Informatik 1

13 – Programme als Daten und der λ-Kalkül

Winter 2020/21

Torsten Grust
Universität Tübingen, Germany

Wir schalten auf die nächste Sprachebene ***Die Macht der Abstraktion — fortgeschritten*** um. Änderungen bzw. neu:

1. Neues **Ausgabeformat für Listen** (···) in der REPL:

> (list x_1 x_2 … x_n)
(x_1 x_2 … x_n)


```
> empty
()

> █
```

2. **Polymorpher Gleichheitstest** equal? für beliebige Werte:

```
(: equal? (%a %b -> boolean))
```

Quoting: Programme *sind* Daten

3. Sei e ein beliebiger Ausdruck. Dann liefert (**quote** e) die **Repräsentation** von e — e wird *nicht* ausgewertet:

```
(quote 42)      ⇝  42
(quote "Leia")  ⇝  "Leia"
(quote #t)      ⇝  #t
```

Literale repräsentieren sich selbst

```
(quote (+ 40 2))          ⇝  (+ 40 2)
(quote (lambda (x) x))    ⇝  (lambda (x) x)
```

Repräsentiert als Liste

- Syntaktischer Zucker: 'e ≡ (quote e).
- ⇒ Kompakte Notation *literaler* Listen (Literale c_i):

$$'(c_1\ c_2\ \ldots\ c_n) \rightsquigarrow (\texttt{list}\ c_1\ c_2\ \ldots\ c_n)$$

$$'() \rightsquigarrow \texttt{empty}$$

Symbole: Repräsentation von Identifiern/Namen

Was genau ist `(first '(* 1 2))`? Was sind **`lambda`**, `x`, `+` in `'(lambda (x) (+ x 1))`?

- Neue Signatur `symbol` zur Repräsentation von **Identifiern** (Namen) in Programmen:

  - effiziente interne Darstellung (keine Duplikate),
  - effizient vergleichbar (mittels `equal?`),

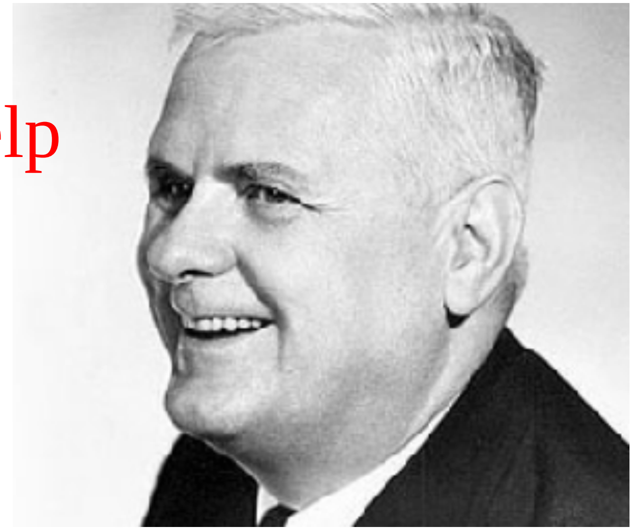
  - kein Zugriff auf die einzelnen Zeichen des Symbols.

- Operationen auf Symbolen:

```
(: symbol?         (%a -> boolean))      (symbol? '*) ⇝ #t
(: symbol->string  (symbol -> string))   } inverse
(: string->symbol  (string -> symbol))   } Funktionen
```

Der **λ-Kalkül** ist eine Notation, die *beliebige* (für einen Computer überhaupt) berechenbare Funktionen darstellen kann.

- Entwickelt in den 1930er Jahren von **Alonzo Church** (*1903, †1995) als neues Fundament der Mathematik (aber die Mathematiker bevorzugten die axiomatische Mengenlehre…). Seither im Einsatz als theoretischer Unterbau von Programmiersprachen.



Alonzo Church

> *“There may, indeed, be other applications of the system [the λ-calculus] than its use as a logic.”*

Die Syntax des λ-Kalküls

Die Menge der **Ausdrücke** E (*expressions*) des λ-Kalküls ist rekursiv definiert (V: unendliche Menge von Variablennamen):

• $\forall\ v \in V$:	$v \in E$	[**Variablen**]
• $\forall\ e_1 \in E,\ e_2 \in E$:	$(\underset{\substack{\uparrow \\ \text{Funktion}}}{e_1}\ \underset{\substack{\uparrow \\ \text{Argument}}}{e_2}) \in E$	[**Applikation**]
• $\forall\ v \in V,\ e_1 \in E$:	$(\lambda \underset{\substack{\uparrow \\ \text{Parameter}}}{v}.\underset{\substack{\uparrow \\ \text{Rumpf}}}{e_1}) \in E$	[**Abstraktion**]

Dies definiert (nur) die **Syntax** des λ-Kalküls. Eine **Semantik** oder Bedeutung müssen wir diesen Ausdrücken erst noch verleihen.

