Lambda-Kalkül

Funktionales Paradigma

Zur Erinnerung …

- Funktionale Programmierung:

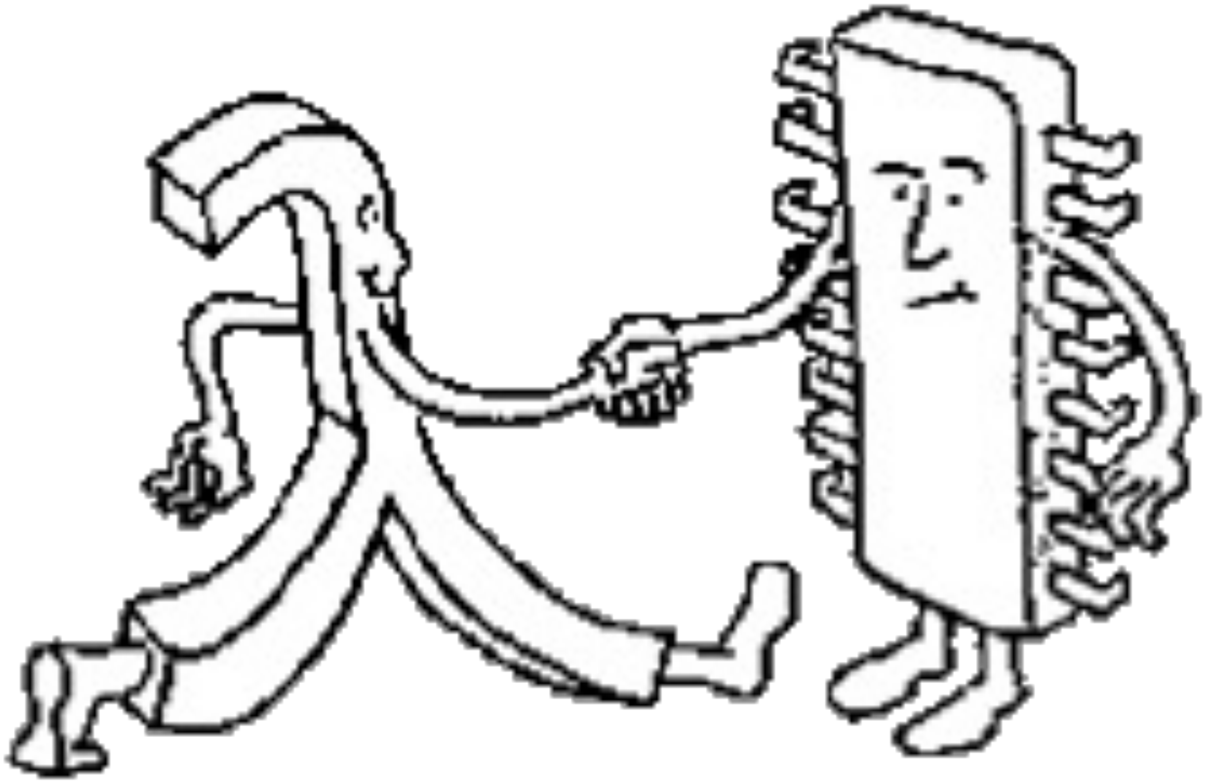
  - Definition von Menge von Funktionen
  - kein Kontext, seiteneffektfrei, lokales Wissen
  - Rekursion (linear, endrekursiv)
  - Funktionen als Bürger erster Klasse
  - partielle Anwendung/Currying
  - Funktionskomposition

Übersicht

Prozedural	Objekt-orientierung	**Funktional**	Logisch	Parallel
Prozeduren	*Objekte und OOP*	*Funktionen*	*Aussagenlogik*	*Parallelität*
Datentypen und Zeiger	*Fünf Konzepte der OOP*	*Funktionale Konzepte*	*Resolution und Unifikation*	*Modelle*
Speicher-verwaltung	*Anwendung der Konzepte*	***Lambda-Kalkül***	*Regelbasiertes Programmieren*	*Mechanismen*
				Analyse
C	*Java*	***Scala***	*Prolog*	*Java*

Anwendung

Dr.-Ing. Christian Braune



Lambda-Kalkül

- Fragestellung: Was kann mit Programmiersprachen (theoretisch) berechnet werden?
- Zur Beantwortung: Einführung eines formalen Modells, das
  - so mächtig ist, dass es auf reale Programmiersprachen zurückführbar ist

  aber

  - so einfach ist, dass damit einfach formale Beweise durchgeführt werden können

- Zwei typische Modelle:
  - Turingmaschine für imperative (prozedurale, OO) Sprachen
  - Lambda-Kalkül für funktionale Sprachen

Dr.-Ing. Christian Braune

Allgemein Lambda-Kalkül

- Einführung durch Alonzo Church und Stephen Cole Kleene in den1930‘er
- Beweis Alan Turing 1937 der Turing-Vollständigkeit von Lambda-definierbaren Funktionen[1]

$\Rightarrow$ Äquivalenz von Lambda-Kalkül und Turingmaschine

$\Rightarrow$ Funktionalen Programmiersprachen genauso mächtig wie prozedurale oder objektorientierte Programmiersprachen!

λ

[1] Computability and λ-Definability
http://www.jstor.org/stable/2268280

Lambda - Syntax

- Lambda-Ausdrücke bestehen aus diesen drei Komponenten:
- Variablen (und Konstanten)

$$a$$

- Funktionsdefinitionen (a.k.a. Lambda Abstraktion)

$$\lambda a.\_$$

- Funktionsapplikationen

$$(\_\ \_)$$

- Blanks sind Platzhalter

- Die einfachste Funktion:

$$\lambda a.\,a$$

- Oder in Python:

```
def lambda_function(a)
        return a
```

- Syntax für Lambda-definierbare Funktionen:

$$\underbrace{\lambda}_{\text{Funktionsdefinition Lambda}}\underbrace{x}_{\text{Funktionsargument}}.\underbrace{x+1}_{\text{Funktionskörper Lambda Term}}$$

- Lambda-Funktion:

  Definition **anonymer**, einstelliger Funktionen

Lambda-Term – Syntax

- Die Syntax eines Lambda-Terms t sei wie folgt definiert:

$$t ::= c \mid x \mid \lambda x.u \mid (u\ v)$$

  - c heißt ***Konstantensymbol***
  - x heißt ***Variablensymbol***
  - $\lambda x.u$ heißt ***Lambda-Abstraktion*** eines Lambda-Term u zu einer (anonymen) Funktion durch das Variablensymbol x
  - $(u\ v)$ heißt ***Lambda-Applikation*** eines Lambda-Terms u auf einen Lambda-Term v - also $u(v)$

Konstantensymbole

- nullstellige Funktionen

- fakultativ für das Lambda-Kalkül (nicht Teil der ersten formalen Definition)
- meist aus dem übergeordneten Calculus:
  - Beispielweise die natürlichen Zahlen für Lambda-Terme im arithmetischen Kontext

- Beispiele: $0,1, FIDO, BILL$

Dr.-Ing. Christian Braune

Variablensymbole

- Beispiele: x, y, z
- Alphabet für die Benennung von Argumenten der Funktionen
- frei oder gebunden (dazu später mehr)

Lambdaapplikation - Funktionsanwendung

- Funktionsapplikationen

$$(\_\ \_)$$

- Beispiel

$$(a\ b)$$

- Bedeutung
  - Wende die Funktion a auf den Wert b an
  - Setze b in die Funktion a ein

Dr.-Ing. Christian Braune

Lambdaapplikation - Funktionsanwendung

- Funktionsapplikationen

$$(\_\ \_)$$

- Beispiel

$$(\lambda a.\, a\ 3)$$

- Bedeutung
  - Setze den Wert 3 in die Identitätsfunktion ein

Lambdaapplikation - Funktionsanwendung

- Funktionsapplikationen

$$(\_\ \_)$$

- Beispiel

$$(3\ \lambda a.\, a)$$

- Bedeutung
  - Wende die Funktion 3 auf die Identitätsfunktion an

Lambda-Applikation

- Falls bei einer Lambda-Applikation $u\ v$,

  - u eine Konstante c ist,
    so ist $u\ v \ := c$

  - u eine Variable x ist,
    so ist $u\ v := \ v$

  - u eine Lambda–Abstraktion $\lambda x.t$ ist,
    so ist $u\ v \ := (\lambda x.t)v = txv$
    also das Einsetzen von v für das Funktionsargument x in u

  - u eine Lambda-Applikation $(s\ t)$ ist,
    so ist $u\ v := \ (s\ t)\ v \ = \ s(t)(v)$
    also das Einsetzen von t in s und danach das Einsetzen von v

Lambda-Applikation - Beispiel

- Funktionsanwendung

- Beispiel: $(\lambda x.x + 1)\ 3$
  - 3 + 1

- Beispiel: $(\lambda x.dog\ x)$ FIDO
  - dog FIDO

- anonyme Funktionsgeneration

- Beispiel: $\lambda x. x + 1$
  - einstellige Funktion mit einem Argument (x) und dem Term x + 1.

