

# Tutorium 04: $\lambda$ -Kalkül

---

Paul Brinkmeier

1. Dezember 2020

Tutorium Programmierparadigmen am KIT

# Heutiges Programm

---

- Übungsblatt 2
- Lambda-Kalkül
- ... und Implementierung in Haskell

# Übungsblatt 2

---

## 1 — Listenkombinatoren

```
module Polynom where

type Polynom = [Double]

cmult polynom c = map (* c) polynom

eval polynom x = foldr go 0 polynom
  where go a_n acc = acc * x + a_n

deriv [] = []
deriv polynom = zipWith (*) [1..] $ tail polynom
```

## 2 — Collatz-Vermutung

```
module Collatz where

collatz = iterate next
  where next aN | even aN    = aN `div` 2
                | otherwise = 3 * aN + 1

num = length . takeWhile (/= 1) . collatz

maxNum a b = bestNum [(m, num m) | m <- [a..b]]
  where bestNum = foldl maxSecond (0, 0)
        maxSecond a b
          | snd a >= snd b = a
          | otherwise      = b
```

## 2 — Collatz-Vermutung

```
module CollatzAlt where

import Collatz (num)
import Data.Function (on)
import Data.List (maximumBy)

maxNum a b =
  maximumBy
    (compare 'on' snd)
    [(m, num m) | m <- [a..b]]
```

- „eleganter“
- In der Klausur aber eher nur Funktionen aus der Prelude verwenden

### 3 — Stream-Kombinatoren

```
module Merge where

import Primes (primes)

merge (x:xs) (y:ys)
  | x <= y      = x : merge xs (y:ys)
  | otherwise   = y : merge (x:xs) ys
merge xs ys = xs ++ ys

primepowers n = mergeAll $ map primesExp [1..n]
  where mergeAll = foldl merge []
        primesExp i = map (^i) primes
```

- Für  $i$  in  $1..n$  unendliche Liste der Primzahlen hoch  $i$  erstellen
- Wegen Laziness: wird nur so weit ausgewertet wie nötig
- Dann: Alle miteinander vereinigen



```
*Merge> pp3 = primepowers 3
*Merge> take 10 pp3
[2,3,4,5,7,8,9,11,13,17]
*Merge> :sprint pp3
pp3 = 2 : 3 : 4 : 5 : 7 : 8 : 9 : 11 : 13 : 17 : _
```

- :sprint a gibt aktuelle Speicherrepräsentation für a aus
- \_ steht dabei für „noch nicht ausgewertet“
- $\rightsquigarrow$  praktisch für Debugging unendlicher Listen

# Wiederholung: Algebraische Datentypen

---

# Cheatsheet: Algebraische Datentypen in Haskell

- data-Definitionen, Datenkonstruktoren
- Algebraische Datentypen: Produkttypen und Summentypen
  - Produkttypen  $\approx$  structs in C
  - Summentypen  $\approx$  enums
- Typkonstruktoren, bspw. `[] :: * -> *`
- Polymorphe Datentypen, bspw. `[a]`, `Maybe a`
- Beispiel:

```
module Shape where

data Shape
  = Circle Double -- radius
  | Rectangle Double Double -- sides
  | Point -- technically equivalent to Circle 0
```

# Cheatsheet: Typklassen 1

- Klasse, Operationen/Methoden, Instanzen
- Beispiele:
  - `Eq t, {(==), (/=)}`, `{Eq Bool, Eq Int, Eq Char, ...}`
  - `Show t, {show}`, `{Show Bool, Show Int, Show Char, ...}`
- Weitere Typklassen: `Ord`, `Num`, `Enum`
- Deklaration/Implementierung:

```
module Truthy where

class Truthy t where
  toBool :: t -> Bool

instance Truthy Int where
  toBool x = x /= 0
```

## Cheatsheet: Typklassen 2

- Vererbung: Typklassen mit Voraussetzungen

```
module Truthy2 where

class Truthy t where
  toBool :: t -> Bool

instance Truthy Int where
  toBool x = x /= 0

instance Truthy t => Truthy (Maybe t) where
  toBool Nothing  = False
  toBool (Just x) = toBool x
```

```
module PlayingCard where

data PlayingCard = PlayingCard Suit Rank

data Suit = Hearts | Diamonds | Clubs | Spades
data Rank
  = Rank7 | Rank8 | Rank9 | Rank10
  | Jack | Queen | King | Ace
```

