

# Vorlesung

# Theoretische Informatik II

**Bernhard Beckert**

**Institut für Informatik**



**Wintersemester 2007/2008**

# Dank

Diese Vorlesungsmaterialien basieren zum Teil auf den Folien zu den Vorlesungen von

**Katrin Erk** (gehalten an der Universität Koblenz-Landau)

**Jürgen Dix** (gehalten an der TU Clausthal)

**Christoph Kreitz** (gehalten an der Universität Potsdam)

Ihnen gilt mein herzlicher Dank.

– *Bernhard Beckert, Oktober 2007*

## Teil IV

# Der $\lambda$ -Kalkül

## Zweck

- **Formale Definition von Funktionen**
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

## Grundlegende Eigenschaften

- Funktion identifiziert mit  $\lambda$ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

## Zweck

- Formale Definition von Funktionen
- **Untersuchung und Anwendung von Funktionsdefinitionen**
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

## Grundlegende Eigenschaften

- Funktion identifiziert mit  $\lambda$ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

# Die Idee

## Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- **Berechnungsmodell**
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

## Grundlegende Eigenschaften

- Funktion identifiziert mit  $\lambda$ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

# Die Idee

## Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- **Grundlage funktionaler Programmiersprachen**
- Einfach und doch mächtig

## Grundlegende Eigenschaften

- Funktion identifiziert mit  $\lambda$ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

# Die Idee

## Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- **Einfach und doch mächtig**

## Grundlegende Eigenschaften

- Funktion identifiziert mit  $\lambda$ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren



# Die Idee

## Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

## Grundlegende Eigenschaften

- Funktion identifiziert mit  $\lambda$ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

## Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

## Grundlegende Eigenschaften

- Funktion identifiziert mit  $\lambda$ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

## Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

## Grundlegende Eigenschaften

- Funktion identifiziert mit  $\lambda$ -Ausdrücken, die sie beschreiben
- **Funktionen angewendet auf Funktionen**
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

## Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

## Grundlegende Eigenschaften

- Funktion identifiziert mit  $\lambda$ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- **Nichts außer Funktionen (keine anderen Datentypen)**

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

## Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

## Grundlegende Eigenschaften

- Funktion identifiziert mit  $\lambda$ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

## Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

## Grundlegende Eigenschaften

- Funktion identifiziert mit  $\lambda$ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

## Definition 14.1 ( $\lambda$ -Ausdruck)

**Variable** Jede Variable

$x$

ist ein  $\lambda$ -Ausdruck

**Abstraktion** Ist  $x$  eine Variable und  $e$  ein  $\lambda$ -Ausdruck, dann ist

$\lambda x. e$

ein  $\lambda$ -Ausdruck

**Anwendung** Sind  $e_1, e_2$   $\lambda$ -Ausdrücke, dann ist

$e_1 e_2$

ein  $\lambda$ -Ausdruck

## Definition 14.1 ( $\lambda$ -Ausdruck)

**Variable** Jede Variable

$x$

ist ein  $\lambda$ -Ausdruck

**Abstraktion** Ist  $x$  eine Variable und  $e$  ein  $\lambda$ -Ausdruck, dann ist

$\lambda x.e$

ein  $\lambda$ -Ausdruck

**Anwendung** Sind  $e_1, e_2$   $\lambda$ -Ausdrücke, dann ist

$e_1 e_2$

ein  $\lambda$ -Ausdruck

## Definition 14.1 ( $\lambda$ -Ausdruck)

**Variable** Jede Variable

$$x$$

ist ein  $\lambda$ -Ausdruck

**Abstraktion** Ist  $x$  eine Variable und  $e$  ein  $\lambda$ -Ausdruck, dann ist

$$\lambda x. e$$

ein  $\lambda$ -Ausdruck

**Anwendung** Sind  $e_1, e_2$   $\lambda$ -Ausdrücke, dann ist

$$e_1 e_2$$

ein  $\lambda$ -Ausdruck

## Assoziativität und Bindungsstärke

- Anwendungen sind linksassoziativ:

$$e_1 e_2 e_3 = (e_1 e_2) e_3$$

- Anwendung bindet stärker als Abstraktion:

$$\lambda x. e_1 e_2 = \lambda x. (e_1 e_2)$$

**WICHTIG!**

Dies muss man wissen, um  $\lambda$ -Ausdrücke verstehen zu können!

