

(fp c<sub>2</sub> 1.0)

(iter 1.0 (c<sub>2</sub> 1.0))

(iter 1.0 (avg 1.0 (c<sub>1</sub> 1.0)))

(iter 1.0 (avg 1.0 2.0))

(iter 1.0 1.5)

### Local Names

"let" bindings are created simultaneously & "let" body is evaluated with respect to those binding.

### Lambda Calculus

- ① Core of all functional languages.
- ② Useful for specifying semantics.
- has only 3 syntactic rules in lambda calculus.

$M \rightarrow x$  (Variable)  
 $\lambda x. M$  (Abstraction)  
 $M_1. (M_2)$  (Application)

optional.  $\lambda$  the 1<sup>st</sup> one is  $(M)$   $\rightarrow$  parenthesis to represent this has to be the 5<sup>th</sup> one is constant control flow.

$\rightarrow$  identity function

$\lambda x. x$  in scheme

(lambda (x) x)

to apply this function on an argument

$(\lambda x. x+1) 2$

fall into 3<sup>rd</sup> category  $\lambda M_1. M_2$   
How is this evaluated? the ~~parameter~~ <sup>argument</sup>

is bound to argument.

How to take multiple arguments?

(define add (lambda (x)

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

(lambda (y) (+ x y)))

$$(\lambda x. \lambda y. x+y)^2 \quad 3$$

$$(\lambda y. 2+y)^3$$

(2+3)  
5

Precedence & Associativity rules

① Application binds tighter than abstraction.

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

$\neq (\lambda x. x) y \rightarrow$  If we want this to be done just use '()'.

② Application associates to left

$$x y z = (x y) z$$

$$\neq x (y z)$$

$\beta$ -reduction

$$(\lambda x. M) N = [N/x] M$$

$\hookrightarrow$  replace occurrences of  $x$  in  $M$

with ' $x$ '  $\rightarrow$  General substitution model type.

$$(\lambda x. x+1) 2 = [2/x] x+1$$

$$1) \underbrace{(\lambda x. x)}_{M_1} \underbrace{((\lambda x. x) (\lambda z. (\lambda x. x) z))}_{M_2} \quad M_1 \text{ applied on } M_2$$

$$\Rightarrow_{\beta} (\lambda x. x) (\lambda z. (\lambda x. x) z)$$

$$=_{\beta} \lambda z. (\lambda x. x) z$$

$=_{\beta} \lambda z z \rightarrow$  Normal form  $\rightarrow$  Cannot reduce any further



$\lambda xyz \equiv \lambda x. \lambda y. \lambda z. (x (y z))$  (takes 3 arguments)

①  $(\lambda xyz. x z (y z)) (\lambda x. x) (\lambda x. x)$

$\equiv_{\beta} \lambda yz. (\lambda x. x) z (y z) (\lambda x. x)$  *rule ②*

Met-1  
 $\equiv_{\beta} \lambda yz. z (y z) (\lambda x. x)$  *leftmost + innermost redex*

$\equiv_{\beta} \lambda z. z ((\lambda x. x) z)$

$\equiv_{\beta} \lambda z. z z$

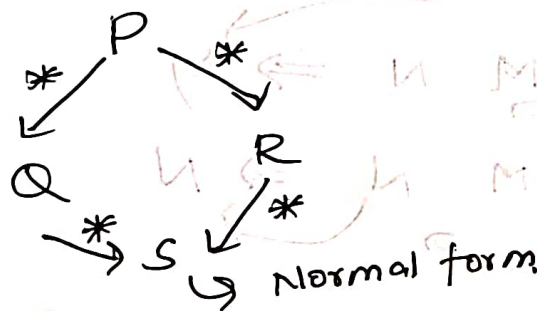
Met-2  
 $\equiv_{\beta} \lambda z. (\lambda x. x) z ((\lambda x. x) z)$

$\equiv_{\beta} \lambda z. z ((\lambda x. x) z)$

*leftmost + outermost redex*  
*call by name*

$\equiv_{\beta} \lambda z. z z$

### Church - Rosser theorem



Normal forms are unique, if they exist.  
 (For some terms normal forms don't exist.)

ex:  $(\lambda x. x x) (\lambda x. x x) \rightarrow$   
 $\equiv_{\beta} (\lambda x. x x) (\lambda x. x x)$

### Lambda Calculus (contd.)

→ Programming based on axioms → program such that it satisfies all the axioms required

ex: programmed numbers, zero satisfying add, mul, sub rules & also axioms of zero.