- Beispiel: $\lambda x. (dog\ x)$
  - einstellige Funktion mit einem Argument (x) und dem Term dog x.

- Beispiel: $\lambda x. \lambda y. (x + \ y)$
  - zweistellige Funktion als Aufrufhierachie zweier einstelliger Funktionen (Prinzip des Currying!)

Lambda-Terme – Beispiele

- $\lambda x.\, x$ – Identitätsfunktion
- $\lambda y.\, \lambda x.\, x$ – Funktion, die einem Argument (hier y) die Identitätsfunktion zuordnet
- $(\lambda y.\, y)\ (\lambda x.\, x)$ – Identitätsfunktion angewendet auf die Identitätsfunktion
- $\lambda z.\, (z\ \lambda x.\, x)$ – Funktion, die eine beliebige Funktion z auf die Identitätsfunktion anwendet

Berechnungen mit Lambda-Termen

- **Hinweis -** Lambda-Applikation ist links-assoziativ und bindet stärker als Lambda-Abstraktion:

$$s\ t\ u := (s\ t)\ u$$

$$\lambda x. t\ u := \lambda x. (t\ u)$$

- Beispiele:

$$(\lambda a. f)\ (\lambda b. c)\ x = f_a^{\lambda b.c} x$$
$$(\lambda a. f)\ ((\lambda b. c)\ x) = f_a^{c}$$

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

falls c ein Konstantensymbol oder ein nicht in x freier Lambda-Term ist

Beispiele Berechnung komplexer Terme

- Ausgehend auf vorhandenen wohlbekannten arithmetischen Funktionen (+,-,...)* lassen sich komplexe Berechnungen ausführen:

$$
\begin{aligned}
&(\lambda w.w)\ (\lambda x.\lambda y.x\ y)\ (\lambda z.z+1)\ 3\\
&= (\lambda w.w)\ (\lambda x.\lambda y.x\ y)\ (\lambda z.z+1)\ 3\\
&= (\lambda x.\lambda y.x\ y)(\lambda z.z+1)3\\
&= (\lambda y.(\lambda z.z+1\ )\ y)\ 3\\
&= (\lambda z.z+1)\ 3\\
&= 3+1=4
\end{aligned}
$$

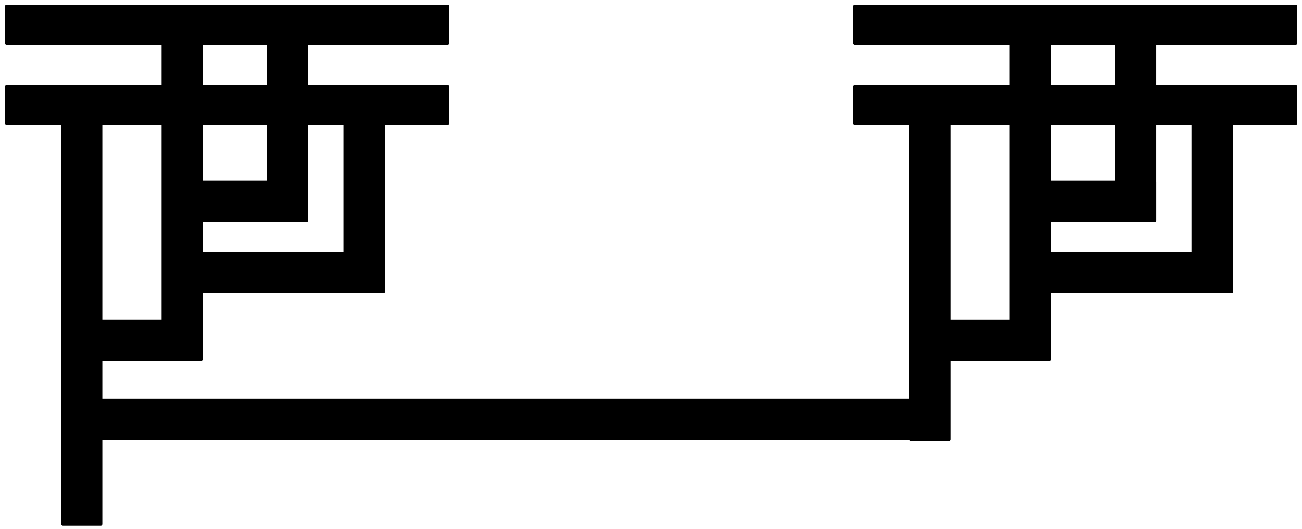
*werden hier zur besseren Lesbarkeit in der Infix-Schreibweise verwendet

Darstellungsformen für Lambda-Ausdrücke

- Lambda-Ausdrücke

$$\left(\left(\lambda x.\left(\lambda y.\left(y((xx)y)\right)\right)\right)\left(\lambda x.\left(\lambda y.\left(y((xx)y)\right)\right)\right)\right)$$

- Tromp Diagramme



Darstellungsformen für Lambda-Ausdrücke

- Lambda-Ausdrücke

$$\left(\left(\lambda x.\left(\lambda y.\left(y((xx)y)\right)\right)\right)\left(\lambda x.\left(\lambda y.\left(y((xx)y)\right)\right)\right)\right)$$

- Tromp Diagramme

Darstellungsformen für Lambda-Ausdrücke

- Lambda-Ausdrücke

$$\left(\left(\lambda x.\left(\lambda y.\left(y((xx)y)\right)\right)\right)\left(\lambda x.\left(\lambda y.\left(y((xx)y)\right)\right)\right)\right)$$

- Tromp Diagramme

Darstellungsformen für Lambda-Ausdrücke

- Lambda-Ausdrücke

$$\left(\left(\lambda x.\left(\lambda y.\left(y\left(\left(xx\right)y\right)\right)\right)\right)\left(\lambda x.\left(\lambda y.\left(y\left(\left(xx\right)y\right)\right)\right)\right)\right)$$

