

Intermediate dabbling in Haskell

Jonas Betzendahl
Stefan Dresselhaus

October 11, 2014

Übersicht I

Organisatorisches

Was ist funktionale Programmierung?

Was sind besondere Merkmale von Haskell?

Tools & Ressourcen

Hello World

Wer sind wir?

Wer sind wir?

2 Studenten, die finden, dass zu wenig gutes Haskell gelehrt wird.

Wer sind wir?

2 Studenten, die finden, dass zu wenig gutes Haskell gelehrt wird.

Wir sind hier um das zu ändern.

Aufbau der Vorlesung

- ▶ 12 Vorlesungen

Aufbau der Vorlesung

- ▶ 12 Vorlesungen
- ▶ 6 praxisnahe Vorlesungen mit zugehöriger Übung

Aufbau der Vorlesung

- ▶ 12 Vorlesungen
- ▶ 6 praxisnahe Vorlesungen mit zugehöriger Übung
- ▶ 6 weiterführende Vorlesungen, die euch zeigen, wie tief der Hasenbau wirklich ist

Aufbau der Vorlesung

- ▶ 12 Vorlesungen
- ▶ 6 praxisnahe Vorlesungen mit zugehöriger Übung
- ▶ 6 weiterführende Vorlesungen, die euch zeigen, wie tief der Hasenbau wirklich ist
- ▶ Abgabe Einzelprojekt zur Erbringung der Studienleistung

Die Bearbeitung der Übungen ist freiwillig - aber im Hinblick auf das Projekt sehr empfehlenswert.

Ihr werdet **ALLE** Übungen im Projekt (in abgewandelter Form) praktisch brauchen.

Funktionale Programmierung, Definition

Funktionale Programmierung ist ein Programmierparadigma, bei dem Programme ausschließlich aus Funktionen bestehen. Dadurch werden die aus der imperativen Programmierung bekannten Nebenwirkungen vermieden. [...] Eine funktionale Programmiersprache ist eine Programmiersprache, die Sprachelemente zur Kombination und Transformation von Funktionen anbietet. Eine rein funktionale Programmiersprache ist eine Programmiersprache, die die Verwendung von Elementen ausschließt, die im Widerspruch zum funktionalen Programmierparadigma stehen.

aus Wikipedia: "Funktionale Programmierung"

Was ist eine (math.) Funktion?

Eine mathematische Funktion ist ein Funktion, die für jede definierte Eingabe eine fest definierte Ausgabe zurückliefert.

Was ist eine (math.) Funktion?

Eine mathematische Funktion ist ein Funktion, die für jede definierte Eingabe eine fest definierte Ausgabe zurückliefert.

Beispiele:

Was ist eine (math.) Funktion?

Eine mathematische Funktion ist ein Funktion, die für jede definierte Eingabe eine fest definierte Ausgabe zurückliefert.

Beispiele:

► $x \rightarrow x^2$

Was ist eine (math.) Funktion?

Eine mathematische Funktion ist ein Funktion, die für jede definierte Eingabe eine fest definierte Ausgabe zurückliefert.

Beispiele:

- ▶ $x \rightarrow x^2$
- ▶ $\text{fst } (a, b) \rightarrow a$

Was ist eine (math.) Funktion?

Eine mathematische Funktion ist ein Funktion, die für jede definierte Eingabe eine fest definierte Ausgabe zurückliefert.

Beispiele:

- ▶ $x \rightarrow x^2$
- ▶ $\text{fst } (a, b) \rightarrow a$
- ▶ $\text{head } (x : xs) \rightarrow x$

Was ist eine (math.) Funktion?

Eine mathematische Funktion ist ein Funktion, die für jede definierte Eingabe eine fest definierte Ausgabe zurückliefert.

Beispiele:

- ▶ $x \rightarrow x^2$
- ▶ $\text{fst } (a, b) \rightarrow a$
- ▶ $\text{head } (x : xs) \rightarrow x$

Beispiele für Nicht-Funktionen:

Was ist eine (math.) Funktion?