## Assoziativität und Bindungsstärke

- Anwendungen sind linksassoziativ:

$$e_1 e_2 e_3 = (e_1 e_2) e_3$$

- Anwendung bindet stärker als Abstraktion:

$$\lambda x. e_1 e_2 = \lambda x. (e_1 e_2)$$

WICHTIG!

Dies muss man wissen, um  $\lambda$ -Ausdrücke verstehen zu können!

## Assoziativität und Bindungsstärke

- Anwendungen sind linksassoziativ:

$$e_1 e_2 e_3 = (e_1 e_2) e_3$$

- Anwendung bindet stärker als Abstraktion:

$$\lambda x. e_1 e_2 = \lambda x. (e_1 e_2)$$

**WICHTIG!**

Dies muss man wissen, um  $\lambda$ -Ausdrücke verstehen zu können!

## Assoziativität und Bindungsstärke

- Anwendungen sind linksassoziativ:

$$e_1 e_2 e_3 = (e_1 e_2) e_3$$

- Anwendung bindet stärker als Abstraktion:

$$\lambda x. e_1 e_2 = \lambda x. (e_1 e_2)$$

**WICHTIG!**

**Dies muss man wissen, um  $\lambda$ -Ausdrücke verstehen zu können!**

## Einige wichtige $\lambda$ -Ausdrücke

### Identitätsfunktion:

$$I = \lambda x. x$$

### Selbstanwendung:

$$D = \lambda x. x x$$

### Auslassen des 2. Argumentes:

$$K = \lambda x. \lambda y. x$$

## Einige wichtige $\lambda$ -Ausdrücke

Identitätsfunktion:

$$I = \lambda x.x$$

Selbstanwendung:

$$D = \lambda x.x x$$

Auslassen des 2. Argumentes:

$$K = \lambda x.\lambda y.x$$

## Einige wichtige $\lambda$ -Ausdrücke

### Identitätsfunktion:

$$I = \lambda x.x$$

### Selbstanwendung:

$$D = \lambda x.x x$$

### Auslassen des 2. Argumentes:

$$K = \lambda x.\lambda y.x$$

## Definition 14.2

- Ein Vorkommen einer Variablen  $x$  ist gebunden, wenn es im Skopus von  $\lambda x$  ist.
- Andernfalls ist es frei.

## $\alpha$ -Konversion

$\lambda$ -Ausdrücke, die gleich sind bis auf Umbenennen gebundener Variablen

- werden identifiziert
- beschreiben dieselbe Funktion

## Definition 14.2

- Ein Vorkommen einer Variablen  $x$  ist gebunden, wenn es im Skopus von  $\lambda x$  ist.
- **Andernfalls ist es frei.**

## $\alpha$ -Konversion

$\lambda$ -Ausdrücke, die gleich sind bis auf Umbenennen gebundener Variablen

- werden identifiziert
- beschreiben dieselbe Funktion

## Definition 14.2

- Ein Vorkommen einer Variablen  $x$  ist gebunden, wenn es im Skopus von  $\lambda x$  ist.
- Andernfalls ist es frei.

## $\alpha$ -Konversion

$\lambda$ -Ausdrücke, die gleich sind bis auf Umbenennen gebundener Variablen

- werden identifiziert
- beschreiben dieselbe Funktion

## Definition 14.2

- Ein Vorkommen einer Variablen  $x$  ist gebunden, wenn es im Skopus von  $\lambda x$  ist.
- Andernfalls ist es frei.

