Praktische Informatik 3: Funktionale Programmierung

Vorlesung 1 / 17. Oktober 2023

Einführung

Thomas Barkowsky

Wintersemester 2023/24

heute in dieser Vorlesung...

- Organisatorisches
  - Übungen, Scheinkriterien, ...
- Programme als Funktionen
  - Funktionale Programmierung, Haskell, Definition von Funktionen, ...
- Berechnung als Auswertung
  - Auswertung von Ausdrücken, Typisierung, Signaturen, ...

Organisatorisches

Das PI3 Team

- Vorlesung: Thomas Barkowsky
  - barkowsky@uni-bremen.de
  - Cartesium 3.45, Tel. 64233

- Tutorien
  - Thomas Barkowsky (barkowsky@uni-bremen.de)
  - Tede von Knorre (tede@uni-bremen.de)
  - Yaroslav Purgin (purgin@uni-bremen.de)
  - Tim Sperling (tsperlin@uni-bremen.de)

Termine

- Vorlesung: Di 14-16 NW2 C0290 (Hörsaal 1)
- Tutorien:
  - Di 12-14 MZH 1110: Tede von Knorre
  - Di 12-14 MZH 5500: Yaroslav Purgin
  - Mi 14-16 MZH 1110: Thomas Barkowsky
  - Mi 14-16 MZH 4140: Yaroslav Purgin / Tede von Knorre
  - Mi 14-16 online: Tim Sperling
- Anmeldung zu den Übungsgruppen
  - über Stud.IP

Termine: E-Klausuren

- Probeklausur: Do 07.12.2023, 10:00 und 10:45
- Hauptklausur: Mo 11.03.2024, 10:00 und 11:45
- Wiederholungsklausur: Do 18.04.2024, 10:00

- Anmeldung rechtzeitig vorher

Übungen

- Ausgabe der Übungsblätter über Stud.IP immer Montags
  - Übungsblatt 0 seit gestern online (ohne Bewertung)
- 6 Einzelübungsblätter
  - Besprechung und Bearbeitung in den Tutorien
  - Abgabe immer Montags (bis Mitternacht) in der Folgewoche
- 3 Gruppenübungsblätter (doppelte Punktzahl)
  - Abgabe am Montag der jeweils übernächsten Woche
  - Gruppengröße max. 3 Studierende

Übungen: Abgabe und Bewertung

- Abgabe über FB3 Gitlab-Repository
  - eigenes Gitlab-Repository erzeugen
  - Tutor einladen
  - jeweilige Aufgabe aus Stud.IP herunterladen, im Repository auspacken, Aufgabe lösen und wieder hochladen
  - Tutor checked Lösung aus, korrigiert, bewertet und lädt wieder hoch
  - siehe Anleitung „HowTo Haskell“, Abschnitt 2
- Bewertung:
  - Korrektheit der Lösung (80%)
  - Stil und Dokumentation (20%)

Scheinkriterien

- Einzel-Übungsblätter, Gruppen-Übungsblätter, elektronische Klausur (“Programmierübung”)
  - 1. Meilenstein: Die 5 besten der 6 Einzel-Übungsblätter werden bewertet und mindestens 50% der darin erreichbaren Punkten müssen erreicht werden
  - 2. Meilenstein: Bewertung der drei Gruppen-Übungsblätter
  - Mindestens 50% der möglichen Punkte aller Übungsblätter (Einzel- und Gruppen-Übungsblätter) müssen erreicht werden
  - 3. Meilenstein: Bewertung der E-Klausur; mindestens 50% der möglichen Punkte müssen erreicht werden
- Gesamtnote gemittelt aus Übungsblättern und E-Klausur

