE UNIVERSITÄT DRESDEN

PROGRAMMIERUNG

ÜBUNG 1: EINLEITUNG

Eric Kunze

eric.kunze@tu-dresden.de

TU Dresden, 13. April 2022

letzte Änderung:
19.04.2022, 18:00

WER BIN ICH?

- **Eric** Kunze
- `eric.kunze@tu-dresden.de`
- Fragen, Wünsche, Vorschläge, . . .
- **Telegram:**
  `@oakoneric` bzw. `t.me/oakoneric`

Meine Materialien sind auf meiner Website `https://oakoneric.github.io/prog22` zu finden.

- Slides (mit Lösungen); evtl. zusätzliche Materialien (nach Bedarf)
- **kein Anspruch auf Vollständigkeit & Korrektheit**

Source Code auf Github `https://github.com/oakoneric-tutorials/programmierung-ss22`

INFOS & CO

OPAL-Kurs: `https://tud.link/y471`

- alle Informationen zur Lehrveranstaltung
- Link zur Vorlesung: Freitag, 2. DS via Zoom
- Übungsblätter
- Forum

Materialien:

- Slides der Vorlesung (ersetzen ehemaliges Skript)
- Aufgabensammlung

Verlegung am Ostermontag:

Ersatztermin am *Mittwoch, 20.04.2022 um 13 Uhr*

- Information per Mail zu Raum/online

HINWEISE

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)

HINWEISE

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 PROGRAMMIERUNG

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

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

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

eine Funktion, die natürliche Zahlen auf natürliche Zahlen abbildet.

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

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

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

eine Funktion, die natürliche Zahlen auf natürliche Zahlen abbildet.

Haskell: Diese würde in Haskell wie folgt aussehen:

```haskell
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.

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

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.

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

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.

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

definieren.

Als Haskell-Funktion sieht das dann so aus:

```haskell
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

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

Die entspricht der mathematische Funktion

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

HASKELL INSTALLIEREN UND COMPILIEREN

Glasgow Haskell Compiler (ghc(i)) :

`https://www.haskell.org/ghc/`

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`

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

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**

DAS PRINZIP DER REKURSION

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

- **Rekursionsfall**
  - Reduktion eines großen Prolems 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 Prolems 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

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

Beachte, dass dabei von oben nach unten getestet wird!

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.

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

```
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 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: $n \leadsto n - 1$

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

- 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: $n \rightsquigarrow n-1$

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

- 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?

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

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

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.

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:

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

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

AUFGABE 2B — SUMMIERTE FAKULTÄTEN (1)

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

- Rekursionsfall:

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

- 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:

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

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

AUFGABE 2B — SUMMIERTE FAKULTÄTEN (2)

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

- Rekursionsfall:

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

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

Lösung:

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

Aufgabe 3

AUFGABE 3 – FIBONACCI-ZAHLEN

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

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

$\Rightarrow$ Rekursionsvorschrift schon gegeben.

Verfahren ohne Rekursion.

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

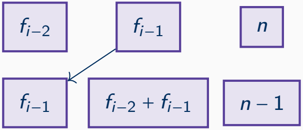
$\boxed{f_{i-1}}$ $\boxed{f_{i-2} + f_{i-1}}$ $\boxed{n - 1}$

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.



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

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

```haskell
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)
```

Zusatzaufgabe 1

ZUSATZAUFGABE 1

Ziel: Anzahl der vollständigen Binärbäume mit n Knoten

Idee: Wie erhalten wir volle Binärbäume? — Ein voller Binärbaum ist

- entweder ein Blatt
- oder er besteht aus einer Wurzel und *zwei* Kindern

Umsetzung:

- Rekursionsfall: $n \geq 3$ Knoten
  - ein Wurzelknoten
  - $n - 1$ Knoten für linken und rechten Teilbaum (systematisch alle Möglichkeiten durchlaufen)
- Basisfall:
  - $n = 0$: es gibt keinen Baum mit keinen Knoten
  - $n = 1$: Baum mit einem Knoten = Blatt (davon gibt es genau einen)

ZUSATZAUFGABE 1

```haskell
countBinTrees :: Int -> Int
countBinTrees 0 = 0
countBinTrees 1 = 1
countBinTrees n = go (n-1)
  where
    go 0 = 0
    go m = go (m-1) + countBinTrees (n - 1 - m) *
                      countBinTrees m
```

Hinweis: go durchläuft alle Möglichkeiten $n - 1$ Knoten so auf zwei (Kind-)Bäume zu verteilen, dass der linke Teilbaum m Knoten und der rechte Teilbaum die übrigen $n - 1 - m$ Knoten besitzt.