

# Tutorium 05: $\lambda$ -Kalkül

---

Paul Brinkmeier

29. November 2022

Tutorium Programmierparadigmen am KIT

# Übungsblätter

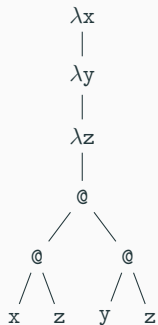
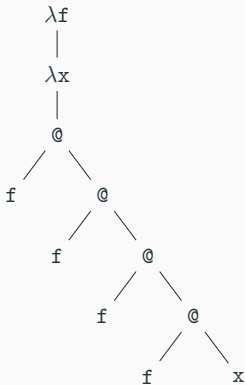
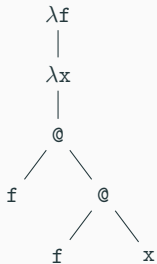
---

# Wiederholung

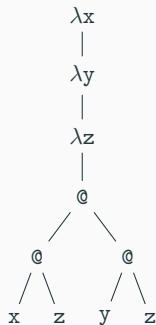
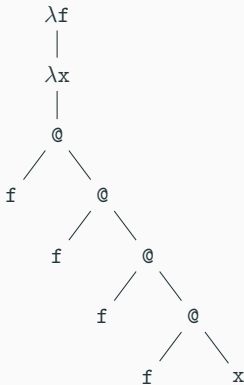
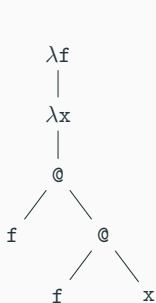
---

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

# $\lambda$ -Terme sind Bäume

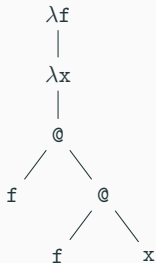


# $\lambda$ -Terme sind Bäume

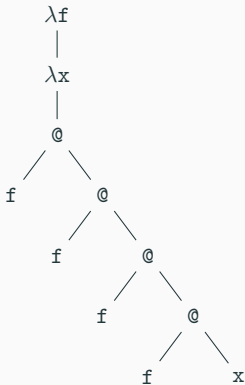


$\lambda f. \lambda x. f (f x)$

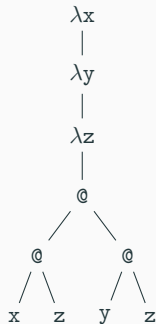
# $\lambda$ -Terme sind Bäume



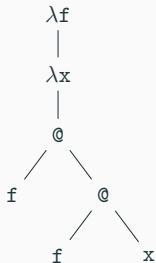
$\lambda f. \lambda x. f (f x)$



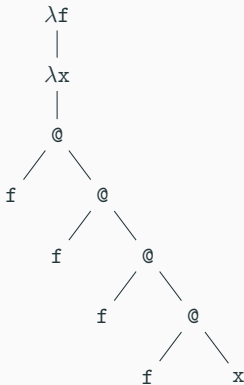
$\lambda f. \lambda x. f (f (f (f x)))$



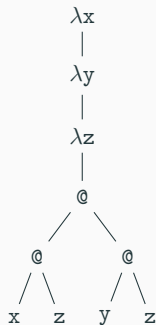
# $\lambda$ -Terme sind Bäume



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



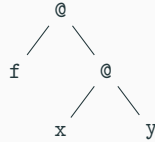
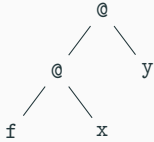
$\lambda f. \lambda x. f (f (f (f x)))$



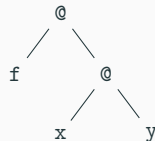
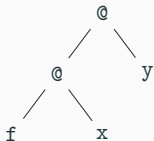
$\lambda x. \lambda y. \lambda z. x z (y z)$



# Klammerung von Lambda-Termen



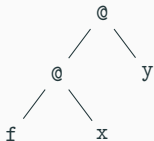
# Klammerung von Lambda-Termen



$$(f\ x)\ y = f\ x\ y$$

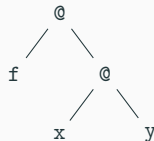
„ $f$  angewendet auf  $x$  und  $y$ “

# Klammerung von Lambda-Termen



$$(f\ x)\ y = f\ x\ y$$

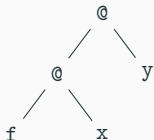
„ $f$  angewendet auf  $x$  und  $y$ “



$$f\ (x\ y) \neq f\ x\ y$$

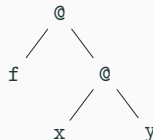
„ $f$  angewendet auf das Ergebnis von  $x\ y$ “

# Klammerung von Lambda-Termen



$$(f\ x)\ y = f\ x\ y$$

„ $f$  angewendet auf  $x$  und  $y$ “



$$f\ (x\ y) \neq f\ x\ y$$

„ $f$  angewendet auf das Ergebnis von  $x\ y$ “

- Funktionsanwendung ist linksassoziativ (linksgeklammert)
- $\leadsto$  Wie bei Haskell!