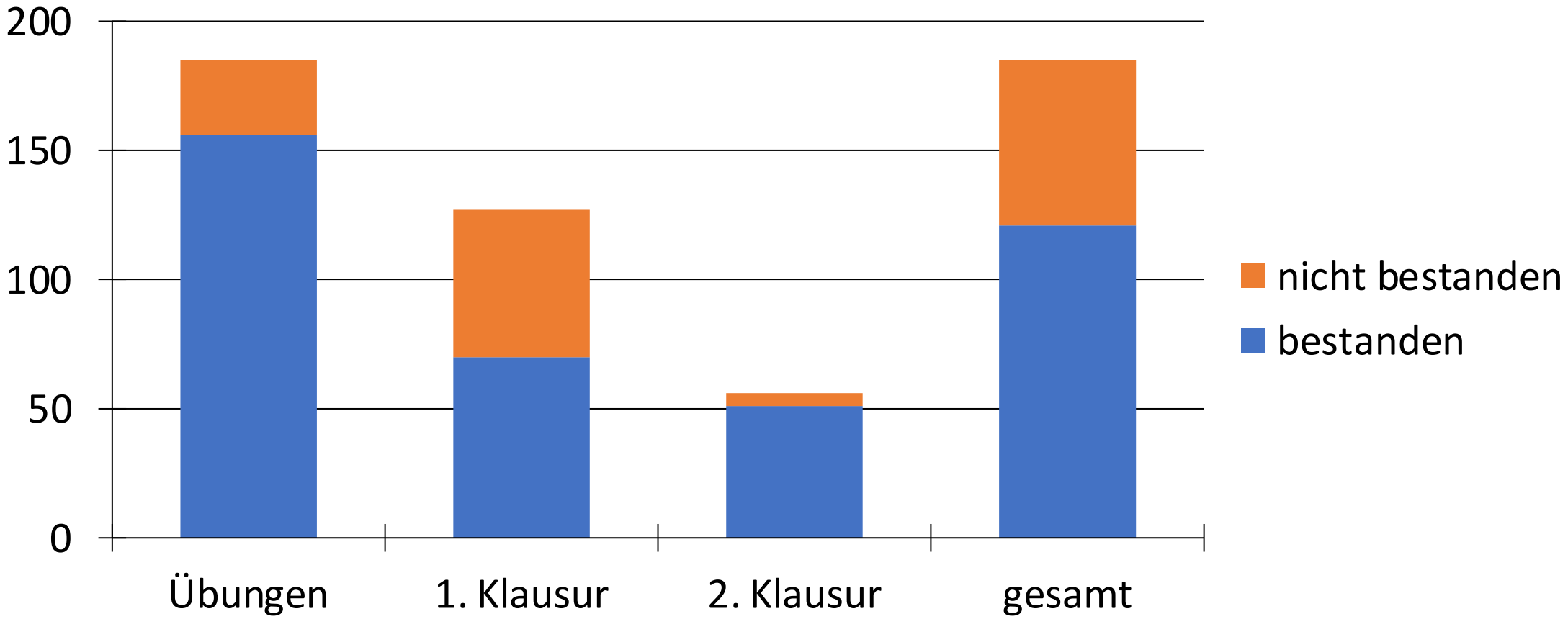
Notenspiegel (in % aller Punkte)

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.

Regeln für den Übungsbetrieb

- Quellen angeben bei
  - gruppenübergreifender Zusammenarbeit
  - Internetrecherche, Literatur, sonstige Quellen
- Täuschungsversuch:
  - null Punkte, kein Schein, Meldung an das Prüfungsamt
- Deadline verpasst?
  - bei triftigem Grund (z.B. Krankheit mehrerer Gruppenmitglieder)
  - rechtzeitige Meldung an den Tutor oder an mich
  - sonst: null Punkte

Statistik von PI3 im Wintersemester 2022/23



Stud.IP

- Eintragung in Übungsgruppen
- Übungsaufgaben (Aufgabenblätter und Programmiervorlagen)
- Vorlesungsfolien (Folien und Programmbeispiele)
- Ankündigungen

Links & Ressourcen

- Die Haskell Homepage: www.haskell.org
  - Hauptanlaufstelle, u.a. für
  - Implementierungen für div. Betriebssysteme (www.haskell.org/downloads/)
  - Haskell-Communities, u.v.m.
  - https://hoogle.haskell.org (Suchmaschine für Haskell-Funktionen)
- Die Sprachdefinition: www.haskell.org/onlinereport/haskell2010/
  - ... auch als PDF: www.haskell.org/definition/haskell2010.pdf

Haskell Bücher

- Simon Thompson (2011). *Haskell: The Craft of Functional Programming*, 3rd ed., Addison Wesley.
- Graham Hutton (2016). *Programming in Haskell*, 2nd ed., Cambridge University Press.
- Miran Lipovaca (2011). *Learn You a Haskell for Great Good!*, No Starch Press.
- Bryan O'Sullivan, John Goerzen, Don Stewart (2009). *Real World Haskell*, O'Reilly.

