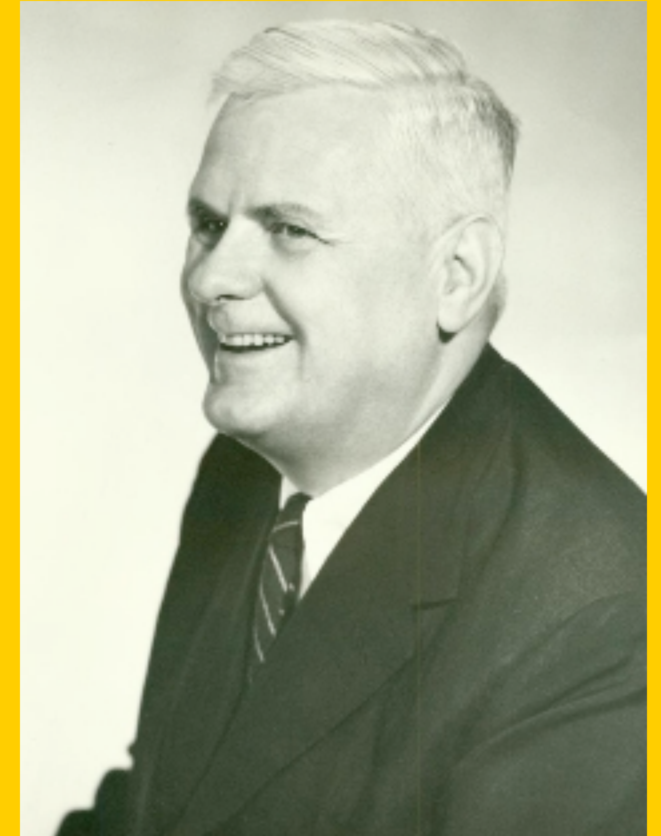
# λ-Fun

## Einführung in den Lambda Kalkül

Jan Hermanns
www.sushi3003.de

# λ-Kalkül

- Alonzo Church ~1930
- Einfacher mathematischer Mechanismus zur Definition und Anwendung von Funktionen
- Grundlage der funktionalen Programmiersprachen



**Alonzo Church 1903-1995**

# Abstraktion

Funktionen haben keine Namen, es wird nur deren Verhalten beschrieben

$\lambda x$ . chocolate-covered x

# Applikation

Anwenden („Aufrufen“) einer Funktion durch „Hintereinanderschreiben“ von Funktion und Eingabewert

λx . chocolate-covered x   cookies

Zur besseren Lesbarkeit klammern wir

(λx . chocolate-covered x) cookies

# Reduktion

Funktionswerte werden durch Einsetzen von Eingabewerten ausgerechnet

(λx . chocolate-covered x) cookies

⇒ chocolate-covered cookies

# Abstakte Syntax

Definition aller „wohlgeformten" Ausdrücke

| | | | |
|---|---|---|---|
| term | ::= | variable | (Variablen) |
| | \| | „λ" variable „." term | (Abstraktion) |
| | \| | term term | (Applikation) |

# Currying

$\lambda y . \lambda x . y\ x$

$(\lambda y . \lambda x . y\ x)$ curry

$\lambda x .$ curry $x$

$(\lambda x .$ curry $x)$ chicken

curry chicken

# Beispiele

Nicht alle Variablennamen müssen definiert sein, die folgenden Terme sind ebenfalls gültig:

```
λx . j h
```

```
a b
```

# Beispiele

(λf . λx . f (f x)) λy . sweet y

λx . (λy . sweet y) ((λy . sweet y) x)

λx . (λy . sweet y) (sweet x)

λx . sweet (sweet x)

(λx . sweet (sweet x)) cookies

sweet (sweet cookies)

# Achtung

$(\lambda f\,.\,\lambda x\,.\,f\,(f\,x))\;x$

$\lambda x\,.\,x\,(x\,x)$

$\lambda f\,.\,\lambda x\,.\,f\,(f\,x)$ α-Konversion

$\lambda a\,.\,\lambda b\,.\,a\,(a\,b)$

$(\lambda a\,.\,\lambda b\,.\,a\,(a\,b))\;x$

$\lambda b\,.\,x\,(x\,b)$

# Church booleans

| | |
|---|---|
| true | λx. λy. x |
| false | λx. λy. y |
| if-then-else | λc. λt. λe. c t e |
| and | λa. λb. a b a |
| or | λa. λb. a a b |
| not | λx. λa. λb. x b a |
| … | … |

