Lambda

Contents

Theorie

Aufbau

Ein λ-Ausdruck **E** ist **:**

1) Ein Variablen-Name

x, z, y, . . ., a, b, . . .

2) Eine Lambda-Abstraktion

λ x. E mit E Lambda-Ausdruck

3) Eine Applikation

E_1 E_2 mit E_1 und E_2 Lambda-Ausdrücke

Äquivalenz

Umbenennung

α-Äquivalenz

Namen gebundener Variablen

- dienen letztlich nur der Dokumentation
- entscheidend sind die Bindungen

α-Äquivalenz

t_1 und t_2 heißen α-äquivalent ($t_1 \stackrel{\alpha}{=} t_2$), wenn t_1 in t_2 durch konsistente Umbenennung der λ-gebundenen Variablen überführt werden kann.

Beispiele:

$$\lambda x.\ x \stackrel{\alpha}{=} \lambda y.\ y$$

$$\lambda x.\ (\lambda z.\ f\ (\lambda y.\ z\ y)\ x) \stackrel{\alpha}{=} \lambda y.\ (\lambda x.\ f\ (\lambda z.\ x\ z)\ y)$$

aber

$$\lambda x.\ (\lambda z.\ f\ (\lambda y.\ z\ y)\ x) \stackrel{\alpha}{\neq} \lambda x.\ (\lambda z.\ g\ (\lambda y.\ z\ y)\ x)$$

$$\lambda x.\ (\lambda z.\ f\ (\lambda y.\ z\ y)\ x) \stackrel{\alpha}{\neq} \lambda z.\ (\lambda z.\ f\ (\lambda y.\ z\ y)\ z)$$

Semantik

η-Äquivalenz

Extensionalitäts-Prinzip:

- Zwei Funktionen sind gleich, falls Ergebnis gleich für alle Argumente

η-Äquivalenz

Terme $\lambda x.\ f\ x$ und f heißen η-äquivalent ($\lambda x.\ f\ x \stackrel{\eta}{=} f$), falls x nicht freie Variable von f

Beispiele:

$$\lambda x.\ \lambda y.\ \underline{f\ z\ x}\ y \stackrel{\eta}{=} \lambda x.\ f\ z\ x$$

$$f\ z \stackrel{\eta}{=} \lambda x.\ \underline{f\ z}\ x$$

$$\lambda x.\ x \stackrel{\eta}{=} \lambda x.\ \underline{(\lambda x.\ x)}\ x$$

aber:

$$\lambda x.\ \underline{g\ x}\ x \stackrel{\eta}{\neq} g\ x$$

Haskell-Beispiel: „Rechts-Kürzen“ von Variablen

```
f x y = g (x+42) y   ⟺   f x = \y -> g (x+42) y   ⟺   f x = g (x+42)
```

Church

Zahlen

Church-Zahlen

Eine (natürliche) Zahl drückt aus, wie oft die Funktion s angewendet wird.

$$c_0 = \lambda s.\ \lambda z.\ z$$
$$c_1 = \lambda s.\ \lambda z.\ s\ z$$
$$c_2 = \lambda s.\ \lambda z.\ s\ (s\ z)$$
$$c_3 = \lambda s.\ \lambda z.\ s\ (s\ (s\ z))$$
$$\vdots$$
$$c_n = \lambda s.\ \lambda z.\ s^n\ z$$

Nachfolgerfunktion:

$$succ = \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)$$

n Church-Zahl, d.h. von der Form $\lambda s.\ \lambda z.\ \ldots$

$$succ\ c_2 = \underline{(\lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z))\ (\lambda s.\ \lambda z.\ s\ (s\ z))}$$
$$\Rightarrow \lambda s.\ \lambda z.\ s\ (\underline{(\lambda s.\ \lambda z.\ s\ (s\ z))\ s}\ z)$$
$$\Rightarrow \lambda s.\ \lambda z.\ s\ (\underline{(\lambda z.\ s\ (s\ z))\ z})$$
$$\Rightarrow \lambda s.\ \lambda z.\ s\ (s\ (s\ z)) = c_3$$

