Lambda Calculus

Oded Padon & Mooly Sagiv

(original slides by Kathleen Fisher, John Mitchell, Shachar Itzhaky, S. Tanimoto , Stephen A. Edwards)

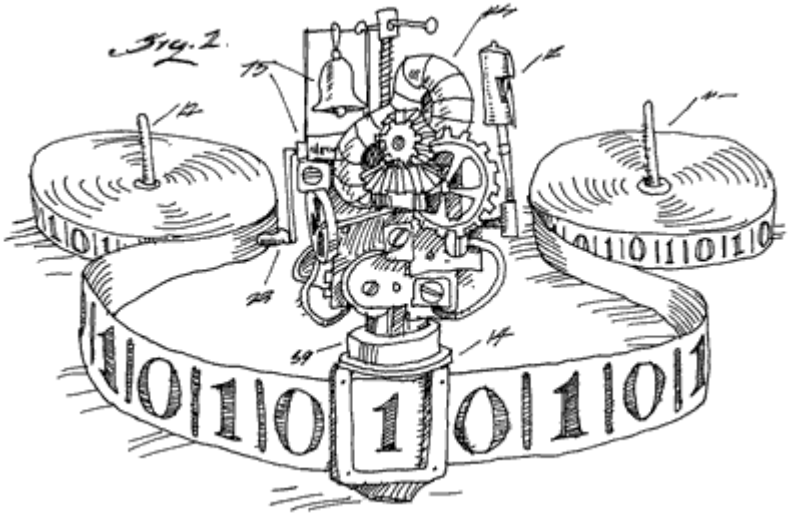
Benjamin Pierce

Types and Programming Languages

http://www.cs.cornell.edu/courses/cs3110/2008fa/recitations/rec26.html

Computation Models

- Turing Machines
- Wang Machines
- Counter Programs
- Lambda Calculus



Alonzo Church

1903–1995
Professor at Princeton (1929–1967) and UCLA (1967–1990)
Invented the Lambda Calculus

Notable students:

- Alan Turing
- Stephen Cole Kleene
- Martin Davis
- Barkley Roser
- Dana Scott
- Leon Hankin
- Michael Rabin
- John Kemeny

Simple Example factorial

C

```
int fact(int x) {
    if (x==0) return  1; else return n * fact(n-1);
}
```

No state mutations

ML

```
fact x : int : int =
    if x=0 then 1 else n * fact n-1
```

No state mutations

Expression only

Another Example Compose

```
comp f: int → int g: int → int: int →int =
    fun x → g (f x)
```

```
comp fun x → x + 1 fun x → x + 1
```

```
comp fun x → x + 1 fun x → x + 1 2
```

```
comp fun x → 2 *x fun y → 4 * y
```

```
comp fun x → 2 *x fun y → 4 * y 3
```

What is λ calculus

- A complete computational model
- An assembly language for functional programming
  - Powerful
  - Concise
  - Counterintuitive
- Can explain many interesting PL features

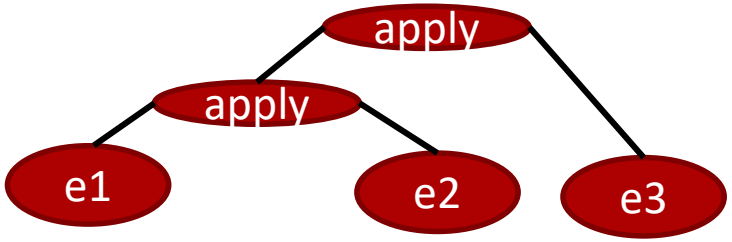
Basics

- Repetitive expressions can be compactly represented using functional abstraction
- Example:
  - (5 * 4* 3 * 2 * 1) + (7 * 6 * 5 * 4 * 3 * 2 *1) =
  - factorial(5) + factorial(7)
  - factorial(n) = if n = 0 then 1 else n * factorial(n-1)
  - factorial= λn. if n = 0 then 1 else n * factorial(n-1)
  - factorial= λn. if n = 0 then 1 else n * apply (factorial (n-1))

Untyped Lambda Calculus

t ::=	terms
x	variable
λ x. t	abstraction
t t	application

Terms can be represented as abstract syntax trees



Syntactic Conventions

- Applications associates to left

  $e_1\ e_2\ e_3 \equiv (e_1\ e_2)\ e_3$

- The body of abstraction extends as far as possible
  - $\lambda x.\ \lambda y.\ x\ y\ x \equiv \lambda x.\ (\lambda y.\ (x\ y)\ x)$

Untyped Lambda Calculus

t ::=	terms
x	variable
λ x. t	abstraction
t t	application

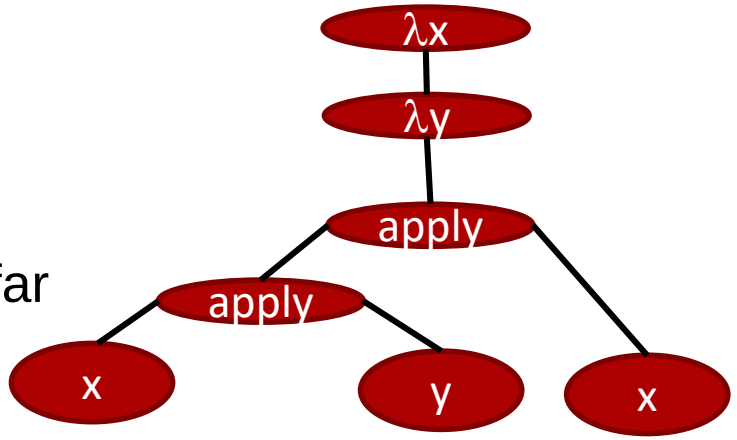
Terms can be represented as abstract syntax trees

Syntactic Conventions

- Applications associates to left

  $e_1\ e_2\ e_3 \equiv (e_1\ e_2)\ e_3$

- The body of abstraction extends as far as possible
  - $\lambda x.\ \lambda y.\ x\ y\ x \equiv \lambda x.\ (\lambda y.\ (x\ y)\ x)$



Untyped Lambda Calculus++

t ::=	terms
x	variable
λ x. t	abstraction
t t	application
number	number
op t_1 t_2 … t_k	built-in operator op t_1, t_2, …, t_k
true	true
false	false

Lambda Calculus in Python

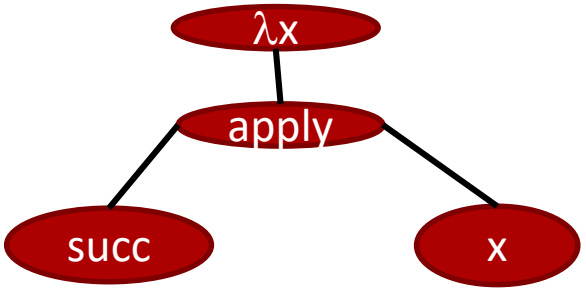
$(\lambda\ x.\ x)\ y$ (lambda x: x) (y)

Lambda Calculus in ML

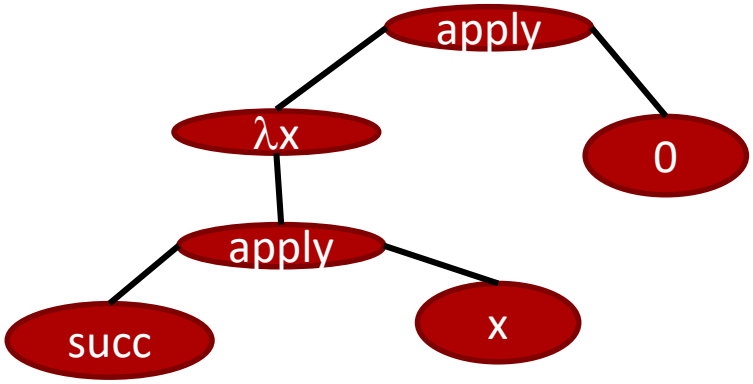
$(\lambda\ x.\ x)\ y$ (fun x → x) (y)

Functions on int

λx. succ x



(λx. succ x) 0



⇒ 1

Define a function which computes 2?

Lambda Expressions

- Function application are written in prefix form
  - "Add 4 to 5" (+ 4 5)
- Evaluation: Select a "redex" and apply function

(+(* 5 6) (* 8 3)) ⇒ (+ 30 (* 8 3))

⇒ (+(30 24)

⇒ 54

- Different evaluation orders

(+(* 5 6) (* 8 3)) ⇒ (+(* 5 6) 24)

⇒ (+ 30 24)

⇒ 54

Simon Peyton Jones, The Implementation of Functional Programming Languages, Prentice-Hall, 1987.

Function Application and Currying

- Function application written as juxtaposition

  f x

- Every function has exactly one argument
  - Multiple arguments can be simulated by Curring
  - (+ x) is a function which adds x to its argument

    (+ 4 5) = (+4) 5 ⇒ 9
  - What is +4?
  - What is +

Lambda Abstraction

- The only other thing in the lambda calculus is lambda abstraction
  - a notation for defining unnamed functions

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

The function of x that adds 1 to x

Boolean Expressions

- (if true x y) $\Rightarrow$ x
- (if false x y) $\Rightarrow$ y
- Boolean negation?
- Boolean conjunction
- Boolean disjunction

Beta Reduction

- Substitute formal by actual arguments

$(\lambda x.\ +\ x\ \ 1)\ 4 \Rightarrow_\beta (\ +\ 4\ \ 1) \Rightarrow_\beta 5$

- x can occur multiple times

$(\lambda x.\ +\ x\ \ x)\ 4 \Rightarrow_\beta (\ +\ 4\ \ 4) \Rightarrow_\beta 8$

- Or not at all

$(\lambda x.\ +\ 1\ \ 6)\ 4 \Rightarrow_\beta (\ +\ 1\ \ 6) \Rightarrow_\beta 7$

Beta-Reduction(Cont)

- Fuzzy but mechanical
- Extra parenthesis and tree-notations can help

$(\lambda x.\lambda y.\ +\ x\ \ y)\ \ 3\ 4$ = $((\lambda x.(\lambda y.\ ((+\ x)\ y)))\ \ 3)\ 4$

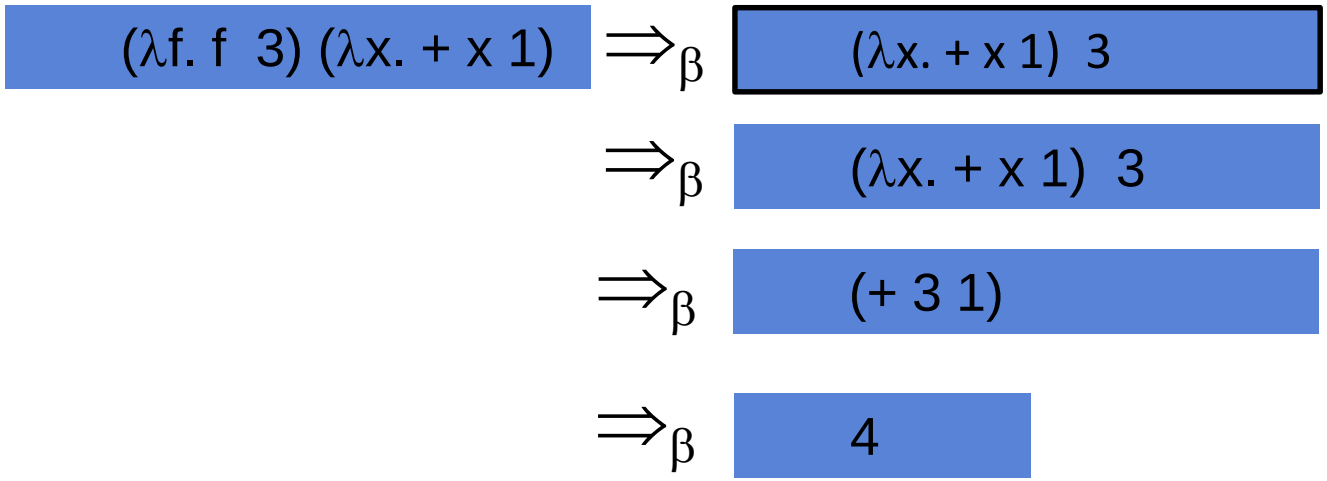
$\Rightarrow_\beta$ $(\lambda y.\ ((+\ 3)\ y))\ 4$