# Monopolykarten

```
module Monopoly where

data MonopolyCard
  = Street String Rent Int Color
  | Station String
  | Utility String

data Rent = Rent Int Int Int Int Int Int

data Color
  = Brown | LightBlue | Pink | Orange
  | Red | Yellow | Green | Blue
```

# Boolesche Logik

```
module BoolExpr where

data BoolExpr
  = Const Bool
  | Var String
  | Neg BoolExpr
  | BinaryOp BoolExpr BinaryOp BoolExpr

data BinaryOp = AND | OR | XOR | NOR
```

Beispiele:

- $a \wedge b$  entspricht `BinaryOp (Var "a") AND (Var "b")`
- $a \vee (b \wedge 0)$  entspricht  
`BinaryOp (Var "a") AND (BinaryOp (Var "b") OR (Const False))`



# $\lambda$ -Kalkül

---

- „Funktionales Gegenstück zur Turingmaschine“
- Wurde u.a. genutzt um Unlösbarkeit des Halteproblems zu zeigen
- Gibt saftig Punkte in der Klausur
  - 13P. im 19SS
  - 10P. (+15P.) im 18WS
  - 20P. (+15P.) im 18SS

Ein Term im  $\lambda$ -Kalkül hat eine der drei folgenden Formen:

Notation	Besteht aus	Bezeichnung
$x$	$x$ : Variablenname	Variable
$\lambda p.b$	$p$ : Variablenname $b$ : $\lambda$ -Term	Abstraktion
$f a$	$f, a$ : $\lambda$ -Terme	Funktionsanwendung

- „ $\lambda$ -Term“: rekursive Datenstruktur
- Semantik definieren wir später

Ein Term im  $\lambda$ -Kalkül hat eine der drei folgenden Formen:

Notation	Besteht aus	Bezeichnung
$x$	$x$ : Variablenname	Variable
$\lambda p.b$	$p$ : Variablenname $b$ : $\lambda$ -Term	Abstraktion
$f\ a$	$f, a$ : $\lambda$ -Terme	Funktionsanwendung

- „ $\lambda$ -Term“: rekursive Datenstruktur
- Semantik definieren wir später
- Jetzt: Ergänzt das Modul Lambda um die fehlenden Typen
  - +Fragen zur ÜB-Korrektur

```
module Lambda where

data LambdaTerm
  = Var String    -- Variable
  | App () ()     -- Funktionsanwendung: f a
  | Abs () ()     -- Abstraktion: \p.b
```

- [github.com/pbrinkmeier/pp-tut](https://github.com/pbrinkmeier/pp-tut)
- Modul x liegt in demos/x.hs

# Begriffe im $\lambda$ -Kalkül

Begriff	Formel	Bedeutung
$\alpha$ -Äquivalenz	$t_1 \stackrel{\alpha}{=} t_2$	$t_1, t_2$ sind gleicher Struktur
$\eta$ -Äquivalenz	$\lambda x.f \stackrel{\eta}{=} f$	„Unterversorgung“
Freie Variablen	$fv(\lambda p.b) = b$	Menge der nicht durch $\lambda$ s gebundenen Variablen
Substitution	$(\lambda p.b) [b \rightarrow c] = \lambda p.c$	Ersetzung freier Variablen
Redex	$(\lambda p.b) t$	„Reducible expression“
$\beta$ -Reduktion	$(\lambda p.b) t \Rightarrow b [p \rightarrow t]$	„Funktionsanwendung“

- $fv(t)$  bezeichnet die frei vorkommenden Variablen im Term  $t$
- Frei vorkommend  $\approx$  nicht durch ein  $\lambda$  gebunden
  - $fv(x) = \{x\}$ , wenn  $x$  Variable
  - $fv(f\ x) = fv(f) \cup fv(x)$
  - $fv(\lambda p.b) = fv(b) \setminus \{p\}$
- Beispiele:
  - $fv(\lambda x.x) = \emptyset$
  - $fv(\lambda x.y) = \{y\}$