## $\alpha$ -Konversion

$\lambda$ -Ausdrücke, die gleich sind bis auf Umbenennen gebundener Variablen

- werden identifiziert
- beschreiben dieselbe Funktion

## Definition 14.3

Das Ergebnis der Substitution von  $x$  in  $e$  durch  $e'$

in Zeichen:  $[e'/x]e$

entsteht dadurch, dass

- 1 gebundene Variablen so umbenannt werden, dass sie nur einmal vorkommen
- 2  $x$  in  $e$  syntaktisch durch  $e'$  ersetzt wird

## Definition 14.3

Das Ergebnis der Substitution von  $x$  in  $e$  durch  $e'$

in Zeichen:  $[e'/x]e$

entsteht dadurch, dass

- 1 gebundene Variablen so umbenannt werden, dass sie nur einmal vorkommen
- 2  $x$  in  $e$  syntaktisch durch  $e'$  ersetzt wird

## Definition 14.3

Das Ergebnis der Substitution von  $x$  in  $e$  durch  $e'$

in Zeichen:  $[e'/x]e$

entsteht dadurch, dass

- 1 gebundene Variablen so umbenannt werden, dass sie nur einmal vorkommen
- 2  $x$  in  $e$  syntaktisch durch  $e'$  ersetzt wird

## $\beta$ -Reduktion

Der  $\lambda$ -Ausdruck

$$(\lambda x. e) e'$$

kann durch  $\beta$ -Reduktion zu dem Ausdruck

$$[e'/x] e$$

reduziert werden

$$\text{In Zeichen: } (\lambda x. e) e' \rightarrow_{\beta} [e'/x] e$$

Wenn  $e \rightarrow_{\beta}^* e'$ , dann

- werden  $e, e'$  identifiziert
- beschreiben dieselbe Funktion

## $\beta$ -Reduktion

Der  $\lambda$ -Ausdruck

$$(\lambda x. e) e'$$

kann durch  $\beta$ -Reduktion zu dem Ausdruck

$$[e'/x] e$$

reduziert werden

$$\text{In Zeichen: } (\lambda x. e) e' \rightarrow_{\beta} [e'/x] e$$

Wenn  $e \rightarrow_{\beta}^* e'$ , dann

- werden  $e, e'$  identifiziert
- beschreiben dieselbe Funktion

## $\beta$ -Reduktion

Der  $\lambda$ -Ausdruck

$$(\lambda x. e) e'$$

kann durch  $\beta$ -Reduktion zu dem Ausdruck

$$[e'/x] e$$

reduziert werden

$$\text{In Zeichen: } (\lambda x. e) e' \rightarrow_{\beta} [e'/x] e$$

Wenn  $e \rightarrow_{\beta}^* e'$ , dann

- werden  $e, e'$  identifiziert
- beschreiben dieselbe Funktion

## Beispiel 14.4

$$(\lambda f.f (\lambda x.x)) (\lambda x.x) \rightarrow_{\beta}^* (\lambda y.y)$$

## Beispiel 14.5

*Der Ausdruck*

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

*kann nicht weiter reduziert werden*

## Beispiel 14.4

$$(\lambda f.f (\lambda x.x)) (\lambda x.x) \rightarrow_{\beta}^* (\lambda y.y)$$

## Beispiel 14.5

*Der Ausdruck*

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

*kann nicht weiter reduziert werden*

## Eigenschaften der Reduktion mittels $\alpha$ - und $\beta$ -Reduktion

- $\beta$ -Reduktion terminiert nicht immer
- $\beta$ -Reduktion ist nicht deterministisch
- Jedoch:  $\rightarrow_{\beta}^*$  ist konfluent (hat die Church-Rosser-Eigenschaft):

Gilt

$$e \rightarrow_{\beta}^* e_1 \quad \text{und} \quad e \rightarrow_{\beta}^* e_2$$

dann gibt es ein  $e'$  mit

$$e_1 \rightarrow_{\beta}^* e' \quad \text{und} \quad e_2 \rightarrow_{\beta}^* e'$$