Wichtige Eigenschaft:

```
c_3 (fun a) nil = (\s.\z.s(s(s z))) (fun a) nil = fun a (fun a (fun a nil))
c_3 fun2 x = (\s.\z.s(s(s z))) fun2 x = fun (fun (fun x))
```

Booleans

Kodierung boolescher Werte

Church-Booleans

Idee: Kodiere boolsche Werte als Fallunterscheidungs-Funktionen.

`True` wird zu $c_{true} = \lambda t.\ \lambda f.\ t$

`False` wird zu $c_{false} = \lambda t.\ \lambda f.\ f$

- **if** b **then** x **else** y wird zu $b\ x\ y$

  if `True` **then** x **else** y ergibt:

$$c_{true}\ x\ y \Rightarrow (\lambda f.\ x)\ y \Rightarrow x$$

- b_1 `&&` b_2 ist äquivalent zu **if** b_1 **then** b_2 **else** `False`

$\Rightarrow$ b_1 `&&` b_2 wird zu $b_1\ b_2\ c_{false}$

`True && True` ergibt:

$$\begin{aligned} & c_{true}\ c_{true}\ (\lambda t.\ \lambda f.\ f) \\ \Rightarrow\ & (\lambda f.\ (\lambda t.\ \lambda f.\ t))\ (\lambda t.\ \lambda f.\ f) \\ \Rightarrow\ & \lambda t.\ \lambda f.\ t = c_{true} \end{aligned}$$

c_true a b = a

c_false a b = b

Methoden

Arithmetische Operationen

$$\begin{array}{lll} \text{Addition:} & \mathit{plus} & = \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z) \\ \text{Multiplikation:} & \mathit{times} & = \lambda m.\ \lambda n.\ \lambda s.\ n\ (m\ s) \\ & & \stackrel{\eta}{=} \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ n\ (m\ s)\ z \\ \text{Potenzieren:} & \mathit{exp} & = \lambda m.\ \lambda n.\ n\ m \\ & & \stackrel{\eta}{=} \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ n\ m\ s\ z \end{array}$$

Idee zu *exp*:

$$\begin{aligned} \mathit{exp}\ c_m\ c_n \stackrel{2}{\Rightarrow}\ & c_n\ c_m = (\lambda s.\ \lambda z.\ s^n\ z)\ c_m \\ \Rightarrow\ & \lambda z.\ (c_m)^n\ z \\ \stackrel{\alpha}{=}\ & \lambda s.\ c_m\ (\cdots (c_m\ (c_m\ (c_m\ s)))\cdots) \\ \text{(Regel für } \mathit{times}\text{)} \stackrel{\alpha\beta\eta}{=}\ & \lambda s.\ c_m\ (\cdots (c_m\ (c_{(m\cdot m)}\ s)))\cdots) \\ \text{(Induktion)} \stackrel{\alpha\beta\eta}{=}\ & \lambda s.\ c_{(m\cdot m\cdot m\cdots m)}\ s \stackrel{\eta}{=} c_{m^n} \end{aligned}$$

sub

sub c2 c1 = c2 - c1

```
sub = \m. \n. n pred m
```

isZero

$$isZero = \lambda n.\ n\ (\lambda x.\ c_{false})\ c_{true}$$

$$\begin{aligned} isZero\ c_0 &= (\lambda n.\ n\ (\lambda x.\ c_{false})\ c_{true})\ (\lambda s.\ \lambda z.\ z) \\ &\Rightarrow (\lambda s.\ \lambda z.\ z)\ (\lambda x.\ c_{false})\ c_{true} \\ &\Rightarrow^2 c_{true} \end{aligned}$$