Eine mathematische Funktion ist ein Funktion, die für jede definierte Eingabe eine fest definierte Ausgabe zurückliefert.

Beispiele:

- ▶ $x \rightarrow x^2$
- ▶ $\text{fst } (a, b) \rightarrow a$
- ▶ $\text{head } (x : xs) \rightarrow x$

Beispiele für Nicht-Funktionen:

- ▶ Zufall. `random()` liefert immer andere Werte.

Was ist eine (math.) Funktion?

Eine mathematische Funktion ist ein Funktion, die für jede definierte Eingabe eine fest definierte Ausgabe zurückliefert.

Beispiele:

- ▶ $x \rightarrow x^2$
- ▶ $\text{fst } (a, b) \rightarrow a$
- ▶ $\text{head } (x : xs) \rightarrow x$

Beispiele für Nicht-Funktionen:

- ▶ Zufall. `random()` liefert immer andere Werte.
- ▶ Input. Die Usereingabe ist nicht immer identisch.

Was ist eine (math.) Funktion?

Eine mathematische Funktion ist ein Funktion, die für jede definierte Eingabe eine fest definierte Ausgabe zurückliefert.

Beispiele:

- ▶ $x \rightarrow x^2$
- ▶ $\text{fst } (a, b) \rightarrow a$
- ▶ $\text{head } (x : xs) \rightarrow x$

Beispiele für Nicht-Funktionen:

- ▶ Zufall. `random()` liefert immer andere Werte.
- ▶ Input. Die Usereingabe ist nicht immer identisch.
- ▶ Output. Wenn jede geschriebene Datei denselben Inhalt hätte wäre das Schreiben sinnlos.

Was ist eine Lambda-Funktion?

Eine Lambda-Funktion ist eine anonyme Funktion, die meist in-place definiert wird.

Was ist eine Lambda-Funktion?

Eine Lambda-Funktion ist eine anonyme Funktion, die meist in-place definiert wird.

Eine Lambda-Funktion wird in Haskell mit `\` eingeleitet.

Was ist eine Lambda-Funktion?

Eine Lambda-Funktion ist eine anonyme Funktion, die meist in-place definiert wird.

Eine Lambda-Funktion wird in Haskell mit `\` eingeleitet.

Beispiele:

Was ist eine Lambda-Funktion?

Eine Lambda-Funktion ist eine anonyme Funktion, die meist in-place definiert wird.

Eine Lambda-Funktion wird in Haskell mit `\` eingeleitet.

Beispiele:

```
1  \x -> x*x  
2  \a b -> a + b
```

Was sind Funktionen höherer Ordnung?

Funktionen höherer Ordnung sind Funktionen, die Funktionen als Parameter nehmen.

Was sind Funktionen höherer Ordnung?

Funktionen höherer Ordnung sind Funktionen, die Funktionen als Parameter nehmen.

Am einfachsten kann man sich das an z.B. Listen verdeutlichen.

Was sind Funktionen höherer Ordnung?

Funktionen höherer Ordnung sind Funktionen, die Funktionen als Parameter nehmen.

Am einfachsten kann man sich das an z.B. Listen verdeutlichen. Wenn wir eine Funktion auf die Liste anwenden wollen, dann wollen wir sie auf jedes Element anwenden.

Was sind Funktionen höherer Ordnung?

Funktionen höherer Ordnung sind Funktionen, die Funktionen als Parameter nehmen.

Am einfachsten kann man sich das an z.B. Listen verdeutlichen. Wenn wir eine Funktion auf die Liste anwenden wollen, dann wollen wir sie auf jedes Element anwenden.

```
1  map :: (a -> b) -> [a] -> [b]
2  map f []      = []
3  map f (x:xs) = f x : map f xs
```

Was sind Funktionen höherer Ordnung?

