

# **The Lambda Calculus**

**CIS400 (Compiler Construction)  
Kris Micinski, Fall 2021**

This is a lecture/livecoding week. We will work through the slides and do some lecture, along with collaborative livecoding as we go. At the end of each lecture, I'll post the code.

# The Lambda Calculus

- A system for calculating based entirely on computing with functions.
- Developed as a foundation for mathematics (originally used to model the natural numbers) by **Alonzo Church** in 1936.
- Church's thesis: *"Every effectively calculable function (effectively decidable predicate) is general recursive"*, i.e., can be computed using the  $\lambda$ -calculus. Used to show there exist unsolvable problems.
- One of the simplest Turing-equivalent languages!
  - Church, with his student Alan Turing, proved the equivalent expressiveness of Turing machines and the  $\lambda$ -calculus (called the **Church-Turing thesis**).
- Still makes up the heart of all functional programming languages!

# The Lambda Calculus

*lambdas* are just anonymous functions!

$e \in \mathbf{Exp} ::= (\lambda (x) e)$	$\lambda$ -abstraction
$\quad \quad \quad   (e e)$	function application
$\quad \quad \quad   x$	variable reference

$x \in \mathbf{Var} ::= \langle \mathbf{variables} \rangle$

# Textual-reduction semantics

- One way of designing a formal semantics is as a relation over terms in the language—one that reduces the term textually.
- This is usually ***small-step***—each eval step must terminate (meaning there are no *premises above the line* in our rules of inference and no recursive use of the interpreter within a step.)
- Consider a small-step semantics for our arithmetic language:

$$a \in \mathbf{AExp} ::= n \mid a + a \mid a - a \mid a \times a$$

$$n, m \in \mathbf{Num} ::= \langle \mathbf{integer\ constants} \rangle$$

## Exercise



Which of the following are **AExps**:

- 10
- 20.5
- $10 + 3$
- $10 + 3 * 4^2$
- $5 - 3 * 2 + 3 * 1$

$$a \in \mathbf{AExp} ::= n \mid a + a \mid a - a \mid a \times a$$
$$n, m \in \mathbf{Num} ::= \langle \text{integer constants} \rangle$$

## Exercise



Which of the following are **AExps**:

- **10**
- 20.5 — No, not integer constant
- **10 + 3**
- $10 + 3 * 4^2$  — Exponent not allowed
- **5 - 3 \* 2 + 3 \* 1**

$$a \in \mathbf{AExp} ::= n \mid a + a \mid a - a \mid a \times a$$
$$n, m \in \mathbf{Num} ::= \langle \mathbf{integer\ constants} \rangle$$

# Textual-reduction semantics

$$a \in \mathbf{AExp} ::= n \mid a + a \mid a - a \mid a \times a$$

$$n, m \in \mathbf{Num} ::= \langle \mathbf{integer\ constants} \rangle$$

- Rules to reduce terms in this language match operations that have two numeric operands already and apply the operation, textually substituting a numeric value for the operation; e.g.:

$$\frac{}{a_0 \times a_1 \Rightarrow n_0 * n_1} \quad \mathbf{where} \ a_0 \text{ is } n_0 \text{ and } a_1 \text{ is } n_1$$

- For example:  $2 * 3 + 4 * 5 \Rightarrow 2 * 3 + 20 \Rightarrow 6 + 20 \Rightarrow 26$
- Is there another way to evaluate  $2*3 + 4*5$  using similar rules?

# The Lambda Calculus

*lambdas* are just anonymous functions!

$e \in \mathbf{Exp} ::= (\lambda (x) e)$	$\lambda$ -abstraction
$\quad \quad \quad   (e e)$	function application
$\quad \quad \quad   x$	variable reference

$x \in \mathbf{Var} ::= \langle \mathbf{variables} \rangle$

# The Lambda Calculus

The lambda-calculus is the functional core of Racket (as of other functional languages).

Just the following subset of Racket is Turing-equivalent!

$$\begin{array}{ll} e \in \mathbf{Exp} ::= (\lambda (x) e) & (\text{lambda } (x) e) \\ & | (e e) & (e0 e1) \\ & | x & x \end{array}$$
$$x \in \mathbf{Var} ::= \langle \mathbf{variables} \rangle$$

# Lambda Abstraction

An expression, *abstracted* over all possible values for a formal parameter, in this case,  $x$ .

