

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

Inhalt und Lernziele

- ▶ Definition von **Funktionen**
 - ▶ Syntaktische Feinheiten
- ▶ Bedeutung von Haskell-Programmen
 - ▶ Striktheit
- ▶ Leben ohne Variablen
 - ▶ Funktionen statt Schleifen
 - ▶ Zahllose Beispiele

Lernziele

Wir wollen einfache Haskell-Programme schreiben können, eine Idee von ihrer Bedeutung bekommen, und ein Leben ohne veränderliche Variablen führen.

I. Definition von Funktionen

Definition von Funktionen

- ▶ Zwei wesentliche Konstrukte:
 - ▶ Fallunterscheidung
 - ▶ Rekursion

Satz

Fallunterscheidung und Rekursion auf natürlichen Zahlen sind **Turing-mächtig**.

- ▶ Funktionen müssen **partiell** sein können.
 - ▶ Insbesondere nicht-terminierende Rekursion
- ▶ Fragen: wie schreiben Funktionen in Haskell auf (**Syntax**), und was bedeutet das (**Semantik**)?

Haskell-Syntax: Charakteristika

- ▶ **Leichtgewichtig**
 - ▶ Wichtigstes Zeichen:
- ▶ Funktionsapplikation: $f\ a$
 - ▶ Klammern sind **optional**
 - ▶ **Höchste** Priorität (engste Bindung)
- ▶ Abseitsregel: Gültigkeitsbereich durch Einrückung
 - ▶ Keine Klammern ($\{ \dots \}$) (optional)
 - ▶ Auch in anderen Sprachen (Python, Ruby)

Haskell-Syntax: Funktionsdefinition

Generelle Form:

- ▶ **Signatur:**

```
max :: Int -> Int -> Int
```

- ▶ **Definition:**

```
max x y = if x < y then y else x
```

- ▶ Kopf, mit Parametern
- ▶ Rumpf (evtl. länger, mehrere Zeilen)
- ▶ Typisches **Muster**: Fallunterscheidung, dann rekursiver Aufruf
- ▶ Was gehört zum Rumpf (**Geltungsbereich**)?

Haskell-Syntax I: Die Abseitsregel

Funktionsdefinition:

```
f x1 x2 x3...xn = e
```

- ▶ **Gültigkeitsbereich** der Definition von f :
alles, was gegenüber f eingerückt ist.

- ▶ Beispiel:

```
f x = hier faengts an  
    und hier gehts weiter  
    immer weiter  
g y z = und hier faengt was neues an
```

- ▶ Gilt auch **verschachtelt**.
- ▶ Kommentare sind **passiv** (heben das Abseits nicht auf).

Haskell-Syntax II: Kommentare

- Pro Zeile: Ab `--` bis Ende der Zeile

```
f x y = irgendwas -- und hier der Kommentar!
```

- Über mehrere Zeilen: Anfang `{-`, Ende `-}`

```
{-
  Hier faengt der Kommentar an
  erstreckt sich ueber mehrere Zeilen
  bis hier -}
f x y = irgendwas
```

- Kann geschachtelt werden.

Haskell-Syntax III: Bedingte Definitionen

- Statt verschachtelter Fallunterscheidungen ...

```
f x y = if B1 then P else
        if B2 then Q else R
```

... **bedingte Gleichungen**:

```
f x y
| B1 = P
| B2 = Q
```

- Auswertung der Bedingungen von oben nach unten
- Wenn keine Bedingung wahr ist: **Laufzeitfehler!** Deshalb:

```
| otherwise = R
```

Haskell-Syntax IV: Lokale Definitionen

- Lokale Definitionen mit `where` oder `let`:

```
f x y
| g = P y
| otherwise = f x where
  y = M
  f x = N x
```

```
f x y =
  let y = M
      f x = N x
  in if g then P y
     else f x
```

- `f`, `y`, ... werden **gleichzeitig** definiert (Rekursion!)
- Namen `f`, `y` und Parameter (`x`) **überlagern** andere
- Es gilt die **Abwärtsregel**
 - Deshalb: Auf gleiche Einrückung der lokalen Definition achten!

Jetzt seid ihr dran!