$(\lambda y. x+y) x =_{\beta} x+x \rightarrow$  acc<sup>n</sup> to brute  $\beta$  red<sup>n</sup>  
 but what is overlooked here is  
 free bound. the  $x$  inside the fn<sup>n</sup> body is  
 a free variable & the  $x$  substitute  
 in place of  $y$  is a bound variable.  
 this makes the initial free variable  
 bound.

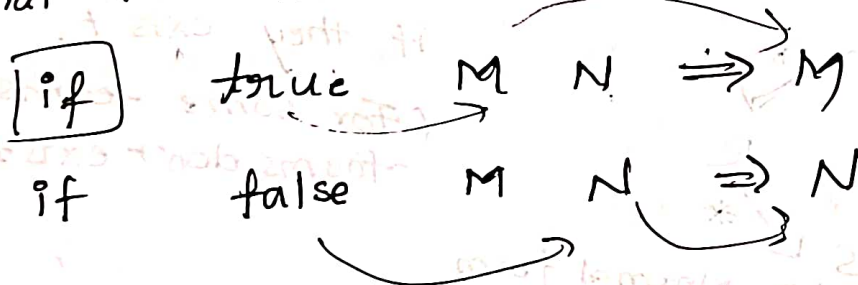
To avoid this we need to perform  $\alpha$ -conversion  
 $\alpha$ -conversion: Replace 'free variables before  
 applying the  $\beta$  procedure to something  
 which cannot be passed as an argument.

$$(\lambda y. x+y) x \equiv_{\alpha} (\lambda y. z+y) x =_{\beta} z+x$$

Let's define

Booleans:

↳ what are the tasks a boolean need to perform.



$$\text{true} = \lambda x. \lambda y. x = \lambda x. \lambda y. x$$

$$\text{false} = \lambda x. \lambda y. y = \lambda x. \lambda y. y$$

~~if we just change the order of passing~~  
 we can return the correct var. term  
 $(M/N)$  by applying the corresponding  
 function.

ex: if true  $((\lambda x y. x) M N)$  if false  $((\lambda x y. y) M N)$

$$(\lambda x y. x) M N$$

$$(\lambda y. M) N$$

$$M$$

$$(\lambda x y. y) M N$$

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

$$N$$

OR

OR  $x$   $y$

$\lambda x y$  if  $x$  true  $y$

$(\lambda x y. \text{if } x \text{ true } y) \text{ true}$

$=_{\beta} (\lambda y. \text{if true true } y)$

$=_{\beta} (\lambda y. \text{true}) y$

$= \text{true}$

$(\lambda x y. \text{if } x \text{ true } y) \text{ false}$

$=_{\beta} (\lambda y. \text{if false true } y)$

$=_{\beta} (\lambda y. y)$

Recursion in  $\lambda$  calculus

First of all to recursively call the same function in traditional programming we just call it by name of function. we never defined names of functions in  $\lambda$  calculus.  $\rightarrow$  we

1)  $Y$ -combinator  $\leftarrow$  closed expression (without free variable)

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

$$Y F =_{\beta} (\lambda x. F(x x)) (\lambda x. F(x x))$$

How?  $\hookrightarrow =_{\beta} F((\lambda x. F(x x)) (\lambda x. F(x x)))$



$$= F(Y F) \dots F(F(Y F)) \dots$$

Q.  $F = \lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } x * f(x)$

$$Y F 3 = \beta ((\lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } x * f(x - 1))) 3$$

$$= F(Y F) 3$$

$$= \beta 3 * (Y F 2)$$

$$= 3 * (F(Y F) 2)$$

$$= \beta 3 * 2 * (Y F 1) \dots$$

some thing came up as SKI calculus.  $\rightarrow$  programming language based

$$I = \lambda x. x$$

$$K = \lambda x. \lambda y. x$$

$$S = \lambda x. \lambda y. \lambda z. x z (y z)$$

$$SKSK = K K (SK)$$

$$= K.$$

there is even 'i-combinator'  $\rightarrow$  uses only i

on  $\lambda$  calculus has only 3 variables 's', 'k', 'i'