

# Praktische Informatik 3: Funktionale Programmierung

## Vorlesung 1 vom 02.11.2020: Einführung

Christoph Lüth



Wintersemester 2020/21

# Was ist Funktionale Programmierung?

- ▶ Programme als Funktionen — Funktionen als Programme
  - ▶ **Keine** veränderlichen Variablen
  - ▶ **Rekursion** statt while-Schleifen
- ▶ Funktionen als Daten — Daten als Funktionen
  - ▶ Erlaubt **Abstraktionsbildung**
- ▶ Denken in Algorithmen, nicht in Zustandsveränderung

# Lernziele

- ▶ **Konzepte** und **typische Merkmale** des funktionalen Programmierens kennen, verstehen und anwenden können:
  - ▶ Modellierung mit **algebraischen Datentypen**
  - ▶ **Rekursion**
  - ▶ Starke **Typisierung**
  - ▶ **Funktionen höher Ordnung** (map, filter, fold)
- ▶ Datenstrukturen und Algorithmen in einer funktionalen Programmiersprache **umsetzen** und auf einfachere praktische Probleme **anwenden** können.

## Modulhandbuch Informatik (Bachelor)

Die Vorlesung *Praktische Informatik 3* vermittelt essenzielles Grundwissen und Basisfähigkeiten, deren Beherrschung für nahezu jede vertiefte Beschäftigung mit Informatik Voraussetzung ist.

# I. Organisatorisches

# Personal

## ▶ **Vorlesung:**

Christoph Lüth <clueth@uni-bremen.de>

[www.informatik.uni-bremen.de/~clueth/](http://www.informatik.uni-bremen.de/~clueth/) (MZH 4186, Tel. 59830)

## ▶ **Tutoren:**

Thomas Barkoswky <barkowsky@informatik.uni-bremen.de>

Tobias Brandt <Tobias.Brandt@dfki.de>

Alexander Krug <krug@uni-bremen.de>

Robert Sachtleben <rob\_sac@uni-bremen.de>

Muhammad Tarek Soliman <soliman@uni-bremen.de>

## ▶ **Webseite:** [www.informatik.uni-bremen.de/~cxl/lehre/pi3.ws20](http://www.informatik.uni-bremen.de/~cxl/lehre/pi3.ws20)

# Corona-Edition

- ▶ Vorlesungen sind **asynchron**
  - ▶ Videos werden Montags zur Verfügung gestellt
  - ▶ Vorlesungen in mehreren Teilen mit Kurzübungen
- ▶ Übungen: Präsenz/Online
  - ▶ Präsenzbetrieb für 56 Stud./Woche
  - ▶ 3 Tutorien mit Präsenzbetrieb
    - ▶ Präsenztutorium ist **optional!**
    - ▶ Präsenztermine gekoppelt an TI2 (gleiche Kohorte)
  - ▶ 3 Online-Tutorien

# Termine

- ▶ **Vorlesung:** Online
- ▶ **Tutorien:**

Di	12– 14	MZH 1470	Robert	Online	Tobias
Do	10– 12	MZH 1470	Thomas	Online	Robert
	10– 12	MZH 1090	Tarek	Online	Alexander
- ▶ Alle Tutorien haben einen Zoom-Raum (für Präsenztutorien als Backup) — siehe Webseite
- ▶ Diese Woche **alle** Tutorien online — Präsenzbetrieb startet **nächste Woche**
- ▶ **Anmeldung** zu den Übungsgruppen über stud.ip (ab 18:00)
- ▶ **Sprechstunde:** Donnerstags 14-16 (via Zoom, bei Bedarf)

# Scheinkriterien

- ▶ Übungsblätter:
  - ▶ 6 Einzelübungsblätter (fünf beste werden gewertet)
  - ▶ 3 Gruppenübungsblätter (doppelt gewichtet)
- ▶ Übungsblätter der letzten Semester können **nicht** berücksichtigt werden
- ▶ Elektronische Klausur am Ende (Individualität der Leistung)
- ▶ Mind. 50% in den Einzelübungsblättern, in allen Übungsblättern und mind. 50% in der E-Klausur
- ▶ Note: 25% Übungsblätter und 75% E-Klausur
- ▶ **Notenspiegel** (in Prozent aller Punkte):