$\Rightarrow_\beta$ $((+\ 3)\ 4)$

$\Rightarrow_\beta$ 7

Functions can be arguments

- Arbitrary functions as arguments/return



Free and Bound Variables

λx. (+ x y) 2

- x is bound
- y is free
  - Determined in the enclosed context
  - Like global variables
- Both x and y are free in (+ x y)

Ill-Formed Beta Reductions

- Sometimes beta-reductions cannot be applied
- Like runtime error/undefined semantics in imperative programs
  - (1 5)
  - (x 5 λy. y)
  - (+ true 2)
  - (test 5 1 2)
- Later will refine lambda calculus to avoid such terms by the compiler

Beta Reduction Formally

Substitution

- Replace a term by a term
  - $x + ((x + 2) * y)[x \mapsto 3, y \mapsto 7] = ?$
  - $x + ((x + 2) * y)[x \mapsto z + 2] = ?$
  - $x + ((x + 2) * y)[t \mapsto z + 2] = ?$
    $x + ((x\ \ + 2) * y)[x \mapsto y]$
  - More tricky in programming languages
  - Why?

Free vs. Bound Variables

- An occurrence of x is bound in t if it occurs in $\lambda x.\ t$
  - otherwise it is free
  - λx is a binder
- Examples
  - Id= $\lambda x.\ x$
  - $\lambda y.\ x\ (y\ z)$
  - $\lambda z.\ \lambda x.\ \lambda y.\ x\ (y\ z)$
  - $(\lambda x.\ x)\ x$

FV: $t \rightarrow 2^{Var}$ is the set free variables of t

$FV(x) = \{x\}$

$FV(\lambda\ x.\ t) = FV(t) - \{x\}$

$FV\ (t_1\ t_2) = FV(t_1) \cup FV(t_2)$

Beta-Reduction

$(\lambda\, x.\, t_1)\; t_2 \Rightarrow_\beta [x \mapsto t_2]\; t_1$ (β-reduction)

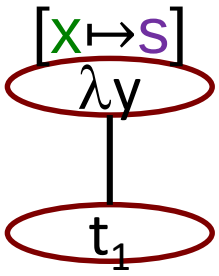
$[x \mapsto s]\; x = s$

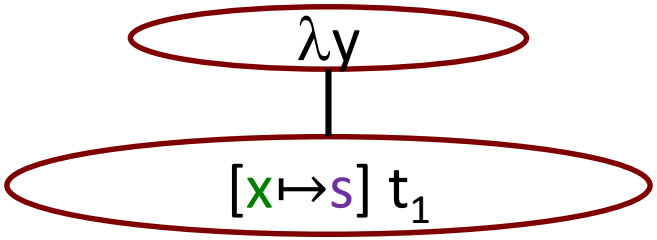
$[x \mapsto s]\; y = y$ if $y \neq x$

$[x \mapsto s]\; (\lambda x.\, t_1) = \lambda x.\, t_1$

$[x \mapsto s]\; (\lambda y.\, t_1) = \lambda y.\, [x \mapsto s]\; t_1$ if $y \neq x$ and $y \notin FV(s)$

$[x \mapsto s]\; (t_1\; t_2) = ([x \mapsto s]\; t_1)\; ([x \mapsto s]\; t_2)$





Beta-Reduction

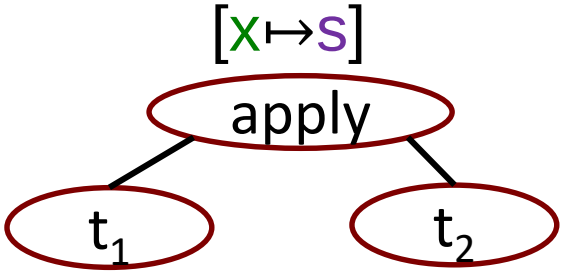
$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

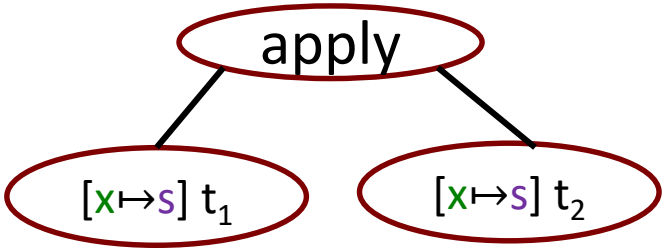
$[x \mapsto s]\ x = s$

$[x \mapsto s]\ y = y$ if $y \neq x$

$[x \mapsto s]\ (\lambda y.\ t_1) = \lambda y.\ [x \mapsto s]\ t_1$ if $y \neq x$ and $y \notin FV(s)$

$[x \mapsto s]\ (t_1\ t_2) = ([x \mapsto s]\ t_1)\ ([x \mapsto s]\ t_2)$



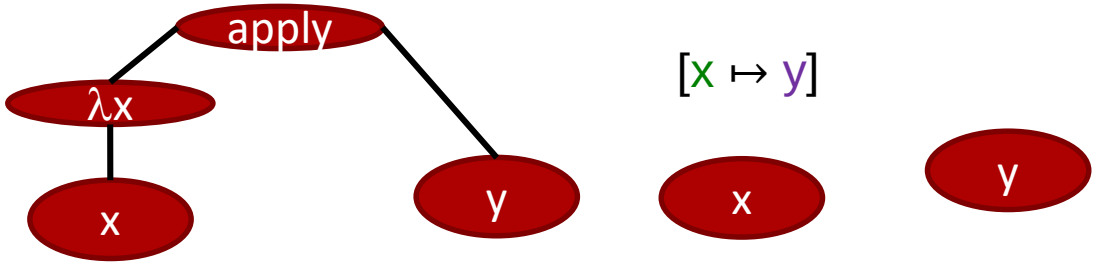


Example Beta-Reduction

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

redex

$(\lambda\ x.\ x)\ y \Rightarrow_\beta\ y$

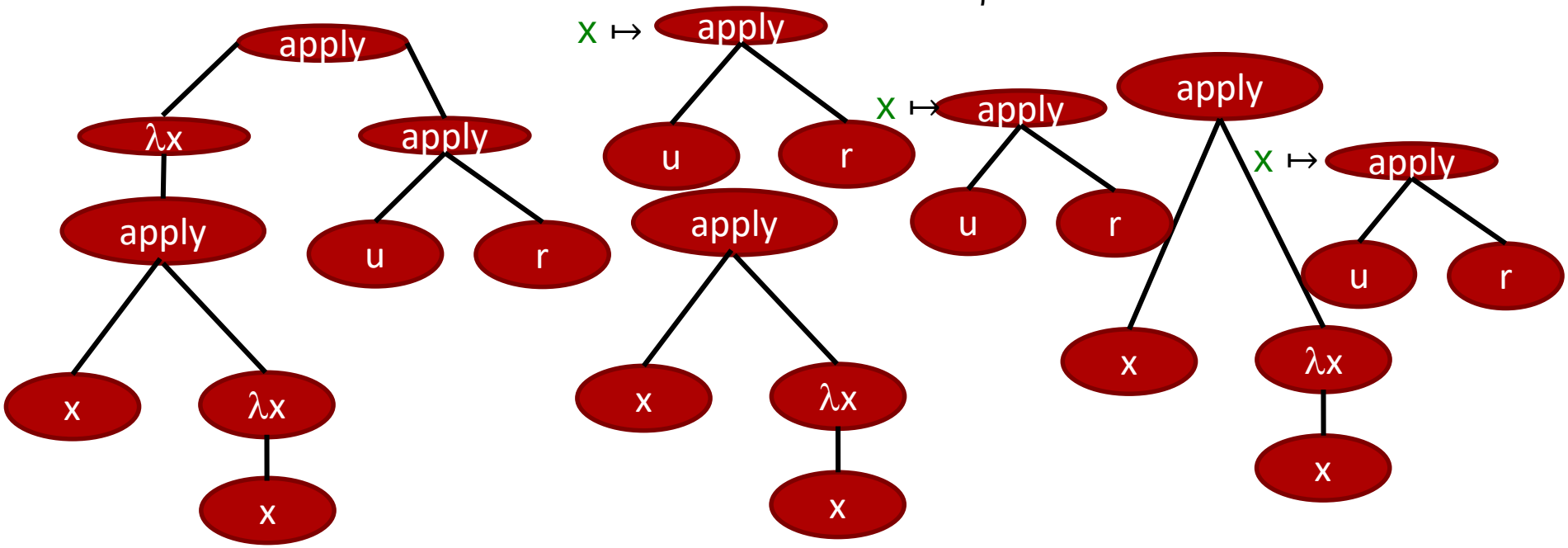


Example Beta-Reduction (ex 2)

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

redex

$(\lambda\ x.\ x\ (\lambda\ x.\ x)\ )\ (u\ r) \Rightarrow_\beta$

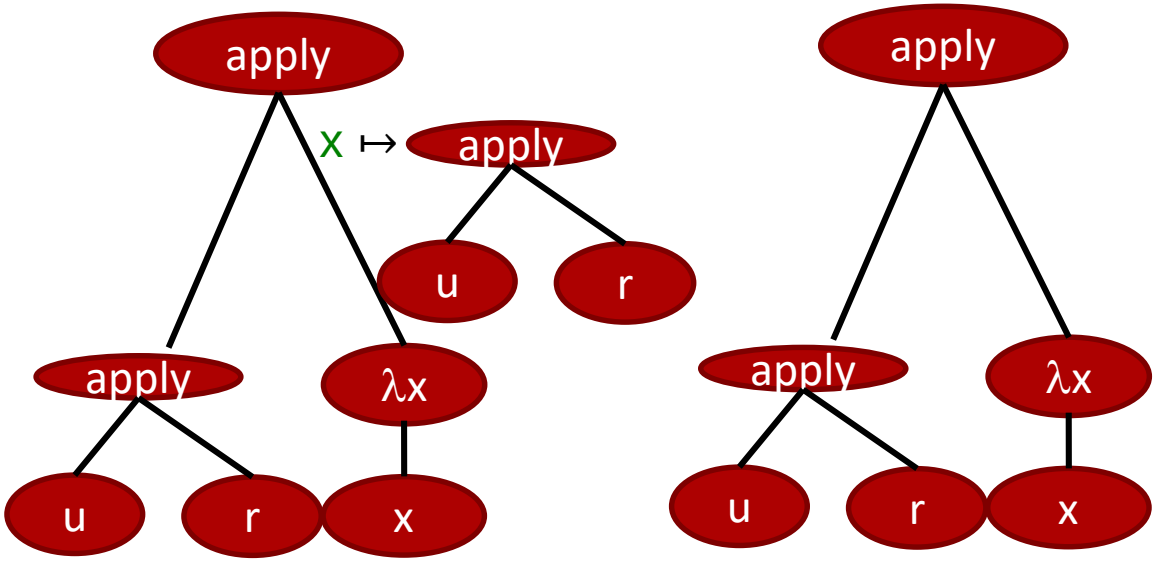


Example Beta-Reduction (ex 2) (cont)

