



Praktische Informatik 3: Funktionale Programmierung

Vorlesung 1 (18.11.2022): Einführung

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität
Bremen

Wintersemester 2022/23

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 und Termine

► Vorlesung:

Di 14– 16 NW2 C0290 Christoph Lüth <clueth@uni-bremen.de>
www.informatik.uni-bremen.de/~clueth/
MZH 4186, Tel. 218-59830

► Tutoren:

Di	12– 14	MZH 1110	Tede von Knorre	<tede@uni-bremen.de>
	12– 14	MZH 5600	Raphael Baass	<rbaass@uni-bremen.de>
Mi	14– 16	MZH 1110	Thomas Barkoswky	<barkowsky@uni-bremen.de>
	14– 16	MZH 1450	Alexander Krug	<krug@uni-bremen.de>
	14– 16	Online	Muhammad Tarek Soliman	<soliman@uni-bremen.de>

► Webseite: www.informatik.uni-bremen.de/~clueth/lehre/pi3.ws22

Scheinbedingungen

- ▶ Übungsblätter:
 - ▶ 6 Einzelübungsblätter (fünf beste werden gewertet) und
 - ▶ 3 Gruppenübungsblätter (doppelt gewichtet)
- ▶ Übungsblätter der letzten Semester werden nur in **Ausnahmefällen** berücksichtigt:
 - ▶ Klausur durch **Krankheit** oder **Corona** verpasst.
- ▶ Individualität der Leistung: **Elektronische Klausur** am Ende

Elektronische Klausur

- ▶ **Termin:** 13.03.2023, 14:00 und 15:45
- ▶ **Ort:** Testzentrum am Boulevard neben der Bibliothek
- ▶ **Dauer:** 90 Minuten
- ▶ **Ablauf:**
 - ▶ Einfache Programmierübungen in der Art der Übungsaufgaben
 - ▶ Einige Multiple-Choice Fragen als **Bonus**

Scheinbedingungen

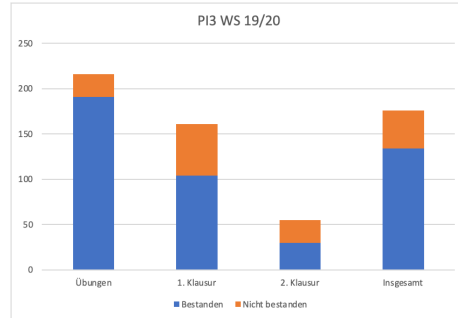
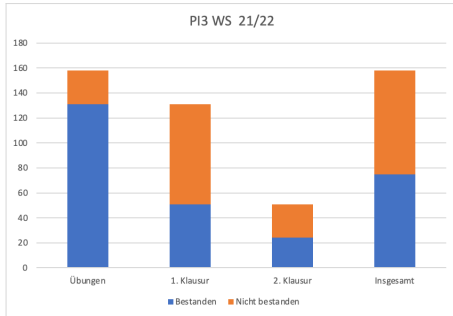
- ▶ Mindestens 50% in den Einzelübungsblättern, in allen Übungsblättern und mindestens 50% in der E-Klausur
- ▶ Note: 50% Übungsblätter und 50% E-Klausur
- ▶ **Notenspiegel** (in Prozent aller Punkte):

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	n/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.

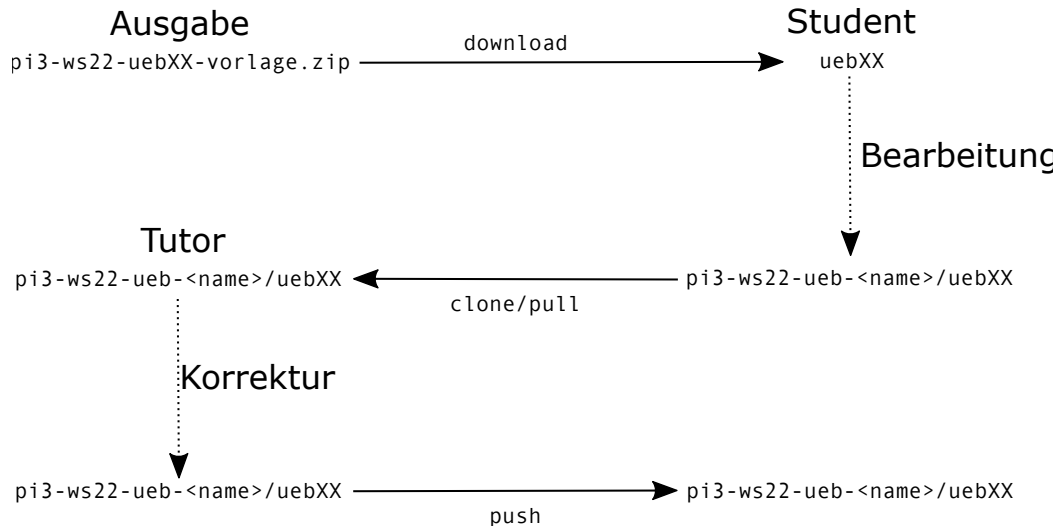
Statistik von PI3 im Wintersemester 21/22



Übungsbetrieb

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

Ablauf des Übungsbetriebs



Warnung



- ▶ PI3 ist **nicht** die Fortsetzung von PI1 und PI2.
- ▶ Funktionale Programmierung ist **anders** und kann als **schwer** empfunden werden.
- ▶ Regelmäßige Bearbeitung der **Übungsblätter** hilft.
- ▶ Sucht **rechtzeitig** Unterstützung!