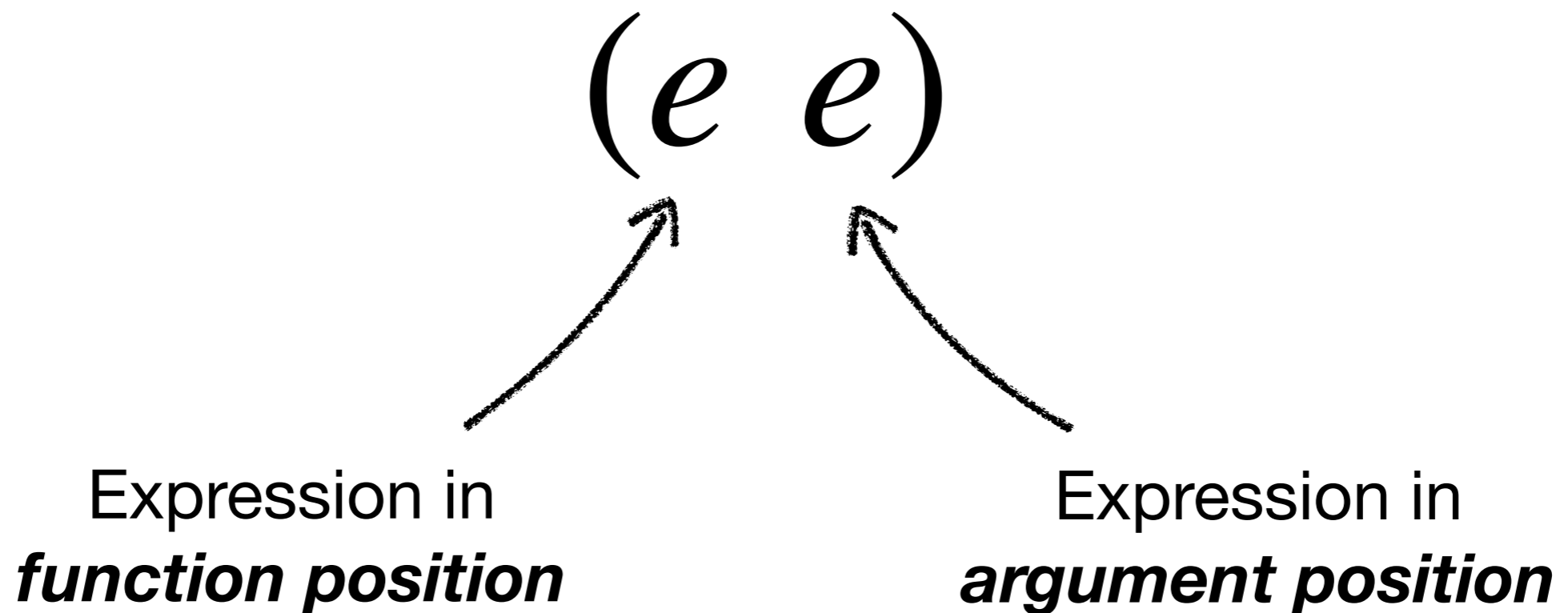
$$(\lambda (x) e)$$

Formal parameter

Function body

# Application

When the first expression is evaluated to a value (in this language, all values are functions!) it may be invoked / applied on its argument.



# Variables

Variables are only defined/assigned when a function is applied and its parameter bound to an argument.

$x$



Variable reference

$((\lambda (f) (f (f (\lambda (x) x)))) (\lambda (x) x))$

We define a rule for step-by-step evaluation called ***Beta-reduction***



$\beta$

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



$\beta$

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



$\beta$

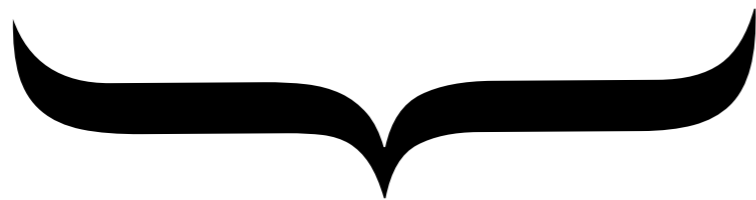
$(\lambda (x) x)$

**Textual substitution.** This says:  
*replace every  $x$  in  $E_0$  with  $E_1$ .*

$((\lambda (x) E_0) E_1)$

$\rightarrow_\beta$

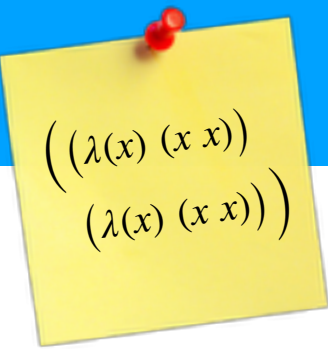
$E_0[x \leftarrow E_1]$



redex

(**re**ducible **ex**pression)

# Example



$$\left( \begin{array}{l} (\lambda(x) (x x)) \\ (\lambda(x) (x x)) \end{array} \right)$$

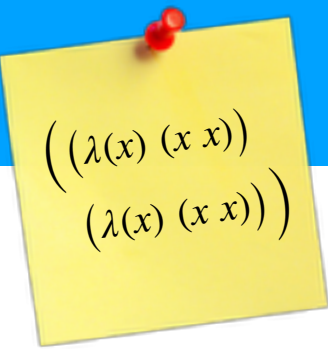
$$((\lambda (x) x) (\lambda (x) x))$$



$\beta$

$$x [x \leftarrow (\lambda (x) x)]$$

# Example



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

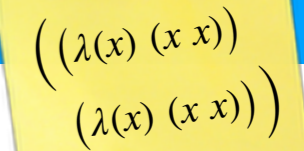
$((\lambda (x) x) (\lambda (x) x))$



$\beta$

$(\lambda (x) x)$

## Example



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

Can you beta-reduce the following term more than once:

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

$\beta$  reduction may continue indefinitely (i.e., in non-terminating programs)

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

$\beta$

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

$\beta$

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

$\beta$

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

$\beta$

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



$\beta$

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



$\beta$

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



$\beta$

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



$\beta$

This specific program is  
known as  $\Omega$  (Omega)

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

$\beta$

$\Omega$  is the smallest non-terminating program!

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

Note how it reduces to itself in a single step!

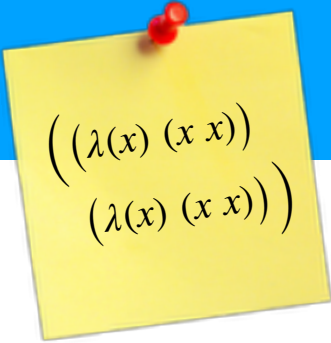
$((\lambda (x) (x\ x)) (\lambda (x) (x\ x)))$

$\beta$

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

$\beta$

# Example



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

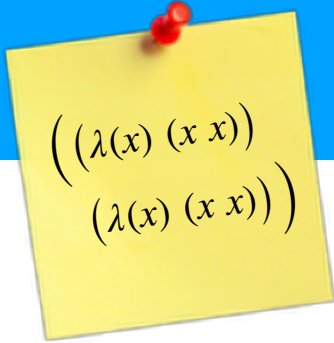
Evaluation with  $\beta$  reduction is nondeterministic!

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

$\beta$

$((\lambda(x) x) ((\lambda(y) y) (\lambda(z) z)))$

# Example



$$\left( \begin{array}{l} (\lambda(x) (x x)) \\ (\lambda(x) (x x)) \end{array} \right)$$

Evaluation with  $\beta$  reduction is nondeterministic!

