0. GLIEDERUNG

1) EINFÜHRUNG

2) SYNTAX DES λ-KALKÜLS

3) BETA - REDUKTION

4) CURRYING

5) REICHWEITE VON VARIABLEN

6) FREIE UND GEBUNDENE VARIABLEN

7) ARITHMETIK TEIL 1

8) BOOLEANS

9) REKURSION

10) ARITHMETIK TEIL 2

I. EINFÜHRUNG

λ – Kalkül ist die kleinste Programmiersprache der Welt

Entstehung:

1900: David Hilbert stellt die Frage ob es möglich ist mittels eines automatischen Verfahrens aus den Axiomen alle Sätze der Mathematik herzuleiten.
Er wollte also durch Kombination einzelner Axiome immer neue Wahrheiten entwickeln.

Church und Turing beantworten diese Frage mit Nein.

Turing entwickelte die Turingmaschine, welche mit imperativen Sprachen wie C oder Pascal verwandt ist.

Nach 1930: Church entwickelte das λ-Kalkül um die Frage, was ein automatisches Verfahren ist, zu beantworten.

Er zeigt, dass eine Maschine (Computer) nie in der Lage wäre aus den Axiomen alle Sätze der Mathematik herzuleiten, egal wie viel Leistung er hätte und wie lange man ihm Zeit geben würde.

Das λ-Kalkül wir in Sprachen wie LISP, SCHEME, ML oder LEGO verwendet.
Dient zur mathematisch Exakten Beschreibung der denotationellen Semantik von funktionalen Programmier-sprachen. Es dient aber auch selber wiederum als funktionale Programmiersprache. Somit könnte man also die Semantik des λ-Kalküls im λ-Kalkül definieren.

II. SYNTAX DES λ-KALKÜLS

<name> := a | b | c | d | ..... | a_1 | b_1 |..... ▲ | ■ | ● | ☻

BEISPIEL : x

<expression> := <name> | <function> | <application>

BEISPIEL : λ y . y

<function> := λ <name> . <expression>

BEISPIEL : (λ y . y) (λ y . y)

<application> := <expression> . <expression>

BEISPIEL : (λ x . x (y))

III. BETA - REDUKTION

Bis auf λ sind alle verwendeten Buchstaben willkürlich und können, innerhalb einer Expression, jederzeit ausgetauscht werden. Buchstaben sind im λ-Kalkül nur Platzhalter.

BEISPIEL 1: $f(x) = x + 1 \rightarrow f = \lambda x . x + 1$

BEISPIEL 2: $(\lambda x . x)\, y = (\lambda t . t)\, g = (\lambda \blacksquare . \blacksquare)\, \blacktriangle$

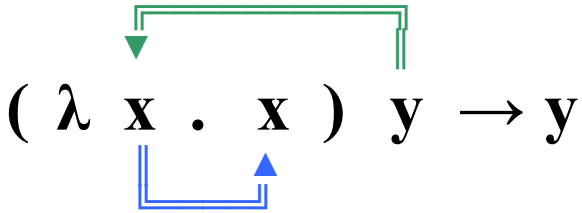
Das verwendete Beispiel (λ x . x) wird als Identitätsfunktion bezeichnet. Es liefert als Ergebnis immer den eingesetzten Wert :

$(\lambda x . x)\, y \rightarrow [\, y / x \,] \rightarrow y$

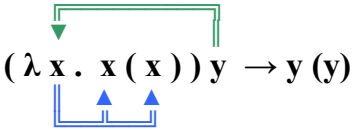
$(\lambda t . t)\, g \rightarrow [\, g / t \,] \rightarrow g$

$(\lambda \blacksquare . \blacksquare)\, \blacktriangle \rightarrow [\, \blacktriangle / \blacksquare \,] \rightarrow \blacktriangle$

Das jeweilige Ergebnis erhält man durch Reduktion. Dabei wird der hinter der <Funktion> stehende <Name> in die <Funktion> eingesetzt.