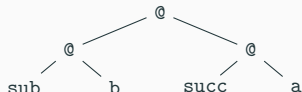
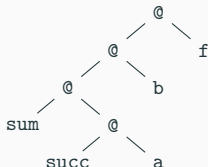
# Klammerungsbeispiel

```
sum a b f
  | a > b = 0
  | otherwise = f a + sum (a+1) b f
-- Bspw. sum 1 10 id == 55
```

$sum = \lambda a. \lambda b. \lambda f. ifZero (sub\ b\ (succ\ a))$

$c_0$

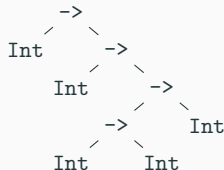
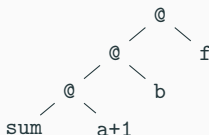
$(add\ (f\ a)\ (sum\ (succ\ a)\ b\ f))$



# Klammerung von Funktionstypen

```
sum a b f
  | a > b = 0
  | otherwise = f a + sum (a+1) b f
-- Bspw. sum 1 10 id == 55
```

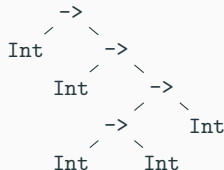
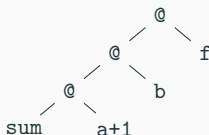
```
sum :: Int -> Int -> (Int -> Int) -> Int
sum :: Int -> (Int -> ((Int -> Int) -> Int))
```



# Klammerung von Funktionstypen

```
sum a b f
  | a > b = 0
  | otherwise = f a + sum (a+1) b f
-- Bspw. sum 1 10 id == 55
```

```
sum :: Int -> Int -> (Int -> Int) -> Int
sum :: Int -> (Int -> ((Int -> Int) -> Int))
```



- Auch Typterme sind Bäume
- *Funktionstypen* sind rechtsassoziativ

# Church-Zahlen

---



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

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

$$\text{succ} = \lambda n. \lambda s. \lambda z. s (n s z)$$

# **Auswertungsstrategien**

---

$$\begin{array}{c} \text{Redex 1} \\ \underbrace{\hspace{1.5cm}} \\ \text{Redex 2} \\ \underbrace{\hspace{1.5cm}} \\ \text{succ} \ (\text{succ} \ c_0) \end{array}$$

mit

$$\begin{aligned} c_0 &= \lambda s. \lambda z. z \\ \text{succ} &= \lambda n. \lambda s. \lambda z. s \ (n \ s \ z) \end{aligned}$$

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

$$\begin{aligned} & \underline{\text{succ}} (\underline{\text{succ}} c_0) \\ \Rightarrow_{\beta} & \lambda s. \lambda z. s ((\underline{\text{succ}} c_0) s z) \\ \Rightarrow_{\beta} & \lambda s. \lambda z. s (\underline{((\lambda s. \lambda z. s (\underline{c_0} s z))) s z}) \\ \Rightarrow_{\beta}^2 & \lambda s. \lambda z. s (s (\underline{c_0} s z)) \\ \Rightarrow_{\beta}^2 & \lambda s. \lambda z. s (s z) \not\Rightarrow \end{aligned}$$

Normalreihenfolge: Linkester Redex zuerst.

$$\begin{array}{c} \underline{\text{succ}} (\underline{\text{succ}} c_0) \\ \Rightarrow_{\beta} \quad \lambda s. \lambda z. s ((\underline{\text{succ}} c_0) s z) \not\Rightarrow_{\text{CbN}} \end{array}$$

Call-by-Name: Linkester Redex zuerst, aber:

- Funktionsinhalte werden nicht weiter reduziert
- $\leadsto$  Betrachte nur Redexe, die nicht von einem  $\lambda$  umgeben sind
- So funktioniert auch Laziness in Haskell (grob)

$$\begin{aligned} & \underline{\text{succ}} (\underline{\text{succ}} c_0) \\ \Rightarrow_{\beta} & \underline{\text{succ}} (\lambda s. \lambda z. s (\underline{c_0} s z)) \\ \Rightarrow_{\beta} & \lambda s. \lambda z. \underline{(\lambda s. \lambda z. s (\underline{c_0} s z))} s z \not\Rightarrow_{\text{CbV}} \end{aligned}$$