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

$\beta$  **or!**

$\beta$

$((\lambda(x) x) ((\lambda(y) y) (\lambda(z) z)))$

$((\lambda(w) w) (\lambda(x) x)) (\lambda(z) z))$

## Exercise



Perform each possible  $\beta$ -reduction

$$((\lambda (x) ((\lambda (y) (x y)) x)) (\lambda (z) (z z)))$$

How many different  $\beta$ -reductions are possible from the above?

# Exercise



$((\lambda (x) ((\lambda (y) (x\ y))\ x))\ (\lambda (z) (z\ z)))$

$\beta$

$((\lambda (x) (x\ x))\ (\lambda (z) (z\ z)))$

Can reduce inner redex...

# Exercise



$((\lambda (x) ((\lambda (y) (x\ y))\ x))\ (\lambda (z) (z\ z)))$

$\downarrow \beta$

$((\lambda (y) ((\lambda (z) (z\ z))\ y))\ (\lambda (z) (z\ z)))$

Or the outer redex.

# Exercise


$$((\lambda (x) ((\lambda (y) (x\ y))\ x))\ (\lambda (z) (z\ z)))$$

$\beta$

$$((\lambda (y) ((\lambda (z) (z\ z))\ y))\ (\lambda (z) (z\ z)))$$

Can't reduce this since we don't (yet) know about the particular value (function)  $z$  in call position.

# Free Variables

We define the free variables of a lambda expression via the function  $\mathbf{FV}$ :

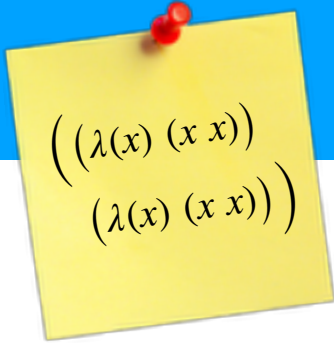
$$\mathbf{FV} : \mathbf{Exp} \rightarrow \mathcal{P}(\mathbf{Var})$$

$$\mathbf{FV}(x) \triangleq \{x\}$$

$$\mathbf{FV}((\lambda (x) e_b)) \triangleq \mathbf{FV}(e_b) \setminus \{x\}$$

$$\mathbf{FV}(e_f e_a) \triangleq \mathbf{FV}(e_f) \cup \mathbf{FV}(e_a)$$

# Example



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

$$\mathbf{FV}((x \ y)) = \{x, y\}$$

$$\mathbf{FV}((\lambda(x) \ x) \ y) = \{y\}$$

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

$$\mathbf{FV}((\lambda(y) \ ((\lambda(x) \ (z \ x)) \ x))) = \{z, x\}$$

## Exercise

What are the free variables of each of the following terms?


$$((\lambda (x) x) y)$$
$$((\lambda (x) (x x)) (\lambda (x) (x x)))$$
$$((\lambda (x) (z y)) x)$$

## Exercise



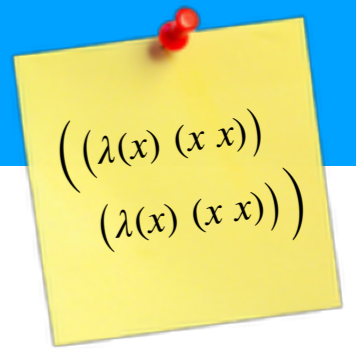
What are the free variables of each of the following terms?

$((\lambda (x) x) y)$   
**{y}**

$((\lambda (x) (x x)) (\lambda (x) (x x)))$   
**{}**

$((\lambda (x) (z y)) x)$   
**{x, y, z}**

# Example

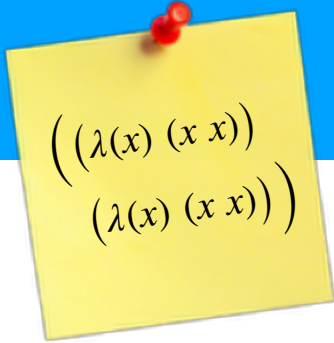


The problem with (naive) textual substitution

$$((\lambda (a) (\lambda (a) a)) (\lambda (b) b))$$

↓ β

# Example

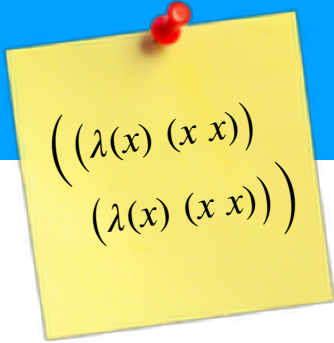


$$\left( \begin{array}{l} (\lambda(x) (x x)) \\ (\lambda(x) (x x)) \end{array} \right)$$

The problem with (naive) textual substitution

$$((\lambda (a) (\lambda (a) a)) (\lambda (b) b))$$
$$\downarrow \beta$$
$$(\lambda (a) a) [a \leftarrow (\lambda (b) b)]$$

# Example



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

The problem with (naive) textual substitution

$((\lambda (a) (\lambda (a) a)) (\lambda (b) b))$

$\downarrow \beta$

$(\lambda (a) (\lambda (b) b))$



# Capture-avoiding substitution

$$E_0 [x \leftarrow E_1]$$

$$x[x \leftarrow E] = E$$

$$y[x \leftarrow E] = y \text{ where } y \neq x$$

$$(E_0 \ E_1)[x \leftarrow E] = (E_0[x \leftarrow E] \ E_1[x \leftarrow E])$$

$$(\lambda \ (x) \ E_0)[x \leftarrow E] = (\lambda \ (x) \ E_0)$$

$$(\lambda \ (y) \ E_0)[x \leftarrow E] = (\lambda \ (y) \ E_0[x \leftarrow E])$$

where  $y \neq x$  and  $y \notin FV(E)$

$\beta$ -reduction cannot occur when  $y \in FV(E)$  

# Example

$\left( \begin{array}{l} (\lambda(x) (x x)) \\ (\lambda(x) (x x)) \end{array} \right)$