- $fv(t)$  bezeichnet die frei vorkommenden Variablen im Term  $t$
- Frei vorkommend  $\approx$  nicht durch ein  $\lambda$  gebunden
  - $fv(x) = \{x\}$ , wenn  $x$  Variable
  - $fv(f\ x) = fv(f) \cup fv(x)$
  - $fv(\lambda p.b) = fv(b) \setminus \{p\}$
- Beispiele:
  - $fv(\lambda x.x) = \emptyset$
  - $fv(\lambda x.y) = \{y\}$
- Implementiert `fv :: LambdaTerm -> Set String`
  - Benutzt `Set`, `union`, `delete` und `fromList` aus `Data.Set`



# Substitution

- Substitution ersetzt alle freien Variablen in einem Term
- $t[a \rightarrow b]$  — Ersetze  $a$  durch  $b$  in  $t$
- Beispiele:
  - $a[a \rightarrow b] = b$
  - $a[b \rightarrow c] = a$
  - $(f\ x)[f \rightarrow g][x \rightarrow y] = g\ y$

# Substitution

- Substitution ersetzt alle freien Variablen in einem Term
- $t[a \rightarrow b]$  — Ersetze  $a$  durch  $b$  in  $t$
- Beispiele:
  - $a[a \rightarrow b] = b$
  - $a[b \rightarrow c] = a$
  - $(f\ x)[f \rightarrow g][x \rightarrow y] = g\ y$
  - $(\lambda x.f\ x)[x \rightarrow y] = \lambda x.f\ x$  ( $x$  ist nicht frei)
  - $(\lambda x.f\ x)[f \rightarrow g] = \lambda x.g\ x$  ( $f$  ist frei)

# Substitution

- Substitution ersetzt alle freien Variablen in einem Term
- $t[a \rightarrow b]$  — Ersetze  $a$  durch  $b$  in  $t$
- Beispiele:
  - $a[a \rightarrow b] = b$
  - $a[b \rightarrow c] = a$
  - $(f\ x)[f \rightarrow g][x \rightarrow y] = g\ y$
  - $(\lambda x.f\ x)[x \rightarrow y] = \lambda x.f\ x$  ( $x$  ist nicht frei)
  - $(\lambda x.f\ x)[f \rightarrow g] = \lambda x.g\ x$  ( $f$  ist frei)
- Implementiert

```
substitute :: (String, Term) -> Term -> Term
```

  - `type Term = LambdaTerm`
  - `fv` braucht ihr dafür nicht

- $t_1 \stackrel{\alpha}{=} t_2$  — Strukturelle Äquivalenz der Terme  $t_1$  und  $t_2$
- Umformung von  $t_1$  in  $t_2$  allein durch Substitution der (gebundenen) Variablen möglich

- $t_1 \stackrel{\alpha}{=} t_2$  — Strukturelle Äquivalenz der Terme  $t_1$  und  $t_2$
- Umformung von  $t_1$  in  $t_2$  allein durch Substitution der (gebundenen) Variablen möglich
- Bspw.:
  - $x \stackrel{\alpha}{\neq} y$ , da  $x$  und  $y$  frei sind
  - $\lambda x.x \stackrel{\alpha}{=} \lambda y.y$ , durch Umbenennen von  $x$  zu  $y$
  - $f (\lambda x.y) \stackrel{\alpha}{=} f (\lambda p.y)$
  - $\lambda x.y \stackrel{\alpha}{\neq} \lambda x.z$

- $\lambda x.f \ x \stackrel{\eta}{=} f$ , wenn  $x \notin fv(f)$
- Wie bei Haskell:  
    `all list = foldl (&&) True list  $\Leftrightarrow$`   
    `all = \list -> foldl (&&) True list  $\Leftrightarrow$`   
    `all = foldl (&&) True`
- Also:
  - $\eta$ -Äquivalenz: eher Umformungsschritt als Gleichheitskriterium
  - Formelle Definition von Unterversorgung

- Bisher:  $\lambda$ -Terme als (seltsame) Datenstruktur  
Jetzt: Ausführungssemantik