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

redex

$(\lambda\ x.\ x\ (\lambda\ x.\ x)\ )\ (u\ r) \Rightarrow_\beta u\ r\ (\lambda\ x.\ x)$



Alpha- Renaming

Alpha conversion:
Renaming of a bound variable and its bound occurrences

$\lambda x.\lambda y.y \quad \Rightarrow_\alpha \quad \lambda x.\lambda z.z$

Simplifies terms

Example Beta-Reduction (ex 2) with renaming

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

redex

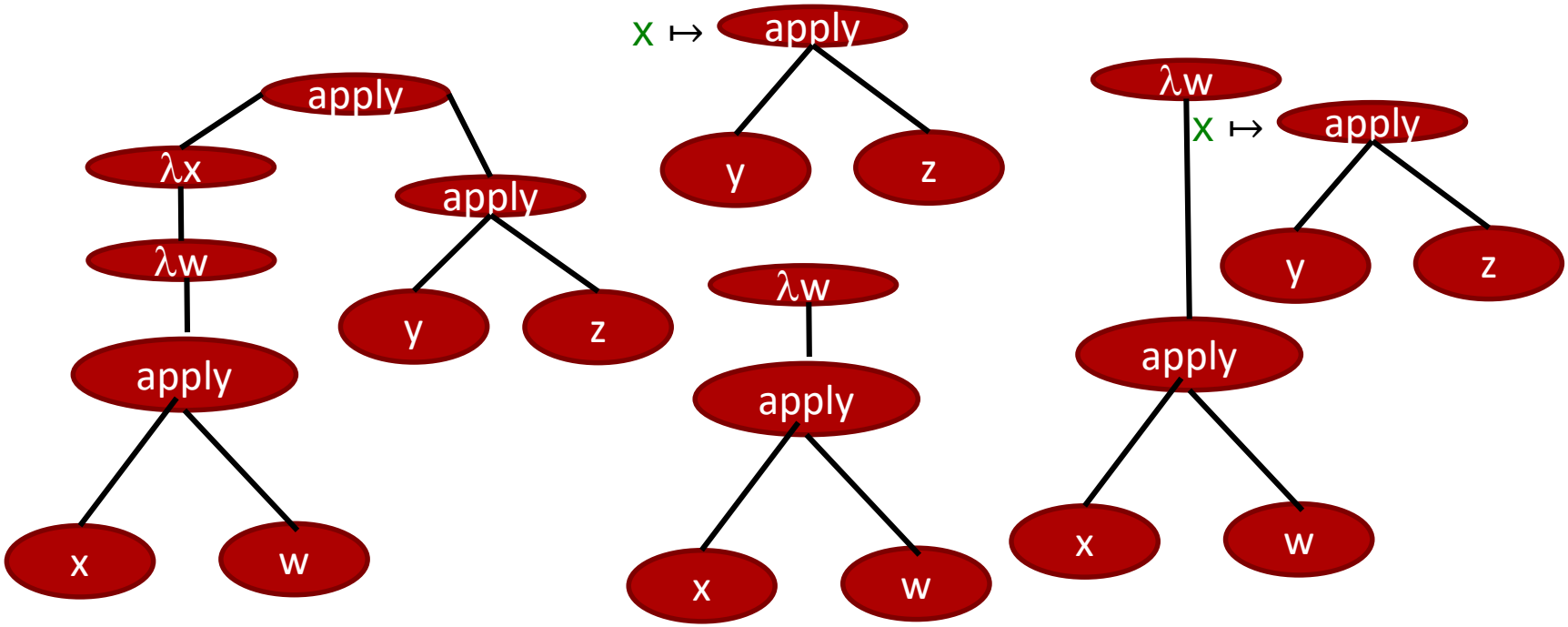
$(\lambda\ x.\ x\ (\lambda\ x.\ x)\ )\ (u\ r) \Rightarrow_\beta$

Example Beta-Reduction (ex 3)

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

redex

$(\lambda\ x.\ (\lambda w.\ x\ w))\ (y\ z) \Rightarrow_\beta \lambda w.\ y\ z\ w$



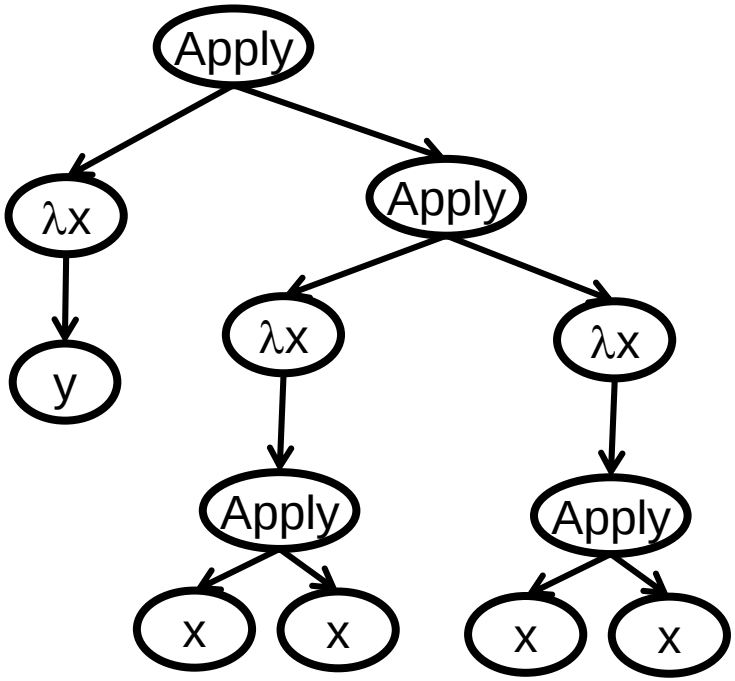
Simple Exercise

- Adding scopes to λ expressions
- Proposed syntax "let x = t_1 in t_2"
- Informal meaning:
  - all the occurrences of x in t_2 are replaced by t_1
- Example: let a = λ x. (λw. x w) in a a =
- How can we simulate let?

Divergence

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

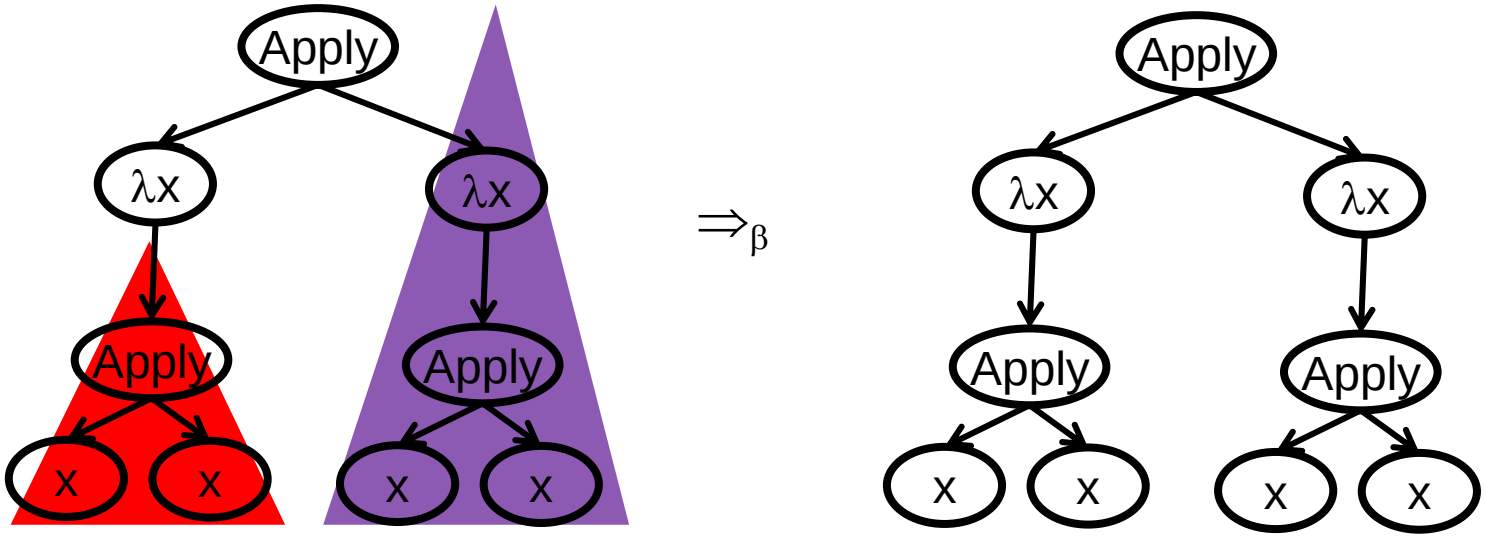
$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$



Divergence

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

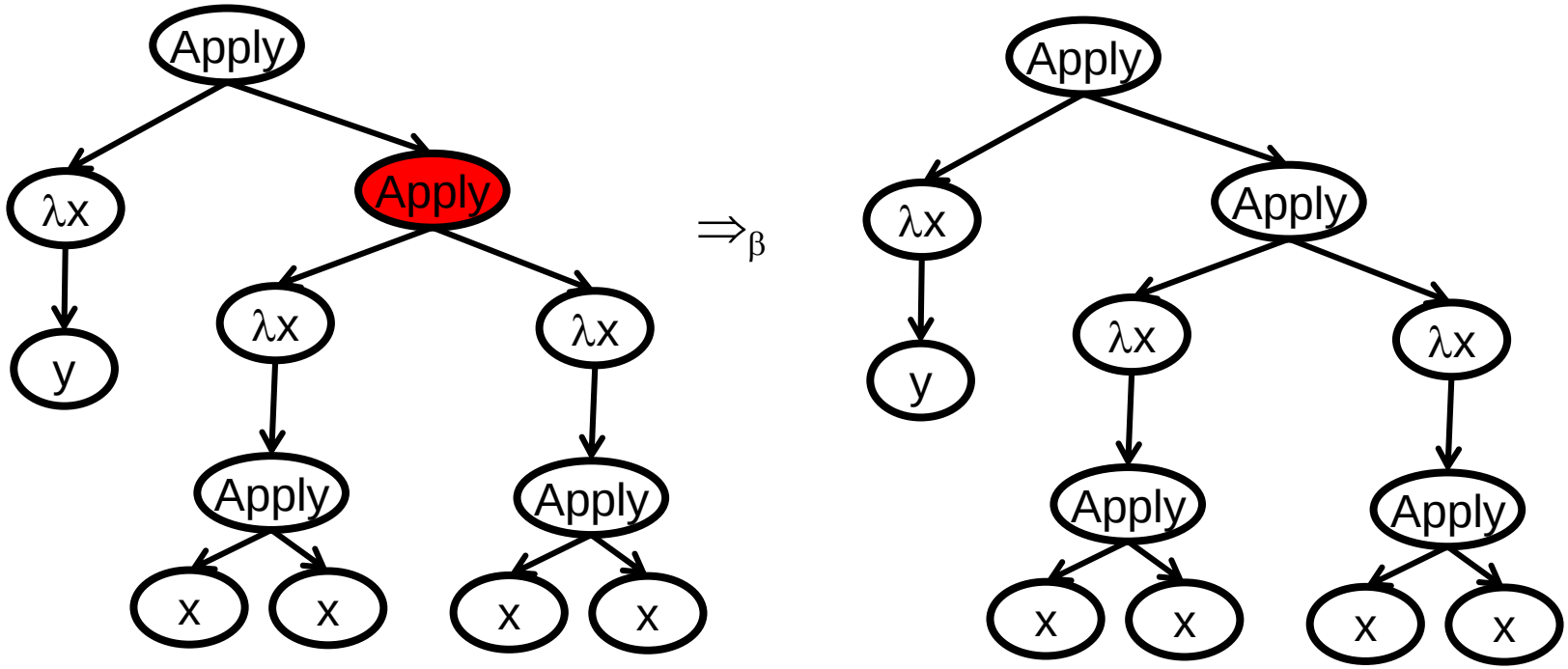
$(\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x))$