Call-by-Value: Linkester Redex zuerst, aber:

- Funktionsinhalte werden nicht weiter reduziert
- $\leadsto$  Betrachte nur Redexe, die nicht von einem  $\lambda$  umgeben sind
- Berechne Argumente vor dem Einsetzen
- $\leadsto$  Betrachte nur Redexe, deren Argument *unter CbV* nicht weiter reduziert werden muss



- Auswertungsstrategien (von lässig nach streng):
  - *Volle  $\beta$ -Reduktion*
  - *Normalreihenfolge*
  - *Call-by-Name*
  - *Call-by-Value*
- Datenstrukturen:
  - *Church-Booleans*
  - *Church-Zahlen*
  - *Church-Listen*
- Rekursion durch *Y-Kombinator*

# Klausuraufgaben zum $\lambda$ -Kalkül

---

$$S = \lambda x. \lambda y. \lambda z. x z (y z)$$

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

$$I = \lambda x. x$$

- SKI-Kalkül kann alles, was  $\lambda$ -Kalkül auch kann, allein mit den Kombinatoren  $S$ ,  $K$  und  $I$
- Definiere  $U = \lambda x. x S K$
- Aufgabe: Beweise, dass man  $S$ ,  $K$  und  $I$  durch  $U$  darstellen kann:

$$S = \lambda x. \lambda y. \lambda z. x z (y z)$$

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

$$I = \lambda x. x$$

- SKI-Kalkül kann alles, was  $\lambda$ -Kalkül auch kann, allein mit den Kombinatoren  $S$ ,  $K$  und  $I$
- Definiere  $U = \lambda x. x S K$
- Aufgabe: Beweise, dass man  $S$ ,  $K$  und  $I$  durch  $U$  darstellen kann:

- $U U x \xRightarrow{?} x$
- $U (U (U U)) = U (U I) \xRightarrow{?} K$
- $U (U (U (U U))) = U K \xRightarrow{?} S$

$$\text{pair} = \lambda a. \lambda b. \lambda f. f \ a \ b$$

$$\text{fst} = \lambda p. p \ (\lambda x. \lambda y. x)$$

$$\text{snd} = \lambda p. p \ (\lambda x. \lambda y. y)$$

$$\text{fst} \ (\text{pair} \ a \ b) = \quad \quad \quad a$$

$$\text{snd} \ (\text{pair} \ a \ b) = \quad \quad \quad b$$

- Schreibe `curry` und `uncurry`, sodass:
  - $(\text{curry } f) \ a \ b = f \ (\text{pair } a \ b)$
  - $(\text{uncurry } g) \ (\text{pair } a \ b) = g \ a \ b$

$$\text{nil} = \lambda n. \lambda c. n$$

$$\text{cons} = \lambda x. \lambda xs. \lambda n. \lambda c. (c \times xs)$$

- Schreibe *head* und *tail*, sodass:

- $\text{head} (\text{cons } A \ B) \xRightarrow{*} A$
- $\text{tail} (\text{cons } A \ B) \xRightarrow{*} B$

$$\text{nil} = \lambda n. \lambda c. n$$

$$\text{cons} = \lambda x. \lambda xs. \lambda n. \lambda c. (c \ x \ xs)$$

- Schreibe *head* und *tail*, sodass:

- $\text{head} (\text{cons } A \ B) \xRightarrow{*} A$

- $\text{tail} (\text{cons } A \ B) \xRightarrow{*} B$

- Schreibe *replicate*, sodass:

- $\text{replicate } c_n \ A = \underbrace{\text{cons } A \ (\text{cons } A \ \dots (\text{cons } A \ \text{nil}))}_{n \text{ mal}}$

- Erinnerung:  $c_n \ f \ x = \underbrace{f \ (f \ \dots (f \ x))}_{n \text{ mal}}$

$$\text{nil} = \lambda n. \lambda c. n$$

$$\text{cons} = \lambda x. \lambda xs. \lambda n. \lambda c. (c \ x \ xs)$$

- Schreibe *head* und *tail*, sodass:
  - $\text{head} (\text{cons } A \ B) \xRightarrow{*} A$
  - $\text{tail} (\text{cons } A \ B) \xRightarrow{*} B$
- Schreibe *replicate*, sodass:
  - $\text{replicate } c_n \ A = \underbrace{\text{cons } A \ (\text{cons } A \ \dots (\text{cons } A \ \text{nil}))}_{n \text{ mal}}$
- Erinnerung:  $c_n \ f \ x = \underbrace{f \ (f \ \dots (f \ x))}_{n \text{ mal}}$
- Werte aus:  $\text{replicate } c_3 \ A \xRightarrow{*} ?$