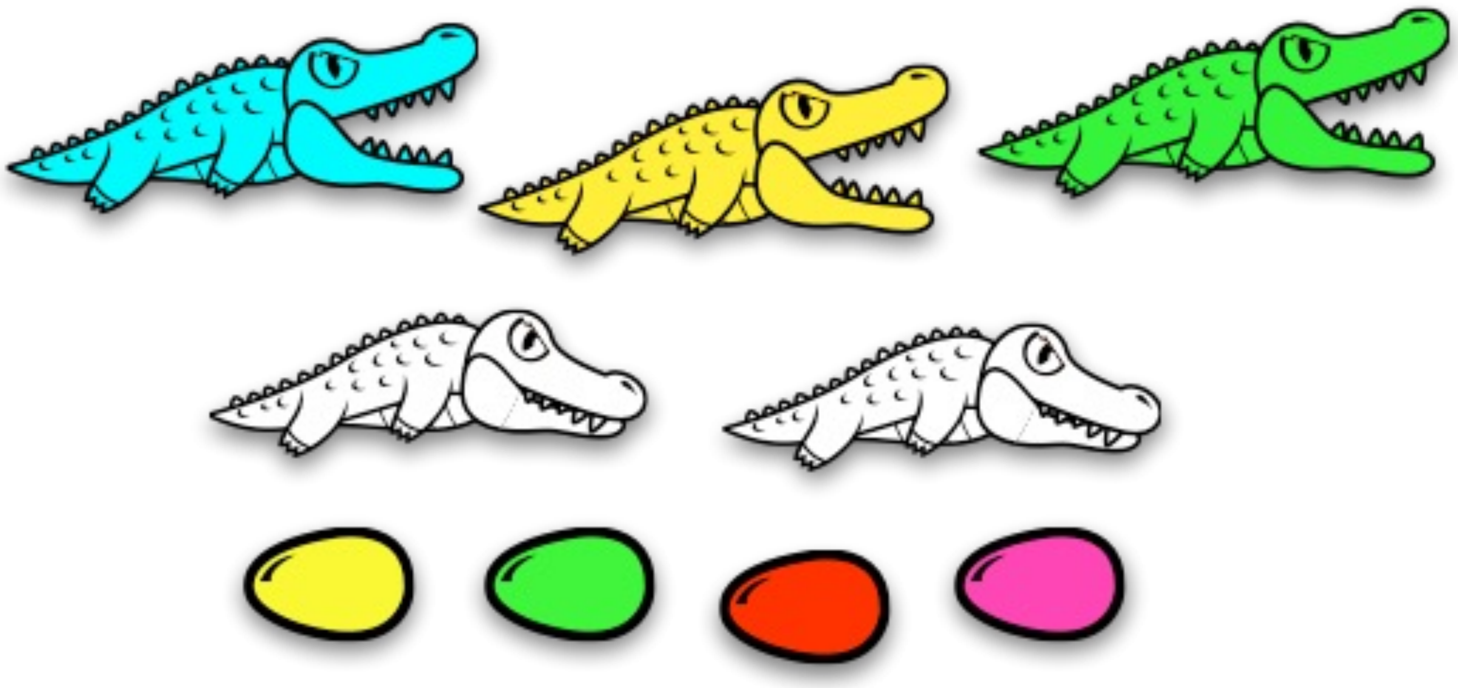
- Tromp Diagramme

Darstellungsformen für Lambda-Ausdrücke

- Lambda-Ausdrücke

$$\Big(\Big(\lambda x.\Big(\lambda y.\Big(y((xx)y)\Big)\Big)\Big)\Big(\lambda x.\Big(\lambda y.\Big(y((xx)y)\Big)\Big)\Big)\Big)$$

- Dinosaurier! (not to scale)



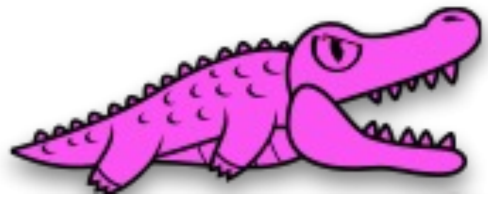
- Lambda-Ausdrücke bestehen aus diesen drei Komponenten:

- Variablen (und Konstanten)

$$a$$

- Funktionsdefinitionen (a.k.a. Lambda Abstraktion)