Capture-avoiding substitution

$((\lambda (a) (\lambda (a) a)) (\lambda (b) b))$

$\downarrow \beta$

$(\lambda (a) a)$



## Exercise



How can you beta-reduce the following expression using capture-avoiding substitution?

$$\begin{aligned} &((\lambda (y) \\ &\quad ((\lambda (z) (\lambda (y) (z\ y)))\ y)) \\ &(\lambda (x)\ x)) \end{aligned}$$

## Exercise



How can you beta-reduce the following expression using capture-avoiding substitution?

$$\begin{aligned} &((\lambda (y) \\ &\quad ((\lambda (z) (\lambda (y) (z\ y)))\ y)) \\ &(\lambda (x) x)) \end{aligned}$$

$\downarrow \beta$

$$((\lambda (z) (\lambda (y) (z\ y))) (\lambda (x) x))$$

## Exercise



How can you beta-reduce the following expression using capture-avoiding substitution?

$$(\lambda (y) ((\lambda (z) (\lambda (y) z)) (\lambda (x) y))))$$

## Exercise



How can you beta-reduce the following expression using capture-avoiding substitution?

$$(\lambda (y) ((\lambda (z) (\lambda (y) z)) (\lambda (x) y)))$$

**You cannot!** This redex would require:

$$(\lambda (y) z) [z \leftarrow (\lambda (x) y)]$$

(y is free here, so it would be captured)

## Exercise



How can you beta-reduce the following expression using capture-avoiding substitution?

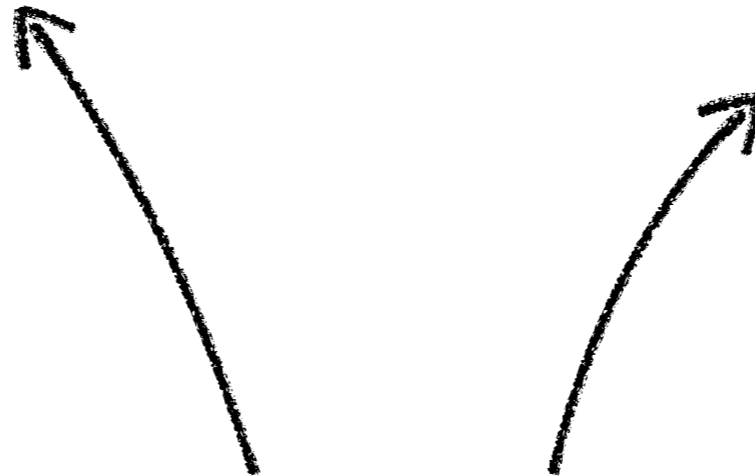
$$(\lambda \ (y) \ ((\lambda \ (z) \ (\lambda \ (y) \ z)) \ (\lambda \ (x) \ y)))$$
$$\rightarrow_{\alpha} (\lambda \ (y) \ ((\lambda \ (z) \ (\lambda \ (w) \ z)) \ (\lambda \ (x) \ y)))$$
$$\rightarrow_{\beta} (\lambda \ (y) \ (\lambda \ (w) \ (\lambda \ (x) \ y)))$$

**Instead we alpha-convert first.**

# $\alpha$ -renaming

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

$(\lambda (a) (\lambda (b) a))$



These two expressions are equivalent—they only differ by their variable names ( $x = a; y = b$ )

# $\alpha$ -renaming

$$(\lambda (x) E_\theta) \rightarrow_\alpha (\lambda (y) E_\theta[x \leftarrow y])$$

$=_\alpha$



$\alpha$  renaming/conversions can be run backward,  
so you might think of it as an equivalence relation

# $\alpha$ -renaming

$\alpha$  renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

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

# $\alpha$ -renaming

$\alpha$  renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!


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

Can't perform naive substitution w/o capturing  $x$ .

# $\alpha$ -renaming

$\alpha$  renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

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



Fix by  $\alpha$  renaming to  $z$

# $\alpha$ -renaming

$\alpha$  renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

$((\lambda (x) (\lambda (z) z)) (\lambda (y) y))$



Fix by  $\alpha$  renaming to  $z$

# $\alpha$ -renaming

$\alpha$  renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

$$((\lambda (x) (\lambda (z) z)) (\lambda (y) y))$$


Could now perform beta-reduction with naive substitution

# $\eta$ - reduction

$$(\lambda (x) (E_0 x)) \rightarrow_{\eta} E_0 \text{ where } x \notin FV(E_0)$$

# $\eta$ - expansion

$$E_0 \rightarrow_{\eta} (\lambda (x) (E_0 x)) \text{ where } x \notin FV(E_0)$$

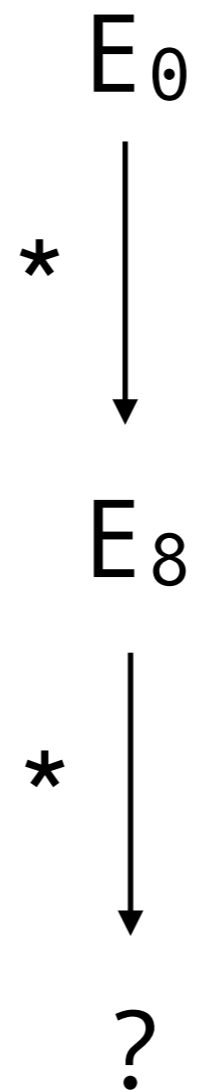
# Reduction

$$(\rightarrow) = (\rightarrow_{\beta}) \cup (\rightarrow_{\alpha}) \cup (\rightarrow_{\eta})$$