Different Evaluation Orders

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

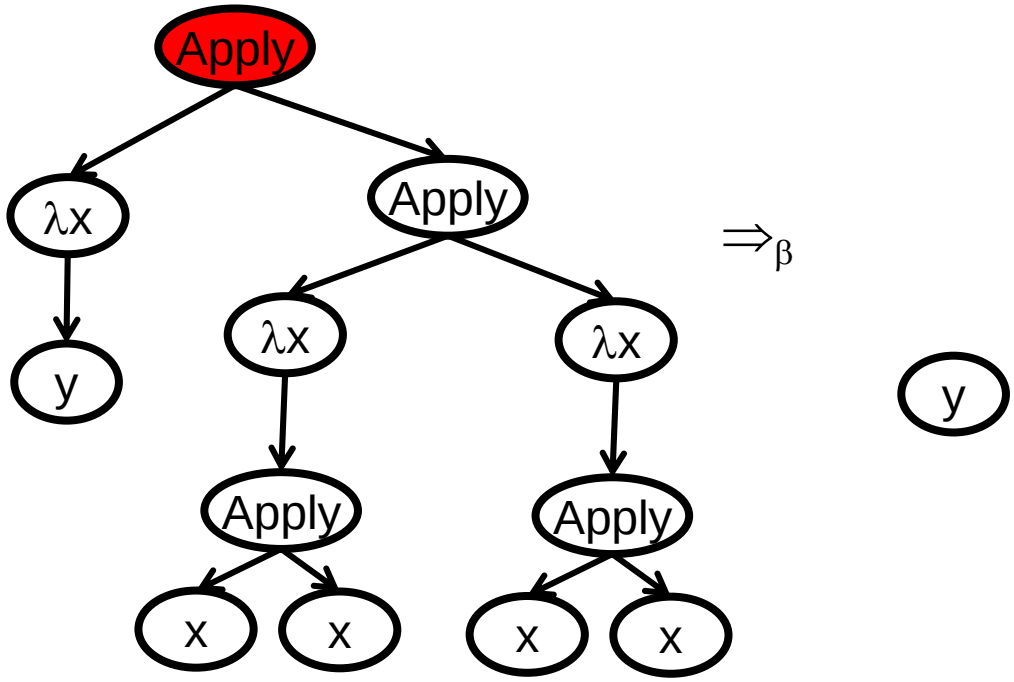
$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$



Different Evaluation Orders

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

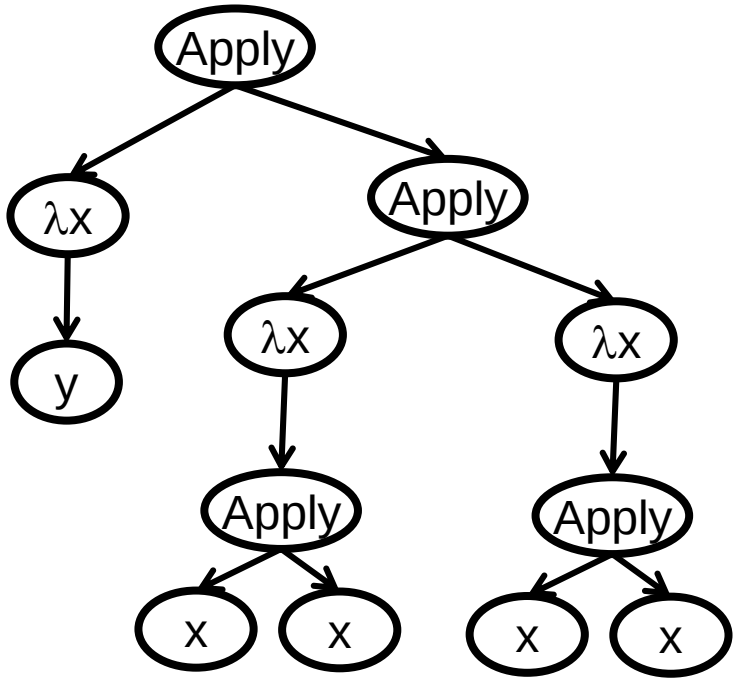
$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$



Different Evaluation Orders

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

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

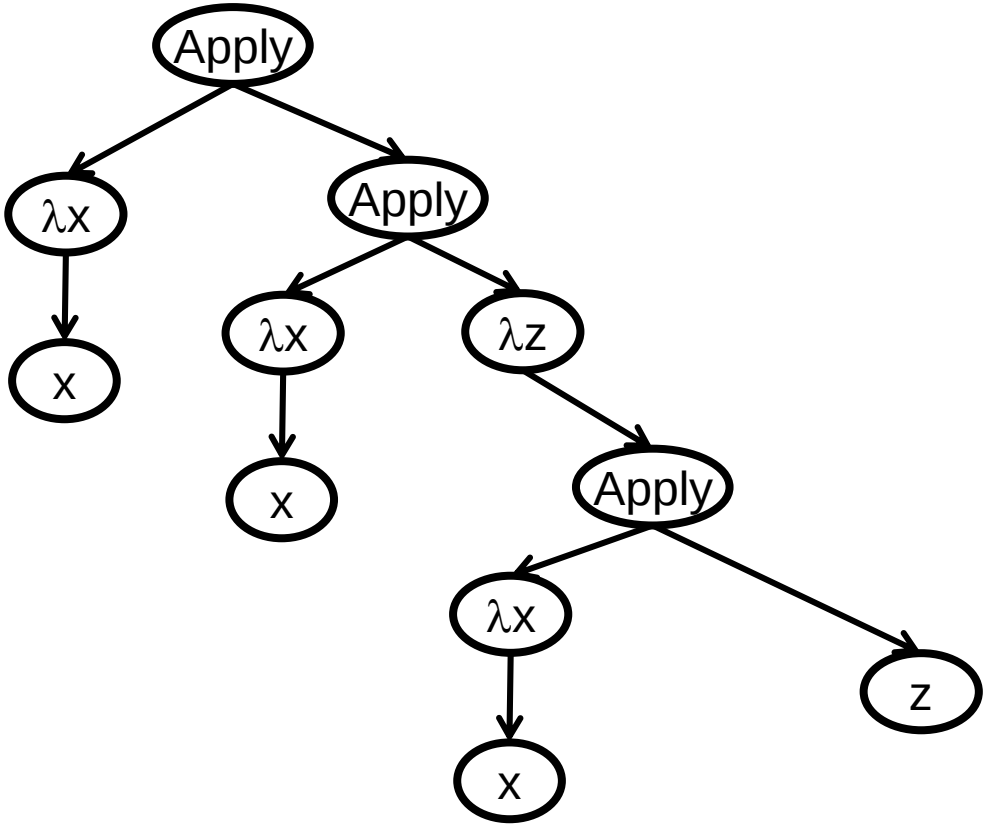



```
def f():
    while True: pass

def g(x):
    return 2

print g(f())
```

Different Evaluation Orders

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$ (β-reduction)

$(\lambda\ x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z)) \equiv id\ (id\ (\lambda z.\ id\ z))$

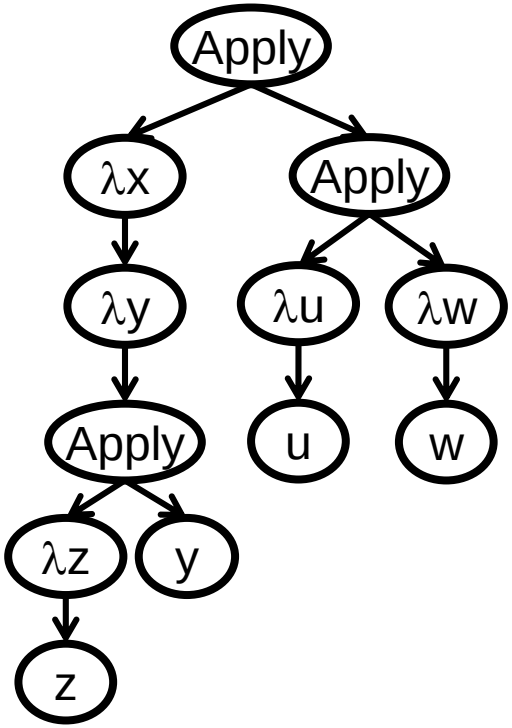


Order of Evaluation

- Full-beta-reduction
  - All possible orders
- Applicative order call by value (Eager)
  - Left to right
  - Fully evaluate arguments before function
- Normal order
  - The leftmost, outermost redex is always reduced first
- Call by name
  - Evaluate arguments as needed
- Call by need
  - Evaluate arguments as needed and store for subsequent usages
  - Implemented in Haskel