$$\lambda a.\_$$



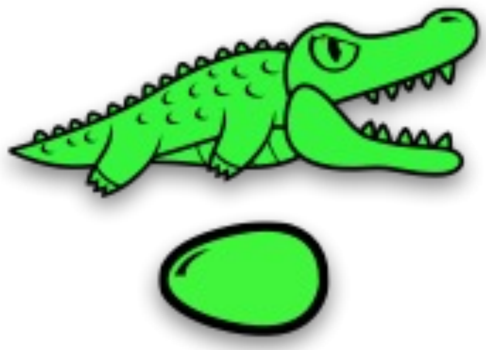
- Funktionsapplikationen

$$(\_\ \_)$$

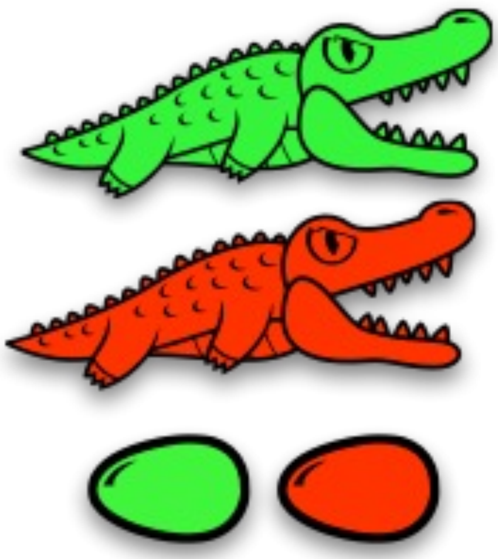
- Blanks sind Platzhalter

Alligatoren bilden Familien

- Eine kleine Familie

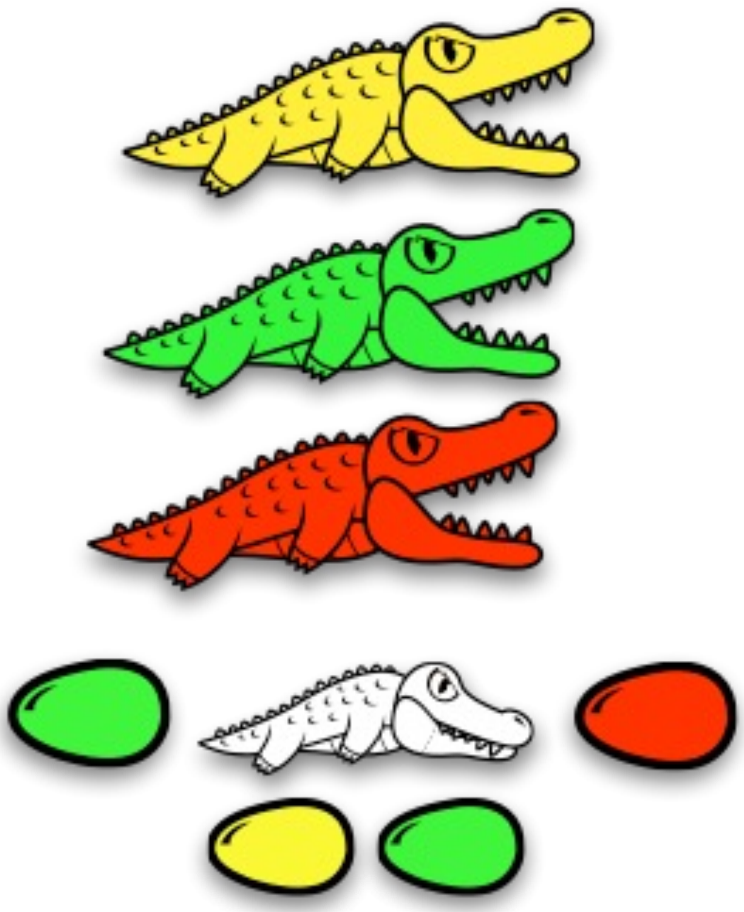


- Eine etwas größere Familie



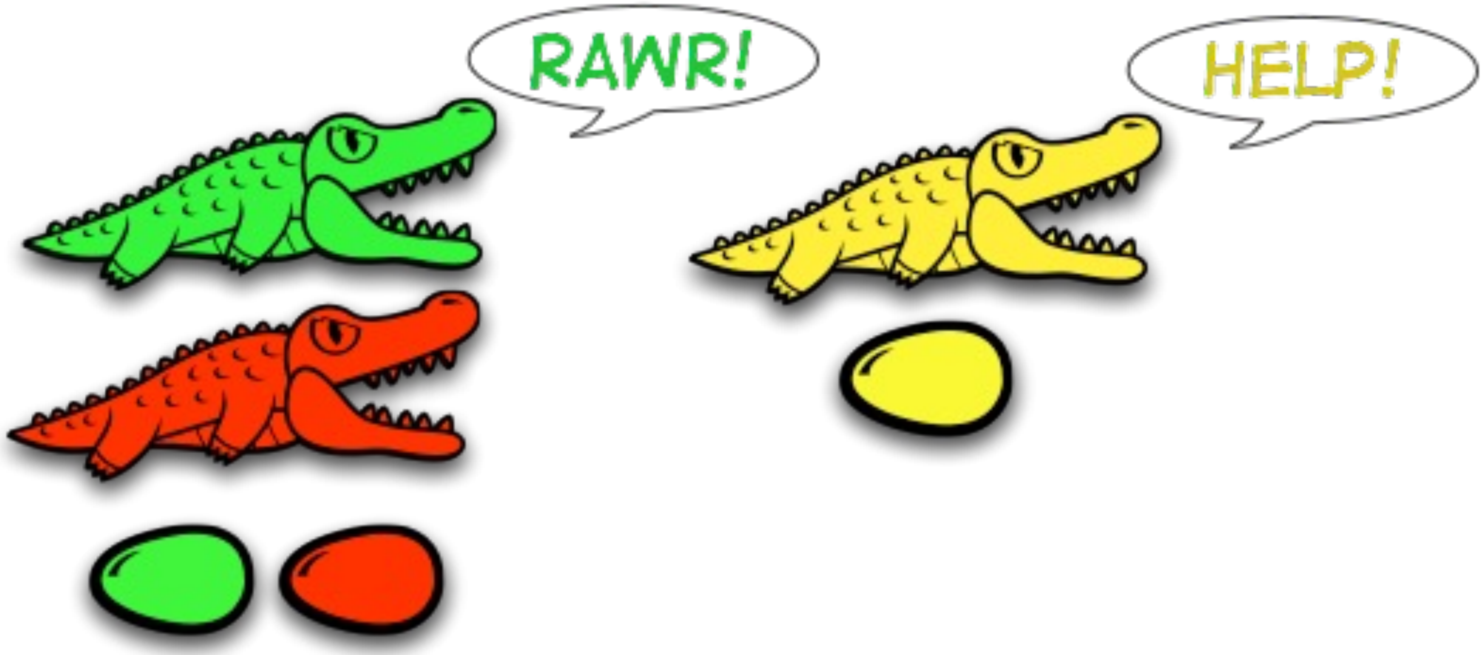
Alligatoren bilden Familien

- Eine riesige Familie

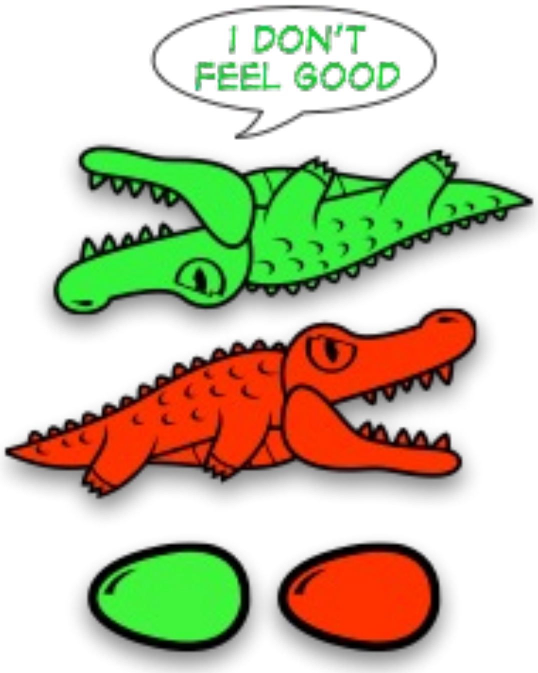


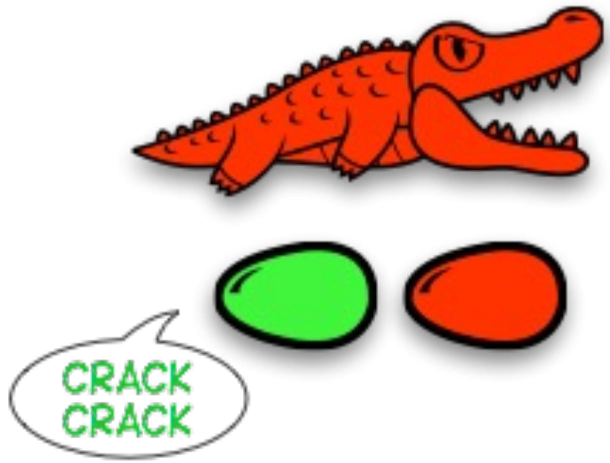
Alligatoren bilden Familien

- Was passiert, wenn wir mehrere Familien haben?

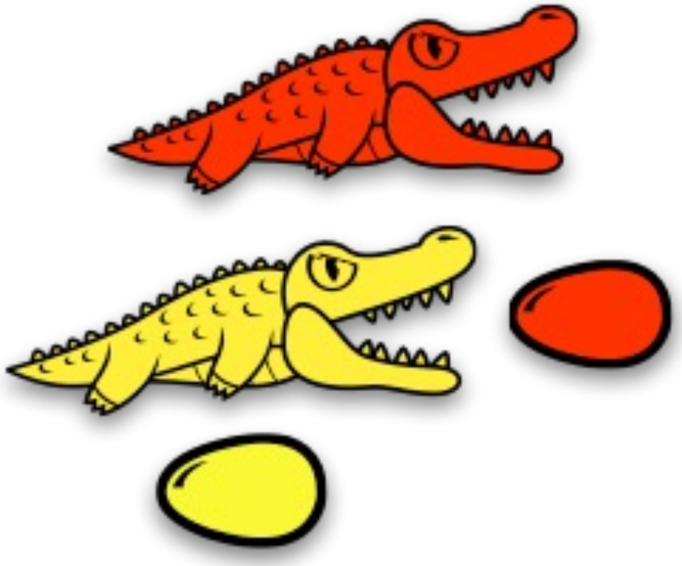


- Wenn ein Alligator eine andere Familie frisst, schlüpft jedes (gleichfarbige) Ei, dass er bewacht