$$\begin{aligned} isZero\ c_2 &= (\lambda n.\ n\ (\lambda x.\ c_{false})\ c_{true})\ (\lambda s.\ \lambda z.\ s\ (s\ z)) \\ &\Rightarrow (\lambda s.\ \lambda z.\ s\ (s\ z))\ (\lambda x.\ c_{false})\ c_{true} \\ &\Rightarrow^2 (\lambda x.\ c_{false})\ ((\lambda x.\ c_{false})\ c_{true}) \\ &\Rightarrow c_{false} \end{aligned}$$

isZero cx = ctrue falls cx = 0 oder cfalse falls cx != 0

if Then Else

Aufbau: Funktion, die c_true oder c_false ergibt (z.B isZero c_n), dann True-Case und False-Case.

```
(isZero arg) True_Term False_term
```

equal

Hierbei muss man jedoch beachten, dass Church-Zahlen
natürliche Zahlen sind, und Subtraktion auf natürlichen Zahlen “saturierend” ist, d.h. $0-n = 0$.
Daher gilt $n = m$ nur dann, wenn sowohl $n-m = 0$ und $m-n = 0$ gelten.

```
eq = \n. \m. (isZero (sub n m)) (isZero (sub m n)) cfalse
--> falls (isZero n - m) then (isZero m - n) else false
```

Selbstapplikation

Divergenz

$$\underline{(\lambda x.\ x)}\ (\lambda x.\ x) \quad \Rightarrow (\lambda x.\ x) \quad \text{in Normalform.}$$

$$\begin{aligned} \underline{(\lambda x.\ x\ x)}\ (\lambda x.\ x\ x) &\Rightarrow \underline{(\lambda x.\ x\ x)}\ (\lambda x.\ x\ x) \\ &\Rightarrow \underline{(\lambda x.\ x\ x)}\ (\lambda x.\ x\ x) \\ &\Rightarrow \ldots \end{aligned}$$

Beachte: Funktionsanwendung ist linksassoziativ!

$$\begin{aligned} \underline{(\lambda x.\ x\ x\ x)}\ (\lambda x.\ x\ x\ x) &\Rightarrow \underline{(\lambda x.\ x\ x\ x)}\ (\lambda x.\ x\ x\ x)(\lambda x.\ x\ x\ x) \\ &\Rightarrow \underline{(\lambda x.\ x\ x\ x)}\ (\lambda x.\ x\ x\ x)(\lambda x.\ x\ x\ x)(\lambda x.\ x\ x\ x) \\ &\Rightarrow \ldots \end{aligned}$$

Y-Kombinator

Mit der Selbstapplikation des Lambdas mit zwei Argumente lässt sich die Rekursion implementieren.

Rekursionsoperator

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

$$\begin{aligned}
Y\ f &= \underline{(\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))}\ f \\
&\Rightarrow \underline{(\lambda x.\ f\ (x\ x))}\ (\lambda x.\ f\ (x\ x)) \\
&\Rightarrow f\ ((\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))) \Leftarrow f\ (Y\ f)
\end{aligned}$$

also $\quad f\ (Y\ f) \stackrel{\beta}{=} Y\ f$

d.h. $\quad$ **Y *f* ist Fixpunkt von *f*.**

Turing-Mächtigkeit

Der untypisierte λ-Kalkül ist turing-mächtig.

Rekursion

1) Implementiere rekursive funktion wie immer, mit einem Abbruch-Bedingung
2) Erstelle eine erweiterte Funktion G als erste Funktion + Lamda für sich selbst
3) Rekursive Funktion = Y G