Funktionen höherer Ordnung sind Funktionen, die Funktionen als Parameter nehmen.

Am einfachsten kann man sich das an z.B. Listen verdeutlichen. Wenn wir eine Funktion auf die Liste anwenden wollen, dann wollen wir sie auf jedes Element anwenden.

```
1 map :: (a -> b) -> [a] -> [b]
2 map f []      = []
3 map f (x:xs) = f x : map f xs
```

`map` ist eine Funktion höherer Ordnung, weil sie als ersten Parameter eine Funktion nimmt, die dann auf die Liste angewendet wird.

Haskell ist zum groteil "pure"

Eine Funktion ist "pure", wenn sie sich wie eine mathematische Funktion verhlt. Viele Funktionen in Haskell sind "pure" und man kann "unpure" Funktionen an der Signatur erkennen.

Haskell ist zum groteil "pure"

Eine Funktion ist "pure", wenn sie sich wie eine mathematische Funktion verhlt. Viele Funktionen in Haskell sind "pure" und man kann "unpure" Funktionen an der Signatur erkennen.

- Es gibt keine Seiteneffekte mit puren Funktionen

Haskell ist zum groteil "pure"

Eine Funktion ist "pure", wenn sie sich wie eine mathematische Funktion verhlt. Viele Funktionen in Haskell sind "pure" und man kann "unpure" Funktionen an der Signatur erkennen.

- ▶ Es gibt keine Seiteneffekte mit puren Funktionen
- ▶ keine NullPointerException, keine Deadlocks, keine Crashes

Haskell ist zum groteil "pure"

Eine Funktion ist "pure", wenn sie sich wie eine mathematische Funktion verhlt. Viele Funktionen in Haskell sind "pure" und man kann "unpure" Funktionen an der Signatur erkennen.

- ▶ Es gibt keine Seiteneffekte mit puren Funktionen
- ▶ keine NullPointerException, keine Deadlocks, keine Crashes
- ▶ Ein purees Programm braucht nicht zu laufen, da es keine Ausgabe liefern kann.

Haskell ist zum groteil "pure"

Eine Funktion ist "pure", wenn sie sich wie eine mathematische Funktion verhlt. Viele Funktionen in Haskell sind "pure" und man kann "unpure" Funktionen an der Signatur erkennen.

- ▶ Es gibt keine Seiteneffekte mit puren Funktionen
- ▶ keine NullPointerException, keine Deadlocks, keine Crashes
- ▶ Ein pures Programm braucht nicht zu laufen, da es keine Ausgabe liefern kann.

Nchste Woche lernen wir, wie wir trotzdem programmieren knnen.

Starke und statische Typisierung

Haskell ist stark und statisch typisiert.

Starke und statische Typisierung

Haskell ist stark und statisch typisiert.

- ▶ Alles hat somit einen Typen (Int, String, Liste von Int, Maybe String, ...)

Starke und statische Typisierung

Haskell ist stark und statisch typisiert.

- ▶ Alles hat somit einen Typen (Int, String, Liste von Int, Maybe String, ...)
- ▶ Typen sind fest. Es gibt kein Autocasting: "08" \neq 8

Typen können selbst definiert werden

In Haskell haben selbst-definierte Datentypen dieselbe Stellung wie Builtin-Datentypen.

Typen können selbst definiert werden

In Haskell haben selbst-definierte Datentypen dieselbe Stellung wie Builtin-Datentypen.

Beispiel:

Typen können selbst definiert werden

In Haskell haben selbst-definierte Datentypen dieselbe Stellung wie Builtin-Datentypen.