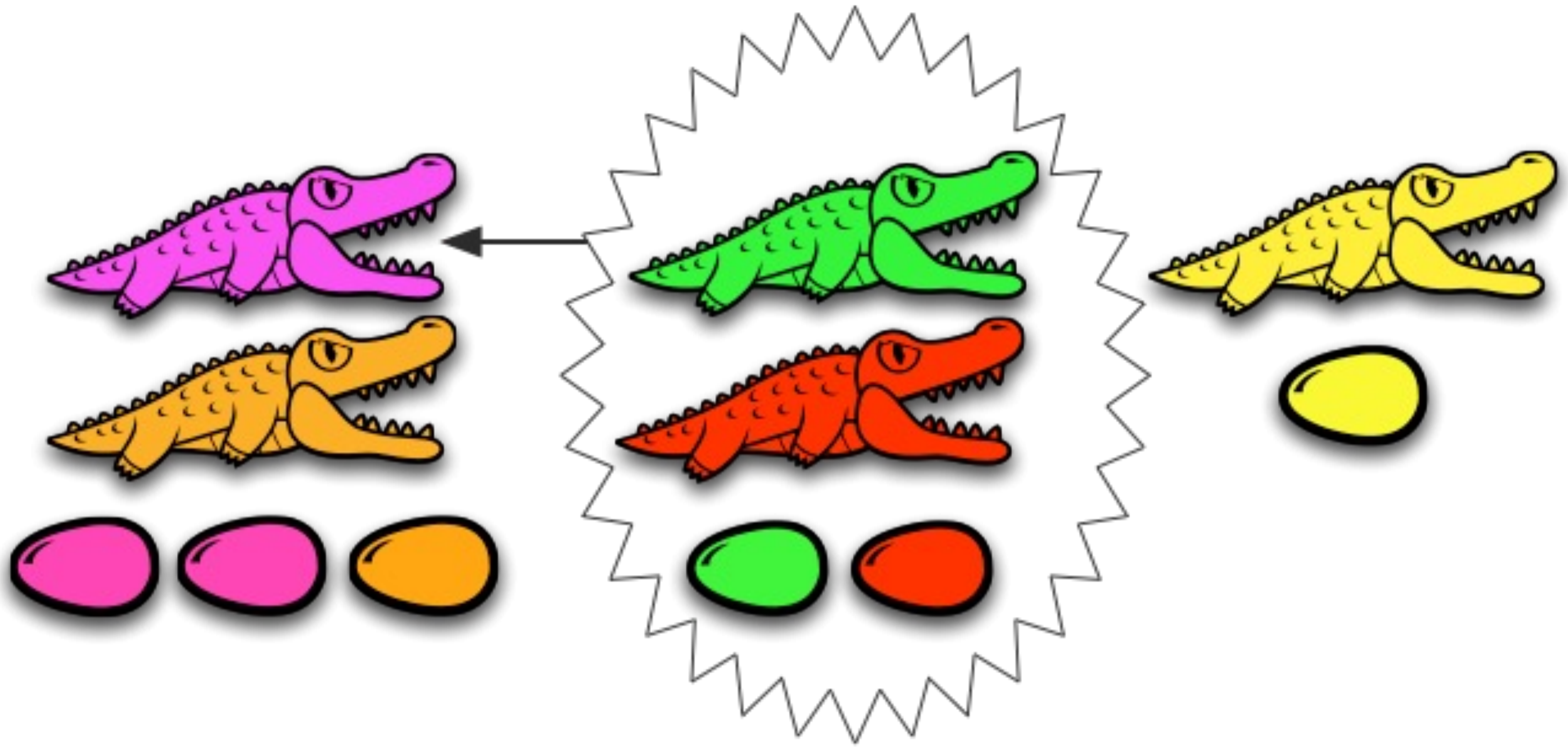




- Wenn ein Alligator eine andere Familie frisst, schlüpft jedes (gleichfarbige) Ei, dass er bewacht – und heraus kommt, was der Alligator gefressen hat



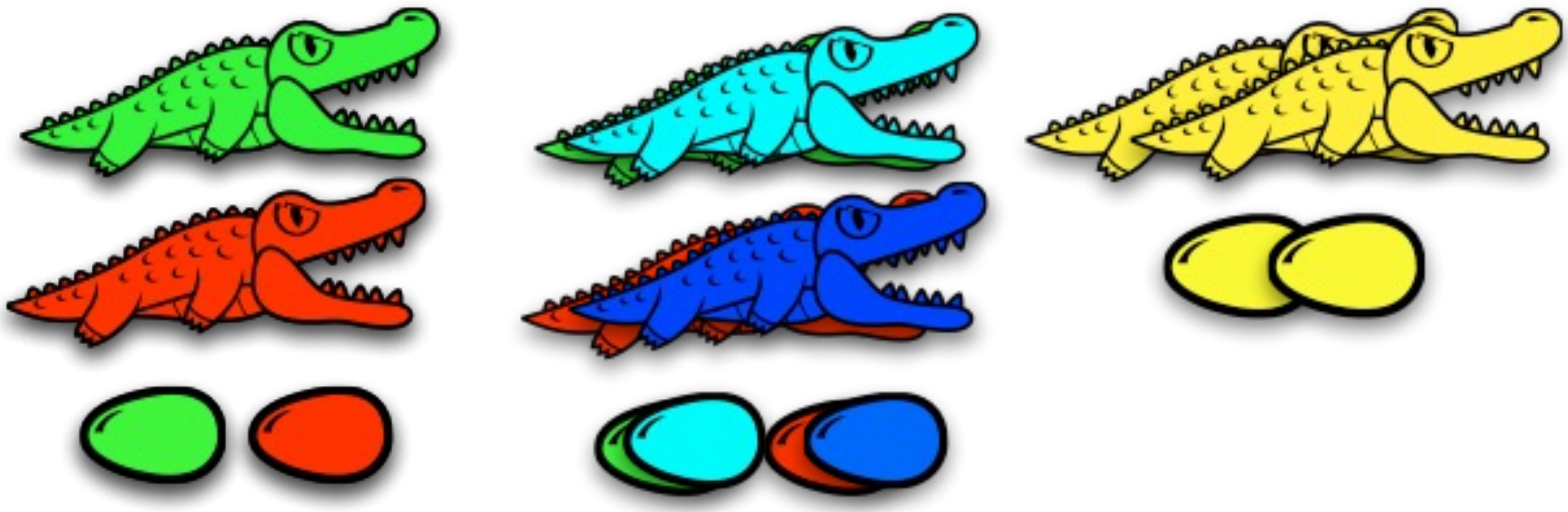
- Alligatoren fressen von links nach rechts und von oben nach unten



Die Farbregel

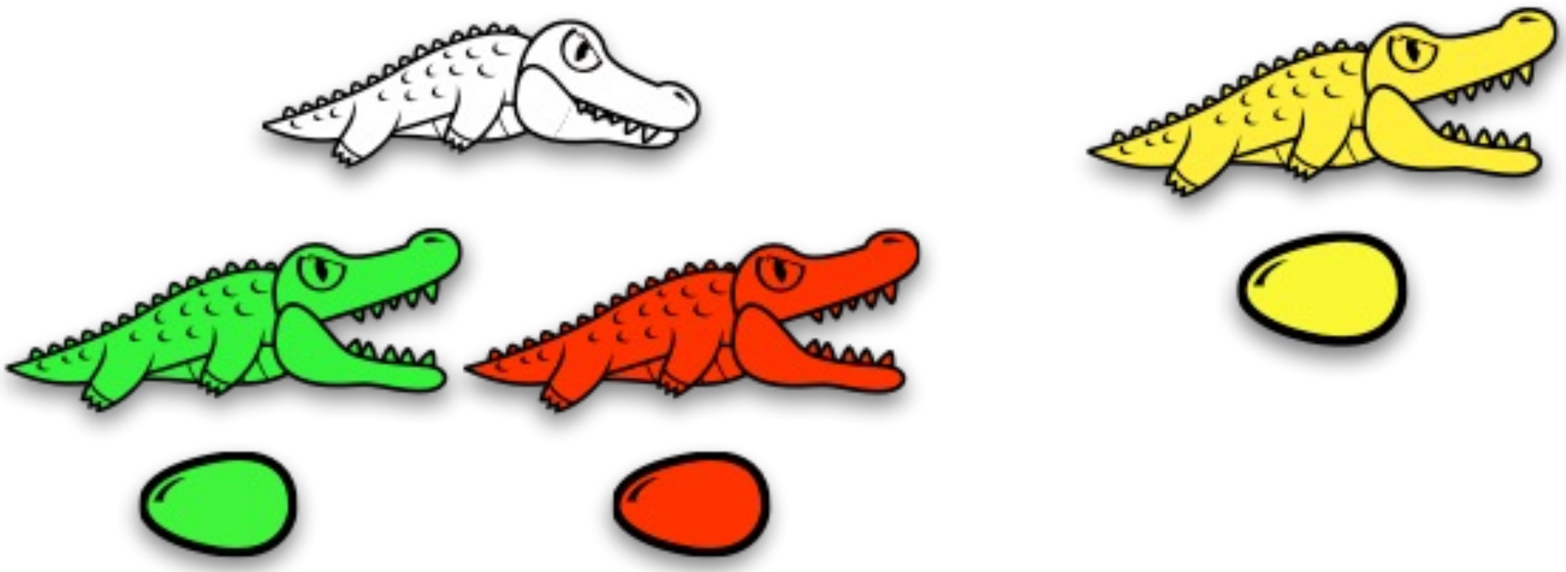
- Was passiert in diesem Fall?