Beispiel: Fakultät im λ-Kalkül

```
fak =          \ n -> if isZero n then 1 else n * fak (n - 1)
G   = \ fak -> \ n -> if isZero n then 1 else n * fak (n - 1)
```

$$\begin{aligned}
Y &= \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)) \\
G &= \lambda fak.\ \lambda n.\ (isZero\ n)\ c_1\ (times\ n\ (fak\ (sub\ n\ c_1))) \\
fak &= Y\ G
\end{aligned}$$

$$\begin{aligned}
fak\ c_2 = Y\ G\ c_2 &\Rightarrow (\lambda x.\ G\ (x\ x))\ (\lambda x.\ G\ (x\ x))\ c_2 \\
&\Rightarrow G\ ((\lambda x.\ G\ (x\ x))\ (\lambda x.\ G\ (x\ x)))\ c_2 \\
&\stackrel{2}{\Rightarrow} (isZero\ c_2)\ c_1\ (times\ c_2\ ((\lambda x.\ G\ (x\ x))\ (\lambda x.\ G\ (x\ x))\ (sub\ c_2\ c_1))) \\
&\stackrel{*}{\Rightarrow} times\ c_2\ ((\lambda x.\ G\ (x\ x))\ (\lambda x.\ G\ (x\ x))\ (sub\ c_2\ c_1)) \\
&\stackrel{*}{\Rightarrow} times\ c_2\ (\overbrace{(\lambda x.\ G\ (x\ x))\ (\lambda x.\ G\ (x\ x))}^{Y\ G \Rightarrow}\ c_1) \\
&\stackrel{3}{\Rightarrow} times\ c_2\ ((isZero\ c_1)\ c_1\ (times\ c_1\ ((\lambda x.\ G(x\ x))(\lambda x.\ G(x\ x))\ (sub\ c_1\ c_1)))) \\
&\stackrel{*}{\Rightarrow} times\ c_2\ (times\ c_1\ ((\lambda x.\ G\ (x\ x))\ (\lambda x.\ G\ (x\ x))\ (sub\ c_1\ c_1))) \\
&\stackrel{*}{\Rightarrow} times\ c_2\ (times\ c_1\ ((isZero\ c_0)\ c_1\ \ldots)) \stackrel{*}{\Rightarrow} c_2
\end{aligned}$$

Auswertungstrategien

Auswertung

Auswertung von λ-Termen

Redex Ein λ-Term der Form $(\lambda x.\ t_1)\ t_2$ heißt Redex.

β-Reduktion β-Reduktion entspricht der Auswertung der Funktionsanwendung auf einem Redex:

$$(\lambda x.\ t_1)\ t_2 \Rightarrow t_1\ [x \mapsto t_2]$$

Substitution $t_1\ [x \mapsto t_2]$ erhält man aus dem Term t_1, wenn man alle <u>freien</u> Vorkommen von x durch t_2 ersetzt.

Normalform Ein Term, der nicht weiter reduziert werden kann, heißt in Normalform.

Beispiele:

$$\underline{(\lambda x.\ x)}\ y \Rightarrow \quad x\ [x \mapsto y] \quad = y$$

$$\underline{(\lambda x.\ x\ (\lambda x.\ x))}\ (y\ z) \Rightarrow (x\ (\lambda x.\ x))\ [x \mapsto y\ z] = (y\ z)\ (\lambda x.\ x)$$

Wert

Werte in Haskell:

- Primitive Werte: `2`, `True`
- Funktionen: `(\x -> x)`, `(&&)`, `(\x -> (\y -> y+y) x)`

Werte im λ-Kalkül:

- Abstraktionen: $c_2 = \lambda s.\ \lambda z.\ s\ (s\ z)$, $c_{true} = \lambda t.\ \lambda f.\ t$
  $\lambda x.\ x$, $\quad \lambda b_1.\ \lambda b_2.\ b_1\ b_2\ (\lambda t.\ \lambda f.\ f)$,
  $\lambda x.\ (\lambda y.\ \mathit{plus}\ y\ y)\ x$

Auswertungsstrategie: Keine weitere Reduzierung von Werten

$\Rightarrow$ Reduziere keine Redexe unter Abstraktionen (umgeben von λ): call-by-name, call-by-value

So, what is a value? In the pure lambda calculus, any abstraction is a value. Intuitively, a value is an expressionthat can not be reduced/executed/simplified any further.