Die Syntax des λ-Kalküls

Beispiele für syntaktisch korrekte Ausdrücke des λ-Kalküls:

$y \in E$	
$(\lambda y.y) \in E$	Identitätsfunktion
$(\lambda y.z) \in E$	Funktion ignoriert Argument y, liefert z
$((f\ x)\ y) \in E$	Anwendung von f auf x und y (Currying)
$(\lambda f.(f\ x)) \in E$	Anwendung von Arg f auf x (H.O.F, Typ ⓘ)

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

Variablen sind entweder **gebunden/frei**

- Verabredete Abkürzung im λ-Kalkül (Currying):

$$(e_1\ e_2\ e_3\ \ldots\ e_n) \equiv (\cdots((e_1\ e_2)\ e_3)\ \ldots\ e_n)$$
$$(f\ x\ y) \equiv ((f\ x)\ y)$$

3 ┆ Freie und Gebundene Variablen

- Im Ausdruck $a_1 \equiv ((\lambda x.(f\ x\ y))\ z)$ …
  - … markiert das λx die Variable x als Parameter. Damit ist Variable x (an das Argument z) **gebunden** (*bound*),
  - aber die Variablen f, y und z sind **frei** (*free*).

- Berechne die **Menge der freien / gebundenen Variablen** in einem λ-Ausdruck:

$$
\begin{aligned}
free(v) &= \{v\} && && 1\\
free((e_1\ e_2)) &= free(e_1) \cup free(e_2) && && 2\\
free((\lambda v.e_1)) &= free(e_1) \setminus \{v\} && \lambda v \text{ bindet } v && 3
\end{aligned}
$$

$$
\begin{aligned}
bound(v) &= \emptyset && && 1\\
bound((e_1\ e_2)) &= bound(e_1) \cup bound(e_2) && && 2\\
bound((\lambda v.e_1)) &= bound(e_1) \cup \{v\} && \lambda v \text{ bindet } v && 3
\end{aligned}
$$

Freie und Gebundene Variablen (Beispiele)

Beispiel: Freie Variablen in $a_1 \equiv ((\lambda x.(f\ x\ y))\ z)$:

$$
\begin{aligned}
& free(((\lambda x.((f\ x)\ y))\ z)) && \boxed{2} \\
&= free((\lambda x.((f\ x)\ y))) \cup free(z) && \boxed{3} \\
&= (free(((f\ x)\ y)) \setminus \{x\}) \cup \{z\} && \boxed{2}\ \boxed{1} \\
&= ((free((f\ x)) \cup free(y)) \setminus \{x\}) \cup \{z\} && \boxed{2}\ \boxed{1} \\
&= (((free(f) \cup free(x)) \cup \{y\}) \setminus \{x\}) \cup \{z\} && \boxed{1}\ \boxed{1} \\
&= (((\{f\} \cup \{x\}) \cup \{y\}) \setminus \{x\}) \cup \{z\} && \cup\ \cup \\
&= (\{f,x,y\} \setminus \{x\}) \cup \{z\} && \setminus\ \cup \\
&= \{f,y,z\}.
\end{aligned}
$$

- ⚠️ Bindung/Freiheit muss für *jedes Vorkommen* einer Variablen separat entschieden werden. **Beispiel:**

$$a_2 \equiv (\overline{x}\ (\lambda x.\underline{x})): \qquad free(a_2) = \{\overline{x}\},\ bound(a_2) = \{\underline{x}\}$$

$\uparrow$ frei $\qquad \uparrow$ gebunden

Die Applikation einer Funktion (= λ-Abstraktion) auf ein Argument wird durch **β-Reduktion** $\rightsquigarrow_\beta$ beschrieben:

- Die Applikation $((\lambda v.e_1)\ e_2)$ wird durchgeführt, in dem
  1. eine Kopie des Rumpfes e_1 hergestellt wird und
  2. alle **freien Vorkommen von** v **in der Kopie des Rumpfes** e_1 durch e_2 ersetzt werden.