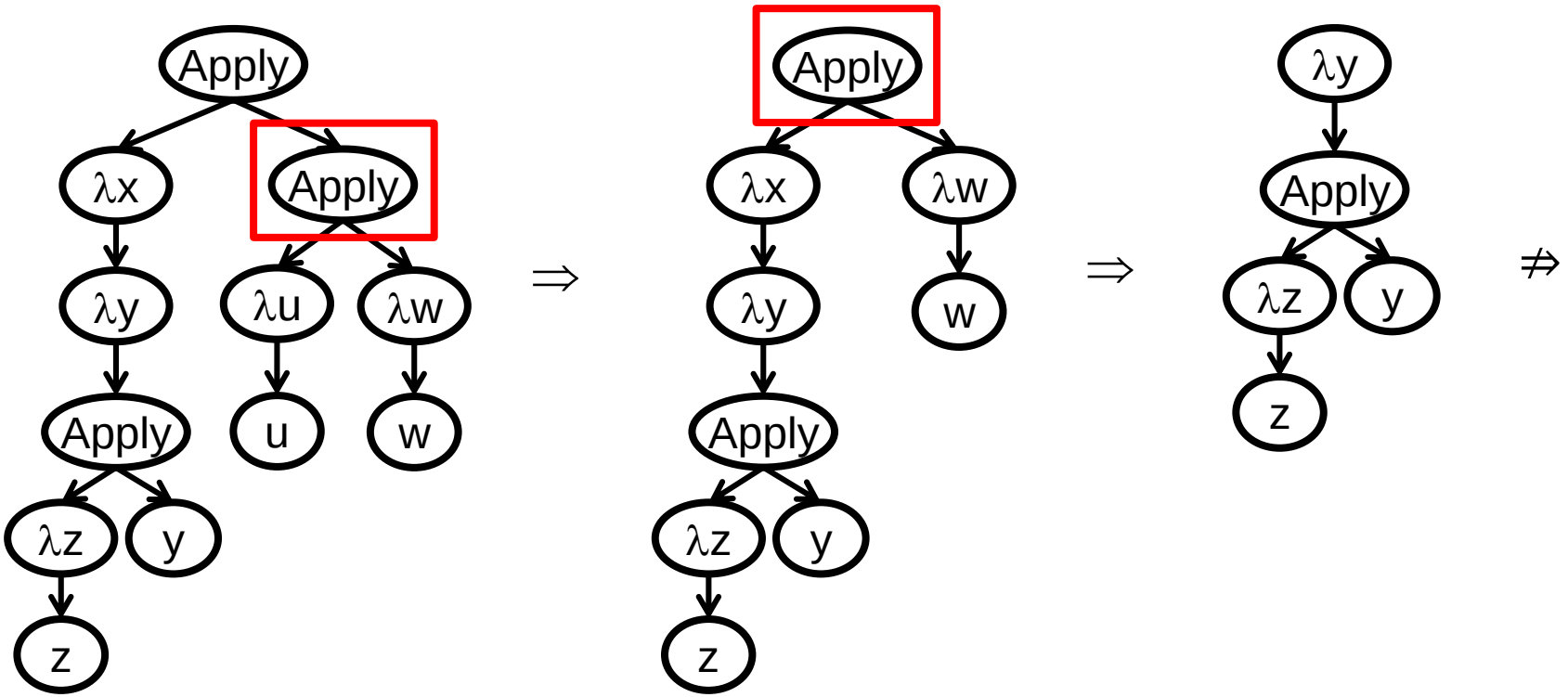
Different Evaluation Orders

(λ x. λ y. (λ z. z) y) ((λ u. u) (λ w. w))



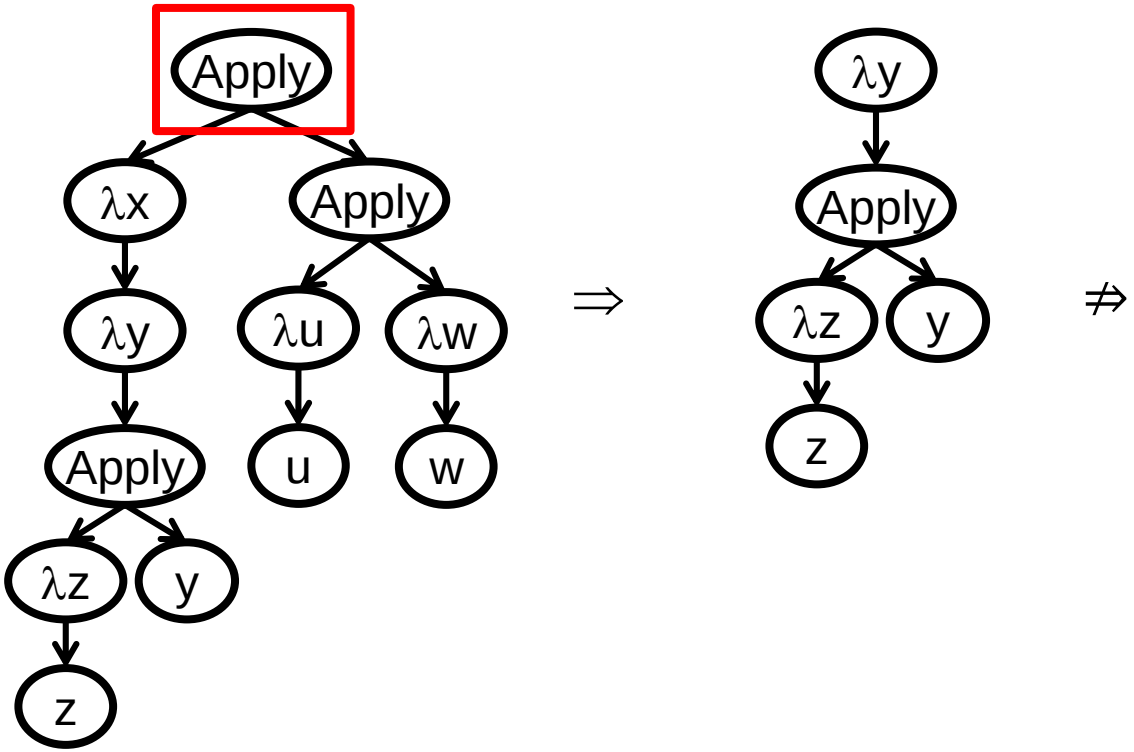
Call By Value

(λ x. λ y. (λ z. z) y) ((λ u. u) (λ w. w))



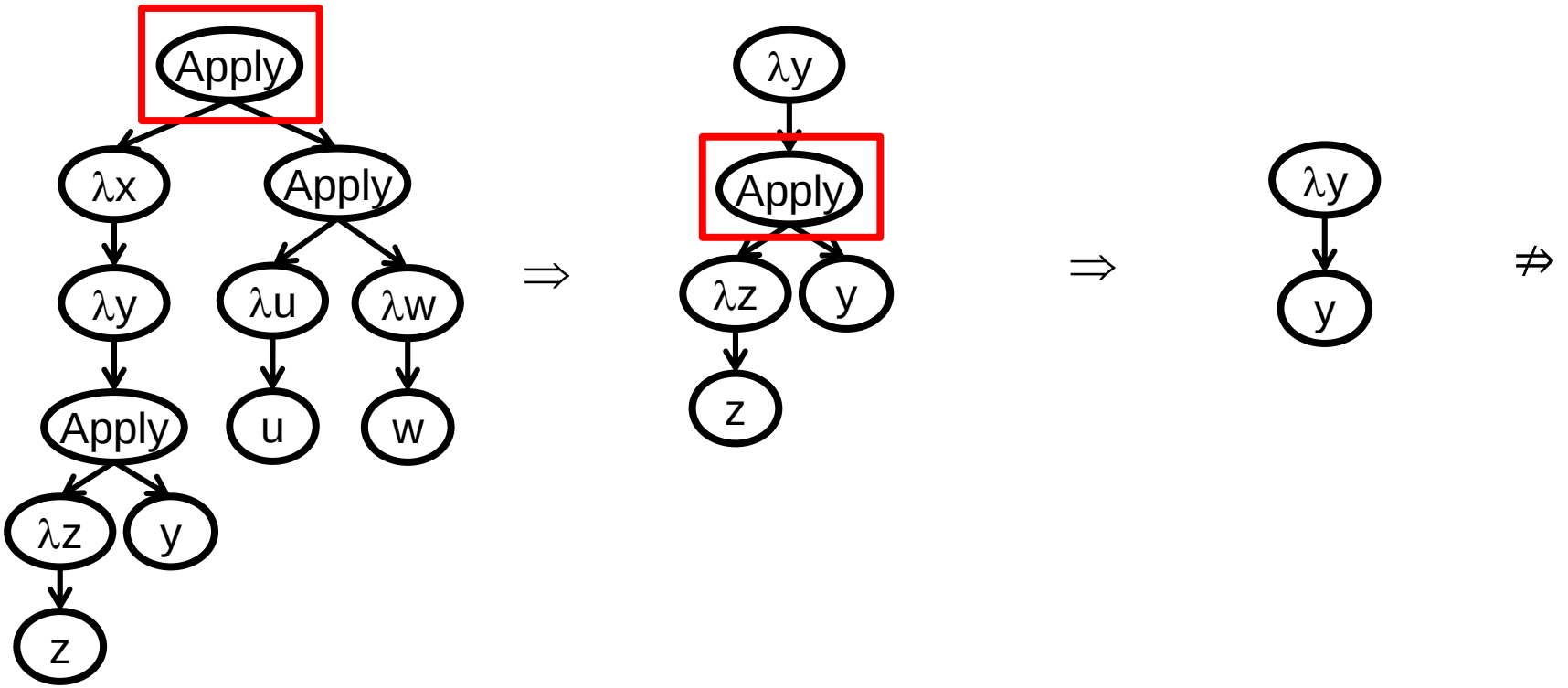
Call By Name (Lazy)

(λ x. λ y. (λ z. z) y) ((λ u. u) (λ w. w))



Normal Order

(λ x. λ y. (λ z. z) y) ((λ u. u) (λ w. w))



Call-by-value Small-step Operational Semantics

t ::=	terms
x	variable
λ x. t	abstraction
t t	application

v ::=	values
λ x. t	abstraction values
other values	

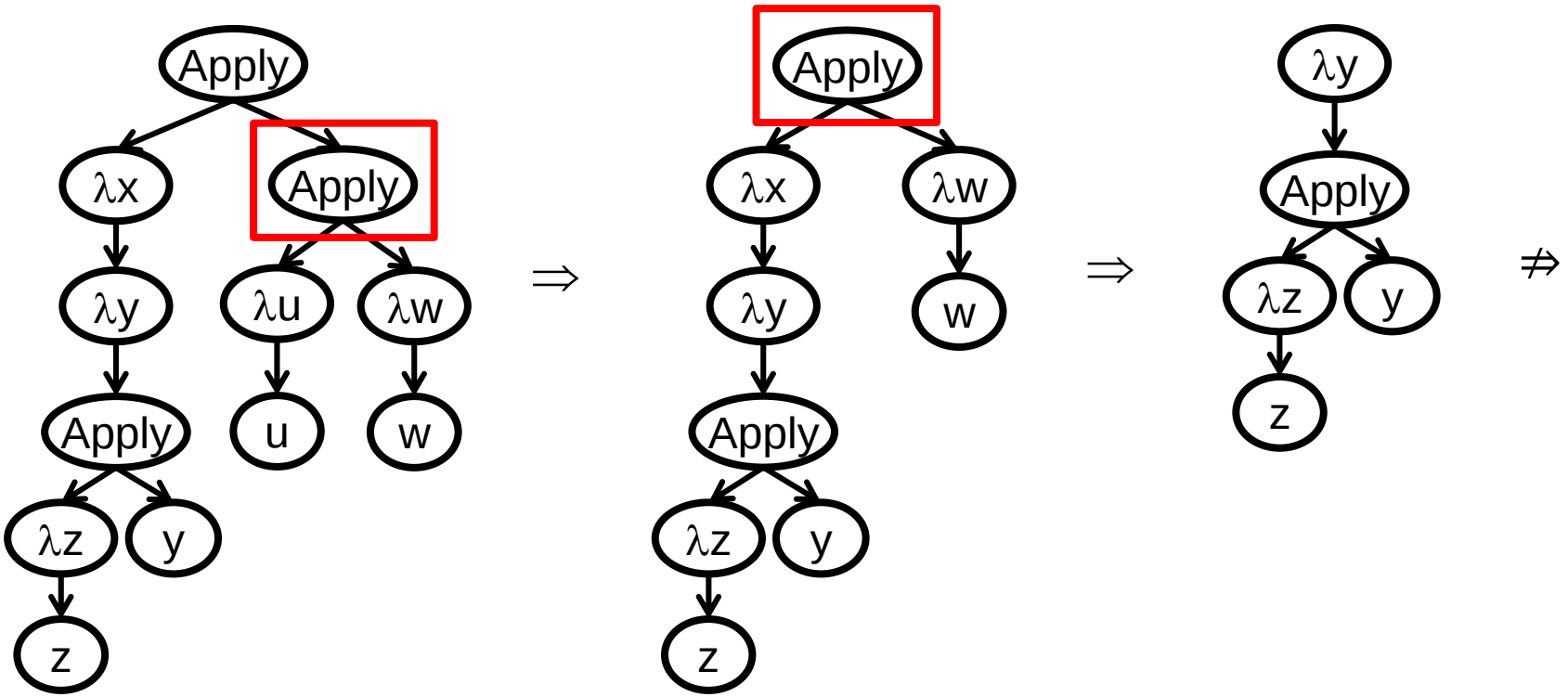
$$(\lambda\, x.\, t_1)\; v_2 \Rightarrow [x \mapsto v_2]\; t_1 \qquad \text{(E-AppAbs)}$$

$$\frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2} \qquad \text{(E-APPL1)}$$

$$\frac{t_2 \Rightarrow t'_2}{v_1\ t_2 \Rightarrow v_1\ t'_2} \qquad \text{(E-APPL2)}$$