- Wenn ein Alligator eine Familie fressen will, die Alligatoren oder Eier seiner eigenen Farbe enthält, müssen wir erst die Farbe dieser Familie ändern



Alte Alligatoren

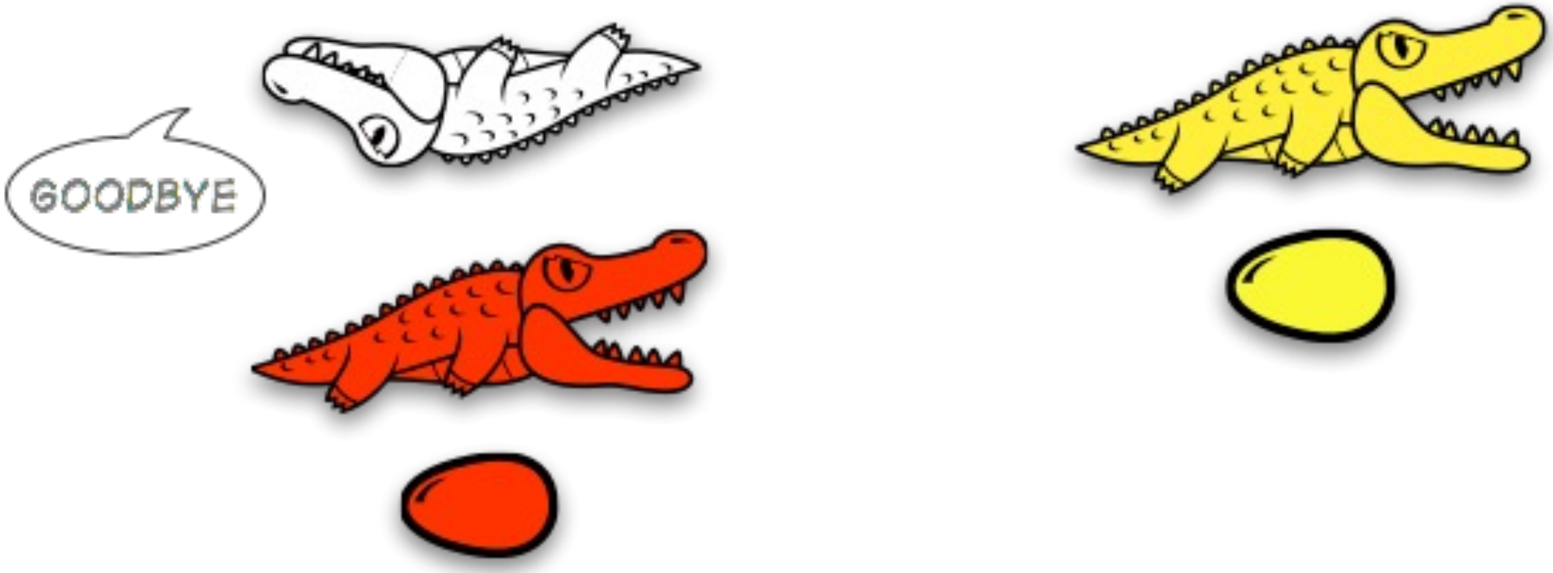
- Manche Alligatoren sind alt und satt und müde
- Ihr einziger Lebenszweck bestellt darin, mehrere Familien zu beschützen



Dr.-Ing. Christian Braune

Alte Alligatoren

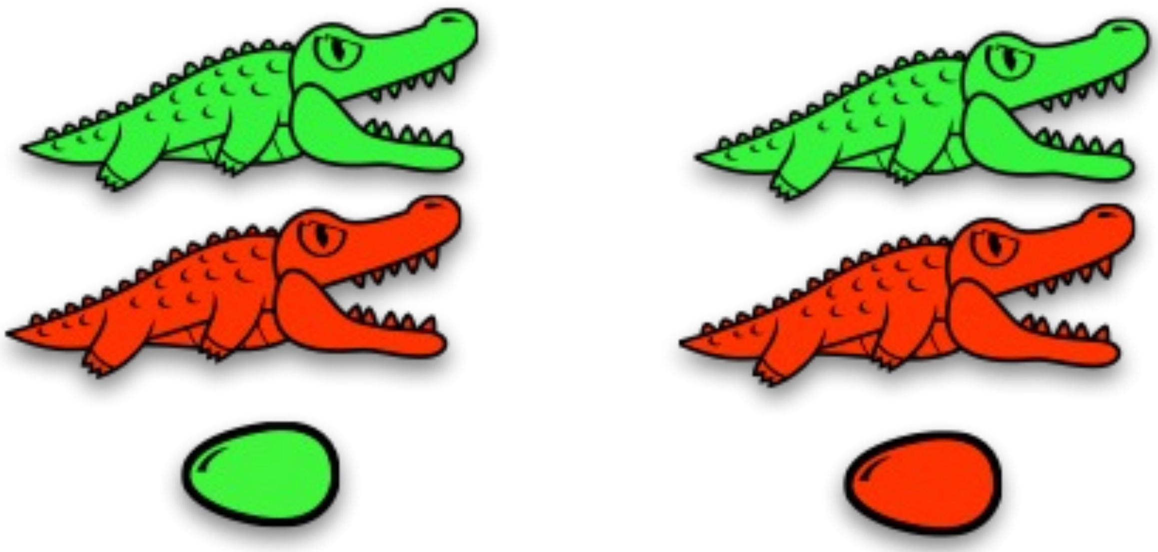
- Manche Alligatoren sind alt und satt und müde
- Ihr einziger Lebenszweck bestellt darin, mehrere Familien zu beschützen
- Wenn alte Alligatoren nur noch eine einzelne Familie beschützen, ...



Wahre und Falsche Alligatorenfamilien

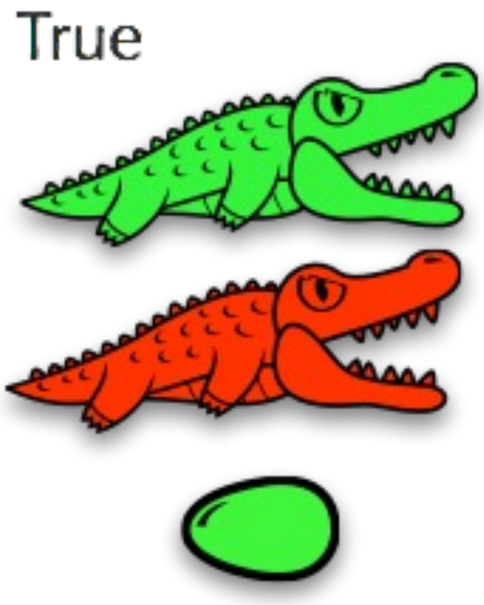
- Diese beiden Familien entsprechen den logischen Funktionen (!)

`true` und **`false`**

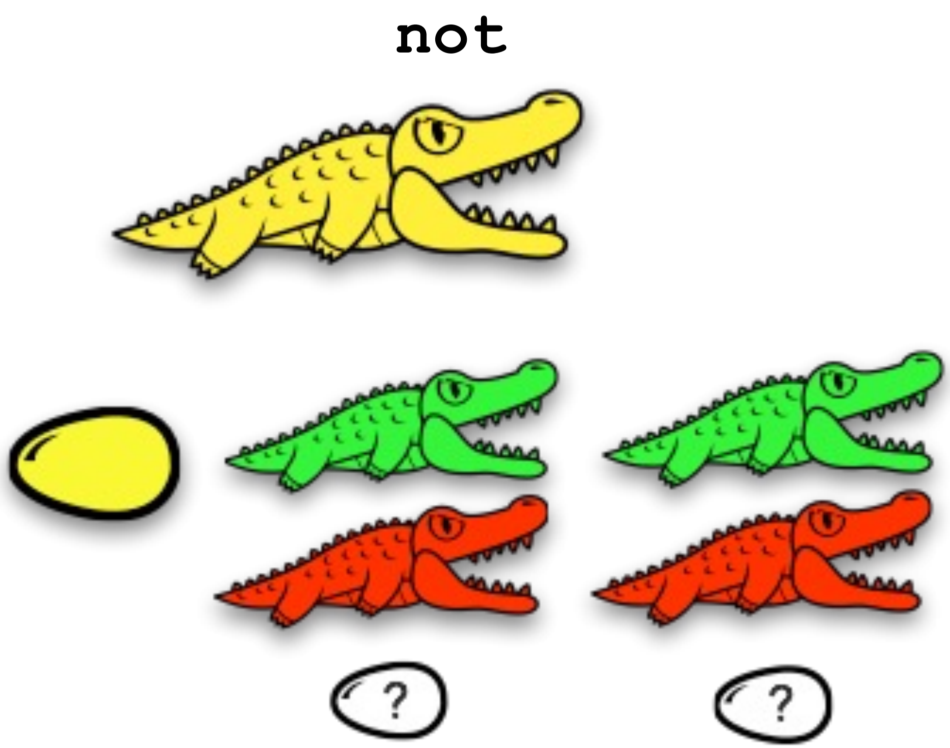


Wahre und Falsche Alligatorenfamilien

- Anwendung der Funktion **`true`** auf die Werte (!) **`True`**/**`False`**



- Diese Alligatorfamilie entspricht der logischen Funktion



- Welche Farbe müssen die beiden Eier haben, damit als Ergbnis **`true`** bzw. **`false`** übrig bleiben, wenn **`not`** die Familie **`false`** bzw. **`true`** frisst?