$$((\lambda x.foo)\ bar)$$
$$\rightsquigarrow_\beta foo.$$

konstante Funktion

$$((\lambda x.((\lambda y.(*\ y\ x))\ a))\ b)$$
$$\rightsquigarrow_\beta ((\lambda y.(*\ y\ b))\ a)$$
$$\rightsquigarrow_\beta (*\ a\ b).$$

Currying

Auswertung im λ-Kalkül: β-Reduktion

- Mehr Beispiele für **β-Reduktion:**

$$((\lambda x.\underbrace{((* \; ((\lambda \overbrace{x.(+ \; x}^{\text{gebunden}} \; a)) \; b)) \; \overset{\text{frei}}{\downarrow}x)}_{\text{Rumpf}}) \; c)$$

ersetze **freie** x im Rumpf durch c

$$\rightsquigarrow_\beta \; ((* \; ((\lambda x.\underbrace{(+ \; \overset{\text{frei}}{\downarrow}x \; a)}_{\text{Rumpf}}) \; b)) \; c)$$

ersetze **freie** x im Rumpf durch b

$$\rightsquigarrow_\beta \; ((* \; (+ \; b \; a)) \; c).$$

  - **NB:** Ohne eine Regel wie apply_prim endet die Reduktion aus der Sicht des λ-Kalküls hier.

Auswertung im λ-Kalkül: β-Reduktion und Variable Capture

- Noch mehr Beispiele für **β-Reduktion:**

$$
\begin{aligned}
&(\overbrace{(\lambda f.(f\ c))}^{\text{Typ ⓘ}}\ (\lambda x.(+\ x\ a)))\\
\rightsquigarrow_\beta\ &((\lambda x.(+\ x\ a))\ c)\\
\rightsquigarrow_\beta\ &(+\ c\ a).
\end{aligned}
$$

Programmieren mit H.O.F

$$
((((\overbrace{\lambda x.(\lambda y.x))}^{\text{ignoriere 2. Arg, liefere 1. Arg}}\ \underbrace{(\lambda z.y)}_{\text{ignoriere Arg, liefere } y})\ \lightning)\ a)
$$

müsste insgesamt y liefern…

$$
\begin{aligned}
\rightsquigarrow_\beta\ &(((\lambda y.(\lambda z.y))\ \lightning)\ a) \quad \text{captured}\\
\rightsquigarrow_\beta\ &((\lambda z.\lightning)\ a)\\
\rightsquigarrow_\beta\ &\lightning.
\end{aligned}
$$

⚠️ **Variable Capture** 👎:
freies y “wandert” unter λy
und wird damit gebunden

Auswertung im λ-Kalkül: Wie funktioniert β-Reduktion *genau*?

$e\{x{\mapsto}a\}$: “*In* e, *ersetze freie Vorkommen von* x *durch* a”. Damit gilt dann $((\lambda x.e)\ a) \rightsquigarrow_\beta e\{x{\mapsto}a\}$:

$$x\{x{\mapsto}a\} = a \qquad [{\mapsto}_1]$$

$$v\{x{\mapsto}a\} = v \qquad [{\mapsto}_2]$$

$$(e_1\ e_2)\{x{\mapsto}a\} = (e_1\{x{\mapsto}a\}\ e_2\{x{\mapsto}a\}) \qquad [{\mapsto}_3]$$

$$(\lambda x.e_1)\{x{\mapsto}a\} = (\lambda x.e_1) \qquad [{\mapsto}_4]$$

$$(\lambda v.e_1)\{x{\mapsto}a\} = \begin{cases} (\lambda v.e_1\{x{\mapsto}a\}) & v \text{ nicht frei in } a \text{ ※} \qquad [{\mapsto}_5] \\ (\lambda v'.e_1\{v{\mapsto}v'\})\{x{\mapsto}a\} & \text{sonst} \qquad [{\mapsto}_6] \end{cases}$$

※ Wenn v nicht frei in a ist, kann v nicht gecaptured werden. Name v' ist neu (wir brauchen einen Pool von Namen).