Themen der Lehrveranstaltung

- Datentypen und Funktionen
- Rekursion und Listen
- Funktionen höherer Ordnung
- Algebraische Datentypen
- Rekursive und zyklische Datenstrukturen
- Abstrakte Datentypen
- Testen und Qualitätssicherung
- I/O, Aktionen und Zustände
- Monaden
- Domänenspezifische Sprachen
- Funktionale Programmierung in der Praxis

Programme als Funktionen

Funktionale Programmierung?

- Ein Programmier**paradigma**
  - vgl. imperative, objektorientierte, logikbasierte, ... Programmierung
- Berechnung durch Anwendung **Funktionen** auf Argumente
- Eine Funktionale Sprache **unterstützt** Funktionale Programmierung

Warum funktionale Programmierung?

- Sichere, fehlerfreie Systeme
- General-purpose Sprache → domänenspezifische Sprachen
- Konzeptuelle Entwicklung und Ausbildung
- Entwicklung eigener informatischer Expertise
- Zunehmende Verbreitung auch in *mainstream* Umgebungen

Warum Haskell?

- Moderne, standardisierte Sprache
- Rein funktional
- Streng typisiert, mit Typinferenz
- Nebenläufige Programmierung
- Lazy Evaluation – Auswertung nur wenn erforderlich
- Große Community, Open Source Entwicklungen, Packages

Haskell

- Erster Entwurf 1990
- Aktuelle Sprachdefinition: Haskell 2010 Language Report
- Benannt nach Haskell B. Curry (1900-1982)
  - Kombinatorische Logik / Lambda-Kalkül
- www.haskell.org
- Glasgow Haskell Compiler (interactive) – GHC(i)
- Standard Distribution: **The Haskell Tool Stack**

Programme als Funktionen

- Eine Funktion erzeugt **einen** Ausgabewert (Ergebnis, Resultat) abhängig von (**einem** oder **mehreren**) Eingabewerten (Argumenten, Parametern)

P: Eingabe → Ausgabe

- Die Verwendung einer Funktion mit Eingabewerten heißt Funktionsaufruf (Funktionsanwendung)
- Werte von Argumenten und Funktionen, die mit diesen Argumenten arbeiten, bilden die Basis

Referenzielle Transparenz

- Ausgabewerte von Funktionen hängen ausschließlich von den aktuellen Eingabewerten ab, nicht vom Kontext des Aufrufs
- Dieselben Eingabewerte erzeugen immer dieselbe Ausgabe
- Keine Seiteneffekte, keine versteckten Zustände
- Alle Abhängikeiten explizit in Funktionsdefinition