Formale Beschreibung der Berechnung

- Umformung und Vereinfachung der Terme formal durch:
  - **α-Konversion:** Möglichkeit zum Verändern des Variablennamens
  - **β-Konversion:** Möglichkeit zum Anwenden von Funktionsparametern (Lambda-Applikation)
- Hierzu Notwendigkeit der formalen Definition der Begriffe
  - freie Variablen
  - Ersetzungsfunktion

Definition: freie Variablen

Seien s, t Lambda-Terme, x ein Variablensymbol und c ein Konstantensymbol. Dann ist die Menge der freien Variablen $freev(t)$ folgendermaßen definiert:

- $freev(c) = \emptyset$
- $freev(x) = \{x\}$
- $freev(t\ u) = freev(t) \cup freev(u)$
- $freev(\lambda x. t) = freev(t) \setminus \{x\}$

$freev$ gibt zu einem beliebigen Lambda-Term die Menge aller freien Variablen an.

x ist nicht frei in t, g.d.w. $x \notin freev(t)$

Dr.-Ing. Christian Braune

Beispiele freie Variablen

- $freev(\lambda x. \lambda y. (x + y)) = \emptyset$
- $freev(\lambda x. y) = \{y\}$
- $freev(\lambda y. (x + y)\ \lambda x. (y + x)) = \{x, y\}$
- $freev((\lambda x. (x + 42))\ (\lambda x. x)) = \emptyset$

Definition: Ersetzungsfunktion

Seien r, s, t Lambda-Terme, c ein Konstantensymbol und x, y Variablensymbole. Dann beschreibt $t[x \leftarrow s]$ das Ersetzen aller Vorkommen von x in t durch s und ist wie folgt definiert:

- $c[x \leftarrow s] = c$
- $y[x \leftarrow s] = y, falls\ y \neq x$
- $x[x \leftarrow s] = s$
- $(t\ r)[x \leftarrow s] = t[x \leftarrow s]\ r[x \leftarrow s]$
- $(\lambda y.t)[x \leftarrow s] = \lambda y.(t[x \leftarrow s])$, wenn y nicht frei in s (oder x in t gar nicht vorkommt)
- $(\lambda y.t)[x \leftarrow s] = \lambda z.(t[y \leftarrow z][x \leftarrow s])$, andernfalls, wobei z ein neues Variablensymbol ist, das nicht frei in t oder s vorkommt

Dr.-Ing. Christian Braune

Beispiele Ersetzungsfunktion

- $x[x \leftarrow 3] = 3$

- $((\lambda y.\, y + x)\ x)[x \leftarrow 3] =$
- $((\lambda y.\, y + x)\ x)[x \leftarrow 3] = (\lambda y.\, y + 3)\ 3$

- $(\lambda x.\, y + 2)[x \leftarrow z] =$
- $(\lambda x.\, y + 2)[x \leftarrow z] = (\lambda x.\, y + 2)$

- $(\lambda x.\, y\ 2)[y \leftarrow \lambda z.\, z + 2] =$
- $(\lambda x.\, y\ 2)[y \leftarrow \lambda z.\, z + 2] = \lambda x.\, (\lambda z.\, z + 2)\ 2$

- $(\lambda x.\, y\ 2)[y \leftarrow \lambda x.\, x + 2] =$
- $(\lambda x.\, y\ 2)[y \leftarrow \lambda x.\, x + 2] = \lambda x.\, (\lambda x.\, x + 2)2$

* inneres x != äußeres x

- $(\lambda x.\, x + y)[y \leftarrow x] =$
- $(\lambda x.\, x + y)[y \leftarrow x] = \lambda z.\, z + x$

* Umbenennung mit neuem Variablensymbol z

Definition: α- und β-Konversion

Seien s, t Lambda-Terme und x, y Variablensymbole

- α-Konversion bzw. α-Äquivalenz – (Umbenennen von Variablensymbolen) sei dann:

  - $(\lambda y. t)[y \leftarrow x] =_\alpha \lambda x. (t[y \leftarrow x])$, wenn $x \notin freev(t)$
    - Dies ist eine entspanntere Version der Farbregel

- β-Konversion bzw. β-Äquivalenz – (Anwendung der Lambda-Applikation) sei dann:

  - $(\lambda y. t)\, s =_\beta t[y \leftarrow s]$
    - Dies ist die Fressregel

Beispiele Konversionen

- α- und β-Äquivalenz von Lambda-Termen:

$$\begin{aligned}\lambda x.\,(\lambda x.\,x)\ 3\ &=_\alpha\ \lambda y.\,(\lambda y.\,y)\ 3\\ &=_\alpha\ \lambda x.\,(\lambda y.\,y)\ 3\\ &=_\alpha\ \lambda y.\,(\lambda x.\,x)\ 3\end{aligned}$$

$$\begin{aligned}\big(\lambda x.\,((\lambda x.\,x)4)\big)\,3\ &=_\beta\ (\lambda x.\,x)4\\ &=_\beta\ 4\end{aligned}$$

Turingmaschinen

- Von Alan Turing 1936/37 vorgestelltes Model
- Eine Turingmaschine kann im wesentlichen nur die folgenden Operationen:
  - Im Speicher einen Adressschritt vorwärts oder rückwärts gehen
  - Das aktuelle Zeichen, was im Speicher steht auslesen
  - An der aktuellen Stelle im Speicher ein Zeichen setzen
- Turingmaschinen haben unendlich viel Speicher und eine Menge von Zuständen
  - der aktuelle Zustand entscheidet darüber, welche der Operationen durchgeführt werden soll und was der nächste Zustand sein soll
- Turingmaschine n sind quasi "speicherprogrammierbare Computer"

Turing-Vollständigkeit

- Ein System heißt turing-vollständig, wenn sie jede turing-berechenbare Funktion berechnen kann

- Eine Funktion ist turing-berechenbar, wenn sie von einer Turingmaschine berechnet werden kann

- Das Lambda-Kalkül ist turing-vollständig.

  [A.M.Turing, On Computable Numbers with an Application to the Entscheidungsproblem, Journal of Symbolic Logic vol.2(1), Cambridge, 1936]

- Bisher: Anwendung von Funktionen

- Aber:
  - keine Entscheidungen möglich
  - keine Form von Iteration möglich

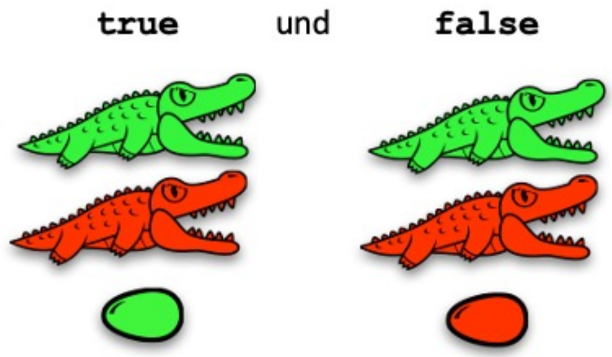
- Ein IF-Statement in Java hat folgenden Aufbau

```
if ( condition ) { then_block } else { else_block }
```

- Wir brauchen also eine Funktion, die anhand eines Wahrheitswertes entscheidet, welche Funktion ausgeführt werden soll

- Wir hatten bisher schon die Ausdrücke TRUE und FALSE

  - TRUE : $\lambda x.\lambda y.x$
  - FALSE: $\lambda x.\lambda y.y$



- Beides sind zweistellige Funktionen.

- Die Funktion TRUE gibt den ersten Parameter zurück.
  Die Funktion FALSE gibt den zweiten Parameter zurück.