Übung 2.1: Syntax

In dem Beispielprogramm auf der vorherigen Folie, welche der Variablen `f`, `x` und `y` auf den rechten Seiten wird wo gebunden?

Lösung:

```
f x y
| g = P y
| otherwise = f x where
  y = M
  f x = N x
```

II. Auswertung von Funktionen

Auswertung von Funktionen

- Auswertung durch **Anwendung** von Gleichungen
- Auswertungsrelation** $s \rightarrow t$:
 - Anwendung einer Funktionsdefinition
 - Anwendung von elementaren Operationen (arithmetisch, Zeichenketten)
- Frage: spielt die **Reihenfolge** eine Rolle?

Auswertung von Ausdrücken

```
inc :: Int -> Int      dbl :: Int -> Int
inc x = x + 1         dbl x = 2 * x
```

- Reduktion von `inc (dbl (inc 3))`
- Von **außen** nach **innen** (outermost-first):

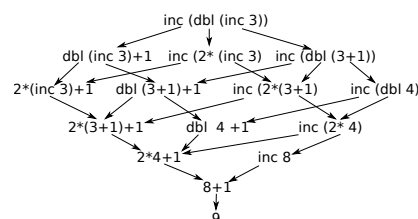

```
inc (dbl (inc 3)) -> dbl (inc 3) + 1
                  -> 2 * (inc 3) + 1
                  -> 2 * (3 + 1) + 1 -> 2 * 4 + 1 -> 8 + 1 -> 9
```
- Von **innen** nach **außen** (innermost-first):


```
inc (dbl (inc 3)) -> inc (dbl (3 + 1)) -> inc (dbl 4)
                  -> inc (2 * 4) -> inc 8
                  -> 8 + 1 -> 9
```

Auswertung von Ausdrücken

```
inc :: Int -> Int      dbl :: Int -> Int
inc x = x + 1         dbl x = 2 * x
```

- Volle Reduktion von `inc (dbl (inc 3))`:



Konfluenz

- ▶ Es kommt immer das gleiche heraus?
- ▶ Sei \rightarrow^* die Reduktion in null oder mehr Schritten.

Definition (Konfluenz)

\rightarrow^* ist **konfluent** gdw:
Für alle r, s, t mit $s \xrightarrow{*} r \xrightarrow{*} t$ gibt es u so dass $s \xrightarrow{*} u \xrightarrow{*} t$.

Konfluenz

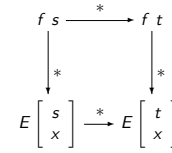
- ▶ Wenn wir von Laufzeitfehlern abstrahieren, gilt:

Theorem (Konfluenz)

Die Auswertungsrelation \rightarrow^* für funktionale Programme ist **konfluent**.

- ▶ Beweisskizze:

Sei $f \ x = E$ und $s \xrightarrow{*} t$:



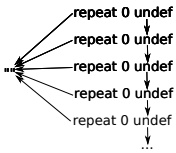
Auswirkung der Auswertungsstrategie

- ▶ Auswertungsstrategie ist also egal?
- ▶ Beispiel:

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

undef :: String
undef = undef
```

- ▶ Auswertung von `repeat 0 undef`:



- ▶ outermost-first **terminiert**
- ▶ innermost-first terminiert **nicht**

Termination und Normalform

Definition (Termination)

\rightarrow ist **terminierend** gdw. es **keine unendlichen** Ketten gibt:
 $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \dots \rightarrow t_n \rightarrow \dots$

Theorem (Normalform)

Sei \rightarrow konfluent und terminierend, dann wertet jeder Term zu genau einer **Normalform** aus, die nicht weiter ausgewertet werden kann.

- ▶ Daraus folgt: **terminierende** funktionale Programme werten unter jeder Auswertungsstrategie jeden Ausdruck zum gleichen Wert aus (der Normalform).

Auswirkung der Auswertungsstrategie

- ▶ Auswertungsstrategie nur für **nicht-terminierende** Programme relevant.
- ▶ Leider ist nicht-Termination **nötig** (Turing-Mächtigkeit)
- ▶ Gibt es eine **semantische** Charakterisierung?
- ▶ Auswertungsstrategie und Parameterübergabe:
 - ▶ Outermost-first entspricht **call-by-need**, verzögerte Auswertung.
 - ▶ Innermost-first entspricht **call-by-value**, strikte Auswertung

Zum Mitdenken...

Übung 2.2:

Warum entspricht outermost-first call-by-need und innermost-first call-by-value?

Lösung: Der Aufruf einer Funktion $f \ x = E$ entspricht hier der Ersetzung der linken Seite f durch die rechte Seite E , mit den Parametern x entsprechend ersetzt.

Wenn wir beispielsweise Auswertung des Ausdrucks `dbl (dbl (dbl (7+3)))` betrachten, dann wird innermost-first zuerst `7+3` reduziert, dann `dbl 10` etc., d.h. jeweils die **Argumente** der Funktion — Funktionen bekommen nur Werte übergeben.

Bei outermost-first wird zuerst das äußerste `dbl` reduziert, was dem Aufruf der Funktion `dbl` mit dem nicht ausgewerteten Argument `dbl (dbl (7+3))` entspricht (verzögerte Auswertung).

III. Semantik und Striktheit

Bedeutung (Semantik) von Programmen

- ▶ **Operationale** Semantik:
 - ▶ Durch den **Ausführungsbegriff**
 - ▶ Ein Programm **ist**, was es **tut**.
 - ▶ In diesem Fall: \rightarrow
- ▶ **Denotationelle** Semantik:
 - ▶ Programme werden auf **mathematische Objekte** abgebildet (Denotat).
 - ▶ Für funktionale Programme: **rekursiv** definierte Funktionen

Äquivalenz von operationaler und denotationaler Semantik

Sei P ein funktionales Programm, \rightarrow die dadurch definierte Reduktion, und $\llbracket P \rrbracket$ das Denotat. Dann gilt für alle Ausdrücke t und Werte v

$$t \rightarrow v \iff \llbracket P \rrbracket(t) = v$$

Striktheit

Definition (Striktheit)

Funktion f ist **strikt** \iff Ergebnis ist undefiniert, sobald ein Argument undefiniert ist.

- ▶ **Denotationelle** Eigenschaft (nicht operational)
- ▶ Haskell ist nach **Sprachdefinition nicht-strikt**
 - ▶ `repeat 0 undef muss ""` ergeben.
 - ▶ Meisten Implementationen nutzen verzögerte Auswertung
- ▶ Andere Programmiersprachen:
 - ▶ Java, C, etc. sind **call-by-value** (nach Sprachdefinition) und damit strikt
 - ▶ Fallunterscheidung ist **immer** nicht-strikt, Konjunktion und Disjunktion meist auch.

Jetzt seit ihr dran!

Übung 2.3: Strikte Fallunterscheidung

Warum ist Fallunterscheidung immer nicht-strikt, auch in Java?

Lösung: Betrachte

```
y = x == 0 ? -1 : 100/x;      if (x == 0) {
                             y = -1;
                             } else {
                             y = 100/x;
                             }
```

Wäre die Fallunterscheidung strikt, würden erst **beide** Fälle ausgewertet; es wäre nicht mehr möglich, die Auswertung undefinierter Ausdrücke abzufangen. Das gleich gilt für das Programm rechts.

IV. Leben ohne Variablen

Rekursion statt Schleifen

Fakultät imperativ:

```
r = 1;
while (n > 0) {
  r = n * r;
  n = n - 1;
}
```

Fakultät rekursiv:

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

- ▶ Veränderliche Variablen werden zu Funktionsparametern
- ▶ Iteration (while-Schleifen) werden zu Rekursion
- ▶ Endrekursion verbraucht keinen Speicherplatz

Rekursive Funktionen auf Zeichenketten

- ▶ Test auf die leere Zeichenkette:

```
null :: String → Bool
null xs = xs == ""
```

- ▶ Kopf und Rest einer nicht-leeren Zeichenkette (vordefiniert):

```
head :: String → Char
tail :: String → String
```



Suche in einer Zeichenkette

- ▶ Suche nach einem Zeichen in einer Zeichenkette:

```
count1 :: Char → String → Int
```

- ▶ In einem leeren String: kein Zeichen kommt vor

- ▶ Ansonsten: Kopf vergleichen, zum Vorkommen im Rest addieren