- Bisher:  $\lambda$ -Terme als (seltsame) Datenstruktur  
Jetzt: Ausführungssemantik
- RedEx: „Reducible expression“  $\Leftrightarrow$   
Funktionsanwendung  $(f\ a)$ , mit  $f = \lambda p.b$
- $(\lambda p.b)\ a$



- Bisher:  $\lambda$ -Terme als (seltsame) Datenstruktur  
Jetzt: Ausführungssemantik
- RedEx: „Reducible expression“  $\Leftrightarrow$   
Funktionsanwendung  $(f\ a)$ , mit  $f = \lambda p.b$
- $(\lambda p.b)\ a \Longrightarrow b[p \rightarrow a]$

- Bisher:  $\lambda$ -Terme als (seltsame) Datenstruktur  
Jetzt: Ausführungssemantik
- RedEx: „Reducible expression“  $\Leftrightarrow$   
Funktionsanwendung  $(f\ a)$ , mit  $f = \lambda p.b$
- $(\lambda p.b)\ a \implies b[p \rightarrow a]$
- „Ausführung“ (besser: Auswertung) von  $\lambda$ -Termen: Anwenden der  $\beta$ -Reduktion, bis Term „konvergiert“
- Term konvergiert  $\approx$  Normalform  $\approx$  enthält keinen Redex mehr
  - Notation:  $t \not\Rightarrow$

- Bisher:  $\lambda$ -Terme als (seltsame) Datenstruktur  
Jetzt: Ausführungssemantik
- RedEx: „Reducible expression“  $\Leftrightarrow$   
Funktionsanwendung  $(f\ a)$ , mit  $f = \lambda p.b$
- $(\lambda p.b)\ a \implies b[p \rightarrow a]$
- „Ausführung“ (besser: Auswertung) von  $\lambda$ -Termen: Anwenden der  $\beta$ -Reduktion, bis Term „konvergiert“
- Term konvergiert  $\approx$  Normalform  $\approx$  enthält keinen Redex mehr
  - Notation:  $t \not\Rightarrow$
- $id\ a = (\lambda x.x)\ a \implies x[x \rightarrow a] = a \not\Rightarrow$

## Beispiel: Church-Booleans

$$c_{\text{true}} = \lambda x. \lambda y. x$$

$$c_{\text{false}} = \lambda x. \lambda y. y$$

$$\begin{aligned}\text{AND} &= \lambda a. \lambda b. a \ b \ c_{\text{false}} \\ &= \lambda a. \lambda b. (a \ b) \ c_{\text{false}}\end{aligned}$$

Funktioniert AND?  $\rightsquigarrow$  Wahrheitstabelle aufstellen:

$$\text{AND } c_{\text{true}} \ c_{\text{true}} = (\lambda a. \lambda b. a \ b \ c_{\text{false}}) \ c_{\text{true}} \ c_{\text{true}}$$

## Beispiel: Church-Booleans

$$c_{\text{true}} = \lambda x. \lambda y. x$$

$$c_{\text{false}} = \lambda x. \lambda y. y$$

$$\begin{aligned}\text{AND} &= \lambda a. \lambda b. a \ b \ c_{\text{false}} \\ &= \lambda a. \lambda b. (a \ b) \ c_{\text{false}}\end{aligned}$$

Funktioniert AND?  $\rightsquigarrow$  Wahrheitstabelle aufstellen:

$$\begin{aligned}\text{AND } c_{\text{true}} \ c_{\text{true}} &= (\lambda a. \lambda b. a \ b \ c_{\text{false}}) \ c_{\text{true}} \ c_{\text{true}} \\ \Rightarrow_{\beta} (\lambda b. a \ b \ c_{\text{false}}) [a \rightarrow c_{\text{true}}] \ c_{\text{true}} &= (\lambda b. \underline{c_{\text{true}} \ b \ c_{\text{false}}}) \ c_{\text{true}}\end{aligned}$$

