

# Praktische Informatik 3: Funktionale Programmierung

## Vorlesung 2 vom 23.10.2016: Funktionen

Christoph Lüth

Universität Bremen

Wintersemester 2018/19



## Fahrplan

### Teil I: Funktionale Programmierung im Kleinen

- ▶ Einführung
- ▶ **Funktionen**
- ▶ Algebraische Datentypen
- ▶ Typvariablen und Polymorphie
- ▶ Zyklische Datenstrukturen
- ▶ Funktionen höherer Ordnung I
- ▶ Funktionen höherer Ordnung II

### Teil II: Funktionale Programmierung im Großen

### Teil III: Funktionale Programmierung im richtigen Leben



## Inhalt

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



## Organisatorisches

### Verteilung der Tutorien (laut stud.ip):

					Abweichung
Mi	08–10	MZH 1470	Thomas Barkowsky	16	-18
	10–12	MZH 1090	Tobias Haslop	50	<b>16</b>
	12–14	MZH 1470	Matz Habermann	49	<b>15</b>
	16–18	MZH 1090	Andreas Kästner	18	-16
Do	12–14	MZH 1090	Gerrit Marquardt	50	<b>15</b>
	16–18	MZH 1110	Gerrit Marquardt	23	-11

- ▶ Wenn möglich, frühe/späte Tutorien belegen.

### Bewertung der Übungsblätter:

- ▶ Dokumentation: kurz und knapp für jede Funktion
- ▶ Code: auf guten Stil achten
- ▶ Tests und Testfälle



## Definition von Funktionen



## Definition von Funktionen

### Zwei wesentliche Konstrukte:

- ▶ Fallunterscheidung
- ▶ Rekursion

### Reicht das?

#### Satz

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

- ▶ Funktionen müssen **partiell** sein können.
- ▶ Insbesondere nicht-terminierende Rekursion



## Haskell-Syntax: Charakteristika

- ▶ **Leichtgewichtig**
  - ▶ Wichtigstes Zeichen:
- ▶ Funktionsapplikation:  $f\ a$ 
  - ▶ Klammern sind optional
  - ▶ **Höchste** Priorität (engste Bindung)
- ▶ Abseitsregel: Gültigkeitsbereich durch Einrückung
  - ▶ Keine Klammern (`{ ... }`)
- ▶ 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...
```

... **bedingte Gleichungen**:

```
f x y
| B1 =...
| B2 =...
```

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

```
| otherwise =...
```



## 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 **Abseitsregel**
  - ▶ Deshalb: Auf **gleiche** Einrückung der lokalen Definition achten!



## Bedeutung von Programmen



## Auswertung von Funktionen

- ▶ Auswertung durch **Anwendung** von Gleichungen
- ▶ **Auswertungsrelation**  $s \rightarrow t$ :
  - ▶ Anwendung einer Funktionsdefinition
  - ▶ Anwendung von elementaren Operationen (arithmetisch, Zeichenketten)



## Auswertung von Ausdrücken

```
inc :: Int → Int
inc x = x+1
```

```
dbl :: Int → Int
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 → 9
```
- ▶ Von **innen** nach **außen** (innermost-first):
 

```
inc (dbl (inc 3)) → inc (dbl (3+1))
                  → inc (2*(3+1))
                  → (2*(3+1)) + 1
                  → 2*4+1 → 9
```

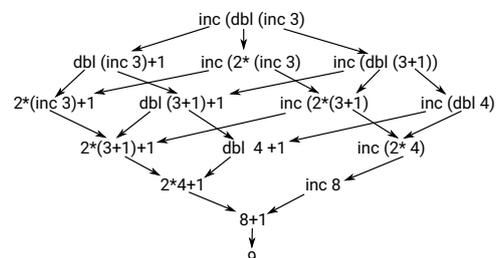


## Auswertung von Ausdrücken

```
inc :: Int → Int
inc x = x+1
```

```
dbl :: Int → Int
dbl x = 2*x
```

- ▶ Reduktion von `inc (dbl (inc 3))`



## Konfluenz

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

### Definition (Konfluenz)

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



## Konfluenz

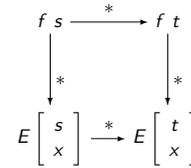
- ▶ Wenn wir von Laufzeitfehlern abstrahieren, gilt:

### Theorem (Konfluenz)

Die Auswertungsrelation  $\overset{*}{\rightarrow}$  für funktionale Programme ist **konfluent**.

- ▶ Beweisskizze:

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



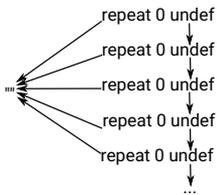
## Auswirkung der Auswertungsstrategie

- ▶ 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:  
 $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \dots t_n \rightarrow \dots$

### Theorem (Normalform)

Sei  $\overset{*}{\rightarrow}$  **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)
- ▶ Auswertungsstrategie und Parameterübergabe:
  - ▶ Outermost-first entspricht **call-by-need**, **verzögerte** Auswertung.
  - ▶ Innermost-first entspricht **call-by-value**, **strikte** Auswertung
- ▶ Gibt es eine **semantische** Charakterisierung?



## 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,  $\overset{*}{\rightarrow}$  die dadurch definierte Reduktion, und  $\llbracket P \rrbracket$  das Denotat. Dann gilt für alle Ausdrücke  $t$  und Werte  $v$   
 $t \overset{*}{\rightarrow} v \iff \llbracket P \rrbracket(t) = v$



## Striktheit

### Definition (Striktheit)

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

- ▶ **Denotationelle** Eigenschaft (nicht operational)
- ▶ Haskell ist **nicht-strikt** (nach Sprachdefinition)
  - ▶ `repeat0 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.



# Leben ohne Variablen



## Rekursion statt Schleifen

Fakultät imperativ:

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

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

Fakultät rekursiv:

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



## 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)
```



## Suche in einer Zeichenkette

- ▶ Etwas lesbarer mit Guards:

```
count2 c s
  | null s = 0
  | head s == c = 1 + count2 c (tail s)
  | otherwise = count2 c (tail s)
```

- ▶ 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  $\text{not } (\text{null } s)$ , dann  $\text{head } s : \text{tail } s == s$
- ▶ 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  $\text{not } (\text{null } s)$ , dann  $\text{init } s \# (\text{last } s : "") == s$



## 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.

- ▶ Erster Versuch:

```
palin1 :: String → Bool
palin1 s
  | length s ≤ 1 = True
  | head s == last s = palin1 (init (tail s))
  | otherwise = False
```



## Strings analysieren: das Palindrom

### ▶ Zweiter Versuch:

```
palin2 :: String → Bool
palin2 s =
  length s ≤ 1 || head s == last s && palin2 (init (tail s))
```

- ▶ Terminiert wegen Nicht-Striktheit von `||`

### ▶ Erweiterte Version:

```
palin3 s = palin2 (clean s)
```

- ▶ 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)
```



## Zusammenfassung

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

- ▶ Auswertungsrelation →
- ▶ Auswertungsstrategien: innermost-first, outermost-firsta
- ▶ 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