$$(\rightarrow^*)$$

reflexive/transitive closure

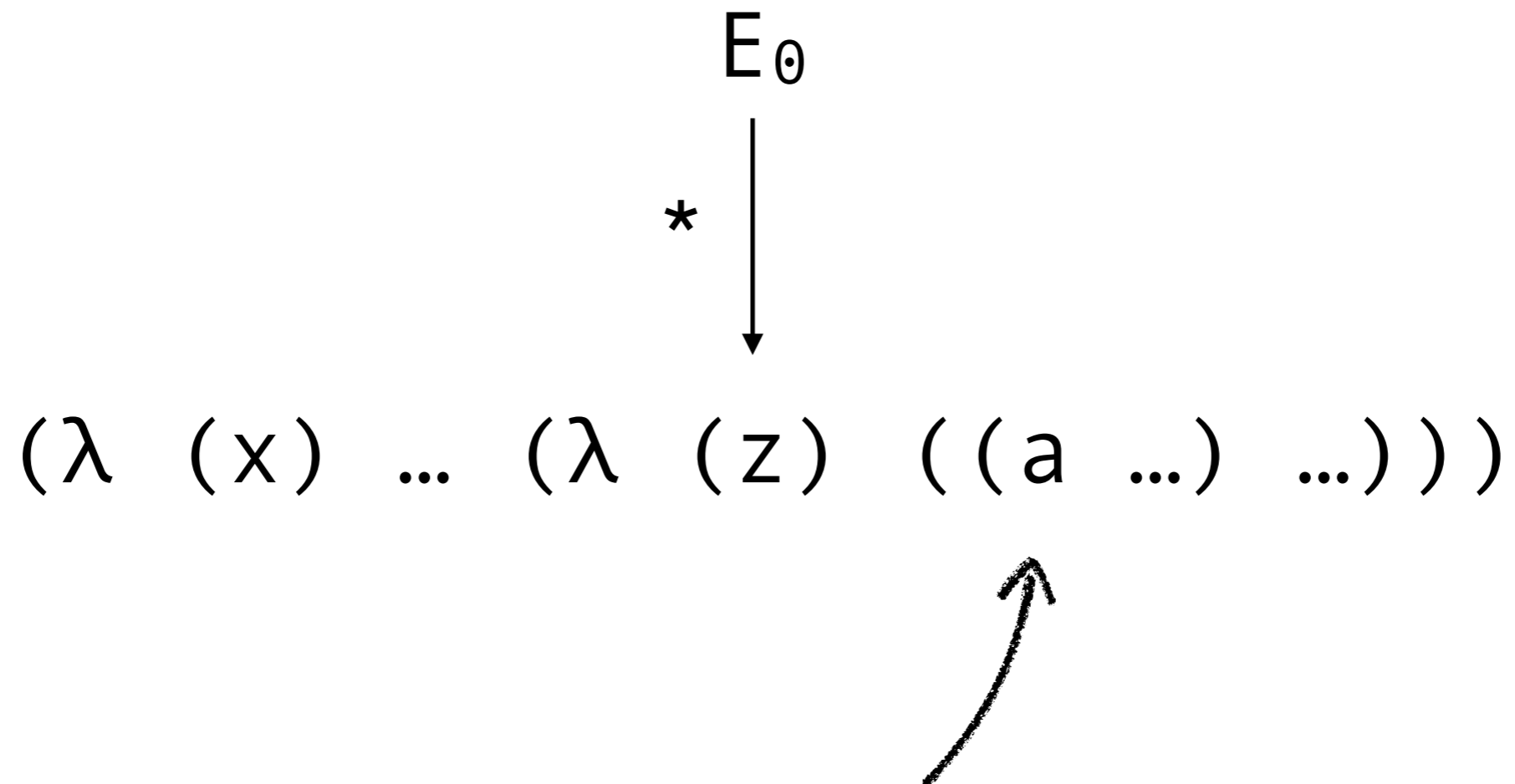
# Evaluation



# Evaluation to *normal form*

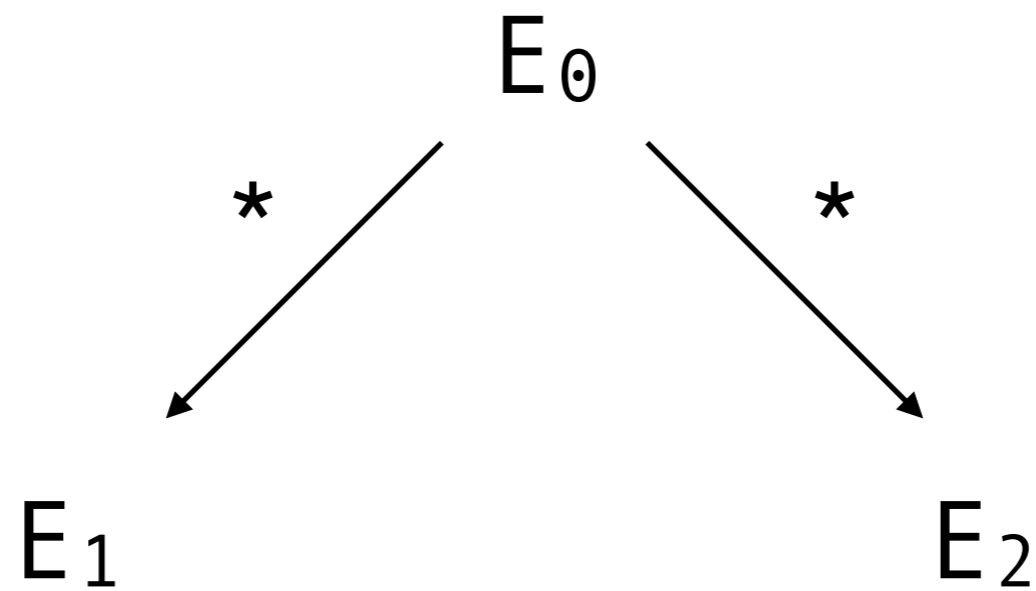
$$\begin{array}{c} E_{\theta} \\ \downarrow * \\ (\lambda \ (x) \ \dots) \end{array}$$

# Evaluation to *normal form*



In ***normal form***, no function position can be a lambda;  
this is to say: *there are no unreduced redexes left!*