Auswertung im λ-Kalkül: β-Reduktion im Beispiel

Drei β-Reduktionen werten $((((\lambda x.(\lambda y.x))\ (\lambda z.y))\ \lightning)\ a)$ aus:[1]

$$((\underline{((\lambda x.(\lambda y.x))\ (\lambda z.y))}\ \lightning)\ a) \qquad \text{erwartetes Ergebnis: } y$$

$$
\begin{aligned}
&\rightsquigarrow_\beta (((\lambda y.x)\{x\mapsto(\lambda z.y)\}\ \lightning)\ a) && ⚠\ y \text{ frei in } (\lambda z.y)\\
&\ \rightarrow_6 (((\lambda y'.x\{y\mapsto y'\})\{x\mapsto(\lambda z.y)\}\ \lightning)\ a) && y'\text{: neuer Name}\\
&\ \rightarrow_2 (((\lambda y'.x)\{x\mapsto(\lambda z.y)\}\ \lightning)\ a) && y' \text{ nicht frei in } (\lambda z.y)\\
&\ \rightarrow_5 (((\lambda y'.x\{x\mapsto(\lambda z.y)\})\ \lightning)\ a) &&\\
&\ \rightarrow_1 (\underline{((\lambda y'.(\lambda z.y))\ \lightning)}\ a) &&
\end{aligned}
$$

$$
\begin{aligned}
&\rightsquigarrow_\beta ((\lambda z.y)\{y'\mapsto\lightning\}\ a) && z \text{ nicht frei in } \lightning\\
&\ \rightarrow_5 ((\lambda z.y\{y'\mapsto\lightning\})\ a) &&\\
&\ \rightarrow_2 \underline{((\lambda z.y)\ a)} &&
\end{aligned}
$$

$$
\begin{aligned}
&\rightsquigarrow_\beta y\{z\mapsto a\}\\
&\ \rightarrow_2 y.
\end{aligned}
$$

[1] ____ markiert den zu reduzierenden Ausdruck (***red**ucible **ex**pression*, *redex*).

Auswertung eines Ausdrucks e im λ-Kalkül: Iteriere β-Reduktion, bis e in seine **Normalform** überführt wurde:

$$e \text{ ist in } \textbf{Normalform} \;\Leftrightarrow\; \underbrace{\nexists\, e' \in E: e \rightsquigarrow_\beta e'}_{e \text{ kann nicht weiter reduziert werden}}$$

- **Beispiele:**

x	ist in NF
$(\lambda x.x)$	ist in NF
$((\lambda x.x)\ z)$	ist nicht in NF
$((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$	ist nicht in NF (und besitzt auch keine)

- Ist es OK, von *der* (eindeutigen) Normalform zu sprechen?

Auswertung im λ-Kalkül: Die Normalform ist eindeutig

Beispiel: Reduziere die beiden *Redexe* in Reihenfolge ①② und ②①. Unterscheidet sich das Endergebnis der Reduktion?

$$
\begin{array}{ccc}
\overbrace{((\lambda z.(z\ a))\ \underbrace{((\lambda x.(\lambda y.x))\ b)}_{②})}^{①} & \xrightarrow{②}_{\beta} & \underline{((\lambda z.(z\ a))\ (\lambda y.b))} \\
①\downarrow_{\beta} & & \downarrow_{\beta} \\
(\underline{((\lambda x.(\lambda y.x))\ b)}\ a) & \longrightarrow_{\beta} & \underline{((\lambda y.b)\ a)} \\
 & & \downarrow_{\beta} \\
 & & b \quad \text{in NF}
\end{array}
$$

- **Zufall?** Nein! ☞ Satz von Church-Rosser.

Auswertung im λ-Kalkül: Satz von Church-Rosser

Satz von Church-Rosser: Wenn Ausdruck e sich (in mehreren) Schritten ($\twoheadrightarrow_{\beta*}$) zu e_1 und e_2 reduzieren lässt, dann gibt es einen Ausdruck e', in den sich e_1 und e_2 reduzieren lassen:

$$\begin{tikzcd}
 & e_1 \arrow[dr, "\beta*"] & \\
e \arrow[ur, "\beta*"] \arrow[dr, "\beta*"'] & & e' \\
 & e_2 \arrow[ur, "\beta*"'] &
\end{tikzcd}$$

Church-Rosser "Diamant" ◇

[ohne Beweis]

- Konsequenz: Sollten e_1 *und* e_2 in Normalform sein, dann gilt $e_1 = e' = e_2$ ($e_i \twoheadrightarrow_{\beta*} e'$ führt null β-Reduktionen aus). Also ist $e_1 = e' = e_2$ ***die*** Normalform von e.