Haskell Programmierung

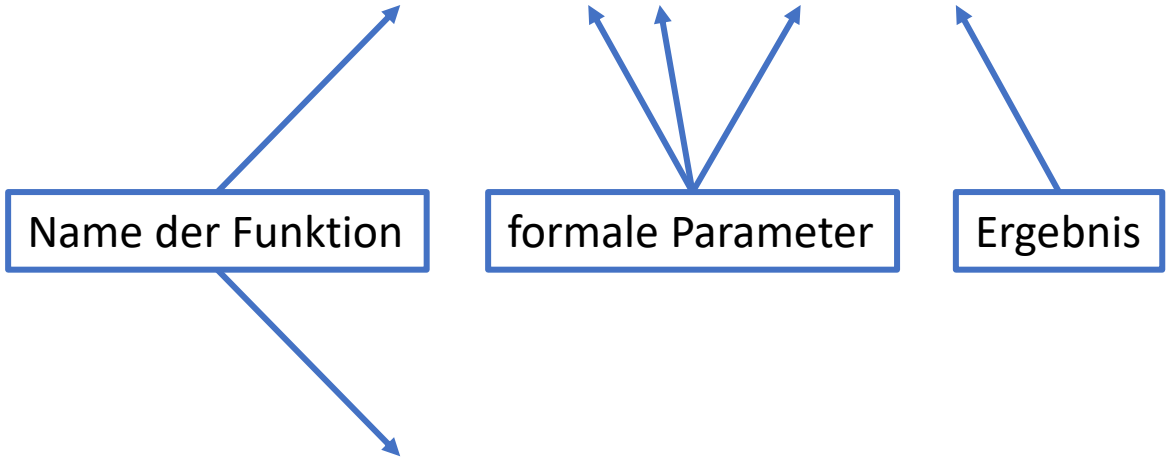
- Haskell Programme bestehen aus Definitionen von Funktionen und Werten
- Eine Haskell Definition verbindet einen Namen (identifier) mit einem Ausdruck oder Wert eines bestimmten Typs
- Beispiel:

```
square :: Integer -> Integer
square n = n*n
```

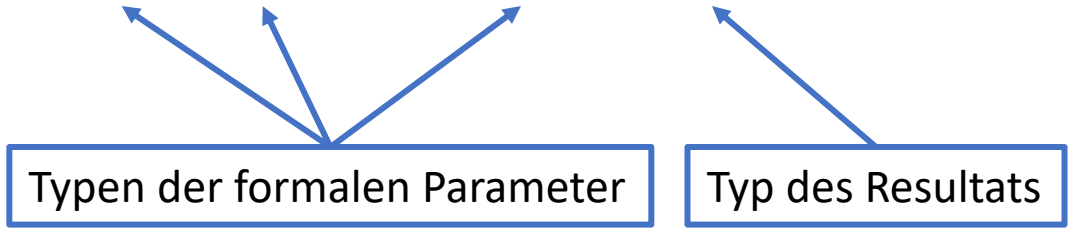
- Funktionsnamen beginnen mit Kleinbuchstaben
- Typbezeichnungen beginnen mit Großbuchstaben

Funktionsdefinition allgemein

- Definition der Funktion: name x1 x2 ... xk = e



- Typdeklaration: name :: t1 -> t2 -> ... -> tk -> t



Verwendung von Funktionen

- Funktionen und Argumente ohne Klammerung
  - z.B. `f(a,b)+cd` (mathematisch) vs. `f a b + c*d` (Haskell)
- Funktionsanwendung hat höchste Priorität

  - weitere Beispiele:

mathemat.	Haskell
`f(x)`	`f x`
`f(x,y)`	`f x y`
`f(g(x))`	`f (g x)`
`f(x,g(y))`	`f x (g y)`
`f(x)f(y)`	`f x * g y`

Zwei Beispiele

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


repeat :: Int -> String -> String
repeat n s =
    if n == 0 then ""
    else s ++ repeat (n-1) s
```

Berechnung durch Auswertung

Auswertung von Ausdrücken

- Funktionen werden durch Funktions**gleichungen** definiert
- Funktionsgleichungen werden auf **Parameter** angewendet
- Daraus resultierende Ausdrücke werden **ausgewertet**
- Nicht weiter reduzierbare Ausdrücke heißen **Werte**
- Werte werden durch Auswertung von Ausdrücken ermittelt
  - Vorgegebene Werte (Zahlen, Zeichen, ...)
  - Definierte Datentypen (Listen, stukturierte Datentypen, ...)

Berechnung als Auswertung von Ausdrücken

- **Beispiel:**

```
double :: Int -> Int
double n = 2*n


double 3
→  2 * 3
→  6

double (double (3+2))
→  double (double 5)
→  double (2 * 5)
→  double 10
→  2 * 10
→  20
```

Auswertung: Fakultät

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


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 0
→ 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
```