# Evaluation Strategy



# Evaluation Strategy

$((\lambda (x) ((\lambda (y) y) x)) (\lambda (z) z))$

$\rightarrow_{\eta} ((\lambda (y) y) (\lambda (z) z))$

$\rightarrow_{\beta} (\lambda (z) z)$

# Evaluation Strategy

$((\lambda (x) ((\lambda (y) y) x)) (\lambda (z) z))$

$\rightarrow_{\beta} ((\lambda (y) y) (\lambda (z) z))$

$\rightarrow_{\beta} (\lambda (z) z)$

# Evaluation Strategy

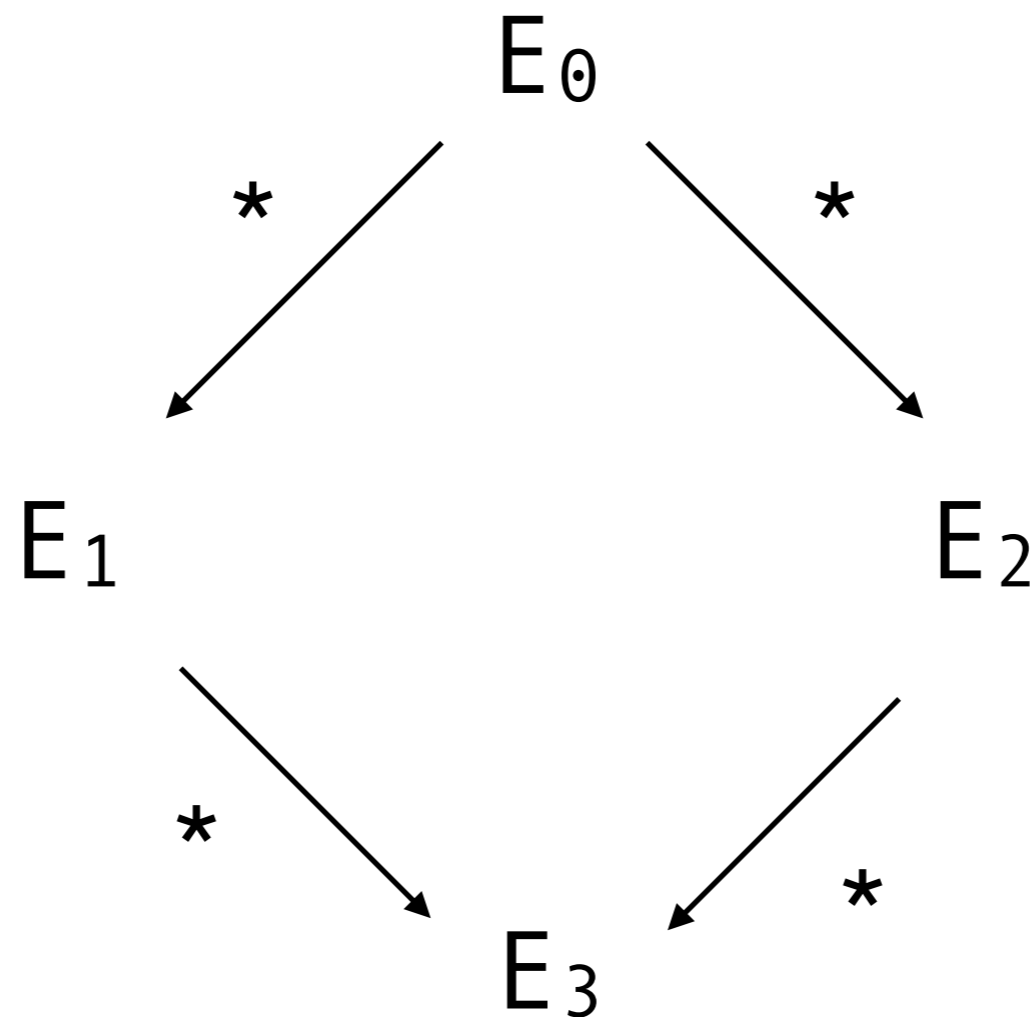
$((\lambda (x) ((\lambda (y) y) x)) (\lambda (z) z))$

