Identity

Js >>

```js
const I = a => a
I(1)   // 1
I(2)   // 2
I(I)   // [function: I]
```

λ

$\lambda a . a$

$\Rightarrow \; I \, I = I$

Js

`I = a => a`

`I(I) = I`

also ===

<u>Haskell:</u> id function: `id 5 == 5`

Parameter

Return expression

$\underbrace{\lambda \; a \; . \; a}_{\text{Lambda Abstraction}}$

function signifier

decimator

Lambda Abstraction

≙ ~~unary~~ anonymous function

↳ it takes a single input

λ calculus syntax

expression	::=	variable	(identifier)
	|	expression expression	(application)
	|	λ variable . expression	(abstraction)
	|	(expression)	(grouping)

Variables

Variables are immutable, there is no concept of assignment

↳ binding

Application:

λ   f a   JS   f(a)

f a b   f(a)(b)

No brackets in λ expression

All functions are Unary -> Currying

↳ example: JS

```
add = a => b => a + b
add (1) (2)   // 3
```

We can specify the application rules with parentheses:

λ $(f\ a)\ b$ fs $(f(a))(b)$

Default: function application is leftassociative

$\Rightarrow$ $(f\ a)\ b \overset{\wedge}{=} f\ a\ b$

This is only needed to force a different order

λ $f\ (a\ b)$ fs $f(a(b))$

Abstraction

λ $\lambda a.b$ fs $a \Rightarrow b$

$\lambda a.b\ x$ $a \Rightarrow b(x)$

$\lambda a.(b\ x)$ $a \Rightarrow (b(x))$

↳ useless ()

$(\lambda a.b)\ x$ $(a \Rightarrow b)(x)$

↳ usefull ()

$\lambda a.\lambda b.\ a$ $a \Rightarrow b \Rightarrow a$

β-reduction

↳ take a function and apply it to its argument

$(\lambda a . a)\ \lambda b . \lambda c . b\ \ (x)\ \lambda e . f$

function   parameter

$\Rightarrow\ (\lambda b . \lambda c . b)\ \ (x)\ \lambda e . f$

$\Rightarrow\ (\lambda c . x)\ \ \lambda e . f$

$\Rightarrow\ x$

↑

„ β normal form" (fully evaluated function)

Mockingbird

(λ) $M := \lambda f . f f$ (JS) `M = f => f(f)`

„ self - application combinator"

(JS) ⇒

```
M = f => f(f)

M(I) == I  // true

try { M(M) } catch (e) { console.log(e.message) }
↳ // call stack size exceeded
```

$M\ I = I\ I = I$

↳ reduces to β normal form

$M\ M = M\ M = M\ M = \ldots$ Ω (omega combinator)

↳ "the end"

⇒ We don't know if an expression has a β normal form (aka. Halting problem)

Abstractions II

(λ) $\lambda a.\lambda b.\lambda c.b$ is equivalent to: $\lambda abc.b$

(JS) a => b => c => b

a => b => c => b

~~(a, b, c) => b~~

The arguments don't come in simultaneously but one after another

⇒ β reduction:

$$((\lambda a.a)\ \underline{\lambda bc.b})(x)\ \lambda e.f$$

$$\Rightarrow (\lambda bc.b)(x)\ \lambda e.f$$

$$\Rightarrow (\lambda c.x)\ \lambda e.f \Rightarrow x$$

Kestrel

λ $K := \lambda ab.a$ JS $K = a \Rightarrow b \Rightarrow a$

=> takes 2 things and always returns the 1st one

$K\,M\,I = M$ $K(M)(I) === M$

$K\,I\,M = I$ $K(I)(M) === I$

Haskell: const 7 2 == 7

This can be used to create a funcion which is fixed on a specific value.

$K\,I\,x\,y = I\,y = y$

=> $K\,I\,x\,y = y$ => „hide"

Kite

Combinations of combinators create combinators

λ $KI := \lambda ab.b$ JS $KI = a \Rightarrow b \Rightarrow b$

$KI\,M\,KI = KI$ $KI(M)(KI) === KI$

Combinator: function with no free variables

Variable in a function body that's not bound to some parameter.

$\lambda b.b$ -> Combinator

$\lambda b.a$ -> No Combinator

$\lambda ab.a$ -> Combinator

$\lambda abc.c(\lambda e.b)$ -> Combinator

sym	λ	use
I	$\lambda a.a$	Identity
M	$\lambda f.ff$	self-application
K	$\lambda ab.a$	first, const
KI	$\lambda ab.b$ $= K\,I\,xy$	second

More combinators can be created by combining Combinators

Cardinal

λ $\lambda fab.\ fba$ -> flipping Arguments

JS `C = f => a => b => f(b)(a)`

$$CKIM = M = \underline{KI}\ IM$$

Haskell: `flip const 1 8 // 8`

Booleans

Example: JS `!x == y || (a && z)`

↓

λ , How???