Call-By-Name

Call-By-Name

Call-by-name Reduziere linkesten äußersten Redex

- Aber nicht falls von einem λ umgeben

$$\underline{(\lambda y.\ (\lambda x.\ y\ (\lambda z.\ z)\ x))}\ ((\lambda x.\ x)\ (\lambda y.\ y))$$
$$\Rightarrow (\lambda x.\ (\underline{(\lambda x.\ x)}\ (\lambda y.\ y))\ (\lambda z.\ z)\ x) \not\Rightarrow$$

Intuition: Reduziere Argumente erst, wenn benötigt

Auswertung in Haskell: Lazy-Evaluation = call-by-name + sharing

- Standard-Auswertungsstrategie für Funktionen/Konstruktoren

```
listOf x = x : listOf x
3 : listOf 3 ⇏

      (div 1 0):(6:[])    ⇏
tail ((div 1 0):(6:[]))  ⇒ 6:[] ⇏
```

Nur die linke **\y** wird reduziert, da es keine äußere Lambda gibt.
Innere \x wird nicht reduziert, da es noch äußere \x gibt.
Äußere \x kann nicht reduziert werden - kein Redex

Call-By-Value

Call-By-Value

Call-by-value Reduziere linkesten Redex

- der nicht von einem λ umgeben
- und dessen Argument ein Wert ist

$$\underline{(\lambda y.\ (\lambda x.\ y\ (\lambda z.\ z)\ x))}\ (\underline{(\lambda x.\ x)}\ (\lambda y.\ y))$$
$$\Rightarrow \underline{(\lambda y.\ (\lambda x.\ y\ (\lambda z.\ z)\ x))}\ (\lambda y.\ y)$$
$$\Rightarrow (\lambda x.\ \underline{(\lambda y.\ y)}\ (\lambda z.\ z)\ x) \not\Rightarrow$$

Intuition: Argumente vor Funktionsaufruf auswerten
Auswertungsstrategie vieler Sprachen: Java, C, Scheme, ML, . . .

Arithmetik in Haskell: Auswertung by-value

```
prodOf x = x * prodOf x
3 + prodOf 3 ⇒ 3 + 3 * prodOf 3 ⇒ 3 + 3 * (3 * prodOf 3) ⇒...

((div 1 0)*6)*0   ⇏
((div 2 2)*6)*0   ⇒ (1*6)*0 ⇒ 6*0 ⇒ 0
```

Linkeste Redex, der keine äußere Lambda hat, und Argument eine Value ist --> **nicht vereinfachbar** ist.

Normalreihenfolge

Normalreihenfolge Immer der linkeste äußerste Redex wird reduziert

$$\underline{(\lambda x.\ x)}\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z))$$
$$\Rightarrow \underline{(\lambda x.\ x)}\ (\lambda z.\ (\lambda x.\ x)\ z)$$
$$\Rightarrow \lambda z.\ \underline{(\lambda x.\ x)}\ z$$
$$\Rightarrow \lambda z.\ z \not\Rightarrow$$

Einfach immer der linkeste Redex reduzieren

Beispiel

```
(\x. x) ((\y. y) z)
         ------- ~ CBV
------- ~~~~~~~~~~~ NRF, CBN
```

Aufgaben

Call-By-Name/Value

```
(\t.\f. f) ((\y. (\x. x x) (\x. x x)) ((\x. x) (\x. x))) (\t.\f. f)
                                       ------   ~~~~~               CBV
 --------   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~               CBN, NRF


(\y. (\z. (\x. x) (\x. x) z) y)
      ---------------------- ~     NRF
kein CBN und kein CBV  (da alles von Lambda umgeben)
```

Unendliche Reduktion

2. (a) Führen Sie 4 β-Reduktionsschritte für folgenden Term aus:

$$\omega' = (\lambda x.\ x\ x)\ (\lambda y.\ m\ (y\ y)\ n)$$

Hinweis: Sparen Sie sich Schreibarbeit, indem Sie sich Abkürzungen für wiederholte Subterme definieren.