Applicative Order wertet λ-Ausdruck e durch wiederholte β-Reduktion aus. $[\![e]\!]^k$ mit maximalem k ist der nächste *Redex*:

1. $[\![v]\!]^k \stackrel{\text{def}}{\equiv} v$ **[ao_var]**
2. $[\![(\lambda v.e_1)]\!]^k \stackrel{\text{def}}{\equiv} (\lambda v.[\![e_1]\!]^k)$ **[ao_λ]**
3. $[\![(e_1\ e_2)]\!]^k \stackrel{\text{def}}{\equiv}$ Sei $f \equiv [\![e_1]\!]^{k+1}$, dann:

$$\begin{cases} [\![e\{v \mapsto \overset{*}{[\![e_2]\!]^{k+2}}\}^{k+1}]\!]^k & f = (\lambda v.e) \quad \textbf{[ao\_β]} \\ (f\ [\![e_2]\!]^{k+1}) & \text{sonst} \quad \textbf{[ao\_apply]} \end{cases}$$

> *Applicative Order* wertet zuerst den **innersten** *Redex* aus (siehe Regel **ao_β**, $*$). Damit wird Argument e_2 ausgewertet, *bevor* die Funktionsanwendung stattfindet.

Für manche Ausdrücke e findet *Applicative Order* die Normalform des Ausdrucks *nicht*. **Beispiel:** Reduktion von $((\lambda x.y)\ \Omega)$:[2]

1. Reduktion via *Applicative Order*:

$$[\![((\lambda x.y)\ \Omega)]\!]^0 = \cdots = [\![y\{x \mapsto [\![\Omega]\!]^2\}^1]\!]^0 \circlearrowright \leftarrow \text{endlose Reduktion, da } [\![\Omega]\!]^k = [\![\Omega]\!]^k$$ ⚠️

2. Reduziere Funktionsanwendung zuerst (*Normal Order*):

$$\underline{((\lambda x.y)\ \Omega)} \rightsquigarrow_\beta y.$$

[2] $\Omega \stackrel{\text{def}}{=} ((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$.

Q: Literale und Operationen darauf stehen im λ-Kalkül nicht zur Verfügung. Kann man damit jemals programmieren?

A: Ja! Definiere λ-Ausdrücke, die **bei Interaktion die erwarteten algebraischen Eigenschaften** zeigen. **Beispiel:**

$$
\begin{array}{lcl|lcl}
\mathit{TRUE} & \stackrel{\text{def}}{\equiv} & (\lambda x.(\lambda y.x)) & \mathit{IF\text{-}ELSE} & \stackrel{\text{def}}{\equiv} & (\lambda x.x) \\
\mathit{FALSE} & \stackrel{\text{def}}{\equiv} & (\lambda x.(\lambda y.y)) & \mathit{AND} & \stackrel{\text{def}}{\equiv} & (\lambda a.(\lambda b.((b\ a)\ b))) \\
\mathit{NOT} & \stackrel{\text{def}}{\equiv} & \text{☞ Übung} & \mathit{OR} & \stackrel{\text{def}}{\equiv} & \text{☞ Übung}
\end{array}
$$

- Diese λ-Ausdrücke verhalten sich wie die Booleans:

$$
\begin{array}{lcl}
[\![(((\mathit{IF\text{-}ELSE}\ \mathit{TRUE})\ \mathit{yes})\ \mathit{no})]\!]^0 & = & \mathit{yes} \\
[\![(((\mathit{IF\text{-}ELSE}\ \mathit{FALSE})\ \mathit{yes})\ \mathit{no})]\!]^0 & = & \mathit{no} \\
[\![((\mathit{AND}\ \mathit{FALSE})\ x)]\!]^0 & = & \mathit{FALSE} \\
[\![((\mathit{AND}\ \mathit{TRUE})\ x)]\!]^0 & = & x
\end{array}
$$

- Repräsentation von **Paaren** $\langle x,y\rangle$ im λ-Kalkül:

$$
\begin{aligned}
PAIR &\stackrel{\text{def}}{\equiv} (\lambda x.(\lambda y.(\lambda s.((s\ x)\ y)))) && \langle x,y\rangle \\
SND &\stackrel{\text{def}}{\equiv} (\lambda p.(p\ (\lambda x.(\lambda y.y)))) && (SND\ \langle x,y\rangle) \rightsquigarrow y \\
FST &\stackrel{\text{def}}{\equiv} (\lambda p.(p\ \underbrace{(\lambda x.(\lambda y.x))}_{TRUE})) && (FST\ \underbrace{\langle x,y\rangle}_{p}) \rightsquigarrow x
\end{aligned}
$$