Beispiel:

```
1  data MyList a = ListItem a (MyList a) | Nil
2      deriving (Show, Eq)
3  l = ListItem 5 (ListItem 4 (ListItem 3 Nil))
4  — analog zu [5,4,3]
5
6  mymap :: (a -> b) -> MyList a -> MyList b
7  mymap f Nil = Nil
8  mymap f (ListItem x xs) = ListItem (f x) (mymap f xs)
9
10 — mymap (\x -> x*x) l
11 — ListItem 25 (ListItem 16 (ListItem 9 Nil))
12 — analog zu [25,16,9]
```

Weitere Typdefinitionen

Für komplexere Datentypen gibt es auch noch die record-syntax.

Weitere Typdefinitionen

Für komplexere Datentypen gibt es auch noch die record-syntax.

Beispiel:

Weitere Typdefinitionen

Für komplexere Datentypen gibt es auch noch die record-syntax.

Beispiel:

```
1  data Person =  
2  { name  :: String  
3  , ort   :: String  
4  , alter :: Int  
5  , cool  :: Bool  
6  }
```

Weitere Typdefinitionen

Für komplexere Datentypen gibt es auch noch die record-syntax.

Beispiel:

```
1 data Person =  
2   { name  :: String  
3   , ort   :: String  
4   , alter :: Int  
5   , cool  :: Bool  
6   }
```

Mit automatisch definierten Accessor-Funktionen:

```
1 let p = Person {name="John", ort="Bielefeld", alter=42, cool=False}  
2 name p           — John  
3 let q = p {cool = True} — NEUE Person mit den geänderten Daten
```

Alle Funktionen sind curried

JEDE Funktion in Haskell nimmt genau 1 Argument.

Alle Funktionen sind curried

JEDE Funktion in Haskell nimmt genau 1 Argument.

- ▶ mehrere Funktionsargumente werden simuliert, indem eine Funktion mit n Argumenten eine Funktion mit $n-1$ Argumenten zurückliefert.

Alle Funktionen sind curried

JEDE Funktion in Haskell nimmt genau 1 Argument.

- ▶ mehrere Funktionsargumente werden simuliert, indem eine Funktion mit n Argumenten eine Funktion mit $n-1$ Argumenten zurückliefert.
- ▶ Funktionen können teil-appliziert werden

Alle Funktionen sind curried

JEDE Funktion in Haskell nimmt genau 1 Argument.

- ▶ mehrere Funktionsargumente werden simuliert, indem eine Funktion mit n Argumenten eine Funktion mit $n-1$ Argumenten zurückliefert.
- ▶ Funktionen können teil-appliziert werden

Beispiele:

Alle Funktionen sind curried

JEDE Funktion in Haskell nimmt genau 1 Argument.

- ▶ mehrere Funktionsargumente werden simuliert, indem eine Funktion mit n Argumenten eine Funktion mit $n-1$ Argumenten zurückliefert.
- ▶ Funktionen können teil-appliziert werden

Beispiele:

```
1  let inc = (+) 1
2  inc 5      —      6
3  inc 41     —      42
4  map inc [1,2,3] — [2,3,4]
5  let add = \x -> (\y -> x + y)
```


Types you do not type

Typannotationen sind in Haskell komplett freiwillig (abgesehen ambiguity)

Types you do not type

Typannotationen sind in Haskell komplett freiwillig (abgesehen ambiguity)

Was für ein Typ hat

```
1 let greet = (++) "Hello"
```

Types you do not type

Typannotationen sind in Haskell komplett freiwillig (abgesehen ambiguity)

Was für ein Typ hat

```
1 let greet = (++) "Hello"
```

Typ: `String -> String`

Types you do not type

Typannotationen sind in Haskell komplett freiwillig (abgesehen ambiguity)

Was für ein Typ hat

```
1 let greet = (++) "Hello"
```

Typ: `String -> String`

```
1 map id
```

Types you do not type

Typannotationen sind in Haskell komplett freiwillig (abgesehen ambiguity)

Was für ein Typ hat

```
1 let greet = (++) "Hello"
```

Typ: `String -> String`

```
1 map id
```

Typ: `[a] -> [a]`

Types you do not type

Typannotationen sind in Haskell komplett freiwillig (abgesehen ambiguity)