Auswertung: Repeat

```
repeat :: Int -> String -> String
repeat n s = if n == 0 then “” else s ++ repeat (n-1) s
```

```
repeat 2 “troet␣”
→ if 2 == 0 then “” else “troet␣” ++ repeat (2-1) “troet␣”
→ “troet␣” ++ repeat 1 “troet␣”
→ “troet␣” ++ if 1 == 0 then “” else “troet␣” ++ repeat (1-1) “troet␣”
→ “troet␣” ++ “troet␣” ++ repeat 0 “troet␣”
→ “troet␣” ++ “troet␣” ++ if n == 0 then “” else s ++ repeat (n-1) s
→ “troet␣” ++ “troet␣” ++ “”
→ “troet␣troet␣”
```

Typisierung in Haskell

- Typen unterscheiden Arten von Werten und Ausdrücken
- Ein **Typ** bezeichnet eine **Menge** von **Werten**
- zum Beispiel:
  - der Haskell-Typ `Bool` bezeichnet die Menge mit den Elementen `True` und `False`
  - der Typ `Bool -> Bool` ist die Menge aller Funktionen, die `Bool` Argumente auf `Bool` Werte abbilden (z.B. die Funktion `not`)
- Notation: `v :: T`
- “v ist Element der Menge T” oder “v ist vom Typ T”

Typinferenz

- Funktionen arbeiten auf definierten Typen
  - Übereinstimmung Funktion – Typen der Eingabewerte
  - Funktion liefert Ausgabewerte eines bestimmten Typs
- **Jeder** Haskell Ausdruck besitzt einen **Typ**
- **Vor** der Auswertung eines Ausdruck wird der Typ des Resultats durch **Typinferenz** bestimmt
- Typinferenz für Funktionen:

$$\frac{\texttt{f :: A -> B} \qquad \texttt{e :: A}}{\texttt{f e :: B}}$$

- Beispiele...

Typinferenz und Typfehler

- Typ von `not False` ?
  - `not :: Bool -> Bool` und `False :: Bool`, also:
    `not False :: Bool`
- Beispiel für Typfehler: `not 3` (`3` nicht Typ `Bool`)

```
Prelude> not 3


<interactive>:5:5: error:
    • No instance for (Num Bool) arising from the literal ‘3’
    • In the first argument of ‘not’, namely ‘3’
      In the expression: not 3
      In an equation for ‘it’: it = not 3
```

Typabfrage in GHCi

- `:type` bzw. `:t`

```
Prelude> :type not
not :: Bool -> Bool
Prelude> :type False
False :: Bool
Prelude> :type not False
not False :: Bool
```

Typisierung: warum?

- Definiert Bedingungen der Verwendung von Funktionen
  - wie viele Argumente
  - Typen der Argumente
- Definiert Typ des Ergebnisses (bei korrekter Anwendung)
- Typfehler werden **vor** der Ausführung erkannt (*type checking*) und treten **niemals** während der Ausführung auf (*type safety*)
- Auch potentiell auswertbare Ausdrücke erzeugen Typfehler, z.B.
  - `if True then 1 else False` (Resultate müssen vom selben Typ sein)
- Ermöglicht Typabstraktion: Typen können verwendet werden ohne zu Wissen, wie die Typen definiert sind

Signaturen in Haskell

- Jede Funktion in Haskell besitzt eine Signatur:

```
fac :: Int -> Int

repeat :: Int -> String -> String
```

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

Zusammenfassung

- Funktionen erzeugen Ausgabewert aus einer Menge von Parametern
  - Referenzielle Transparenz: Ausgabe nicht abhängig von Kontext des Aufrufs
- Berechnung erfolgt durch Auswertung von Ausdrücken
- Typisierung definiert Bedingungen der Verwendung von Funktionen
  - Typ beschreibt Menge von Werten
- Vor der Auswertung von Ausdrücken: Typinferenz
- Jede Funktion in Haskell besitzt eine Signatur

nächstes Mal...

- Datentypen in Haskell
  - Basisdatentypen, Listen, Tupel, Funktionen
  - Polymorphie und Typvariablen
  - Typklassen

- Definition von Funktionen
  - Fallunterscheidung
  - Funktionsdefinition durch *Pattern Matching*
  - Konstruktion von Listen