Beispiellösung: Sei $\theta = \lambda y.\ m\ (y\ y)\ n$:

$$\begin{aligned}
\omega' = \ & (\lambda x.\ x\ x)\ (\lambda y.\ m\ (y\ y)\ n) \\
\Rightarrow\ & \theta\ \theta\ =\ (\lambda y.\ m\ (y\ y)\ n)\ (\lambda y.\ m\ (y\ y)\ n) \\
\Rightarrow\ & m\ (\theta\ \theta)\ n \\
\Rightarrow\ & m\ (m\ (\theta\ \theta)\ n)\ n \\
\Rightarrow\ & m\ (m\ (m\ (\theta\ \theta)\ n)\ n)\ n
\end{aligned}$$

Zähler (SS16 A4)

Im λ-Kalkül lassen sich Nachrichten – ähnlich wie Church-Booleans – darstellen als:

$$\begin{aligned}\mathsf{inc} &= \lambda i.\ \lambda a.\ \lambda g.\ i \\ \mathsf{add} &= \lambda i.\ \lambda a.\ \lambda g.\ a \\ \mathsf{get} &= \lambda i.\ \lambda a.\ \lambda g.\ g\end{aligned}$$

Weiterhin sei definiert:

$$\begin{aligned}\mathsf{react} = \lambda n.\ \lambda x.\ \lambda m.\ m\ &(n\ (\mathsf{succ}\ x)) \\ &(\lambda y.\ n\ (\mathsf{plus}\ x\ y)) \\ &x\end{aligned}$$

Reduktion zeigen

$$\begin{array}{lll}
\mathsf{react}\ n\ c_x\ \mathsf{inc} & \Rightarrow^3 & \mathsf{inc}\ (n\ (\mathsf{succ}\ c_x))\ (\lambda y.\ n\ (\mathsf{plus}\ c_x\ y))\ c_x \\
& \Rightarrow^3 & n\ (\mathsf{succ}\ c_x) \\
& \Rightarrow^* & n\ c_{x+1} \\
\mathsf{react}\ n\ c_x\ \mathsf{add}\ c_y & \Rightarrow^3 & \mathsf{add}\ (n\ (\mathsf{succ}\ c_x))\ (\lambda y.\ n\ (\mathsf{plus}\ c_x\ y))\ c_x\ c_y \\
& \Rightarrow^3 & (\lambda y.\ n\ (\mathsf{plus}\ c_x\ y))\ c_y \\
& \Rightarrow & n\ (\mathsf{plus}\ c_x\ c_y) \\
& \Rightarrow^* & n\ c_{x+y} \\
\mathsf{react}\ n\ c_x\ \mathsf{get} & \Rightarrow^3 & \mathsf{get}\ (n\ (\mathsf{succ}\ c_x))\ (\lambda y.\ n\ (\mathsf{plus}\ c_x\ y))\ c_x \\
& \Rightarrow^3 & c_x
\end{array}$$

„new" Term mit Y Kombinator

(b) Hätten wir einen λ-Term new, der neue Zähler-Objekte konstruiert, so ließe sich ein Zähler mit Stand c_x einfach darstellen als: [3 Punkte]

$$z_x = \mathsf{react}\ \mathsf{new}\ c_x$$

Geben Sie also einen λ-Term new an, sodass[1] für Church-Zahlen c_x:

$$\mathsf{new}\ c_x \Rightarrow^* \mathsf{react}\ \mathsf{new}\ c_x$$

(b) Sei

$$\mathsf{New} = \lambda n.\ \lambda x.\ \mathsf{react}\ n\ x$$

Dann wähle new als Resultat der Reduktion:

$$\mathsf{Y}\ \mathsf{New} \Rightarrow \underbrace{(\lambda x.\ \mathsf{New}\ (x\ x))\ (\lambda x.\ \mathsf{New}\ (x\ x))}_{=:\mathsf{new}}$$