$\rightarrow_{\beta} ((\lambda (x) x) (\lambda (z) z))$

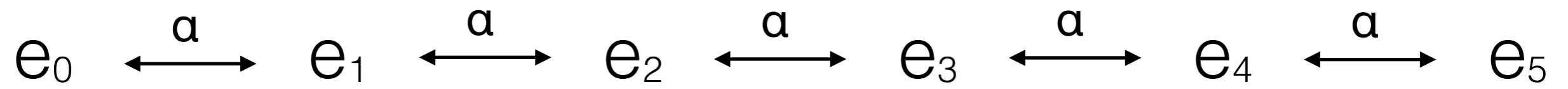
$\rightarrow_{\beta} (\lambda (z) z)$

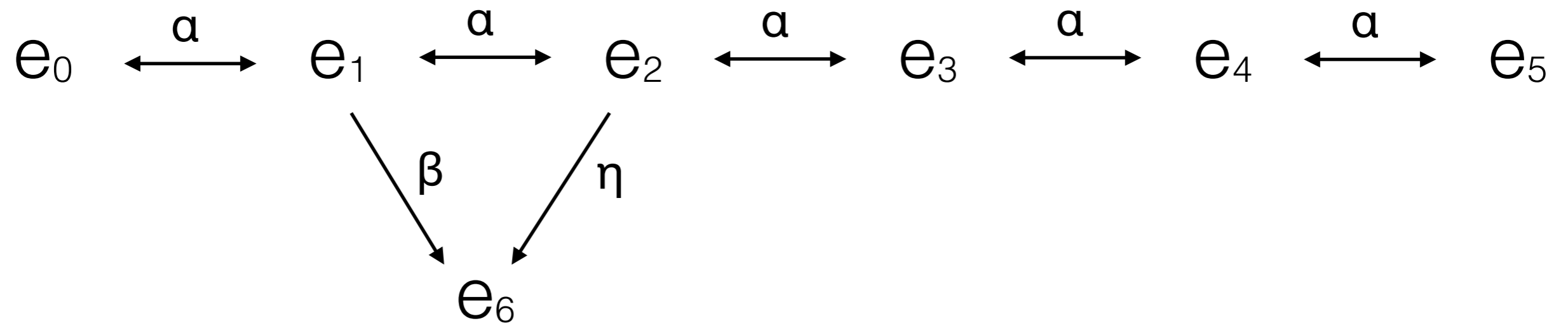
# Confluence

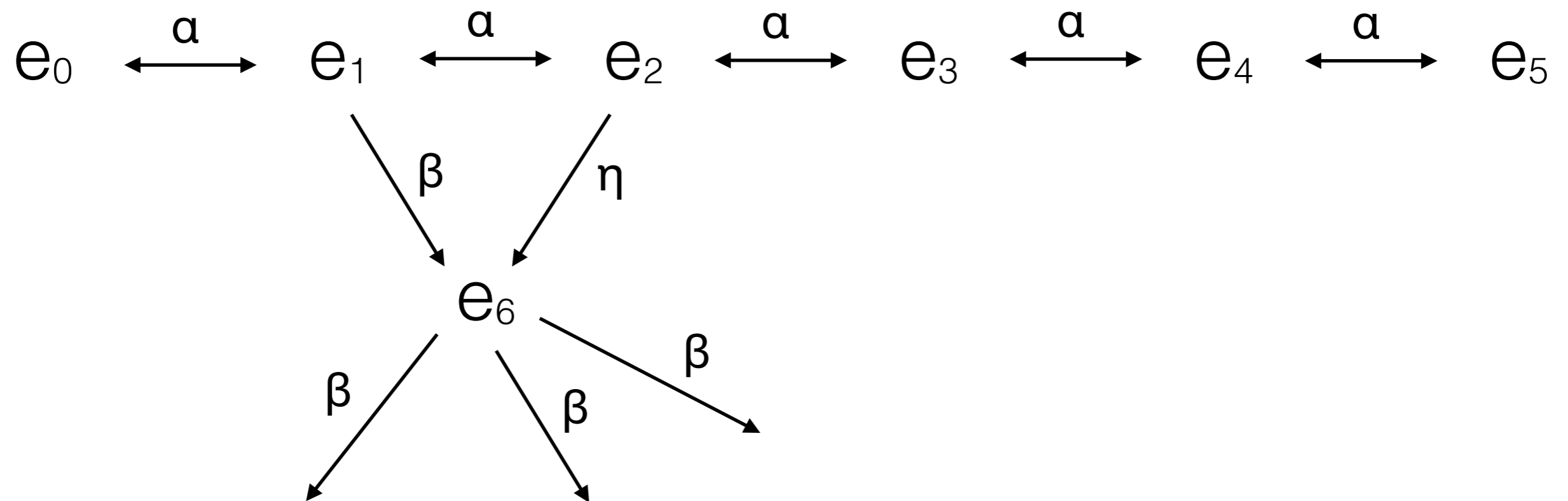
Diverging paths of evaluation must eventually join back together.