Take a look at the following use case for Booleans:

JS `const result = bool ? expr1 : expr2;`

// true // false

Turn it into a function call

(K) (KI)

λ result := func expr1 expr2

=> True := K False := KI

(T) (F)

Negation: (Js) $!p = \text{not}(p)$

$\Rightarrow$ (λ) Not p $\Rightarrow$ $p\ F\ T$

This unknown boolean is a function which when its true selects the first argument (F) and when its false selects the second argument (T)

$\Rightarrow$ (λ) $\text{Not} := \lambda p.\, p\, F\, T$

(Js) `const not = p => p(F)(T)`

Church encoding: Booleans

Sym	λ	use
T	$\lambda ab.a$	True
F	$\lambda ab.b$	False
	$\lambda p.pFT$	Not
C	$\lambda fab.fba$	Not

Better Representation of Not:

① Not $k = kI$

↳ Not $(\lambda ab.a) = \lambda ba.a$

② Not $kI = k$

↳ Not $(\lambda ba.a) = \lambda ab.a$

=> $C\ (kI) = k$

$C\ k = kI$

The Cardinal is already an representation of the boolean not.

Problem: JS

$C\ (T) == F$ // false

$Not(T) == F$ // true

The Not function selects between T / F but the Cardinal (C) creates a new function which behaves identically.

- extentional equality => for every Input they generate the same output

↳ $C\ k = kI$: Extensionaly equal

- intentional equality: both have the same source

  Not T = F is intentionally equal

Boolean And

(λ) $AND := \lambda pq.pqF$

(Js) const and = p => q => p(q)(F)

p false -> select F

p true -> select q (T/F)

This behaves like a logic AND

Beautification: If p is false and it should

select false -> p can select itself

$\lambda pq.pqp$

This creates extentionally equal functions.

Boolean OR

λ $OR := \lambda pq.\ p\,T\,q$

JS `const or = p => q => p(T)(q)`

$\Rightarrow \quad \lambda pq.\ p\,p\,q$

Trick: $\underbrace{(\lambda pq.\ p\,p\,q)}\, x\, y = x\, x\, y$

$M^{*}\, x\, y = x\, x\, y$

$\uparrow$

identically to M, but for clarity

The Mockingbird is an already existing implementation of the OR

Equality:

λ $\lambda pq.\ p(q\,T\,F)(q\,F\,T) =: Equ$

$$\lambda pq.\ p \begin{cases} \to q \begin{cases} \to T \\ \to F \end{cases} \\ \to q \begin{cases} \to F \\ \to T \end{cases} \end{cases}$$

When p is true => just select q (redundancy)

—''— false -> —''— Not q

⇒ Simplification:

$\lambda pq.\ p\,q\,(\text{Not}\,q)$

<u>De Morgan:</u> $\neg(P \wedge Q) = (\neg P) \vee (\neg Q)$

(JS) `!(p && q) = (!p) || (!q)`

(λ) Equ (Not (And (p q)) (Or (Not p) (Not q)))

$\lambda xy.\ x\,y\,((\lambda fab.\ f\,b\,a)\ y)$

$((\lambda fab.\ f\,b\,a)\ ((\lambda xy.\ x\,y\,x)\ p\,q)$

$((\lambda f.\ f\,f)\ ((\lambda fab.\ f\,b\,a)\ p)\ ((\lambda fab.\ f\,b\,a)\ q))$

Here we can see De Morgan's law directly in the lambda expression

<u>Combinator Basis:</u>

$S := \lambda abc.\ a\,c\,(b\,c)$

$K := \lambda ab.\ a$

Example: $I = SKK = SKS$

How can we do numbers?

Not: Nouns one, two, three

Instead: Adverbs Once, twice, thrice

$\boxed{1}$ λ Js

$N1 := \lambda fa.fa$ $n1 = f \Rightarrow a \Rightarrow f(a)$

The function is called once (Identity)

$\boxed{2}$ λ $N2 := \lambda fa.f(fa)$ Js $n2 = f \Rightarrow a \Rightarrow f(f(a))$

The function is applied twice

$\boxed{3}$ λ $N3 := \lambda fa.f(f(fa))$ Js $n3 = f \Rightarrow a \Rightarrow f(f(f(a)))$

Examples:

$N1 \ NOT \ T = NOT \ T = F$

$N2 \ NOT \ T = NOT(NOT \ T) = NOT \ F = T$

⋮

What is zero?

$\lambda fa.a$ -> Applies it no times

identical to False!

Can we dynamically generate numbers?

=> Sucessor function: If we give it a number it generates the next number

$$Succ \; N1 = N2$$
$$Succ \; N2 = N3 = Succ\,(Succ \; N1)$$
...

=> „Peano" Numbers

(λ) $Succ := \lambda n f a . f(n f a)$

(Js) $Succ = n \Rightarrow f \Rightarrow a \Rightarrow f(n(f)(a))$

take the number of function applications and do one more on top of it.