Weiteres Beispiel :



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

ALLGEMEIN :

$$E_1\ E_2\ E_3\ E_4 \equiv (((E_1\ E_2)\ E_3)\ E_4)$$

$$(\lambda s.(\lambda z.s(s(z))))\ f\ a$$

$$\rightarrow (\lambda z.f(f(z)))\ a$$

$$\rightarrow f(f(a))$$

IV. CURRYING

Das λ-Kalkül wertet Funktionen mittels Currying aus. Currying bedeutet, dass pro Schritt immer nur ein Argument der Funktion ausgewertet wird.

BEISPIEL 1:

$$(\lambda x.(\lambda y.x\ y))\ u\ v$$

1. Schritt : $\rightarrow (\lambda y.u\ y)\ v$

Die Auswertung von Funktionen kann also wieder Funktionen ergeben.

2. Schritt : $\rightarrow u\ v$

BEISPIEL 2:

$$f\ x\ y = x + y \rightarrow f\ x\ y = \lambda x\ y.\ x + y$$

$(\lambda 2)\ 3 \rightarrow$ 1. Schritt : $\rightarrow (\lambda y.2 + y)\ 3$

2. Schritt : $\rightarrow (2 + 3)$

3. Schritt : $\rightarrow 5$

X : int → (int → int)

V. REICHWEITE VON VARIABLEN

(λ x . (λ y . x y)) y ← freie Variable
↑
gebundene Variable

Da y eine freie Variable ist, darf sie nicht für die gleichnamige Variable y eingesetzt werden, da diese innerhalb der Funktion durch ein λ eingeführt wurde und somit gebunden ist.

(λ ▲ . (λ ■ . ▲ ■) y

Da Variablen im λ Kalkül nur Platzhalter sind, kann man sie durch beliebige andere Buchstaben, Zeichen oder Symbole ersetzen.

Man führt vor der eigentlichen Reduktion, eine so genannte α – Konversion durch. Hierbei ersetzt man die in der Applikation doppelt vorkommenden Variablen durch unterschiedliche. Es werden also komplett neue Variablen-namen generiert.

(λ x . (λ y . x y)) y → [s / x] [t / y] → (λ s . (λ t . s t)) y

Somit bestehen keinerlei Konflikte mehr zwischen gebundenen und freien Variablen. Eine Reduktion ist also möglich.

VI. FREIE UND GEBUNDENE VARIABLEN

Nicht nur innerhalb einer Applikation sondern auch innerhalb einer Funktion kann es freie und gebundene Variablen geben.

(λ x . x y) **[x ist gebunden, y ist frei]**

Freie Variablen:

1. **<name> ist frei in <name>**

   BEISPIEL : y [y ist frei]

2. **<name> ist frei in λ <name$_1$> . <expression> falls**
   <name> ≠ <name$_1$> und
   <name> frei in <expression>

   BEISPIEL : (λ x . x y) [y ist frei]

3. **<name> ist frei in E_1 E_2 falls**
   <name> in E_1 frei ist oder
   <name> in E_2 frei ist.

   BEISPIEL : (λ x . x) (λ z . y z) [y ist frei]

Gebundene Variablen:

1. **<name> ist gebunden in λ <name$_1$> . <expression>**

   falls <name> = <name$_1$>

   oder <name> in <expression> gebunden ist

   BEISPIEL : (λ x . x) **[x ist gebunden]**

2. **<name> ist gebunden in E_1 E_2**

   falls <name> gebunden ist in E_1

   oder <name> gebunden ist in E_2

   BEISPIEL : (λ y . y) (λ x . x) **[x ist gebunden]**

WICHTIG:

Setze nie eine freie Variable in einen Ausdruck ein, in dem die Variable gebunden ist.

VII. ARITHMETIK TEIL 1

Zahlen werden im λ – Kalkül mit der Identitätsfunktion [(λ x . x)] dargestellt. Die Anzahl der ineinander verschachtelten Funktionen bestimmt den Zahlenwert.

$0 \equiv (\lambda s.(\lambda z.z))$
$1 \equiv (\lambda s.(\lambda z.s(z)))$
$2 \equiv (\lambda s.(\lambda z.s(s(z))))$
$3 \equiv (\lambda s.(\lambda z.s(s(s(z)))))$ usw……

Dies wird als iterative Funktionsapplikation bezeichnet.