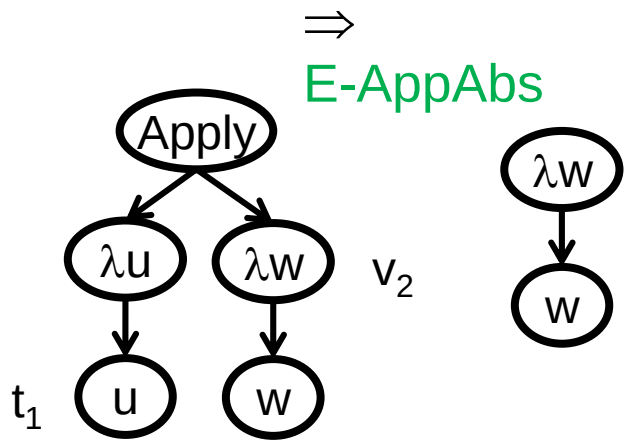
Call By Value

(λ x. λ y. (λ z. z) y) ((λ u. u) (λ w. w))



Call By Value OS

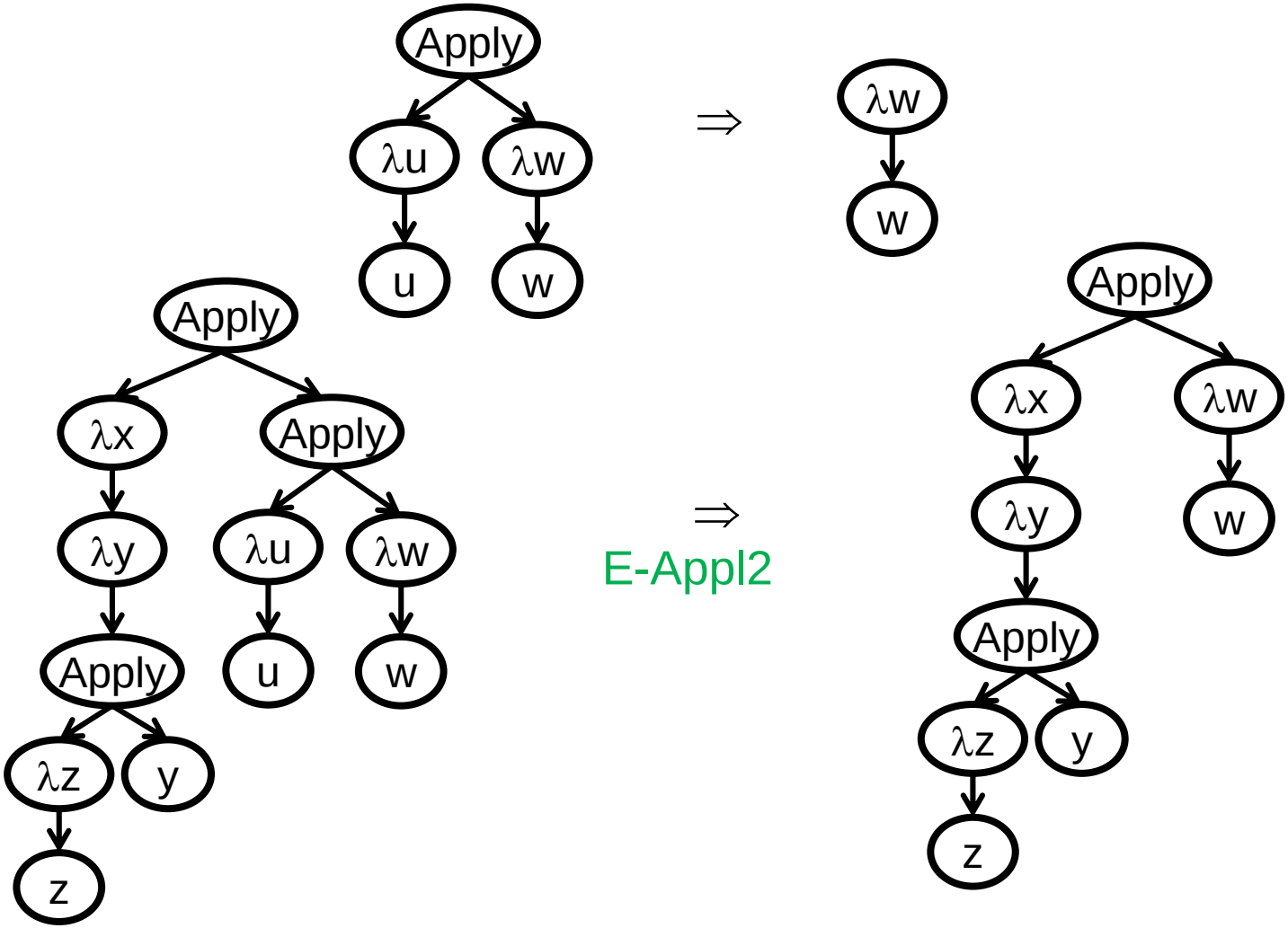
$(\lambda\ x.\ \lambda\ y.\ (\lambda\ z.\ z)\ y)\ \underline{((\lambda\ u.\ u)\ (\lambda\ w.\ w))}$



Call By Value OS (2)

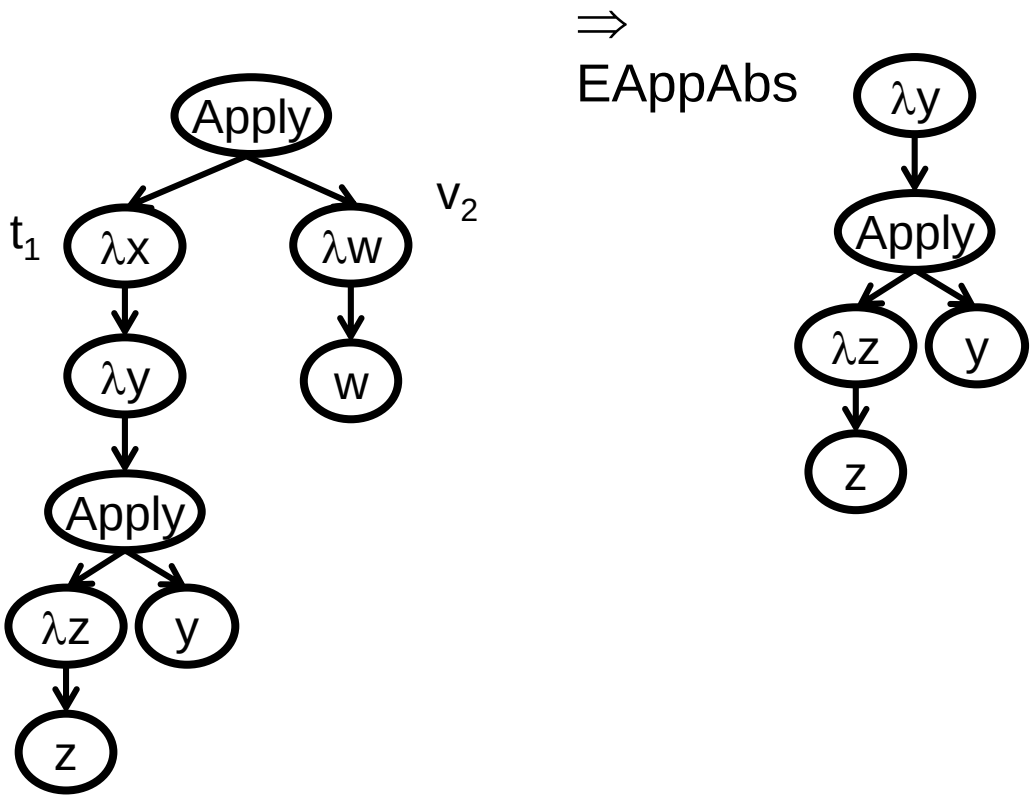
v_1 t_2

$(\lambda\, x.\, \lambda\, y.\, (\lambda\, z.\, z)\, y)\, ((\lambda\, u.\, u)\, (\lambda\, w.\, w))$



Call By Value OS (3)

$(\lambda\ x.\ \lambda\ y.\ (\lambda\ z.\ z)\ y)\ ((\lambda\ u.\ u)\ (\lambda\ w.\ w))$



Call-by-value big-step Operational Semantics

t ::=	terms
x	variable
λ x. t	abstraction
t t	application

v ::=	values
λ x. t	abstraction values
other values	

$$\lambda\, x.\ t \rightarrow \lambda\, x.\ t \qquad \text{(V-Value)}$$

$$\frac{t_1 \rightarrow \lambda\, x.\ t_3 \quad t_2 \rightarrow v_1 \quad [x \mapsto v_1]\, t_3 \rightarrow v_2}{t_1\ t_2 \rightarrow v_2} \qquad \text{(V-App)}$$

A Pseudocode for call-by value interpreter

```
Lambda eval(Lambda t) {
   switch(t) {
        case t= λx. t1:   // A value
                return t
        case t = t1 t2:
                Lambda temp = eval(t1);
                assert temp = λx. t3;
                Lambda v1 = eval(t2) ; // v1 must be a value
                return eval([x ↦ v1] t3) ;
      default: assert false;
    }
}
```

Programming in λ Calculus

- Functions with multiple arguments
- Simulating values
  - Tuples
  - Booleans
  - Numerics
- Recursion

Programming in the λ Multiple arguments

f= λ(x, y). s

-> Currying

f= λx. λy. s

$f\ v\ w = (f\ v)\ w = ((\lambda x.\ \lambda y.s)\ v)\ w \Rightarrow (\lambda y.[x \mapsto v]s)\ w \Rightarrow [x \mapsto v]\ [y \mapsto w]\ s$

$((\lambda x.\ \lambda y.\ x*x+y*y)\ 3)\ 4 \Rightarrow (\lambda y.[x \mapsto 3]\ x*x+y*y)\ 4 = (\lambda y.3*3+y*y)\ 4 \Rightarrow$