## Beispiel: Church-Booleans

$$c_{\text{true}} = \lambda x. \lambda y. x$$

$$c_{\text{false}} = \lambda x. \lambda y. y$$

$$\begin{aligned}\text{AND} &= \lambda a. \lambda b. a \ b \ c_{\text{false}} \\ &= \lambda a. \lambda b. (a \ b) \ c_{\text{false}}\end{aligned}$$

Funktioniert AND?  $\rightsquigarrow$  Wahrheitstabelle aufstellen:

$$\begin{aligned}\text{AND } c_{\text{true}} \ c_{\text{true}} &= (\lambda a. \lambda b. a \ b \ c_{\text{false}}) \ c_{\text{true}} \ c_{\text{true}} \\ \Rightarrow_{\beta} (\lambda b. a \ b \ c_{\text{false}}) [a \rightarrow c_{\text{true}}] \ c_{\text{true}} &= (\lambda b. \underline{c_{\text{true}}} \ b \ c_{\text{false}}) \ c_{\text{true}} \\ \Rightarrow_{\beta} (c_{\text{true}} \ b \ c_{\text{false}}) [b \rightarrow c_{\text{true}}] &= (\lambda x. \lambda y. x) \ c_{\text{true}} \ c_{\text{false}}\end{aligned}$$

## Beispiel: Church-Booleans

$$c_{\text{true}} = \lambda x. \lambda y. x$$

$$c_{\text{false}} = \lambda x. \lambda y. y$$

$$\begin{aligned}\text{AND} &= \lambda a. \lambda b. a \ b \ c_{\text{false}} \\ &= \lambda a. \lambda b. (a \ b) \ c_{\text{false}}\end{aligned}$$

Funktioniert AND?  $\rightsquigarrow$  Wahrheitstabelle aufstellen:

$$\begin{aligned}\text{AND } c_{\text{true}} \ c_{\text{true}} &= (\lambda a. \lambda b. a \ b \ c_{\text{false}}) \ c_{\text{true}} \ c_{\text{true}} \\ \Rightarrow_{\beta} (\lambda b. a \ b \ c_{\text{false}}) [a \rightarrow c_{\text{true}}] \ c_{\text{true}} &= (\lambda b. \underline{c_{\text{true}}} \ b \ c_{\text{false}}) \ c_{\text{true}} \\ \Rightarrow_{\beta} (c_{\text{true}} \ b \ c_{\text{false}}) [b \rightarrow c_{\text{true}}] &= (\lambda x. \lambda y. x) \ c_{\text{true}} \ c_{\text{false}} \\ \Rightarrow_{\beta} (\underline{\lambda y. c_{\text{true}}}) \ c_{\text{true}} &\Rightarrow_{\beta} c_{\text{true}} \quad \checkmark\end{aligned}$$

## Beispiel: Church-Booleans

$$\text{AND } c_{\text{false}} t = (\lambda a. \lambda b. a \ b \ c_{\text{false}}) c_{\text{false}} t$$



## Beispiel: Church-Booleans

$$\begin{aligned}\text{AND } c_{\text{false}} t &= (\lambda a. \lambda b. a \ b \ c_{\text{false}}) \ c_{\text{false}} \ t \\ \Rightarrow_{\beta} (\lambda b. a \ b \ c_{\text{false}}) [a \rightarrow c_{\text{false}}] \ t &= (\lambda b. c_{\text{false}} \ b \ c_{\text{false}}) \ t\end{aligned}$$

## Beispiel: Church-Booleans

$$\begin{aligned}\text{AND } c_{\text{false}} t &= (\lambda a. \lambda b. a \ b \ c_{\text{false}}) \ c_{\text{false}} \ t \\ \Rightarrow_{\beta} (\lambda b. a \ b \ c_{\text{false}}) [a \rightarrow c_{\text{false}}] \ t &= (\lambda b. \underline{c_{\text{false}} \ b \ c_{\text{false}}}) \ t \\ \Rightarrow_{\beta} (c_{\text{true}} \ b \ c_{\text{false}}) [b \rightarrow t] &= (\lambda x. \lambda y. y) \ t \ c_{\text{false}}\end{aligned}$$