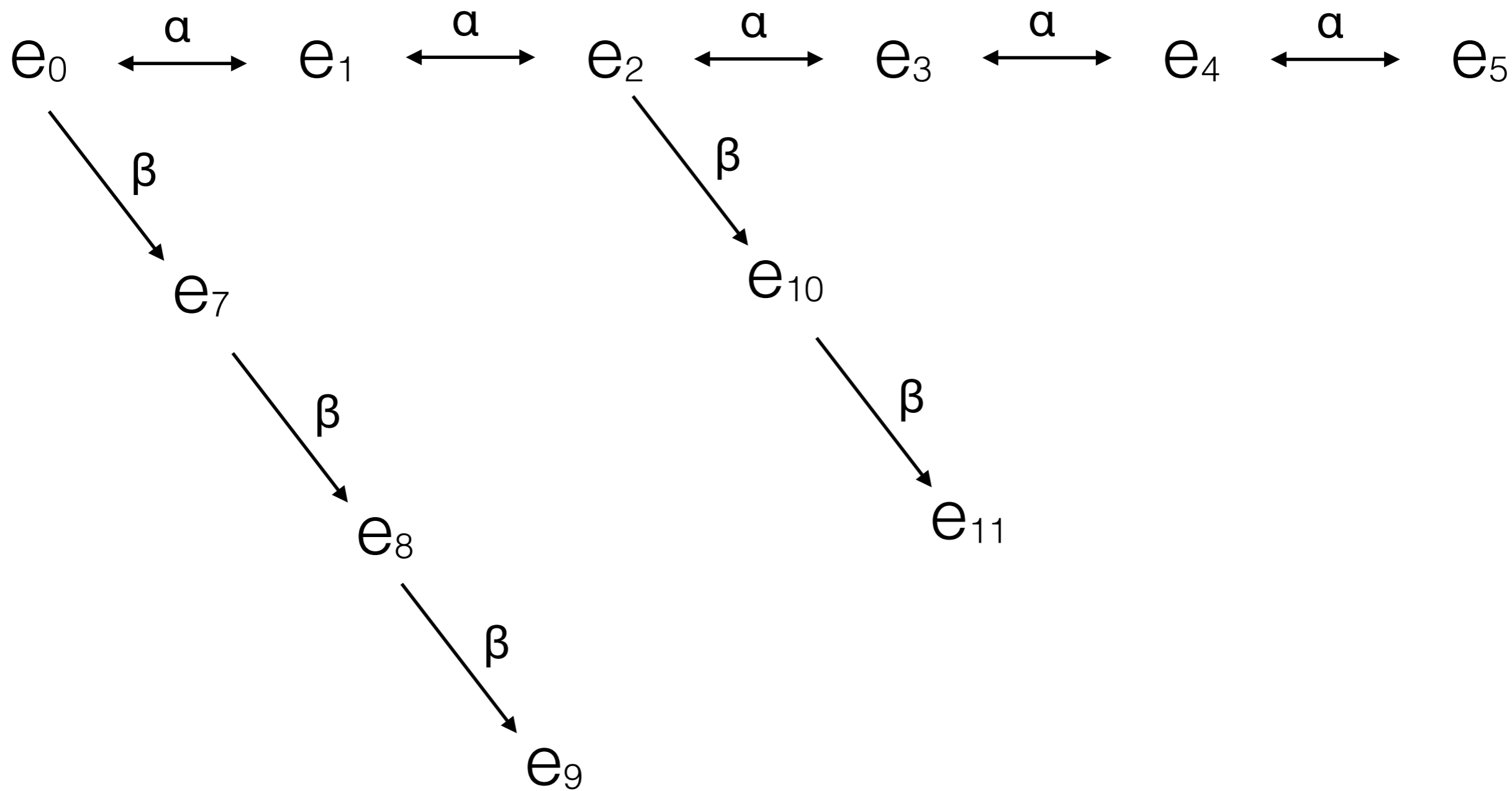


Church-Rosser Theorem

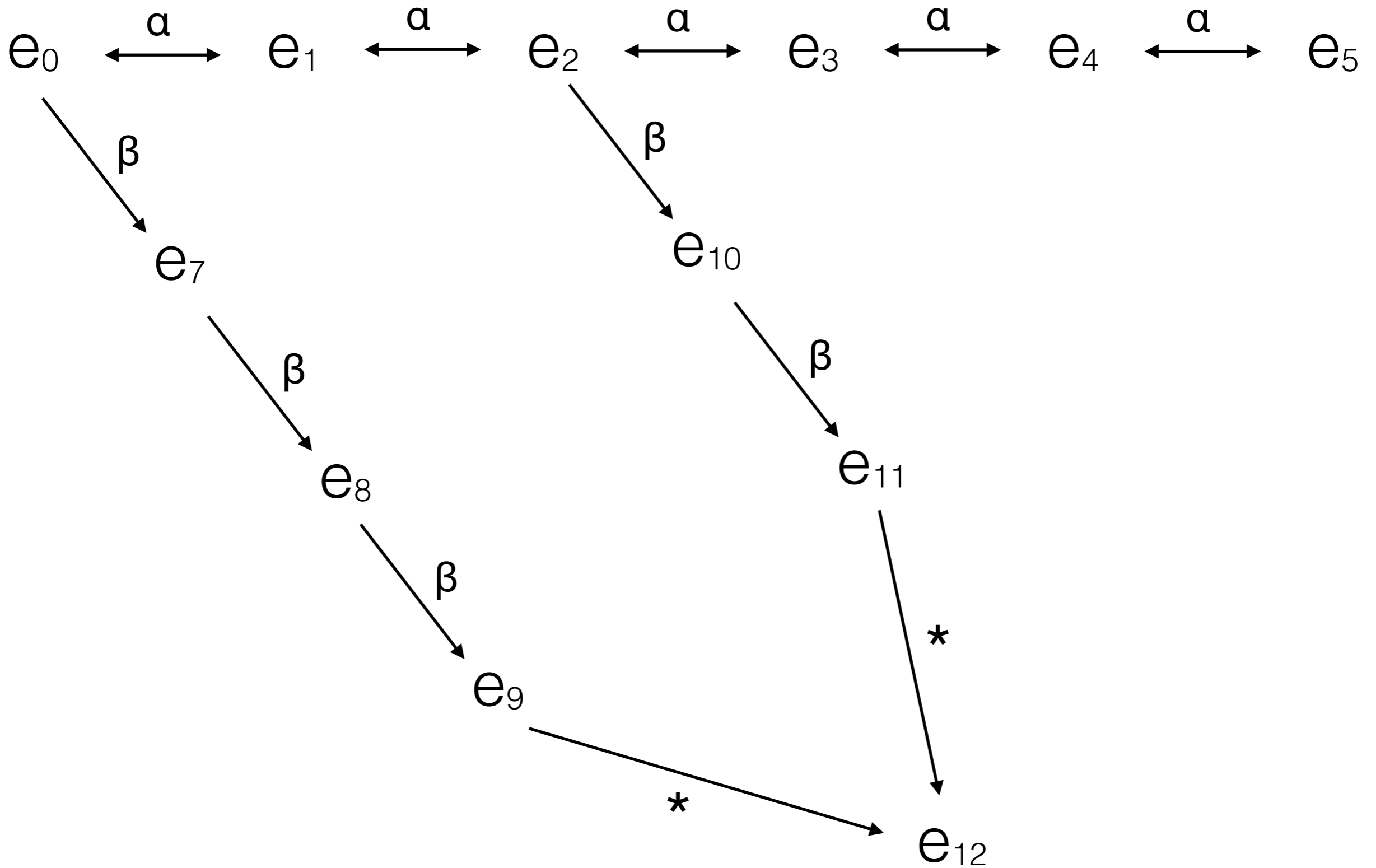








# Confluence (i.e., Church-Rosser Theorem)



## Applicative evaluation order

Always evaluates the *innermost* leftmost redex first.

## Normal evaluation order

Always evaluates the *outermost* leftmost redex first.

## Applicative evaluation order

$((\lambda (x) ((\lambda (y) y) x)) (\lambda (z) z))$

## Normal evaluation order

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

## Call-by-value (CBV) semantics

Applicative evaluation order, *but not under lambdas*.

## Call-by-name (CBN) semantics

Normal evaluation order, *but not under lambdas*.

## Exercise



Write a lambda term other than  $\Omega$  which also does not terminate

(Hint: think about using some form of self-application)

## Exercise



Write a lambda term other than  $\Omega$  which also does not terminate

$$\begin{aligned} &((\lambda (y) ((\lambda (x) (y\ x))\ y)) \\ &(\lambda (y) ((\lambda (x) (y\ x))\ y))) \end{aligned}$$
$$\begin{aligned} &((\lambda (u) ((u\ u)\ u)) \\ &(\lambda (u) ((u\ u)\ u))) \end{aligned}$$
$$\begin{aligned} &((\lambda (x)\ x) \\ &((\lambda (u) (u\ u)) \\ &(\lambda (u) (u\ u)))) \end{aligned}$$