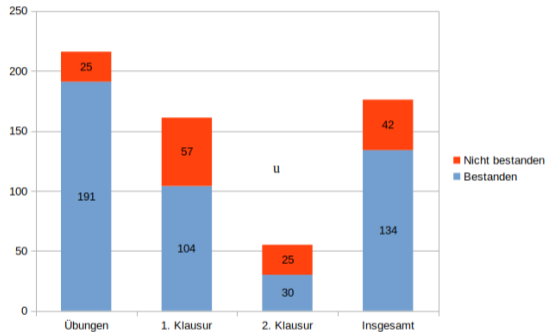
Pkt.%	Note	Pkt.%	Note	Pkt.%	Note	Pkt.%	Note
		89.5-85	1.7	74.5-70	2.7	59.5-55	3.7
≥ 95	1.0	84.5-80	2.0	69.5-65	3.0	54.5-50	4.0
94.5-90	1.3	79.5-75	2.3	64.5-60	3.3	49.5-0	<b>n/b</b>



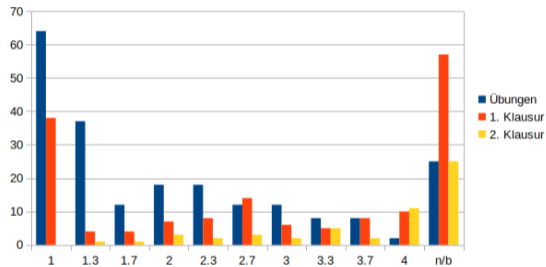
# Spielregeln

- ▶ **Quellen angeben** bei
  - ▶ Gruppenübergreifender Zusammenarbeit
  - ▶ Internetrecherche, Literatur, etc.
- ▶ **Täuschungsversuch:**
  - ▶ Null Punkte, **kein** Schein, **Meldung** an das **Prüfungsamt**
- ▶ **Deadline verpaßt?**
  - ▶ **Triftiger** Grund (z.B. Krankheit)
  - ▶ **Vorher** ankündigen, sonst **null** Punkte.