BEISPIEL : $f\ a$
$f^0 a \rightarrow a$
$f^1 a \rightarrow (f(a))$
$f^2 a \rightarrow (f(f(a)))$
$f^3 a \rightarrow (f(f(f(a))))$

$2\,f\,a \equiv (\lambda s(\lambda z.s(s(z))))\,f\,a$
$\rightarrow (\lambda z.f(f(z)))\,a$
$\rightarrow f(f(a))$

DIE NACHFOLGER FUNKTION

Die Nachfolgerfunktion wird mit S abgekürzt.

$S \equiv (\lambda w . (\lambda y . (\lambda x . y (w y x))))$

$\equiv (\lambda wyx . y (w y x))$

$S\ 2 \equiv (\lambda w y x . y (w y x))\ 2$

$\rightarrow (\lambda y x . y (2\ y\ x))$ **[Church – Rosser – Satz]**

$\rightarrow (\lambda y x . y (y (y (x)))) \rightarrow 3$

Zur Erinnerung :
$3 \equiv (\lambda s . (\lambda z . s (s (s (z))))) = (\lambda s z . s (s (s (z))))$

Die Nachfolgerfunktion von 3, also S3 ist somit 4 und die Nachfolgerfunktion von 4, also S4 ist somit 5

$S3 \rightarrow 4$

$S4 \rightarrow 5$

Mit Hilfe der Nachfolgerfunktion ist es möglich die ersten Rechnungen im λ-Kalkül durchzuführen.

DIE ADDITION

Will man zwei Zahlen x und y addieren, so wertet man x mal die Nachfolgerfunktion von y aus → x S y

BEISPIEL (2 + 3) :

$2\ S\ 3 \rightarrow S (S (3)) \rightarrow S (4) \rightarrow 5$

$(\lambda x y . x\ S\ y)\ 2\ 3$ **[Additionsfunktion]**

$\rightarrow (\lambda y . 2\ S\ y)\ 3$

$\rightarrow 2\ S\ 3$

$\rightarrow 5$

DIE MULTIPLIKATION

(λ x y z . x (y z)) **[MULTIPLIKATIONSFUNKTION]**

BEISPIEL : 2 * 2

(λ x y z . x (y z)) 2 2
[2 wird für x eingesetzt]
→ (λ y z . 2 (y z)) 2
[2 wird für y eingesetzt]
→ (λ z . 2 (2 z))
[für 2 wird die Entsprechung im λ-Kalkül eingesetzt]
→ (λ z . (λ s z . s (s (z))) ((λ s z . s (s (z))) z))
[da z gebunden ist, kann es nicht durch z ersetzt werden. Wir führen also eine α-Konversion durch.]
→ (λ z . (λ s z . s (s (z))) ((λ s t . s (s (t))) z))
[s wird durch z ersetzt]
→ (λ z . (λ s z . s (s (z))) ((λ t . z (z (t)))))
[z ist im ersten Teil der Applikation gebunden, man kann also nicht den zweiten Teil für s einsetzen, da es sonst zu Konflikten mit z käme => α-Konversion]
→ (λ z . (λ s a . s (s (a))) ((λ t . z (z (t)))))
[(λ t . z (z (t))) wird für s eingesetzt]
→ (λ z . (λ a . (λ t . z (z (t))) (λ t . z (z (t))) (a)))
[t wird durch a ersetzt]
→ (λ z . (λ a . (λ t . z (z (t)))) (z (z (a)))
[t wird durch (z (z (a))) ersetzt]
→ (λ z . (λ a . z (z (z (z (a))))))

→ 4

VIII. BOOLEANS

T ≡ (λ x y . x) **Liefert immer x als Ergebnis**

F ≡ (λ x y . y) **Liefert immer y als Ergebnis**

IF

IF ≡ (λ b x y . b x y)

BEISPIEL:

IF (λ b x y . b x y) T 3 4 → T 3 4 → 3

AND

/\ ≡ (λ x y . x y (λ u v . v)) **[(λ u v . v) = F]**

≡ (λ x y . x y F)

BEISPIEL 1 :

/\TT ≡ (λ x y . x y F) T T
→ (λ y . T y F) T
→ T T F
→ T
(Erinnerung: True liefert immer x als Ergebnis)