## Beispiel: Church-Booleans

$$\begin{aligned}\text{AND } c_{\text{false}} t &= (\lambda a. \lambda b. a \ b \ c_{\text{false}}) \ c_{\text{false}} \ t \\ \Rightarrow_{\beta} (\lambda b. a \ b \ c_{\text{false}}) [a \rightarrow c_{\text{false}}] \ t &= (\lambda b. c_{\text{false}} \ b \ c_{\text{false}}) \ t \\ \Rightarrow_{\beta} (c_{\text{true}} \ b \ c_{\text{false}}) [b \rightarrow t] &= (\lambda x. \lambda y. y) \ t \ c_{\text{false}} \\ &\Rightarrow^2 c_{\text{false}}\end{aligned}$$

$$\text{AND } t \ c_{\text{false}} = (\lambda a. \lambda b. a \ b \ c_{\text{false}}) \ t \ c_{\text{false}}$$

## Beispiel: Church-Booleans

$$\begin{aligned}\text{AND } c_{\text{false}} t &= (\lambda a. \lambda b. a \ b \ c_{\text{false}}) \ c_{\text{false}} \ t \\ \Rightarrow_{\beta} (\lambda b. a \ b \ c_{\text{false}}) [a \rightarrow c_{\text{false}}] \ t &= (\lambda b. \underline{c_{\text{false}} \ b \ c_{\text{false}}}) \ t \\ \Rightarrow_{\beta} (c_{\text{true}} \ b \ c_{\text{false}}) [b \rightarrow t] &= (\lambda x. \lambda y. y) \ t \ c_{\text{false}} \\ &\Rightarrow^2 c_{\text{false}}\end{aligned}$$

$$\begin{aligned}\text{AND } t \ c_{\text{false}} &= (\lambda a. \lambda b. a \ b \ c_{\text{false}}) \ t \ c_{\text{false}} \\ \Rightarrow_{\beta} (\lambda b. a \ b \ c_{\text{false}}) [a \rightarrow t] \ c_{\text{false}} &= (\lambda b. \underline{t \ b \ c_{\text{false}}}) \ c_{\text{false}}\end{aligned}$$

## Beispiel: Church-Booleans

$$\begin{aligned}\text{AND } c_{\text{false}} t &= (\lambda a. \lambda b. a b c_{\text{false}}) c_{\text{false}} t \\ \Rightarrow_{\beta} (\lambda b. a b c_{\text{false}}) [a \rightarrow c_{\text{false}}] t &= (\lambda b. c_{\text{false}} b c_{\text{false}}) t \\ \Rightarrow_{\beta} (c_{\text{true}} b c_{\text{false}}) [b \rightarrow t] &= (\lambda x. \lambda y. y) t c_{\text{false}} \\ &\Rightarrow^2 c_{\text{false}}\end{aligned}$$

$$\begin{aligned}\text{AND } t c_{\text{false}} &= (\lambda a. \lambda b. a b c_{\text{false}}) t c_{\text{false}} \\ \Rightarrow_{\beta} (\lambda b. a b c_{\text{false}}) [a \rightarrow t] c_{\text{false}} &= (\lambda b. t b c_{\text{false}}) c_{\text{false}} \\ \Rightarrow_{\beta} (t b c_{\text{false}}) [b \rightarrow c_{\text{false}}] &= t c_{\text{false}} c_{\text{false}}\end{aligned}$$

## Beispiel: Church-Booleans

$$\begin{aligned}\text{AND } c_{\text{false}} t &= (\lambda a. \lambda b. a b c_{\text{false}}) c_{\text{false}} t \\ \Rightarrow_{\beta} (\lambda b. a b c_{\text{false}}) [a \rightarrow c_{\text{false}}] t &= (\lambda b. c_{\text{false}} b c_{\text{false}}) t \\ \Rightarrow_{\beta} (c_{\text{true}} b c_{\text{false}}) [b \rightarrow t] &= (\lambda x. \lambda y. y) t c_{\text{false}} \\ &\Rightarrow^2 c_{\text{false}}\end{aligned}$$

$$\begin{aligned}\text{AND } t c_{\text{false}} &= (\lambda a. \lambda b. a b c_{\text{false}}) t c_{\text{false}} \\ \Rightarrow_{\beta} (\lambda b. a b c_{\text{false}}) [a \rightarrow t] c_{\text{false}} &= (\lambda b. t b c_{\text{false}}) c_{\text{false}} \\ \Rightarrow_{\beta} (t b c_{\text{false}}) [b \rightarrow c_{\text{false}}] &= t c_{\text{false}} c_{\text{false}} \\ &\rightsquigarrow c_{\text{false}}\end{aligned}$$