Das IF im Lambda-Kalkül

- Diese Funktionen können wir benutzen, um unser klassisches IF im Lambda-Kalkül auszudrücken

```
if ( condition ) { then_block } else { else_block }
```

```
if ( z ) { a } else { b }
```

$$\lambda a.\lambda b.\lambda z.z\ a\ b$$

- a ist die Funktion, die zurückgegeben werden soll, wenn z wahr ist
- b ist die Funktion, die zurückgegeben werden soll, wenn z falsch ist

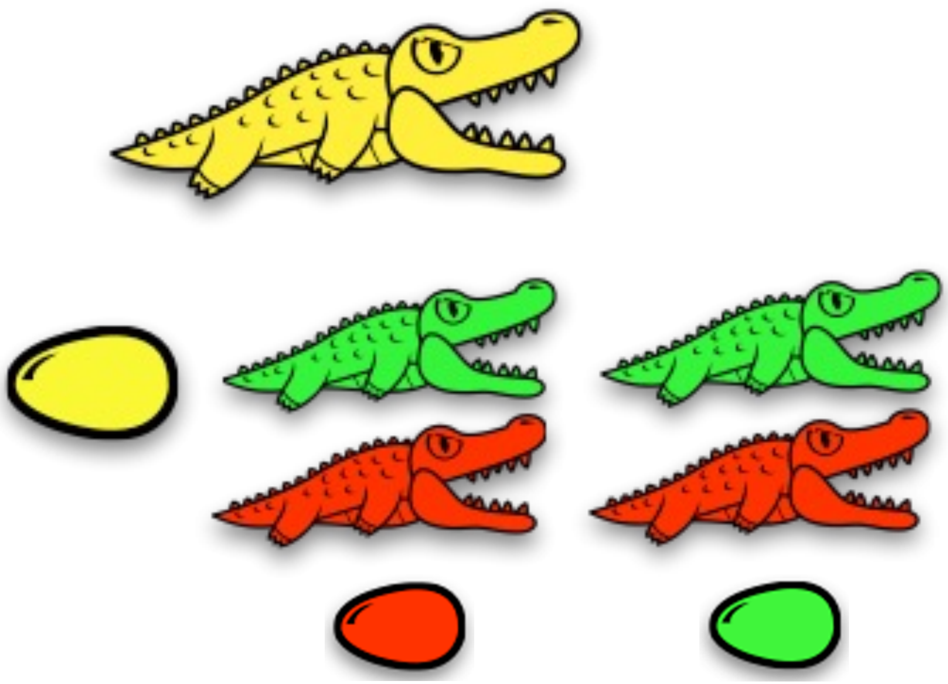
Logische Kombinatoren

- Nur die beiden Werte TRUE und FALSE helfen uns noch nicht weiter

- logische Operatoren:
  - NOT
  - AND und NAND
  - OR und NOR
  - XOR und XNOR

- NOT

$$\lambda a.(a\ FALSE\ TRUE)$$



- AND

$$\lambda a.\lambda b.(a\ b\ FALSE)$$

- Wenn a TRUE ist, wird der erste Parameter zurückgegeben → also der Wert von b

- Wenn a FALSE ist, wird der zweite Parameter zurückgegeben → also FALSE

Logische Kombinatoren

- NAND
- OR
- NOR
- XOR
- XNOR

- Hausaufgabe!

- Bisher: Anwendung von Funktionen

- Aber:
  - keine Entscheidungen möglich
  - keine Form von Iteration möglich

- Wir kennen bereits die Identitätsfunktion

$$\lambda x.\, x$$

- Was passiert hier?

$$\lambda x.\, x\ x$$

- Diese Funktion heißt ω . Sie wendet die übergebene Funktion x auf sich selbst an

- Z.B.

$$\omega(\lambda y.\, y + 5)\ 3$$
$$= (\lambda x.\, x\ x)(\lambda y.\, y + 5)\ 3$$
$$= \big((\lambda y.\, y + 5)(\lambda y.\, y + 5)\big)\ 3$$
$$= (\lambda y.\, (y + 5) + 5)\ 3$$
$$= \big((3 + 5) + 5\big)$$
$$= (8 + 5)$$
$$= 13$$

- Was passiert hier?

$$
\begin{aligned}
\Omega = \ & \omega\omega \\
= \ & (\lambda x.\, x\ x)(\lambda x.\, x\ x) \\
= \ & (\lambda x.\, x\ x)(\lambda x.\, x\ x) \\
= \ & (\lambda x.\, x\ x)(\lambda x.\, x\ x)
\end{aligned}
$$

- Endlosrekursion!

Rekursion im Lambda-Kalkül

- Was passiert hier?

$$\lambda y.\left(\lambda x.y(x\ x)\right)\left(\lambda x.y(x\ x)\right)\ F$$

$$\begin{aligned}
&\lambda y.\left(\lambda x.{\color{blue}y}(x\ x)\right)\left(\lambda x.{\color{blue}y}(x\ x)\right)\ {\color{blue}F}\\
&= \left(\lambda x.F({\color{green}x\ x})\right){\color{green}\left(\lambda x.F(x\ x)\right)}\\
&= F\ \left(\left(\lambda x.F(x\ x)\right)\left(\lambda x.F(x\ x)\right)\right)\\
&= F\left(F\ \left(\left(\lambda x.F(x\ x)\right)\left(\lambda x.F(x\ x)\right)\right)\right)
\end{aligned}$$

Dr.-Ing. Christian Braune
OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG INF FAKULTÄT FÜR INFORMATIK

- Diese Funktion heißt Y-Kombinator

$$
\begin{aligned}
&\lambda y.\left(\lambda x.\, y(x\ x)\right)\left(\lambda x.\, y(x\ x)\right) F \\
&= Y\ F \\
&= F(Y\ F) \\
&= F\left(F(Y\ F)\right) \\
&= \ \dots
\end{aligned}
$$

- Y-Kombinator + IF = Rekursion

In der Praxis…

Lambda-Expression - Allgemein

- Anwendung anonymer Funktionen in vielen Hochsprachen:

  - Python: `lambda x: x + 4`
  - JavaScript: `function(x) { x + 4 }`
  - Ruby: `->(x) { x + 4 }`
  - C++: `[] (x) -> int { x + 4 }`
  - Java 8: `x -> x + 4`

Dr.-Ing. Christian Braune
Otto von Guericke Universität Magdeburg INF Fakultät für Informatik

Lambda-Expression in Scala I

- Allgemein:

```
(p-name: p-typ) => funktionskörper;
```

- Vorteile:
  - kürzere Schreibweise/Übersichtlichkeit
  - direkt (als Parameter/Rückgabewert/in Operationen) benutzbar
  - kein Kontext!

Lambda-Expression – Scala II

- Nutzbar als Funktion:

```
scala> ((x: Int) => x + 5) (7);
: Int = 12
```


Dr.-Ing. Christian Braune

- Nutzbar als Rückgabewert:

```
scala> def sumWithX(x: Int) =
     |     (y: Int) => x + y;
     : (x: Int) Int => Int
scala> sumWithX(7);
     : Int => Int = <function1>
scala> sumWithX(7)(5)
     : Int = 12
```

Lambda-Expression – Scala IV

- Nutzbar als Parameter

```
scala> def strecken(f: Double => Double, k: Double):
     |         Double => Double = k * f(_);
scala> strecken((x) => x + 4 , 2);
     : Double => Double = <function1>
scala> strecken((x) => x + 4, 2)(2);
     : Double = 12.0
```

Zusammenfassung

- Lambda Kalkül ist Turing-mächtiger Beschreibungsformalismus für Programme
- Funktionen sind anonym; d.h. namenlos
- (Beweise i.d.R. viel einfacher als mit Turing-Maschinen)

- Zentrale Elemente
  - Lambda-Abstraktion: Bilden einer anonymen Funktion
  - Lambda-Applikation: Einsetzen von Werten

- Umsetzung als Lambda-Expressions in praktisch allen gängigen Programmiersprachen
    - als Funktion
    - als Rückgabetyp
    - als Parameter

Dr.-Ing. Christian Braune