(((if-then-else) true) richtig) falsch

(((λc . λt . λe. c t e) λx . λy . x) richtig) falsch

((λt . λe. (λx . λy . x) t e) richtig) falsch

((λt . λe. (λy . t) e) richtig) falsch

((λt . λe. t) richtig) falsch

(λe. richtig) falsch

richtig

(((if-then-else) false) richtig) falsch

(((λc . λt . λe. c t e) λx . λy . y) richtig) falsch

((λt . λe. (λx . λy . y) t e) richtig) falsch

((λt . λe. (λy . y) e) richtig) falsch

((λt . λe. e) richtig) falsch

(λe. e) falsch

falsch

# Church numerals

| | |
|---|---|
| 0 | λf. λx. x |
| 1 | λf. λx. f x |
| 2 | λf. λx. f (f x) |
| 3 | λf. λx. f (f (f x)) |
| ... | ... |
| inc | λn. λg. λy. g (n g y) |
| plus | λm. λn. λg. λy. m g (n g y) |
| mul | λm. λn. λg. λy. m (n g) y |
| ... | ... |

# Inkrementieren

(inc) 0

$(\lambda n.\ \lambda g.\ \lambda y.\ g\ (n\ g\ y))\ \lambda f.\ \lambda x.\ x$

$\lambda g.\ \lambda y.\ g\ ((\lambda f.\ \lambda x.\ x)\ g\ y)$

$\lambda g.\ \lambda y.\ g\ ((\lambda x.\ x)\ y)$

$\lambda g.\ \lambda y.\ g\ y$ α-Konversion

$\lambda f.\ \lambda x.\ f\ x$

# Inkrementieren

$(\text{inc})\ 1$

$(\lambda n.\ \lambda g.\ \lambda y.\ g\ (n\ g\ y))\ \lambda f.\ \lambda x.\ f\ x$

$\lambda g.\ \lambda y.\ g\ ((\lambda f.\ \lambda x.\ f\ x)\ g\ y)$

$\lambda g.\ \lambda y.\ g\ ((\lambda x.\ g\ x)\ y)$

$\lambda g.\ \lambda y.\ g\ (g\ y)$ α-Konversion

$\lambda f.\ \lambda x.\ f\ (f\ x)$

# Church pairs

| | |
|---|---|
| pair | λa. λb. λc. c a b |
| fst | λa. a true |
| snd | λa. a false |
| nil | λa. true |
| isnil? | λp. p (λx. λy. false) |
| ... | ... |

# nil

Wir bauen Listen aus Paaren, wobei das Ende einer Liste mit nil gekennzeichnet wird

(pair 1) ((pair 2) ((pair 3) ((pair 4) nil)))

# pair

pair 2 nil

((λa. λb. λc. c a b) 2) nil

(λb. λc. c 2 b) nil

λc. c 2 nil

# Listen

pair 1 (pair 2 nil)

((λa. λb. λc. c a b) 1) (pair 2 nil)

(λb. λc. c 1 b) (pair 2 nil)

λc. c 1 (pair 2 nil) α-Konversion

λd. d 1 (pair 2 nil)

λd. d 1 (λc. c 2 nil)

# snd

snd (pair 0 nil)

(λa. a false) λc. c 0 nil

(λa. a (λx. λy. y)) λc. c 0 nil

(λc. c 0 nil) (λx. λy. y)

((λx. λy. y) 0) nil

(λy. y) nil

nil

# isnil?

isnil? nil

(λp. p (λx. λy. false)) nil

(λp. p (λx. λy. false)) λa. true

(λa. true) (λx. λy. false)

true

# isnil?

isnil? (pair 2 nil)

(λp. p (λx. λy. false)) (pair 2 nil)

(pair 2 nil) (λx. λy. false)

(λc. c 2 nil) (λx. λy. false)

(λx. λy. false) 2 nil

(λy. false) nil

false

# length

Wie berechnet man die Länge einer Liste?