- Welcher Redex soll zuerst ausgewertet werden?
- $\rightsquigarrow$  verschiedene Auswertungsstrategien

- Welcher Redex soll zuerst ausgewertet werden?
- $\rightsquigarrow$  verschiedene Auswertungsstrategien
- **Volle  $\beta$ -Reduktion** — Beliebiger Redex
- **Normalreihenfolge** — „Linkester“ Redex



- Welcher Redex soll zuerst ausgewertet werden?
- $\rightsquigarrow$  verschiedene Auswertungsstrategien
- **Volle  $\beta$ -Reduktion** — Beliebiger Redex
- **Normalreihenfolge** — „Linkester“ Redex
- **Call-by-Name** — Nur äußerster „linkester Redex“
- **Call-by-Value** — „Linkester Redex“, der eine Normalform als Argument hat

# Normalreihenfolge

```
module LambdaN where

data LambdaTerm
  = Var String
  | App LambdaTerm LambdaTerm
  | Abs String LambdaTerm
```

- Implementiert  
normalBeta :: LambdaTerm -> LambdaTerm
- Führt einen  $\beta$ -Reduktionsschritt in Normalreihenfolge (linkester Redex) aus
- Wenn kein Redex vorkommt, wird derselbe Term zurückgegeben
- Bindet LambdaShow ein für Show LambdaTerm

# Church-Zahlen im $\lambda$ -Kalkül

---

$$\begin{aligned}c_0 &= ? \\c_1 &= s(c_0) \\c_2 &= s(s(c_0)) \\c_3 &= s(s(s(c_0))) \\c_8 &= s(s(s(s(s(s(s(s(c_0))))))))\end{aligned}$$

1. Die 0 ist Teil der natürlichen Zahlen
2. Wenn  $n$  Teil der natürlichen Zahlen ist,  
ist auch  $s(n) = n + 1$  Teil der natürlichen Zahlen

- „Zahlen“ im  $\lambda$ -Kalkül werden durch Funktionen in Normalform dargestellt
- $c_n f x = f$   $n$ -mal angewendet auf  $x$
- Bspw.  $(c_3 g y) = g (g (g y)) = g^3 y$   
Mit  $c_3 = \lambda f. \lambda x. f (f (f x))$
- Schreibt eine  $\lambda$ -Funktion *succ*, die eine Church-Zahl nimmt und zu deren Nachfolger auswertet

- „Zahlen“ im  $\lambda$ -Kalkül werden durch Funktionen in Normalform dargestellt
- $c_n f x = f$   $n$ -mal angewendet auf  $x$
- Bspw.  $(c_3 g y) = g (g (g y)) = g^3 y$   
Mit  $c_3 = \lambda f. \lambda x. f (f (f x))$
- Schreibt eine  $\lambda$ -Funktion *succ*, die eine Church-Zahl nimmt und zu deren Nachfolger auswertet
- Überträgt die Funktion in euren Haskell-Code und wertet *succ*  $c_0$  durch wiederholtes Anwenden von `normalBeta` aus
- Vergleicht euer Ergebnis mit dem von Wavelength:
  - [pp.ipd.kit.edu/lehre/misc/lambda-ide/Wavelength.html](http://pp.ipd.kit.edu/lehre/misc/lambda-ide/Wavelength.html)

- Terme  $t$ : Variable ( $x$ ), Funktion ( $\lambda x.t$ ), Anwendung ( $t\ t$ )
- $\alpha$ -Äquivalenz: Gleiche Struktur
- $\eta$ -Äquivalenz: Unterversorgung
- Freie Variablen, Substitution, RedEx
- $\beta$ -Reduktion:  
 $(\lambda p.b)\ t \Rightarrow b[p \rightarrow t]$
- Church-Booleans, Church-Zahlen, Y-Kombinator