# Statistik von PI3 im Wintersemester 19/20



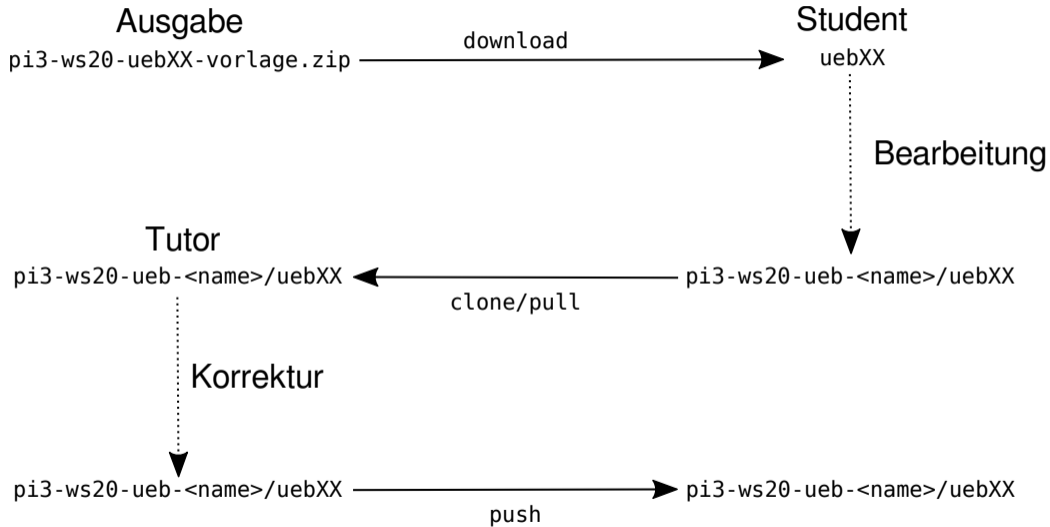
Notenverteilung



# Übungsbetrieb

- ▶ Ausgabe der Übungsblätter über die Webseite **Montag mittag**
- ▶ Besprechung der Übungsblätter in den Tutorien
- ▶ 6 Einzelübungsblätter:
  - ▶ Bearbeitungszeit bis **Montag folgender Woche 12:00**
  - ▶ Die fünf besten werden gewertet
- ▶ 3 Gruppenübungsblätter (doppelt gewichtet):
  - ▶ Bearbeitungszeit bis **Montag übernächster Woche 12:00**
  - ▶ Übungsgruppen: max. **drei Teilnehmer**
- ▶ **Abgabe** elektronisch
- ▶ **Bewertung:** Korrektheit, Angemessenheit (“Stil”), Dokumentation

# Ablauf des Übungsbetriebs



# II. Einführung

# Fahrplan

- ▶ **Teil I: Funktionale Programmierung im Kleinen**
  - ▶ Einführung
  - ▶ Funktionen
  - ▶ Algebraische Datentypen
  - ▶ Typvariablen und Polymorphie
  - ▶ Funktionen höherer Ordnung I
  - ▶ Rekursive und zyklische Datenstrukturen
  - ▶ Funktionen höherer Ordnung II
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

# Warum funktionale Programmierung lernen?

- ▶ Funktionale Programmierung macht aus Programmierern Informatiker
- ▶ Blick über den Tellerrand — was kommt in 10 Jahren?
- ▶ **Herausforderungen** der Zukunft:
  - ▶ Nebenläufige und **reaktive** Systeme (Mehrkernarchitekturen, serverless computing)
  - ▶ Massiv verteilte Systeme („Internet der Dinge“)
  - ▶ Große Datenmengen („Big Data“)

# The Future is Bright — The Future is Functional

- ▶ Funktionale Programmierung enthält die **wesentlichen** Elemente moderner Programmierung:
  - ▶ Datenabstraktion und Funktionale Abstraktion
  - ▶ Modularisierung
  - ▶ Typisierung und Spezifikation
- ▶ Funktionale Ideen jetzt im Mainstream:
  - ▶ Reflektion — LISP
  - ▶ Generics in Java — Polymorphie
  - ▶ Lambda-Fkt. in Java, C++ — Funktionen höherer Ordnung



# Geschichtliches: Die Anfänge

- ▶ **Grundlagen** 1920/30
  - ▶ Kombinatorlogik und  $\lambda$ -Kalkül (Schönfinkel, Curry, Church)
- ▶ Erste funktionale **Programmiersprachen** 1960
  - ▶ LISP (McCarthy), ISWIM (Landin)
- ▶ **Weitere** Programmiersprachen 1970– 80
  - ▶ FP (Backus); ML (Milner, Gordon); Hope (Burstall); Miranda (Turner)



Moses Schönfinkel



Haskell B. Curry



Alonzo Church



John McCarthy



John Backus



Robin Milner



Mike Gordon

# Geschichtliches: Die Gegenwart

- ▶ **Konsolidierung** 1990
  - ▶ CAML, Formale Semantik für Standard ML
  - ▶ Haskell als Standardsprache
- ▶ **Kommerzialisierung** 2010
  - ▶ OCaml
  - ▶ Scala, Clojure (JVM)
  - ▶ F# (.NET)

# Warum Haskell?

- ▶ **Moderne** Sprache
- ▶ Standardisiert, mehrere **Implementationen**
  - ▶ Interpreter: `ghci`, `hugs`
  - ▶ Compiler: `ghc`, `nhc98`
  - ▶ Build: `stack`
- ▶ **Rein** funktional
  - ▶ **Essenz** der funktionalen Programmierung



# Programme als Funktionen

- ▶ Programme als Funktionen:

$$P : \text{Eingabe} \rightarrow \text{Ausgabe}$$

- ▶ Keine veränderlichen **Variablen** — kein versteckter **Zustand**
- ▶ Rückgabewert hängt ausschließlich von Werten der Argumente ab, nicht vom Aufrufkontext (**referentielle Transparenz**)
- ▶ Alle **Abhängigkeiten explizit**

# Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2
```

# Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)
```

# Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)  
      → if False then 1 else 2* fac 1
```

# Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)  
      → if False then 1 else 2* fac 1  
      → 2* fac 1
```



# Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)
      → if False then 1 else 2* fac 1
      → 2* fac 1
      → 2* if 1 == 0 then 1 else 1* fac (1-1)
```

# Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)
      → if False then 1 else 2* fac 1
      → 2* fac 1
      → 2* if 1 == 0 then 1 else 1* fac (1-1)
      → 2* if False then 1 else 1* fac (1-1)
```

# Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)
      → if False then 1 else 2* fac 1
      → 2* fac 1
      → 2* if 1 == 0 then 1 else 1* fac (1-1)
      → 2* if False then 1 else 1* fac (1-1)
      → 2* 1* fac 0
```

# Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)
      → if False then 1 else 2* fac 1
      → 2* fac 1
      → 2* if 1 == 0 then 1 else 1* fac (1-1)
      → 2* if False then 1 else 1* fac (1-1)
      → 2* 1* fac 0
      → 2* 1* if 0 == 0 then 1 else 0* fac (0-1)
```

# Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)
      → if False then 1 else 2* fac 1
      → 2* fac 1
      → 2* if 1 == 0 then 1 else 1* fac (1-1)
      → 2* if False then 1 else 1* fac (1-1)
      → 2* 1* fac 0
      → 2* 1* if 0 == 0 then 1 else 0* fac (0-1)
      → 2* 1* if True then 1 else 0* fac (0-1)
```

# Beispiel: Programmieren mit Funktionen

- ▶ **Programme** werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)
      → if False then 1 else 2* fac 1
      → 2* fac 1
      → 2* if 1 == 0 then 1 else 1* fac (1-1)
      → 2* if False then 1 else 1* fac (1-1)
      → 2* 1* fac 0
      → 2* 1* if 0 == 0 then 1 else 0* fac (0-1)
      → 2* 1* if True then 1 else 0* fac (0-1)
      → 2* 1* 1 → 2
```

## Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"
```

## Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"
```

```
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"
```



## Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"
```

```
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"
```

```
→ if False then "" else "hallo_" ++ repeat 1 "hallo_"
```

## Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"
```

```
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"
```

```
→ if False then "" else "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ repeat 1 "hallo_"
```

## Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"
```

```
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"
```

```
→ if False then "" else "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ if 1 == 0 then "" else "hallo_" ++ repeat (1-1) "hallo_"
```

## Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"
```

```
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"
```

```
→ if False then "" else "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ if 1 == 0 then "" else "hallo_" ++ repeat (1-1) "hallo_"
```

```
→ "hallo_" ++ if False then "" else "hallo_" ++ repeat 1 "hallo_"
```

## Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"
```

```
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"
```

```
→ if False then "" else "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ if 1 == 0 then "" else "hallo_" ++ repeat (1-1) "hallo_"
```

```
→ "hallo_" ++ if False then "" else "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ ("hallo_" ++ repeat 0 "hallo_")
```

## Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"
```

```
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"
```

```
→ if False then "" else "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ if 1 == 0 then "" else "hallo_" ++ repeat (1-1) "hallo_"
```

```
→ "hallo_" ++ if False then "" else "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ ("hallo_" ++ repeat 0 "hallo_")
```

```
→ "hallo_" ++ ("hallo_" ++ if 0 == 0 then "" else "hallo_" ++ repeat (0-1) "hallo_")
```

## Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"
```

```
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"
```

```
→ if False then "" else "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ if 1 == 0 then "" else "hallo_" ++ repeat (1-1) "hallo_"
```

```
→ "hallo_" ++ if False then "" else "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ ("hallo_" ++ repeat 0 "hallo_")
```

```
→ "hallo_" ++ ("hallo_" ++ if 0 == 0 then "" else "hallo_" ++ repeat (0-1) "hallo_")
```