λx. ((if-then-else (isnil? x)) 0) (inc (??? (snd x)))

λf. λx. ((if-then-else (isnil? x)) 0) (inc (f (snd x)))

(λf. λx. ((if-then-else (isnil? x)) 0) (inc (f (snd x)))) EGAL

λx. ((if-then-else (isnil? x)) 0) (inc (EGAL (snd x)))

Mit dieser Funktion können wir die Länge von leeren Listen berechnen!

Daher nennen wir die Funktion len0

# len1

(λf. λy. ((if-then-else (isnil? y)) 0) (inc (f (snd y)))) len0

λy. ((if-then-else (isnil? y)) 0) (inc (len0 (snd y)))

Mit dieser Funktion können wir die Länge von leeren Listen und Listen mit einem Element berechnen!

# len2

(λf. λz. ((if-then-else (isnil? z)) 0) (inc (f (snd z)))) len1

λz. ((if-then-else (isnil? z)) 0) (inc (len1 (snd z)))

Mit dieser Funktion können wir die Länge von Listen mit bis zu zwei Elementen berechnen!

# mklen

λf. λz. ((if-then-else (isnil? z)) 0) (inc (f (snd z)))

Mit dieser Funktion können wir also neue Funktionen bauen, die die Länge von bestimmten Listen ausrechnen können.

# len0 revisited

(λm. m EGAL) mklen

mklen EGAL

(λf. λx. ((if-then-else (isnil? x)) 0) (inc (f (snd x)))) EGAL

λx. ((if-then-else (isnil? x)) 0) (inc (EGAL (snd x)))

len0

# len1 revisited

(λm. m (m EGAL)) mklen

mklen (mklen EGAL)

mklen len0

(λf. λz. ((if-then-else (isnil? z)) 0) (inc (f (snd z)))) len0

λy. ((if-then-else (isnil? y)) 0) (inc (len0 (snd y)))

len1

# len2 revisited

(λm. m (m (m EGAL))) mklen

mklen (mklen (mklen EGAL))

mklen len1

(λf. λz. ((if-then-else (isnil? z)) 0) (inc (f (snd z)))) len1

λy. ((if-then-else (isnil? y)) 0) (inc (len1 (snd y)))

len2

# len2 revisited

(λm. m (m (m EGAL))) mklen

mklen (mklen (mklen EGAL))

mklen len1

(λf. λz. ((if-then-else (isnil? z)) 0) (inc (f (snd z)))) len1

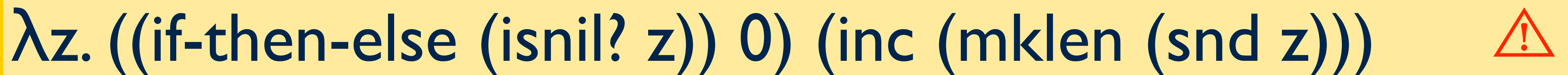
λy. ((if-then-else (isnil? y)) 0) (inc (len1 (snd y)))

len2

# mklen revisited

mklen mklen

(λf. λz. ((if-then-else (isnil? z)) 0) (inc (f (snd z)))) mklen




λz. ((if-then-else (isnil? z)) 0) (inc ((mklen mklen) (snd z)))

So ganz stimmt es noch nicht, aber wir sind fast am Ziel!

# mklen redefined

Wir ändern die Definition von mklen minimal

λf. λz. ((if-then-else (isnil? z)) 0) (inc ((f f) (snd z)))

Und erhalten durch folgendem Aufruf die gesuchte Funktion length!

mklen mklen

# Quellen

- „The Little Schemer“, Friedman & Felleisen, MIT Press
- „An Introduction to Lambda Calculus and Scheme“, Jim Larson 1996, http://www.jetcafe.org/jim/lambda.html
- „Der Lambda-Kalkuel“, Christoph Kreitz 2007, http://tt2.hpi.uni-potsdam.de/archive/video/flash/2822/
- Wikipedia