BEISPIEL 2 :

$\wedge FF \equiv (\lambda x\, y\, .\, x\, y\, F)\, F\, F$
$\rightarrow (\lambda y\, .\, F\, y\, F)\, F$
$\rightarrow F\, F\, F$
$\rightarrow F$

OR

$V \equiv (\lambda x\, y\, .\, x\, (\lambda x\, y\, .\, x)\, y)$
$\equiv (\lambda x\, y\, .\, x\, T\, y)$

BEISPIEL 1 :

$VTF \equiv (\lambda x\, y\, .\, x\, T\, y)\, T\, F$
$\rightarrow (\lambda y\, .\, T\, T\, y)\, F$
$\rightarrow T\, T\, F$
$\rightarrow T$

BEISPIEL 2 :

$VFF \equiv (\lambda x\, y\, .\, x\, T\, y)\, F\, F$
$\rightarrow (\lambda y\, .\, F\, T\, y)\, F$
$\rightarrow F\, T\, F$
$\rightarrow F$

(Erinnerung: False liefert immer y als Ergebnis)

NOT

$\neg \equiv (\lambda x\, .\, x\, F\, T)$

BEISPIEL 1 :

$\neg\, T \equiv (\lambda x\, .\, x\, F\, T)\, T$
$\rightarrow T\, F\, T$
$\rightarrow F$

BEISPIEL 2 :

$\neg\, F \equiv (\lambda x\, .\, x\, F\, T)\, F$
$\rightarrow F\, F\, T$
$\rightarrow T$

CONDITIONAL TEST

Der Conditional Test stellt fest, ob eine Zahl n = 0 ist.

$Z \equiv (\lambda x\, .\, x\, F\, \neg\, F)$

Wichtig: Es wird von links nach rechts assoziiert.

$Z \equiv (\lambda x\, .\, (((x\, F)\, \neg)\, F))$

Erinnerung: Eine Zahl n ist im λ-Kalkül die n-malige Anwendung von I auf ein Argument a.

Mittels Z wird nun überprüft ob 0 = 0 ist:

$Z\, 0 \equiv (\lambda x\, .\, x\, F\, \neg\, F)\, 0$
[0 wird für x eingesetzt.]
$\rightarrow 0\, F\, \neg\, F$
[Linksassoziativität!: F wird 0-Mal auf ¬ angewendet. ¬ bleibt unberührt.]
$\rightarrow \neg\, F$
$\rightarrow T$

Es ist also wahr, dass 0 = 0 ist.
Allgemein: Prüfung von n

Z n ≡ (λ x . x F ¬ F) n
→ n F ¬ F
[F wählt aus (x , y) immer y aus. Gibt es aber nur ein Argument x, so ist das Ergebnis von F a = I. F von ¬ ist somit I. F von I ergibt wieder I. Dies wiederholt man n-Mal, bis stehen bleibt:]
→ I F
[I F = I]
→ F

n ist also ungleich 0. Zu diesem Ergebnis kommt man nur wenn F mindestens einmal auf ¬ angewandt wurde, n also ungleich 0 ist.

IX. REKURSION

Die Rekursive Funktion ist im λ-Kalkül wie folgt definiert:

Y ≡ λ y . (λ x . y (x x)) (λ x . y (x x)))

Y ist der Rekursionsopperator.
Nachfolgend soll R eine beliebige rekursive Funktion sein.

YR = (λ y . (λ x . y (x x)) (λ x . y (x x))) R
[R wird für y eingesetzt.]
→ (λ x . R (x x)) (λ x . R (x x))
[(λ x . R (x x)) wird für x eingesetzt.]
→ R ((λ x . R (x x)) (λ x . R (x x)))
→ R (Y R)

Die Rekursion von R erzeugt die Anwendung von R auf (Y R) → R (Y R)

BEISPIEL 1:

$0 + 1 + 2 + 3 + 4 + \ldots + n = S_n$

$S_n = n + S_{n-1}$
$S_0 = 0$

R ≡ λ r n . Z n 0 (n S (r (P n)))) [Definition von S_n]
[r steht für S_n, n steht für eine Zahl, Z n 0 überprüft ob n = 0 ist]