Was für ein Typ hat

```
1 let greet = (++) "Hello"
```

Typ: `String -> String`

```
1 map id
```

Typ: `[a] -> [a]`

```
1 abs
```

Types you do not type

Typannotationen sind in Haskell komplett freiwillig (abgesehen ambiguity)

Was für ein Typ hat

```
1 let greet = (++) "Hello"
```

Typ: `String -> String`

```
1 map id
```

Typ: `[a] -> [a]`

```
1 abs
```

Typ: `Num a => a -> a`

Types you do not type

Typannotationen sind in Haskell komplett freiwillig (abgesehen ambiguity)

Was für ein Typ hat

```
1 let greet = (++) "Hello"
```

Typ: `String -> String`

```
1 map id
```

Typ: `[a] -> [a]`

```
1 abs
```

Typ: `Num a => a -> a`

Geht die Annotation immer an. Hilft enorm.

Die wichtigsten Tools und Ressourcen

Die wichtigsten Tools und Ressourcen

- ▶ cabal
Paketverwaltungstool von Haskell.

Die wichtigsten Tools und Ressourcen

- ▶ cabal
Paketverwaltungstool von Haskell.
- ▶ Hackage
Paketdatenbank und Online-Doku.

Die wichtigsten Tools und Ressourcen

- ▶ cabal
Paketverwaltungstool von Haskell.
- ▶ Hackage
Paketdatenbank und Online-Doku.
- ▶ Haddock
Dokumentation ähnlich wie JavaDoc - nur besser.

Die wichtigsten Tools und Ressourcen

- ▶ cabal
Paketverwaltungstool von Haskell.
- ▶ Hackage
Paketdatenbank und Online-Doku.
- ▶ Haddock
Dokumentation ähnlich wie JavaDoc - nur besser.
- ▶ ghci
Interpreter des GHC. Hierin macht ihr die ersten Übungen.

Die wichtigsten Tools und Ressourcen

- ▶ cabal
Paketverwaltungstool von Haskell.
- ▶ Hackage
Paketdatenbank und Online-Doku.
- ▶ Haddock
Dokumentation ähnlich wie JavaDoc - nur besser.
- ▶ ghci
Interpreter des GHC. Hierin macht ihr die ersten Übungen.
- ▶ Hoogle/Hayoo
Suchmaschine für Pakete/Funktionen. Sucht auch nach Signaturen.

Einrichtungshilfe

Ihr braucht:

- ▶ einen GHC (7.6, 7.8)
- ▶ cabal (1.18+)
- ▶ einen Texteditor, mit dem ihr umgehen könnt (z.B. vim)

Wir sind im ersten Tutorium gerne bei der Einrichtung behilflich.
Vorraussetzung ist ein laufendes Linux.

Übungen

- ▶ Implementation eines eigenen Datentyps "Tree", der einen Baum darstellt

Übungen

- ▶ Implementation eines eigenen Datentyps "Tree", der einen Baum darstellt
- ▶ eine "map"-Funktion für den Baum

Übungen

- ▶ Implementation eines eigenen Datentyps "Tree", der einen Baum darstellt
- ▶ eine "map"-Funktion für den Baum
- ▶ "toList" und "fromList"-Funktionen für den Baum

Übungen

- ▶ Implementation eines eigenen Datentyps "Tree", der einen Baum darstellt
- ▶ eine "map"-Funktion für den Baum
- ▶ "toList" und "fromList"-Funktionen für den Baum
- ▶ "fold" für den Baum

Übungen

- ▶ Implementation eines eigenen Datentyps "Tree", der einen Baum darstellt
- ▶ eine "map"-Funktion für den Baum
- ▶ "toList" und "fromList"-Funktionen für den Baum
- ▶ "fold" für den Baum
- ▶ genaueres auf dem Übungszettel