```
→ "hallo_" ++ ("hallo_" ++ if True then "" else "hallo_" ++ repeat (-1) "hallo_")
```

## Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"
```

```
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"
```

```
→ if False then "" else "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ if 1 == 0 then "" else "hallo_" ++ repeat (1-1) "hallo_"
```

```
→ "hallo_" ++ if False then "" else "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ ("hallo_" ++ repeat 0 "hallo_")
```

```
→ "hallo_" ++ ("hallo_" ++ if 0 == 0 then "" else "hallo_" ++ repeat (0-1) "hallo_")
```

```
→ "hallo_" ++ ("hallo_" ++ if True then "" else "hallo_" ++ repeat (-1) "hallo_")
```

```
→ "hallo_" ++ ("hallo_" ++ "")
```



## Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s ++ repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo_"
```

```
→ if 2 == 0 then "" else "hallo_" ++ repeat (2-1) "hallo_"
```

```
→ if False then "" else "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ if 1 == 0 then "" else "hallo_" ++ repeat (1-1) "hallo_"
```

```
→ "hallo_" ++ if False then "" else "hallo_" ++ repeat 1 "hallo_"
```

```
→ "hallo_" ++ ("hallo_" ++ repeat 0 "hallo_")
```

```
→ "hallo_" ++ ("hallo_" ++ if 0 == 0 then "" else "hallo_" ++ repeat (0-1) "hallo_")
```

```
→ "hallo_" ++ ("hallo_" ++ if True then "" else "hallo_" ++ repeat (-1) "hallo_")
```

```
→ "hallo_" ++ ("hallo_" ++ "")
```

```
→ "hallo_hallo_"
```

# Auswertung als Ausführungsbegriff

- ▶ **Programme** werden durch **Gleichungen** definiert:

$$f(x) = E$$

- ▶ **Auswertung** durch **Anwenden** der Gleichungen:

- ▶ Suchen nach **Vorkommen** von  $f$ , e.g.  $f(t)$

- ▶  $f(t)$  wird durch  $E \begin{bmatrix} t \\ x \end{bmatrix}$  ersetzt

- ▶ Auswertung kann **divergieren**!

# Ausdrücke und Werte

- ▶ Nichtreduzierbare Ausdrücke sind **Werte**
- ▶ Vorgegebene **Basiswerte**: Zahlen, Zeichen
  - ▶ Durch **Implementation** gegeben
- ▶ Definierte **Datentypen**: Wahrheitswerte, Listen, ...
  - ▶ **Modellierung** von Daten

# Jetzt seid ihr dran!

## Übung 1.1: Auswertung

Hier ist eine weitere Beispiel-Funktion:

```
stars n = if n > 1 then stars (div n 2) ++ "*" else ""
```

`div n m` ist die ganzzahlige Division: `div 7 2` → 3

Berechnet wie oben die Reduktion von `stars 5`

# Jetzt seid ihr dran!

## Übung 1.1: Auswertung

Hier ist eine weitere Beispiel-Funktion:

```
stars n = if n > 1 then stars (div n 2) ++ "*" else ""
```

`div n m` ist die ganzzahlige Division: `div 7 2` → 3

Berechnet wie oben die Reduktion von `stars 5`

Lösung:

`stars 5` →

# Jetzt seit ihr dran!

## Übung 1.1: Auswertung

Hier ist eine weitere Beispiel-Funktion:

```
stars n = if n > 1 then stars (div n 2) ++ "*" else ""
```

div n m ist die ganzzahlige Division:  $\text{div } 7 \ 2 \rightarrow 3$

Berechnet wie oben die Reduktion von `stars 5`

Lösung:

```
stars 5 → if 5 > 1 then stars (div 5 2) ++ "*" else ""  
→
```

# Jetzt seit ihr dran!

## Übung 1.1: Auswertung

Hier ist eine weitere Beispiel-Funktion:

```
stars n = if n > 1 then stars (div n 2) ++ "*" else ""
```

div n m ist die ganzzahlige Division:  $\text{div } 7 \ 2 \rightarrow 3$

Berechnet wie oben die Reduktion von `stars 5`

Lösung:

```
stars 5  →if 5 > 1 then stars (div 5 2) ++ "*" else ""
         →stars 2 ++ "*"
         →
```

# Jetzt seit ihr dran!

## Übung 1.1: Auswertung

Hier ist eine weitere Beispiel-Funktion:

```
stars n = if n > 1 then stars (div n 2) ++ "*" else ""
```

div n m ist die ganzzahlige Division:  $\text{div } 7 \ 2 \rightarrow 3$

Berechnet wie oben die Reduktion von `stars 5`

Lösung:

```
stars 5  →if 5 > 1 then stars (div 5 2) ++ "*" else ""
         →stars 2 ++ "*"
         →(if 2 > 1 then stars (div 2 2) ++ "*" else "") ++ "*"
         →
```



# Jetzt seit ihr dran!

## Übung 1.1: Auswertung

Hier ist eine weitere Beispiel-Funktion:

```
stars n = if n > 1 then stars (div n 2) ++ "*" else ""
```

div n m ist die ganzzahlige Division:  $\text{div } 7 \ 2 \rightarrow 3$

Berechnet wie oben die Reduktion von `stars 5`

Lösung:

```
stars 5  →if 5 > 1 then stars (div 5 2) ++ "*" else ""
         →stars 2 ++ "*"
         →(if 2 > 1 then stars (div 2 2) ++ "*" else "") ++ "*"
         →(stars 1 ++ "*") ++ "*"
         →
```

# Jetzt seit ihr dran!

## Übung 1.1: Auswertung

Hier ist eine weitere Beispiel-Funktion:

```
stars n = if n > 1 then stars (div n 2) ++ "*" else ""
```

div n m ist die ganzzahlige Division:  $\text{div } 7 \ 2 \rightarrow 3$

Berechnet wie oben die Reduktion von `stars 5`

Lösung:

```
stars 5  →if 5 > 1 then stars (div 5 2) ++ "*" else ""
        →stars 2 ++ "*"
        →(if 2 > 1 then stars (div 2 2) ++ "*" else "") ++ "*"
        →(stars 1 ++ "*") ++ "*"
        →((if 1 > 1 then stars (div 1 2) ++ "*" else "") ++ "*") ++ "*"
        →
```

# Jetzt seit ihr dran!

## Übung 1.1: Auswertung

Hier ist eine weitere Beispiel-Funktion:

```
stars n = if n > 1 then stars (div n 2) ++ "*" else ""
```

div n m ist die ganzzahlige Division:  $\text{div } 7 \ 2 \rightarrow 3$

Berechnet wie oben die Reduktion von `stars 5`

Lösung:

```
stars 5  →if 5 > 1 then stars (div 5 2) ++ "*" else ""
        →stars 2 ++ "*"
        →(if 2 > 1 then stars (div 2 2) ++ "*" else "") ++ "*"
        →(stars 1 ++ "*") ++ "*"
        →((if 1 > 1 then stars (div 1 2) ++ "*" else "") ++ "*") ++ "*"
        →("" ++ "*") ++ "*" → "**"
```

# III. Typen

# Typisierung

- ▶ **Typen** unterscheiden Arten von Ausdrücken und Werten:

`repeat n s = ...`      `n` Zahl  
                                 `s` Zeichenkette

- ▶ **Wozu** Typen?
  - ▶ Frühzeitiges Aufdecken “offensichtlicher” Fehler
  - ▶ Erhöhte **Programmsicherheit**
  - ▶ Hilfestellung bei **Änderungen**

## Slogan

“Well-typed programs can't go wrong.”

— *Robin Milner*

# Signaturen

- ▶ Jede Funktion hat eine **Signatur**