```
count1 c s =
  if null s then 0
  else if head s == c then 1 + count1 c (tail s)
  else count1 c (tail s)
```

- ▶ Übung: wie formuliere ich `count` mit Guards? (Lösung in den Quellen)

Suche in einer Zeichenkette

- ▶ Endrekursiv:

```
count3 c s = count3' c s 0
count3' c s r =
  if null s then r
  else count3' c (tail s) (if head s == c then 1+r else r)
```

- ▶ Endrekursiv mit lokaler Definition

```
count4 c s = count4' s 0 where
  count4' s r =
    if null s then r
    else count4' (tail s) (if head s == c then 1+r else r)
```



Strings konstruieren

- ▶ `:` hängt Zeichen vorne an Zeichenkette an (vordefiniert)

```
(:) :: Char → String → String
```

- ▶ Es gilt: Wenn `not (null s)`, dann `head s : tail s == s`

- ▶ Mit `:` wird `(++)` definiert:

```
(++) :: String → String → String
xs ++ ys
  | null xs = ys
  | otherwise = head xs : (tail xs ++ ys)
```

- ▶ `quadrat` konstruiert ein Quadrat aus Zeichen:

```
quadrat :: Int → Char → String
quadrat n c = repeat n (repeat n c) ++ "\n"
```



Strings analysieren

- ▶ Warum immer nur Kopf/Rest?
- ▶ Letztes Zeichen (dual zu head):

```
last1 :: String -> Char
last1 s = if null s then last1 s
         else if null (tail s) then head s
         else last1 (tail s)
```

- ▶ Besser: mit Fehlermeldung

```
last :: String -> Char
last s
| null s = error "last: empty string"
| null (tail s) = head s
| otherwise = last (tail s)
```



Strings analysieren

- ▶ Anfang der Zeichenkette (dual zu tail):

```
init :: String -> String
init s
| null s = error "init: empty string" -- nicht s
| null (tail s) = ""
| otherwise = head s : init (tail s)
```

- ▶ Damit: Wenn not (null s), dann `init s + (last s : "") == s`



Strings analysieren: das Palindrom

- ▶ Palindrom: vorwärts und rückwärts gelesen gleich.
- ▶ Rekursiv:
 - ▶ Alle Wörter der Länge 1 oder kleiner sind Palindrome
 - ▶ Für längere Wörter: wenn erstes und letztes Zeichen gleich sind und der Rest ein Palindrom.

- ▶ Erste Variante:

```
palin1 :: String -> Bool
palin1 s
| length s <= 1 = True
| head s == last s = palin1 (init (tail s))
| otherwise = False
```



Strings analysieren: das Palindrom

- ▶ Problem: Groß/Kleinschreibung, Leerzeichen, Satzzeichen irrelevant.
- ▶ Daher: nicht-alphanumerische Zeichen entfernen, alles Kleinschrift:

```
clean :: String -> String
clean s
| null s = ""
| isAlphaNum (head s) = toLower (head s) : clean (tail s)
| otherwise = clean (tail s)
```

- ▶ Erweiterte Version:

```
palin2 s = palin1 (clean s)
```



Fortgeschritten: Vereinfachung von palin1

- ▶ Das hier ist nicht so schön:

```
palin1 s
| length s <= 1 = True
| head s == last s = palin1 (init (tail s))
| otherwise = False
```

- ▶ Was steht da eigentlich:

```
palin1' s = if length s <= 1 then True
           else if head s == last s then palin1' (init (tail s))
           else False
```

- ▶ Damit:

```
palin3 s = length s <= 1 || head s == last s && palin3 (init (tail s))
```

- ▶ Terminiert nur wegen Nicht-Striktheit von ||



Zusammenfassung

- ▶ **Bedeutung** von Haskell-Programmen:

- ▶ Auswertungsrelation \rightarrow
- ▶ Auswertungsstrategien: innermost-first, outermost-first
- ▶ Auswertungsstrategie für terminierende Programme irrelevant

- ▶ **Striktheit**

- ▶ Haskell ist **spezifiziert** als nicht-strikt
- ▶ Meist implementiert durch verzögerte Auswertung

- ▶ Leben **ohne Variablen**:

- ▶ Rekursion statt Schleifen
- ▶ Funktionsparameter statt Variablen

- ▶ Nächste Vorlesung: Datentypen