YR1 ≡ R (Y R) 1
→ (λ r n . Z n 0 (n S (r (P n)))) (Y R) 1
[(Y R) für r und 1 für n]

→ Z 1 0 (1 S ((Y R) (P 1)))
[da Z 1 ≠ 0 wird das zweite Argument gewählt.]
→ 1 S ((Y R) (P 1))
[(Y R) ist ein rekursiver Aufruf → R (Y R)]
→ 1 S (R (Y R) (P 1))
[Die Rekursion wird einmal durchlaufen. Für R wird S_n eingesetzt.]
→ 1 S ((λ r n . Z n 0 (n S (r (P n)))) (Y R) (P 1))
[(Y R) für r und (P 1) für n.]
→ 1 S (Z (P 1) 0 ((P 1) S ((Y R) (P (P (1))))))
[(P 1) = 0 → Rekursion terminiert, da Z n das erste Argument auswählt.]
→ 1 S 0
→ 1 = S_1

BEISPIEL 2:

YR3 fi (Y R) 3
→ (λ r n . Z n 0 (n S (r (P n)))) (Y R) 3
[für r → (Y R) und für n → 3]
→ Z 3 0 (3 S (Y R (P 3)))
[Z 3 ≠ 0 → zweites Argument]
→ 3 S (Y R (P 3))
[Y R wird reduziert zu R (Y R)
→ 3 S (R (Y R) (P 3))
...
→ 3 S (P 3) S R (Y R) (P (P 3))
...
→ 3 S 2 S 1 S 0
→ 6

X. ARITHMETIK TEIL 2

Die Vorläuferfunktion, oder auch Predecessor Function, wird im λ-Kalkül mit P abgekürzt.

Paare werden im λ-Kalkül als (λ z . z a b) geschrieben.

Um mit der Vorläuferfunktion rechnen zu können, muss man in diese immer Paare einsetzen.

BEISPIELE FÜR PAARE: (0 , 0)
(1 , 0)
(2 , 1)
(3 , 2)
(4 , 3)
allgemein : p = (p_1, p_2)

Wendet man T bzw. F auf die Funktion an, so erhält man immer das erste bzw. zweite Element des Paares:

(λ z . z a b) T → T a b → a

(λ z . z a b) F → F a b → b

Um die Vorläuferfunktion herleiten zu können, benötigt man die folgende Hilfsfunktion:

Φ ≡ λ p . (S (p_1)) (p_1))

In diese Hilfsfunktion wird nun das Paar (1 , 0) eingesetzt.

$\Phi (\lambda t . t 1 0) \equiv$
[Einsetzen von (λ t .t 1 0) in die Hilfsfunktion Φ]
$(\lambda p . (S (p T)) (p T)) (\lambda t . t 1 0)$
[(λ t . t 1 0) wir für p eingesetzt]
$\rightarrow (S (\lambda t . t 1 0) T) ((\lambda t . t 1 0) T))$
[T für t . Wählt das erste Element des Paares aus]
$\rightarrow (S (1) (1))$
[Nachfolgerfunktion S : S (1) = 2]
$\rightarrow (2 , 1)$

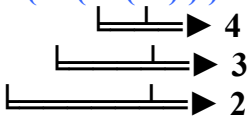
Die Vorläuferfunktion P ist also:

$$P \equiv (\lambda n . n \Phi (\lambda z . z 0 0) F)$$

BEISPIEL :

$P 1 \equiv \lambda n . n \Phi (\lambda z . z 0 0) F) 1$
[1 wird für n eingesetzt]
$\rightarrow 1 \Phi (\lambda z . z 0 0) F$
[1 fällt weg, da Φ nur einmal durchlaufen wird]
$\rightarrow \Phi (\lambda z .z 0 0) F$
[(λ z . z 0 0) wird in Hilfsfunktion Φ eingesetzt]
$\rightarrow (\lambda z . z 1 0) F$
$\rightarrow F 1 0$
$\rightarrow 0$
ERGEBNIS:
Neben der Multiplikation und Addition ist nun auch die Subtraktion im λ-Kalkül möglich.

$5 - 3 = 3 P 5 \rightarrow P (P (P (5)))$



$\rightarrow 2$