The results are only extentionally equal, not intentionally

Bluebird „B-Combinator

(λ) $B := \lambda f g a . f(g a)$ Function Composition

(Js) $B = f \Rightarrow g \Rightarrow a \Rightarrow f(g(a))$

$$B \; Not \; Not \; T = T$$

Here we combined Not Not to the identity I example: Haskell odd = not • even

Beautification:

$$\text{Succ} := \lambda n f a . f\,(\underline{n f}\,a)$$

$$= \lambda n f . B\, f\, (n f)$$

The successor can also be expressed with the Bluebird combinator

How can we add other numbers than +1 ?

$$\text{Add} := \lambda n k . n \text{ succ } k$$

$$\text{Add } n3\ n5 = \text{succ}(\text{succ}(\text{succ } n5))$$

$$= (\text{succ} \cdot \text{succ} \cdot \text{succ})\ n5$$

$$= n3 \text{ succ } n5$$

Multiplication

$$\text{Mult} := \lambda n k f . n(k(f)) = (B n k) f$$

$$\text{Mult } n2\ n3\ f\ a = (f \circ f \circ f \circ f \circ f \circ f)\ a$$

$$= ((f \circ f \circ f)(f \circ f \circ f))\ a$$

$$= ((n3 f) \circ (n3 f))\ a$$

$$= n2\ (n3\ f)\ a$$

We can cancle out the f on both sides

$Mul := \lambda nk.\,Bnk$ and also $n \circ k$

=> Multiplication is identical to the B combinator

Multiplication and the B combinator are α-equivalent -> They are the same when changing the variable names.

<u>Exponentiation:</u>

$Pow := \lambda nk.\,kn$ Thrush Combinator

$Pow\ n2\ n3 = n8 = n2 \times n2 \times n2$

$= n2 \circ n2 \circ n2$

$= n3\ n2$

Also: $Pow = C\ I$

The is zero

$$\text{IS0}\ N_0 = T$$

$$\text{IS0}\ N_1 = F$$

$$\text{IS0}\ N_2 = F$$

...

$$\text{IS0} := \lambda n.\, n\, \underbrace{(KF)}_{\text{return false}}\ \underbrace{T}_{\text{return true}}$$

The kestral is always applied when the Church numeral is unequal to 0 and therefore always returns F. When n=0 the function application is skipped and returns T.

Data-structures

Vireo

$\lambda abf.\, fab$

pair things together

↳ you can move this pair around and use it for other purposes

If you want to access the things in the box:
give it a function and this function
accesses both things.

-> This is an example of using closures as
data structures.

Example: VIM

$$VIM = \lambda f. f\, I\, M$$

The VIM (pair of identity & Mockingbird) is a
function that holds on to the identity and
Mockingbird and provides an Interface for
interacting with them => give me a function
and I give you these two things.

$$\underline{VIM}\ K = (\lambda f. f\, I\, M)\, K = I$$

$$\underline{VIM}\ \underline{KI} = M$$

Access condend of Vireo pairs

First := $\lambda p. p k$ =: FST

Second := $\lambda p. p (k I)$ =: SND

PHI ϕ

PHI := $\lambda p. V \, (SND\ p) \, (Succ\ (SND\ p))$ =: ϕ

$\phi \, (V\ M\ N7) = V\ N7, N8$

$\phi \, (V\ N9\ N2) = V\ N2, N3$

-> shift second thing to 1st and increment 2nd

$N8\ \phi \, (V\ N\emptyset\ N\emptyset) = N8 \, (V\ N\emptyset\ N1)$

$= V\ N7\ N8$

(8 times the ϕ applied to the $(V\ N\emptyset\ N\emptyset)$ pair

=> take the 1st argument:

$FSt \, (n8\ \phi \, (V\ N\emptyset\ N\emptyset)) = N7$

↳ subtraction of n8 by n1

This allows us to create a predecessor function:

$$\text{Pred} := \lambda n. \text{Fst}\ (n\ \Phi\ (V\ N0\ N0))$$

<u>Subtraction in general</u>

$$\text{SUB} := \lambda n k.\ k\ \text{Pred}\ n$$

<u>Less-than or equal:</u>

$$\text{LEQ} := \lambda n k.\ \text{IS0}\ (\text{Sub}\ n\ k)$$

<u>Equality:</u>

$$\text{EQ} := \lambda n k.\ \text{AND}\ (\text{LEQ}\ n\ k)\ (\text{LEQ}\ k\ n)$$

<u>Greater than:</u>

$$\text{GT} := \lambda n k.\ \text{NOT}\ (\text{LEQ}\ n\ k)$$

can we express this otherwise? It looks like the Bluebird (B) but for 2 arguments.

Blackbird combinator:

$B_1 := \lambda f g a b . f (g a b)$

-> $GT = B_1 \; NOT \; LEQ$

Funfact: $B_1 = B B B$

↳ The Blackbird is the composition of composition composed with composition and composition.

BCKI - Basis

These 4 combinators form a Basis:

$KI = KI = CK$

$B_1 = BBB$

$T_h = CI$

$V = B C T_h = B C (C I)$