- Repräsentation von **Listen** (make-pair x xs) und empty:

$$
\begin{aligned}
MAKE\text{-}PAIR &\stackrel{\text{def}}{\equiv} (\lambda x.(\lambda xs.\overbrace{((PAIR\ FALSE)\ ((PAIR\ x)\ xs))}^{\langle FALSE,\langle x,xs\rangle\rangle}))) \\
FIRST &\stackrel{\text{def}}{\equiv} (\lambda xs.(FST\ (SND\ xs))) \\
REST &\stackrel{\text{def}}{\equiv} (\lambda xs.(SND\ (SND\ xs))) \\
EMPTY? &\stackrel{\text{def}}{\equiv} FST \\
EMPTY &\stackrel{\text{def}}{\equiv} I \qquad (EMPTY?\ EMPTY) \rightsquigarrow (I\ TRUE) \rightsquigarrow TRUE
\end{aligned}
$$

Existiert eine Repräsentation der **natürlichen Zahlen** {0,1,2, …} im λ-Kalkül, die arithmetische Operationen erlaubt?

- **Definition:** $\tilde{n}$ ist das ***Church Numeral*** für $n \in \mathbb{N}$:

$$\begin{aligned}\tilde{n} &\stackrel{\text{def}}{=} (\lambda f.(\lambda x.(f^{n}\ x))) \\ &= (\lambda f.(\lambda x.\underbrace{(f \cdots (f\ (f}_{n\text{-fache Applikation von } f}\ x))\cdots)))\end{aligned}$$

  - **Beispiele:**

$$\begin{aligned}\tilde{0} &\equiv (\lambda f.(\lambda x.x)) \\ \tilde{1} &\equiv (\lambda f.(\lambda x.(f\ x))) \\ \tilde{2} &\equiv (\lambda f.(\lambda x.(f\ (f\ x)))) \\ &\vdots\end{aligned}$$

x ist **Fixpunkt** von Funktion f, falls $f(x) = x$ gilt.

- **Beispiele** (reelle Funktionen):

$f(x) = x^2$	zwei Fixpunkte: $x = 0$, $x = 1$
$f(x) = x + 1$	kein Fixpunkt
$f(x) = x$	unendliche viele Fixpunkte

Im λ-Kalkül hat **jede Funktion** F **einen Fixpunkt:** $(Y\ F)$.

- Sei $Y \stackrel{\text{def}}{=} (\lambda f.((\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x)))))$. Dann gilt:

$$(F\ (Y\ F)) \quad = \quad (Y\ F)$$

$(Y\ F)$ ist Fixpunkt von F

Rekursion im λ-Kalkül: Der *Y*-Kombinator

$$
\begin{aligned}
& (\underline{Y}\ F) \\
= \quad & \underline{((\lambda f.((\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x)))))\ F)} \\
\rightsquigarrow_\beta \quad & \underline{((\lambda x.(F\ (x\ x)))\ (\lambda x.(F\ (x\ x))))} \\
\rightsquigarrow_\beta \quad & (F\ \underline{((\lambda x.(F\ (x\ x)))\ (\lambda x.(F\ (x\ x))))}) \\
\leftsquigarrow_\beta \quad & (F\ (\underline{(\lambda f.((\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x)))))}\ F)) \\
= \quad & (F\ (Y\ F)).
\end{aligned}
$$

MIT Scheme

- Zu $\leftsquigarrow_\beta$: Es gilt: $e = ((\lambda f.e\{F \mapsto f\})\ F)$ (β-Abstraktion).

Y ist Haskell B. Curry's ***Y*-Kombinator**, den wir einsetzen können, um **Rekursion im λ-Kalkül** auszudrücken.

Rekursion im λ-Kalkül: Beispiel Fakultätsfunktion *n*!