Alternativ: Für das Verhalten wie in Fußnote[1] reicht schon:

$$\mathsf{new} = \mathsf{Y}\ \mathsf{New}$$

oder einfach:

$$\mathsf{new} = \mathsf{Y}\ \mathsf{react}$$

(c)

$$\begin{array}{lllll}
z_x\ \mathsf{inc} & \underbrace{\Rightarrow^*}_{\text{per a)}} & \mathsf{new}\ c_{x+1} & \underbrace{\Rightarrow^*}_{\text{per b)}} & z_{x+1} \\
z_x\ \mathsf{add}\ c_y & \underbrace{\Rightarrow^*}_{\text{per a)}} & \mathsf{new}\ c_{x+y} & \underbrace{\Rightarrow^*}_{\text{per b)}} & z_{x+y} \\
z_x\ \mathsf{get} & \underbrace{\Rightarrow^*}_{\text{per a)}} & c_x & &
\end{array}$$

Listen

Neben natürlichen Zahlen und Booleans lassen sich auch Listen im λ-Kalkül definieren. Seien also:

$$\begin{aligned} \texttt{nil} &= \lambda\texttt{n}.\ \lambda\texttt{c}.\ \texttt{n} \\ \texttt{cons} &= \lambda\texttt{x}.\ \lambda\texttt{xs}.\ \lambda\texttt{n}.\ \lambda\texttt{c}.\ \texttt{c x xs} \end{aligned}$$

Hierbei ist `nil` die Darstellung der leeren Liste, und `cons x xs` die Darstellung der Liste mit erstem Element `x` und Listenrest `xs` (vgl. Haskell `x:xs`).

(a) Geben Sie Definitionen für λ-Ausdrücke `head` und `tail` an, so dass für beliebige A, B gilt: [4 Punkte]

$$\begin{aligned} \texttt{head (cons } A\ B\texttt{)} &\Rightarrow^* A \\ \texttt{tail (cons } A\ B\texttt{)} &\Rightarrow^* B \end{aligned}$$

Das Verhalten bei Übergabe von `nil` als Argument ist Ihnen überlassen.

Beispiellösung:
$\texttt{head} = \lambda\texttt{l}.\ \texttt{l}\ c_{false}\ (\lambda\texttt{x}.\ \lambda\texttt{xs}.\ \texttt{x})$
$\texttt{tail} = \lambda\texttt{l}.\ \texttt{l}\ c_{false}\ (\lambda\texttt{x}.\ \lambda\texttt{xs}.\ \texttt{xs})$

(b) Geben Sie einen λ-Ausdruck `replicate` an, für den für beliebiges A gilt: [4 Punkte]

$$\texttt{replicate}\ c_n\ A \Rightarrow^* \underbrace{\texttt{cons}\ A\ (\texttt{cons}\ A\ \ldots(\ \texttt{cons}\ A\ \texttt{nil})\ldots)}_{n\ \texttt{mal}}$$

Hinweis: Rufen Sie sich in Erinnerung, welche Struktur c_n hat.

Beispiellösung: c_n hat Struktur $\underbrace{\lambda\texttt{s}.\lambda\texttt{z}.\ \texttt{s}\ (\ldots(\ \texttt{s}\ \texttt{z})\ldots)}_{n\ mal}$, sieht also schon *fast* aus wie der Ausdruck, den wir haben wollen. Es ergibt sich also:
$\texttt{replicate} = \lambda\texttt{n}.\lambda\texttt{x}.\ \texttt{n}\ (\texttt{cons x})\ \texttt{nil}$

Meine Lösung

```
head = \a. a c_false c_true
tail = \a. a c_false c_true

head (cons A B) = (cons A B) c_false c_true =
((\x. \xs. \n. \c. c x xs)  A B) c_false c _true =
(\n. \c. c A B) c_false c_true =
c_true A B = A
```