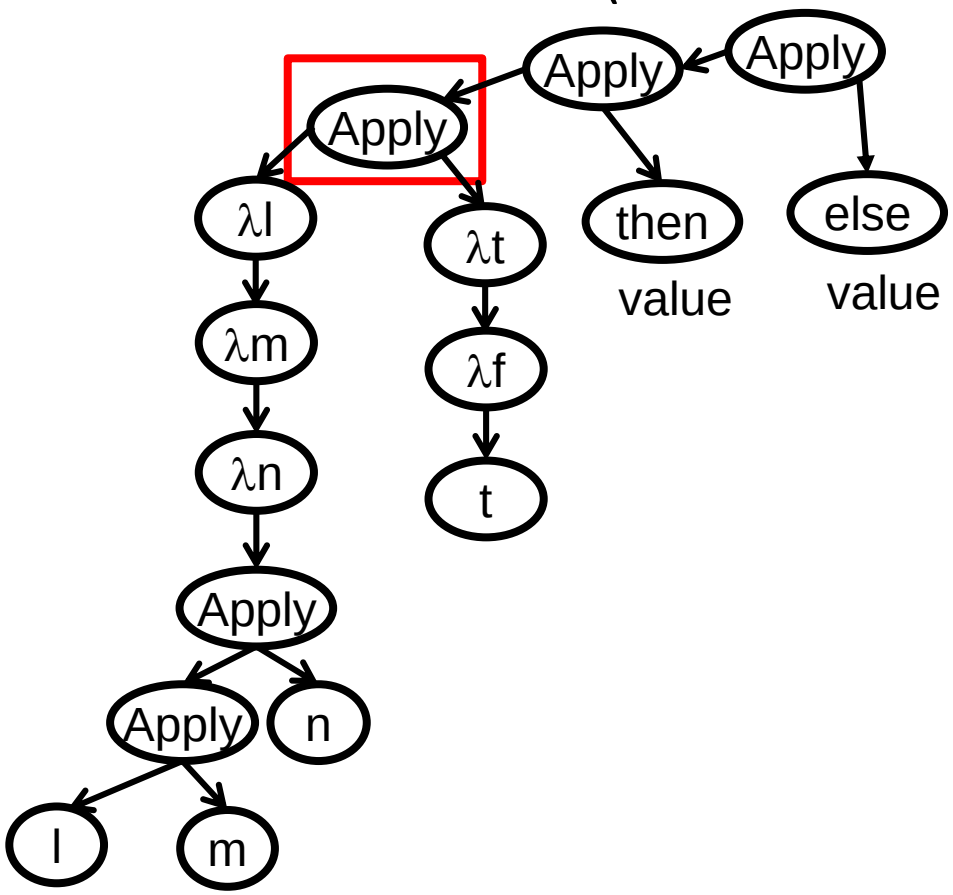
$[y \mapsto 4]\ 3*3+y*y = 3*3+4*4$

Simulating Booleans

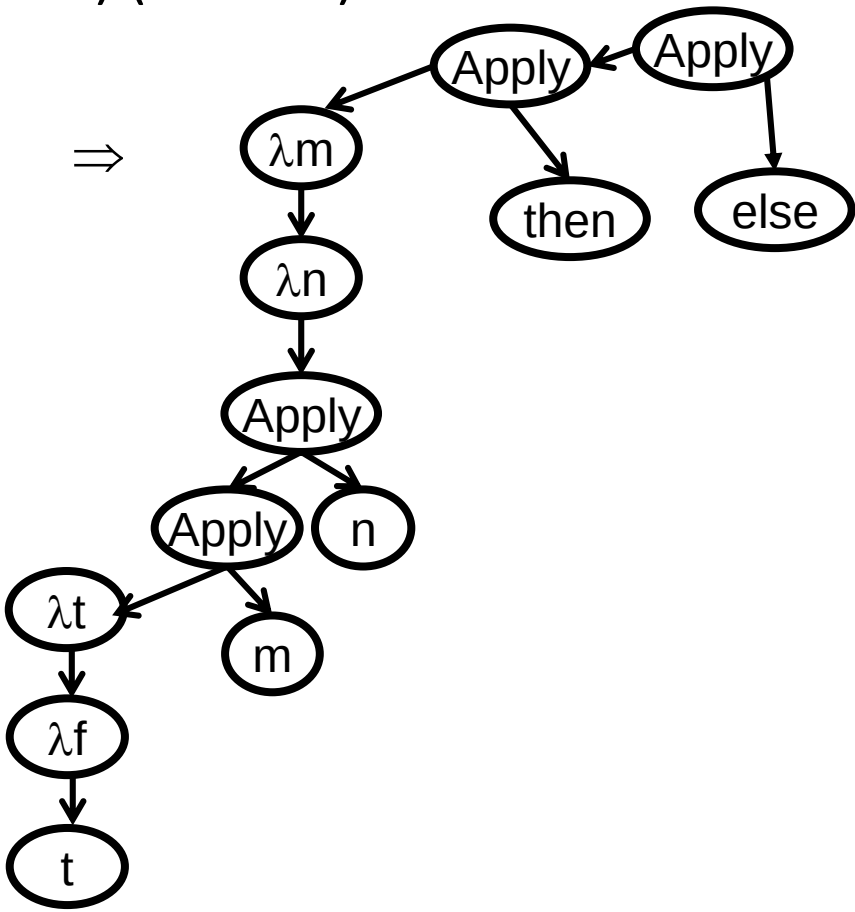
- tru = λt. λf. t
- fls = λt. λf. f
- test = λl. λm. λn. l m n

Showing that test is correct

- tru = λt. λf. t fls = λt. λf. f
- test = λl. λm. λn. l m n
- test tru then else = (λl. λm. λn. l m n) (λt. λf. t) then else

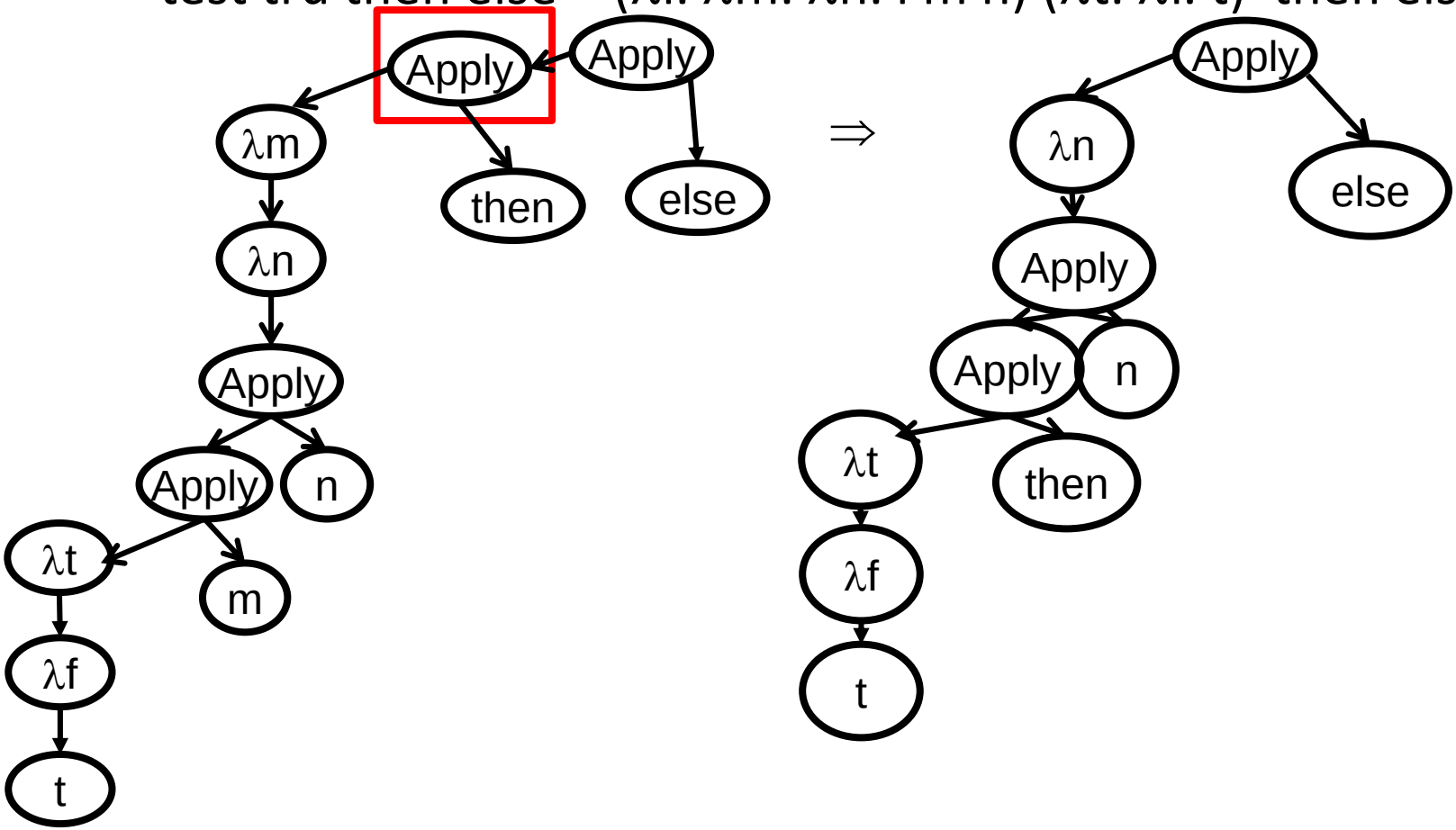


⇒



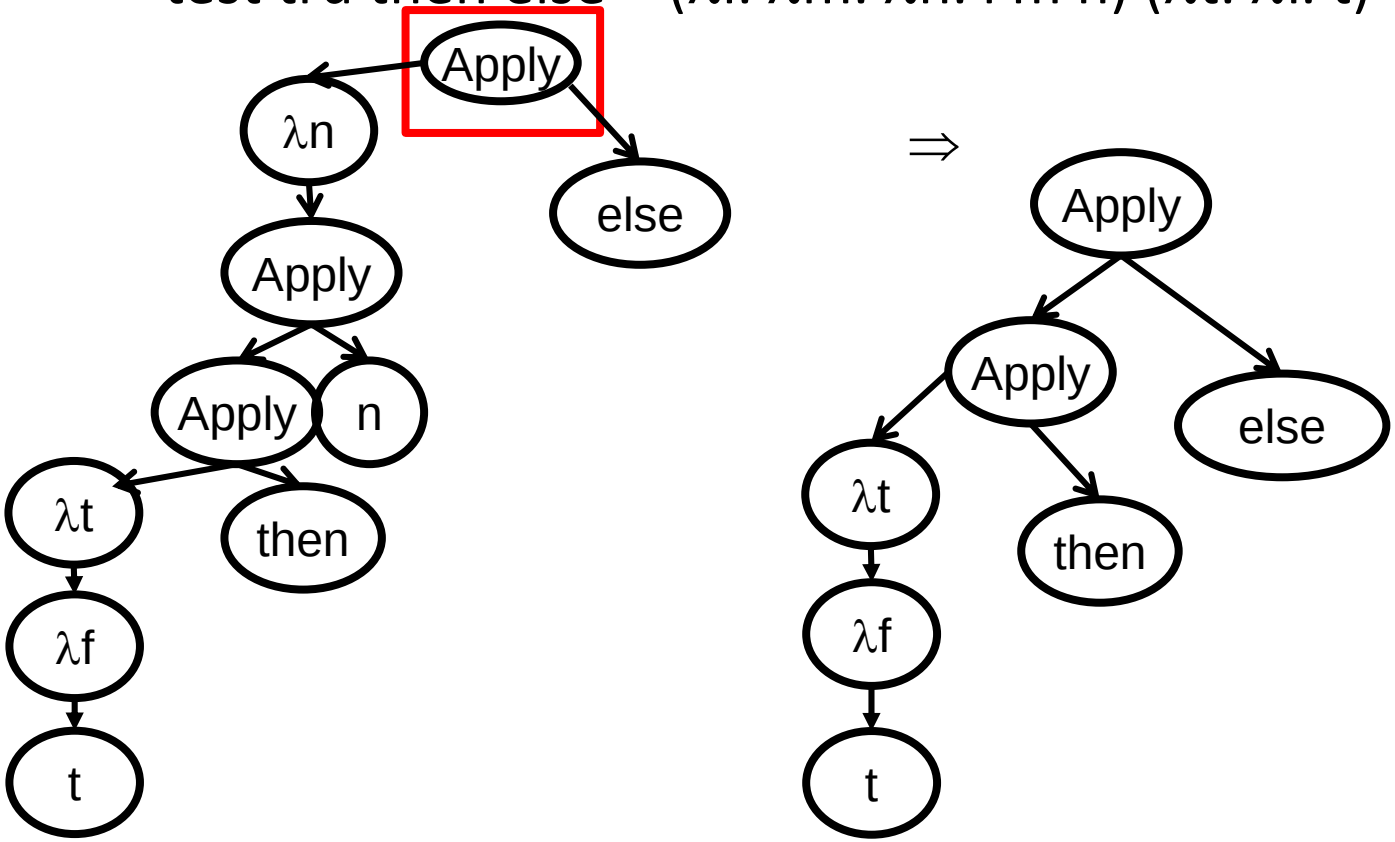
Showing that test is correct(2)