λ-Ausdruck $(FAC\ n)$ berechnet die **Fakultät n! rekursiv:**

$$
\begin{aligned}
FAC \stackrel{\text{def}}{=}\ & (\lambda n.(((\mathit{IF\text{-}ELSE}\ (\mathit{ISZERO?}\ n))\\
& \quad 1)\\
& \quad ((\mathit{MULT}\ n)\ (FAC\ (\mathit{PRED}\ n)))))
\end{aligned}
$$

Rekursion

$$
\begin{aligned}
\leftsquigarrow_\beta\ & (\underbrace{(\lambda f.(\lambda n.(((\mathit{IF\text{-}ELSE}\ (\mathit{ISZERO?}\ n))\ 1)\ ((\mathit{MULT}\ n)\ (\ f\ \ (\mathit{PRED}\ n))))))}_{\equiv\ F}\ FAC)
\end{aligned}
$$

- Es gilt also $FAC = (F\ FAC) \Rightarrow FAC$ ist Fixpunkt von F.
- 💡 Definiere die Fakultätsfunktion als $FAC \stackrel{\text{def}}{=} (Y\ F)$.

❶ Rekursives f. ❷ $F \equiv$ β-Abstraktion von f. ❸ $f \stackrel{\text{def}}{=} (Y\ F)$.

Rekursion im λ-Kalkül: Eine Reduktionsstrategie für Y

⚠️ *Applicative Order* führt für $(Y\ F)$ zu endloser Reduktion:

$$((\underline{Y\ F})\ \widetilde{n}_0) \rightsquigarrow ((F\ (\underline{Y\ F}))\ \widetilde{n}_0) \rightsquigarrow ((F\ (F\ (\underline{Y\ F})))\ \widetilde{n}_0) \rightsquigarrow \cdots\ \infty$$

Benötigt: eine Reduktionsstrategie, die **Funktionsanwendung** **vor Funktionsargumenten** reduziert. Etwa ***Normal Order***:[3]

$$((\underline{Y\ F})\ \widetilde{n}_0) \rightsquigarrow (\underline{(F\ (Y\ F))}\ \widetilde{n}_0) \rightsquigarrow \begin{array}{l} \nearrow\ e_1[((\underline{Y\ F})\ \widetilde{n}_1)] \rightsquigarrow \cdots \\ \quad F \text{ entscheidet} \\ \searrow\ \underline{e_2[\widetilde{n}_0]}.\ \text{Rekursionsabbruch} \end{array}$$

[3] $e_1[e]$ bezeichnet einen Ausdruck e_1, in dem e als Teilausdruck vorkommt.

11 ¦ Reduktionsstrategie *Normal Order*[4]

1. $\llbracket v \rrbracket^k \stackrel{\text{def}}{\equiv} v$ **[no_var]**
2. $\llbracket (\lambda v.e_1) \rrbracket^k \stackrel{\text{def}}{\equiv} (\lambda v.\llbracket e_1 \rrbracket^k)$ **[no_λ]**
3. $\llbracket (e_1\ e_2) \rrbracket^k \stackrel{\text{def}}{\equiv}$ Sei $f \equiv \llparenthesis e_1 \rrparenthesis^{k+1}$, dann:

$$\begin{cases} \llbracket e\{v \overset{*}{\rightarrow} e_2\}^{k+1} \rrbracket^k & f = (\lambda v.e) \quad \textbf{[no\_β]} \\ (\llbracket f \rrbracket^{k+1}\ \llbracket e_2 \rrbracket^{k+1}) & \text{sonst} \quad \textbf{[no\_apply]} \end{cases}$$

- Intern nutzt Regel **no_β** die *Call by Name* Reduktion $\llparenthesis\,\rrparenthesis^k$:

4. $\llparenthesis v \rrparenthesis^k \stackrel{\text{def}}{\equiv} v$ **[bn_var]**

5. $\llparenthesis (\lambda v.e_1) \rrparenthesis^k \stackrel{\text{def}}{\equiv} (\lambda v.e_1)$ **[bn_λ]**
6. $\llparenthesis (e_1\ e_2) \rrparenthesis^k \stackrel{\text{def}}{\equiv}$ Sei $f \equiv \llparenthesis e_1 \rrparenthesis^{k+1}$, dann:

$$\begin{cases} \llparenthesis e\{v \overset{*}{\rightarrow} e_2\}^{k+1} \rrparenthesis^k & f = (\lambda v.e) \quad \textbf{[bn\_β]} \\ (f\ e_2) & \text{sonst} \quad \textbf{[bn\_apply]} \end{cases}$$

[4] Verfügbar als Funktion `no` im File `definitions-13.rkt`.