II. Einführung

► 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-Funktionen in Java, C++ — Funktionen höherer Ordnung

Geschichtliches: Die Anfänge

- ▶ **Grundlagen** 1920/30
 - ▶ Kombinatorlogik und λ -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**)

Beispiel: Programmieren mit Funktionen

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

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

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

```
fact 2
```

Beispiel: Programmieren mit Funktionen

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

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

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

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

Beispiel: Programmieren mit Funktionen

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

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

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

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


Beispiel: Programmieren mit Funktionen

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

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

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

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

Beispiel: Programmieren mit Funktionen

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

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

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

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

Beispiel: Programmieren mit Funktionen

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

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

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

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

Beispiel: Programmieren mit Funktionen

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

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

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

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

Beispiel: Programmieren mit Funktionen

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

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

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

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

Beispiel: Programmieren mit Funktionen

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

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

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

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

Beispiel: Programmieren mit Funktionen

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

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

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

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

Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

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

- ▶ Auswertung:

```
rep 2 "hallo_"
```


Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

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

- ▶ Auswertung:

```
rep 2 "hallo_"
```

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

Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

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

- ▶ Auswertung:

```
rep 2 "hallo_"
```

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

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

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

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

- Auswertung:

```
rep 2 "hallo_"
```

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

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

```
→ "hallo_" ++ rep 1 "hallo_"
```

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

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

- Auswertung:

```
rep 2 "hallo_"
```

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

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

```
→ "hallo_" ++ rep 1 "hallo_"
```

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

Beispiel: Nichtnumerische Werte

► Rechnen mit Zeichenketten

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

► Auswertung:

```
rep 2 "hallo_"
```

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

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

```
→ "hallo_" ++ rep 1 "hallo_"
```

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

```
→ "hallo_" ++ if False then "" else "hallo_" ++ rep 0 "hallo_"
```

Beispiel: Nichtnumerische Werte

► Rechnen mit Zeichenketten

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

► Auswertung:

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

Beispiel: Nichtnumerische Werte

► Rechnen mit Zeichenketten

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

► Auswertung:

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

Beispiel: Nichtnumerische Werte

► Rechnen mit Zeichenketten

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

► Auswertung:

```
rep 2 "hallo_"  
→ if 2 == 0 then "" else "hallo_" ++ rep (2-1) "hallo_"  
→ if False then "" else "hallo_" ++ rep 1 "hallo_"  
→ "hallo_" ++ rep 1 "hallo_"  
→ "hallo_" ++ if 1 == 0 then "" else "hallo_" ++ rep (1-1) "hallo_"  
→ "hallo_" ++ if False then "" else "hallo_" ++ rep 0 "hallo_"  
→ "hallo_" ++ ("hallo_" ++ rep 0 "hallo_")  
→ "hallo_" ++ ("hallo_" ++ if 0 == 0 then "" else "hallo_" ++ rep (0-1) "hallo_")  
→ "hallo_" ++ ("hallo_" ++ if True then "" else "hallo_" ++ rep (-1) "hallo_")
```


Beispiel: Nichtnumerische Werte

► Rechnen mit Zeichenketten

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

► Auswertung:

```
rep 2 "hallo_"  
→ if 2 == 0 then "" else "hallo_" ++ rep (2-1) "hallo_"  
→ if False then "" else "hallo_" ++ rep 1 "hallo_"  
→ "hallo_" ++ rep 1 "hallo_"  
→ "hallo_" ++ if 1 == 0 then "" else "hallo_" ++ rep (1-1) "hallo_"  
→ "hallo_" ++ if False then "" else "hallo_" ++ rep 0 "hallo_"  
→ "hallo_" ++ ("hallo_" ++ rep 0 "hallo_")  
→ "hallo_" ++ ("hallo_" ++ if 0 == 0 then "" else "hallo_" ++ rep (0-1) "hallo_")  
→ "hallo_" ++ ("hallo_" ++ if True then "" else "hallo_" ++ rep (-1) "hallo_")  
→ "hallo_" ++ ("hallo_" ++ "")
```

Beispiel: Nichtnumerische Werte

► Rechnen mit Zeichenketten

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

► Auswertung:

```
rep 2 "hallo_"  
→ if 2 == 0 then "" else "hallo_" ++ rep (2-1) "hallo_"  
→ if False then "" else "hallo_" ++ rep 1 "hallo_"  
→ "hallo_" ++ rep 1 "hallo_"  
→ "hallo_" ++ if 1 == 0 then "" else "hallo_" ++ rep (1-1) "hallo_"  
→ "hallo_" ++ if False then "" else "hallo_" ++ rep 0 "hallo_"  
→ "hallo_" ++ ("hallo_" ++ rep 0 "hallo_")  
→ "hallo_" ++ ("hallo_" ++ if 0 == 0 then "" else "hallo_" ++ rep (0-1) "hallo_")  
→ "hallo_" ++ ("hallo_" ++ if True then "" else "hallo_" ++ rep (-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

III. Typen

Typisierung

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

rep	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**

```
fact :: Int → Int
```

```
rep  :: Int → String → String
```

- ▶ **Typüberprüfung**

- ▶ `fact` nur auf `Int` anwendbar, Resultat ist `Int`
- ▶ `rep` 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 (≥ 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, π 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



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 $f :: a \rightarrow b$