```
fac    :: Int → Int
```

```
repeat :: Int → String → String
```

- ▶ **Typüberprüfung**

- ▶ `fac` nur auf `Int` anwendbar, Resultat ist `Int`
- ▶ `repeat` nur auf `Int` und `String` anwendbar, Resultat ist `String`

# Übersicht: Typen in Haskell

Typ	Bezeichner	Beispiel		
Ganze Zahlen	<code>Int</code>	<code>0</code>	<code>94</code>	<code>-45</code>
Fließkomma	<code>Double</code>	<code>3.0</code>	<code>3.141592</code>	
Zeichen	<code>Char</code>	<code>'a'</code> <code>'x'</code>	<code>'\034'</code>	<code>'\n'</code>
Zeichenketten	<code>String</code>	<code>"yuck"</code>	<code>"hi\nho"\n"</code>	
Wahrheitswerte	<code>Bool</code>	<code>True</code>	<code>False</code>	
Funktionen	<code>a → b</code>			

► Später mehr. **Viel** mehr.

# Das Rechnen mit Zahlen

**Beschränkte** Genauigkeit,  
**konstanter** Aufwand  $\longleftrightarrow$  **beliebige** Genauigkeit,  
**wachsender** Aufwand



# Das Rechnen mit Zahlen

**Beschränkte** Genauigkeit,  
**konstanter** Aufwand  $\longleftrightarrow$  **beliebige** Genauigkeit,  
**wachsender** Aufwand

Haskell bietet die Auswahl:

- ▶ `Int` - ganze Zahlen als Maschinenworte ( $\geq 31$  Bit)
- ▶ `Integer` - beliebig große ganze Zahlen
- ▶ `Rational` - beliebig genaue rationale Zahlen
- ▶ `Float`, `Double` - Fließkommazahlen (reelle Zahlen)

# Ganze Zahlen: Int und Integer

- ▶ Nützliche Funktionen (**überladen**, auch für `Integer`):

```
+ , * , ^ , - :: Int -> Int -> Int
abs           :: Int -> Int — Betrag
div, quot    :: Int -> Int -> Int
mod, rem     :: Int -> Int -> Int
```

Es gilt:  $(\text{div } x \ y) * y + \text{mod } x \ y = x$

- ▶ Vergleich durch  $=$ ,  $\neq$ ,  $\leq$ ,  $<$ , ...
- ▶ **Achtung:** Unäres Minus
  - ▶ Unterschied zum Infix-Operator  $-$
  - ▶ Im Zweifelsfall klammern: `abs (-34)`

# Fließkommazahlen: Double

- ▶ Doppeltgenaue Fließkommazahlen (IEEE 754 und 854)
  - ▶ Logarithmen, Wurzel, Exponentiation,  $\pi$  und  $e$ , trigonometrische Funktionen
- ▶ Konversion in ganze Zahlen:
  - ▶ `fromIntegral :: Int, Integer → Double`
  - ▶ `fromInteger :: Integer → Double`
  - ▶ `round, truncate :: Double → Int, Integer`
  - ▶ Überladungen mit Typannotation auflösen:

```
round (fromInt 10) :: Int
```

- ▶ **Rundungsfehler!**

# Alphanumerische Basisdatentypen: Char

▶ Notation für einzelne **Zeichen**: 'a',...

▶ Nützliche **Funktionen**:

```
ord :: Char → Int  
chr :: Int → Char
```

```
toLower :: Char → Char
```

```
toUpper :: Char → Char
```

```
isDigit :: Char → Bool
```

```
isAlpha :: Char → Bool
```

▶ Zeichenketten: String



## Jetzt seit ihr noch mal dran.

- ▶ ZIP-Datei mit den Quellen auf der Webseite verlinkt (Rubrik *Vorlesung*)
- ▶ Für diese Vorlesung: eine Datei `Examples.hs` mit den Quellen der Funktionen `fac`, `repeat` und `start`.
- ▶ Unter der Rubrik *Übung*: Kurzanleitung PI3-Übungsbetrieb
- ▶ Durchlesen und Haskell Tool Stack installieren, Experimente ausprobieren, 0. Übungsblatt angehen.

### Übung 1.2: Mehr Sterne

Ändert die Funktion `stars` so ab, dass sie eine Zeichenkette aus `n` Sternchen zurückgibt.

# Zusammenfassung

- ▶ **Programme** sind **Funktionen**, definiert durch **Gleichungen**
  - ▶ Referentielle Transparenz
  - ▶ kein impliziter Zustand, keine veränderlichen Variablen
- ▶ **Ausführung** durch **Reduktion** von Ausdrücken
- ▶ Typisierung:
  - ▶ **Basistypen**: Zahlen, Zeichen(ketten), Wahrheitswerte
  - ▶ Jede Funktion  $f$  hat eine Signatur  $f :: a \rightarrow b$