- tru = λt. λf. t fls = λt. λf. f
- test = λl. λm. λn. l m n
- test tru then else = (λl. λm. λn. l m n) (λt. λf. t) then else



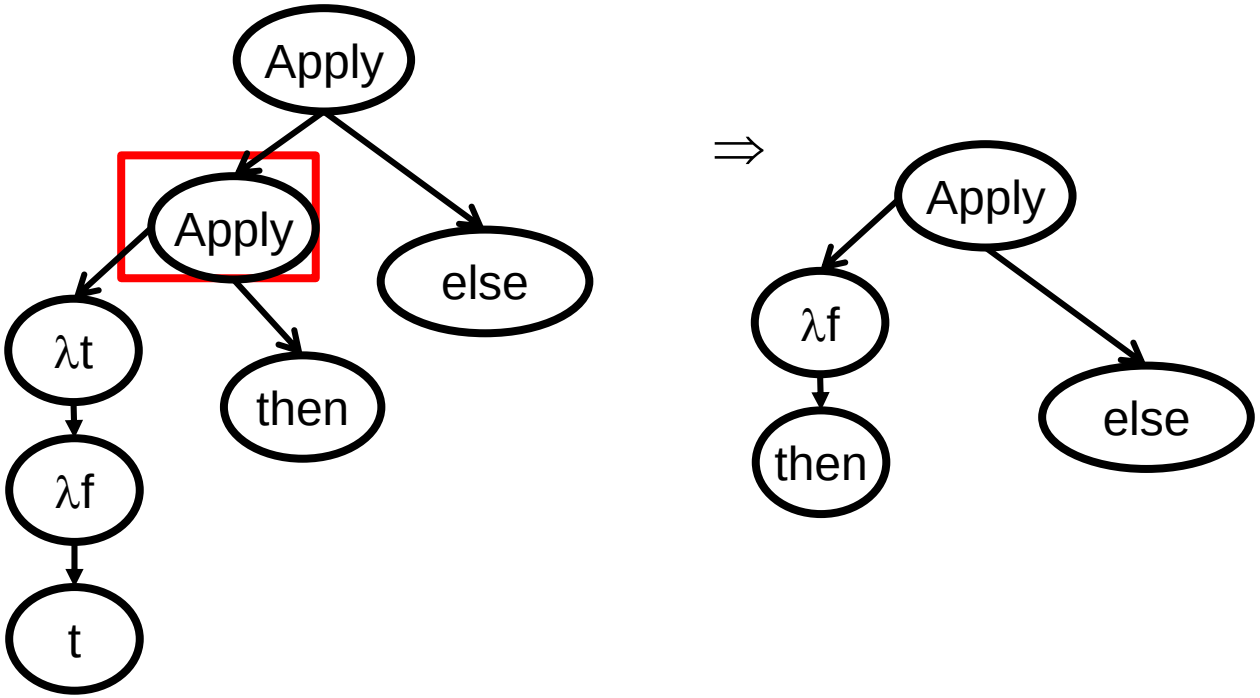
Showing that test is correct(3)

- tru = λt. λf. t fls = λt. λf. f
- test = λl. λm. λn. l m n
- test tru then else = (λl. λm. λn. l m n) (λt. λf. t) then else



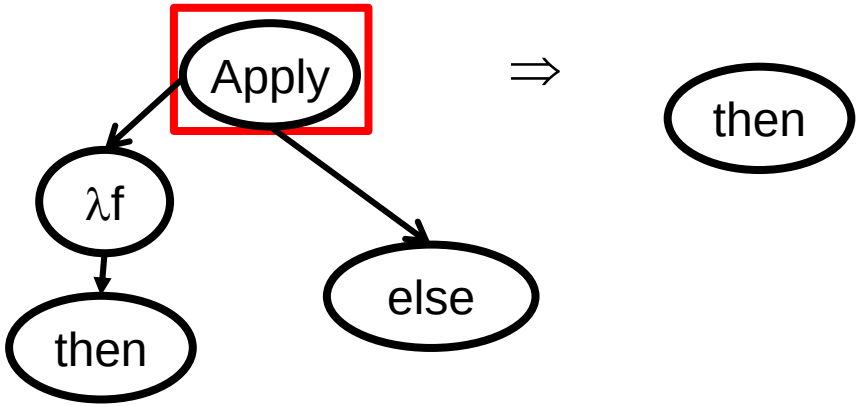
Showing that test is correct(4)

- tru = λt. λf. t fls = λt. λf. f
- test = λl. λm. λn. l m n
- test tru then else = (λl. λm. λn. l m n) (λt. λf. t) then else



Showing that test is correct(5)

- tru = λt. λf. t fls = λt. λf. f
- test = λl. λm. λn. l m n
- test tru then else = (λl. λm. λn. l m n) (λt. λf. t) then else



Showing that test is correct

- tru = λt. λf. t
- fls = λt. λf. f
- test = λl. λm. λn. l m n

test tru then else = (λl. λm. λn. l m n) (λt. λf. t) then else

$\Rightarrow_\beta$ (λm. λn. (λt. λf. t) m n) then else

$\Rightarrow_\beta$ (λn. (λt. λf. t) then n) else

$\Rightarrow_\beta$ (λt. λf. t) then else

$\Rightarrow_\beta$ (λf. then) else

$\Rightarrow_\beta$ then

Simulating Tests

- tru = λt. λf. t
- fls = λt. λf. f
- test = λl. λm. λn. l m n
- test tru then else = (λl. λm. λn. l m n) (λt. λf. t)
- test fls then else = (λl. λm. λn. l m n) (λt. λf. f)

Programming in λ Booleans

- tru = λt. λf. t
- fls = λt. λf. f
- test = λl. λm. λn. l m n
- test tru then else = (λl. λm. λn. l m n) (λt. λf. t)
- test fls then else = (λl. λm. λn. l m n) (λt. λf. f)
- and = λb. λb'. test b b' fls
- or = ?
- not = ?

Church Numerals

- $c_0 = \lambda s.\ \lambda z.\ z$
- $c_1 = \lambda s.\ \lambda z.\ s\ z$
- $c_2 = \lambda s.\ \lambda z.\ s\ (s\ z)$
- $c_3 = \lambda s.\ \lambda z.\ s\ (s\ (s\ z))$
- $c_n = \lambda s.\ \lambda z.\ s\ (s\ \ldots(s\ z)))$

The Successor Function

- succ = $\lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)$

succ c_0 = $(\lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z))\ (\lambda s1.\ \lambda z1.\ z1)$

$\Rightarrow_\beta$ $\lambda s.\ \lambda z.\ s\ ((\lambda s1.\ \lambda z1.\ z1)\ s\ z)$

$\Rightarrow_\beta$ $\lambda s.\ \lambda z.\ s\ ((\lambda z1.\ z1)\ z)$

$\Rightarrow_\beta$ $\lambda s.\ \lambda z.\ s\ z$

The Successor Function (Cont)

- succ = $\lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)$

$(c_2\ succ)\ c_0$

Addition

- plus = λm. λn. λs. λz. m s (n s z)

Multiplication

- times = λm. λn. m (plus n) c_0

Combinators

- A combinator is a function in the Lambda Calculus having no free variables
- Examples
  - $\lambda x.\ x$ is a combinator
  - $\lambda x.\ \lambda y.\ (x\ y)$ is a combinator
  - $\lambda x.\ \lambda y.\ (x\ z)$ is not a combinator
- Combinators can serve nicely as modular building blocks for more complex expressions
- The Church numerals and simulated Booleans are examples of useful combinators

Recursion

- Where is recursion in the lambda calculus?
- Cannot define
  FAC = (test (= n 0) 1 (* n (FAC (- n 1))))
- Even let does not help

Fixpoints

- We say X is a fixpoint of a function F if F(X)= X
- Examples
  - F = λx.x
  - F = λx.2
  - F = λx.(+ 1 x)
- FAC is a fixpoint of the non-recursive equation
  - H=λf.λn. (test (= n 0) 1 (* n (f (- n 1))))
- Compute FAC(n) from H using beta reductions

The Y-combinator

- FAC is a fixpoint of the non-recursive equation
  - H=λf.λn. (test (= n 0) 1 (* n (f (- n 1))))
- A function Y which computes FAC from H
  - FAC = Y H
  - FAC = H FAC
  - Y H = H (Y H)
  - Y is the function which that takes a function f and applies f (f (… (f …)))

The Y-combinator

- FAC is a fixpoint of the non-recursive equation
  - H=λf.λn. (test (= n 0) 1 (* n (f (- n 1))))
- A function Y which computes FAC from H
  - FAC = Y H
  - FAC = H FAC
  - Y H = H (Y H)

FAC 1 = Y H 1 = H (Y H) 1

= (λf.λn. test (= n 0) 1 (* n f (- n 1))) (Y H) 1

$\Rightarrow_\beta$ (λn. test (= n 0) 1 (* n (Y H)(- n 1))) 1

$\Rightarrow_\beta$ test (= 1 0) 1 (* 1 (Y H) (- 1 1)) $\Rightarrow^*_\beta$ * 1 (Y H) 0

= * 1 (H (Y H)) 0 = * 1 ((λf.λn. test (= n 0) 1 (* n f (- n 1))) (Y H)) 0

$\Rightarrow_\beta$ * 1 (λn. test (= n 0) 1 (* n (Y H) (- n 1))) 0

$\Rightarrow_\beta$ * 1 (test (= 0 0) 1 (* 0 (Y H) (- 0 1))) $\Rightarrow^*_\beta$ * 1 1 $\Rightarrow_\beta$ 1

The Y-combinator

- omega= $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

- Recursion can be simulated
  - $Y = \lambda x\ .\ (\lambda y.\ x\ (y\ y))\ (\lambda y.\ x\ (y\ y))$
  - $Y\ f \quad \Rightarrow^*_{\beta}$ "f (Y f)"

Summary: Lambda Calculus

- Powerful
- The ultimate assembly language
- Useful to illustrate ideas
- But can be counterintuitive
- Usually extended with useful syntactic sugars
- Other calculi exist
  - pi-calculus
  - object calculus
  - mobile ambients
  - …