- Darum: Normalformen sind eindeutig

## Eigenschaften der Reduktion mittels $\alpha$ - und $\beta$ -Reduktion

- $\beta$ -Reduktion terminiert nicht immer
- $\beta$ -Reduktion ist nicht deterministisch
- Jedoch:  $\rightarrow_{\beta}^*$  ist konfluent (hat die Church-Rosser-Eigenschaft):  
Gilt

$$e \rightarrow_{\beta}^* e_1 \quad \text{und} \quad e \rightarrow_{\beta}^* e_2$$

dann gibt es ein  $e'$  mit

$$e_1 \rightarrow_{\beta}^* e' \quad \text{und} \quad e_2 \rightarrow_{\beta}^* e'$$

- Darum: Normalformen sind eindeutig

## Eigenschaften der Reduktion mittels $\alpha$ - und $\beta$ -Reduktion

- $\beta$ -Reduktion terminiert nicht immer
- $\beta$ -Reduktion ist nicht deterministisch
- **Jedoch:  $\rightarrow_{\beta}^*$  ist konfluent (hat die Church-Rosser-Eigenschaft):**  
Gilt

$$e \rightarrow_{\beta}^* e_1 \quad \text{und} \quad e \rightarrow_{\beta}^* e_2$$

dann gibt es ein  $e'$  mit

$$e_1 \rightarrow_{\beta}^* e' \quad \text{und} \quad e_2 \rightarrow_{\beta}^* e'$$

- Darum: Normalformen sind eindeutig

## Eigenschaften der Reduktion mittels $\alpha$ - und $\beta$ -Reduktion

- $\beta$ -Reduktion terminiert nicht immer
- $\beta$ -Reduktion ist nicht deterministisch
- Jedoch:  $\rightarrow_{\beta}^*$  ist konfluent (hat die Church-Rosser-Eigenschaft):  
Gilt

$$e \rightarrow_{\beta}^* e_1 \quad \text{und} \quad e \rightarrow_{\beta}^* e_2$$

dann gibt es ein  $e'$  mit

$$e_1 \rightarrow_{\beta}^* e' \quad \text{und} \quad e_2 \rightarrow_{\beta}^* e'$$

- Darum: Normalformen sind eindeutig

## Eigenschaften der Reduktion mittels $\alpha$ - und $\beta$ -Reduktion

- $\beta$ -Reduktion terminiert nicht immer
- $\beta$ -Reduktion ist nicht deterministisch
- Jedoch:  $\rightarrow_{\beta}^*$  ist konfluent (hat die Church-Rosser-Eigenschaft):  
Gilt

$$e \rightarrow_{\beta}^* e_1 \quad \text{und} \quad e \rightarrow_{\beta}^* e_2$$

dann gibt es ein  $e'$  mit

$$e_1 \rightarrow_{\beta}^* e' \quad \text{und} \quad e_2 \rightarrow_{\beta}^* e'$$

- **Darum: Normalformen sind eindeutig**

# Mächtigkeit des $\lambda$ -Kalküls

## Ausdrucksstärke

Im  $\lambda$ -Kalkül kann man ausdrücken:

- Datentypen wie Integer, Boolean, Listen, Bäume, ...
- Verzweigung
- Rekursion

## Turing-Mächtigkeit

Der  $\lambda$ -Kalkül kann Turing-Maschinen simulieren

## Unentscheidbarkeit

Es ist unentscheidbar, ob für beliebig gegebene  $e, e'$  gilt, dass

$$e \rightarrow_{\beta}^* e'$$

# Mächtigkeit des $\lambda$ -Kalküls

## Ausdrucksstärke

Im  $\lambda$ -Kalkül kann man ausdrücken:

- Datentypen wie Integer, Boolean, Listen, Bäume, ...
- Verzweigung
- Rekursion

## Turing-Mächtigkeit

Der  $\lambda$ -Kalkül kann Turing-Maschinen simulieren

## Unentscheidbarkeit

Es ist unentscheidbar, ob für beliebig gegebene  $e, e'$  gilt, dass

$$e \rightarrow_{\beta}^* e'$$

# Mächtigkeit des $\lambda$ -Kalküls

## Ausdrucksstärke

Im  $\lambda$ -Kalkül kann man ausdrücken:

- Datentypen wie Integer, Boolean, Listen, Bäume, ...
- Verzweigung
- **Rekursion**

## Turing-Mächtigkeit

Der  $\lambda$ -Kalkül kann Turing-Maschinen simulieren

## Unentscheidbarkeit

Es ist unentscheidbar, ob für beliebig gegebene  $e, e'$  gilt, dass

$$e \rightarrow_{\beta}^* e'$$

# Mächtigkeit des $\lambda$ -Kalküls

## Ausdrucksstärke

Im  $\lambda$ -Kalkül kann man ausdrücken:

- Datentypen wie Integer, Boolean, Listen, Bäume, ...
- Verzweigung
- Rekursion

## Turing-Mächtigkeit

Der  $\lambda$ -Kalkül kann Turing-Maschinen simulieren

## Unentscheidbarkeit

Es ist unentscheidbar, ob für beliebig gegebene  $e, e'$  gilt, dass

$$e \rightarrow_{\beta}^* e'$$

# Mächtigkeit des $\lambda$ -Kalküls

## Ausdrucksstärke

Im  $\lambda$ -Kalkül kann man ausdrücken:

- Datentypen wie Integer, Boolean, Listen, Bäume, ...
- Verzweigung
- Rekursion

## Turing-Mächtigkeit

Der  $\lambda$ -Kalkül kann Turing-Maschinen simulieren

## Unentscheidbarkeit

Es ist unentscheidbar, ob für beliebig gegebene  $e, e'$  gilt, dass

$$e \rightarrow_{\beta}^* e'$$

# Mächtigkeit des $\lambda$ -Kalküls

## Ausdrucksstärke

Im  $\lambda$ -Kalkül kann man ausdrücken:

- Datentypen wie Integer, Boolean, Listen, Bäume, ...
- Verzweigung
- Rekursion

## Turing-Mächtigkeit

Der  $\lambda$ -Kalkül kann Turing-Maschinen simulieren

## Unentscheidbarkeit

Es ist unentscheidbar, ob für beliebig gegebene  $e, e'$  gilt, dass

$$e \rightarrow_{\beta}^* e'$$

# Darstellung von Aussagenlogik im $\lambda$ -Kalkül

## Wahrheitswerte

$true =_{\text{def}} \lambda x. \lambda y. x$

$false =_{\text{def}} \lambda x. \lambda y. y$

## if-then-else

$\text{if } C \text{ then } U \text{ else } V =_{\text{def}} ???$

# Darstellung von Aussagenlogik im $\lambda$ -Kalkül

## Wahrheitswerte

$true =_{\text{def}} \lambda x. \lambda y. x$

$false =_{\text{def}} \lambda x. \lambda y. y$

## if-then-else

$\text{if } C \text{ then } U \text{ else } V =_{\text{def}} ???$

# Darstellung von Aussagenlogik im $\lambda$ -Kalkül

## Wahrheitswerte

$true =_{\text{def}} \lambda x. \lambda y. x$

$false =_{\text{def}} \lambda x. \lambda y. y$

## if-then-else

$\text{if } C \text{ then } U \text{ else } V =_{\text{def}} C U V$

# Darstellung von Aussagenlogik im $\lambda$ -Kalkül

## Mit true, false, if-then-else alles darstellbar

$\neg x \equiv \text{if } x \text{ then false else true}$

$x \wedge y \equiv \text{if } x \text{ then } y \text{ else false}$

$x \vee y \equiv \text{if } x \text{ then true else } y$

## Damit

$\neg x \equiv x \text{ false true}$

$x \wedge y \equiv x y \text{ false}$

$x \vee y \equiv x \text{ true } y$

# Darstellung von Aussagenlogik im $\lambda$ -Kalkül

## Mit true, false, if-then-else alles darstellbar

$\neg x \equiv \text{if } x \text{ then false else true}$

$x \wedge y \equiv \text{if } x \text{ then } y \text{ else false}$

$x \vee y \equiv \text{if } x \text{ then true else } y$

## Damit

$\neg x \equiv x \text{ false true}$

$x \wedge y \equiv x y \text{ false}$

$x \vee y \equiv x \text{ true } y$

# Darstellung Paaren im $\lambda$ -Kalkül

## Paare

$mkpair(x, y) =_{\text{def}} \lambda b. b x y$

$fst(p) =_{\text{def}} p \text{ true}$

$snd(p) =_{\text{def}} p \text{ false}$

## Ähnlich für

- Tupel
- Listen
- etc.

# Darstellung Paaren im $\lambda$ -Kalkül

## Paare

$$mkpair(x, y) =_{\text{def}} \lambda b. b x y$$
$$fst(p) =_{\text{def}} p \text{ true}$$
$$snd(p) =_{\text{def}} p \text{ false}$$

## Ähnlich für

- Tupel
- Listen
- etc.

# Darstellung Paaren im $\lambda$ -Kalkül

## Paare

$$mkpair(x, y) =_{\text{def}} \lambda b. b x y$$

$$fst(p) =_{\text{def}} p \text{ true}$$

$$snd(p) =_{\text{def}} p \text{ false}$$

## Ähnlich für

- Tupel
- Listen
- etc.

# Darstellung natürlicher Zahlen im $\lambda$ -Kalkül

## Natürliche Zahlen

$$0 =_{\text{def}} \lambda f. \lambda x. x$$

$$1 =_{\text{def}} \lambda f. \lambda x. f x$$

$$2 =_{\text{def}} \lambda f. \lambda x. f (f x)$$

$$3 =_{\text{def}} \lambda f. \lambda x. f (f (f x))$$

⋮

$$n =_{\text{def}} \lambda f. \lambda x. \underbrace{f(\dots (f x)\dots)}_{n \text{ mal}}$$

## Operationen auf natürlichen Zahlen

$$\mathit{succ}(n) =_{\text{def}} \lambda g. \lambda y. n \ g \ (g \ y)$$

$$+(n, m) =_{\text{def}} n \ \mathit{succ} \ m$$

$$*(n, m) =_{\text{def}} n \ (m \ \mathit{succ}) \ 0$$

$$\mathit{iszero}(n) =_{\text{def}} n \ (\lambda b. \mathit{false}) \ \mathit{true}$$

## Der Y-Operator

$$Y =_{\text{def}} \lambda F. (\lambda y. F(y y)) (\lambda x. F(xx))$$

## Y ist Fixpunkt-Operator

$$f(Y f) = Y f$$

für alle  $f$

## Der Y-Operator

$$Y =_{\text{def}} \lambda F. (\lambda y. F(y y)) (\lambda x. F(xx))$$

## Y ist Fixpunkt-Operator

$$f(Yf) = Yf$$

für alle  $f$

## Beispiel

$$fak\ n =_{\text{def}}\ (Y\ F)\ n$$

mit

$$F =_{\text{def}}\ \lambda f.\lambda n.\text{if iszero}(n)\ \text{then } 1\ \text{else } n * (f(n-1))$$

## Allgemein

Der Y-Operator erlaubt es, beliebige  $\mu$ -rekursive Funktionen im  $\lambda$ -Kalkül zu definieren.

## Beispiel

$$fak\ n =_{\text{def}}\ (Y\ F)\ n$$

mit

$$F =_{\text{def}}\ \lambda f.\lambda n.\text{if iszero}(n)\ \text{then } 1\ \text{else } n * (f(n-1))$$

## Allgemein

Der Y-Operator erlaubt es, beliebige  $\mu$ -rekursive Funktionen im  $\lambda$ -Kalkül zu definieren.