Lambda Calculus & Church Encoding
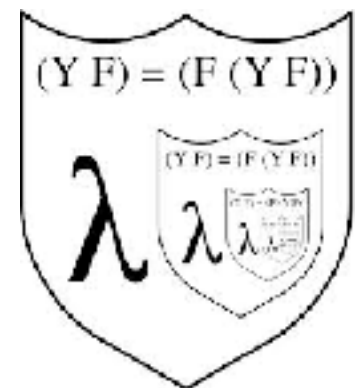
Kristopher Micinski (Slides in part by Thomas Gilray, UAB)
CIS 700 —Program Analysis: Foundations and Applications
Fall '19, Syracuse University

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

Today's insight

- We can define a **tiny language**—namely the lambda calculus—that encapsulates a Turing-equivalent formulation of our language.

- We can give this language a simple machine-based semantics via textual substitution (term-rewriting).

- Therefore, we will have encapsulated a fully-precise model of Core Scheme by the combination of (a) applications of the desugarings, and (b) the simple term-rewriting semantics of the lambda calculus.

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 λ-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 λ-calculus (called the **Church-Turing thesis**).

- Still makes up the heart of all functional programming languages!

The Lambda Calculus

lambdas are just anonymous functions!

$$e \in \textbf{Exp} ::= (\lambda\ (x)\ e) \qquad \lambda\text{-abstraction}$$
$$|\ (e\ e) \qquad \text{function application}$$
$$|\ x \qquad \text{variable reference}$$

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

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:

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

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

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.:

$$\frac{\rule{3cm}{0.4pt}}{a_0 \times a_1 \Rightarrow n_0 * n_1}$$ **where** a_0 is n_0 and a_1 is n_1

- For example: $\quad 2 * 3 + 4 * 5 \Rightarrow 2 * 3 + 20 \Rightarrow 6 + 20 \Rightarrow 26$

- Is there another way to evaluate 2*3 + 4*5 using similar rules?

Big Step Interpreters

Interpreter works in one "big step"

```
(define (eval env e)
  …)


(eval env `(,e0 ,e1))


    (eval env e0)
```

To evaluate (e0 e1), interpreter
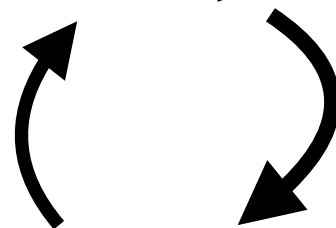recursively calls **itself** to evaluate e0

Big Step Interpreters

Interpreter works in one "big step"

```
(define (eval env e)
   …)
```
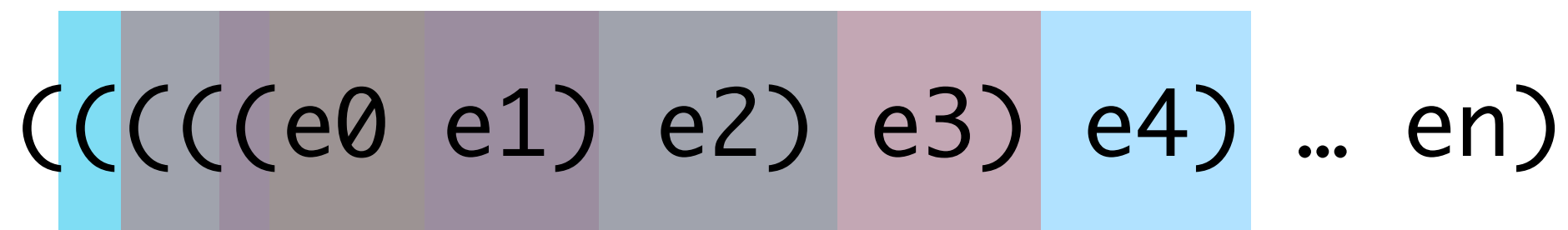
```
(eval env `(,e0 ,e1))
```

Returns result of e0, then continues evaluation…

```
(eval env e0)
```

To evaluate (e0 e1), interpreter recursively calls **itself** to evaluate e0

In a big-step semantics, the recursive nature of expression evaluation is exposed via the **interpreter's stack**.

$$(((((e0 \ e1) \ e2) \ e3) \ e4) \ \ldots \ en)$$

A call to evaluate a nested expression will push a frame onto the interpreter's stack for each subexpression.

We often think of the lambda calculus as **small-step**, as each rule transforms an input to an output term.

The Lambda Calculus

lambdas are just anonymous functions!

$$e \in \textbf{Exp} ::= (\lambda \ (x) \ e) \qquad \lambda\text{-abstraction}$$
$$| \ (e \ e) \qquad \text{function application}$$
$$| \ x \qquad \text{variable reference}$$

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

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!

$$e \in \textbf{Exp} ::= (\lambda\ (x)\ e) \qquad \texttt{(lambda (x) e)}$$

$$\mid (e\ e) \qquad \texttt{(e e)}$$

$$\mid x \qquad \texttt{x}$$

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

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!

$$e \in \textbf{Exp} ::= (\lambda \ (x) \ e) \qquad \texttt{(lambda (x) e)}$$
$$\qquad \qquad | \ (e \ e) \qquad \qquad \texttt{(e e)}$$
$$\qquad \qquad | \ x \qquad \qquad \qquad \texttt{x}$$

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

Lambda Abstraction

An expression, *abstracted* over all possible values for a formal parameter, in this case, x.
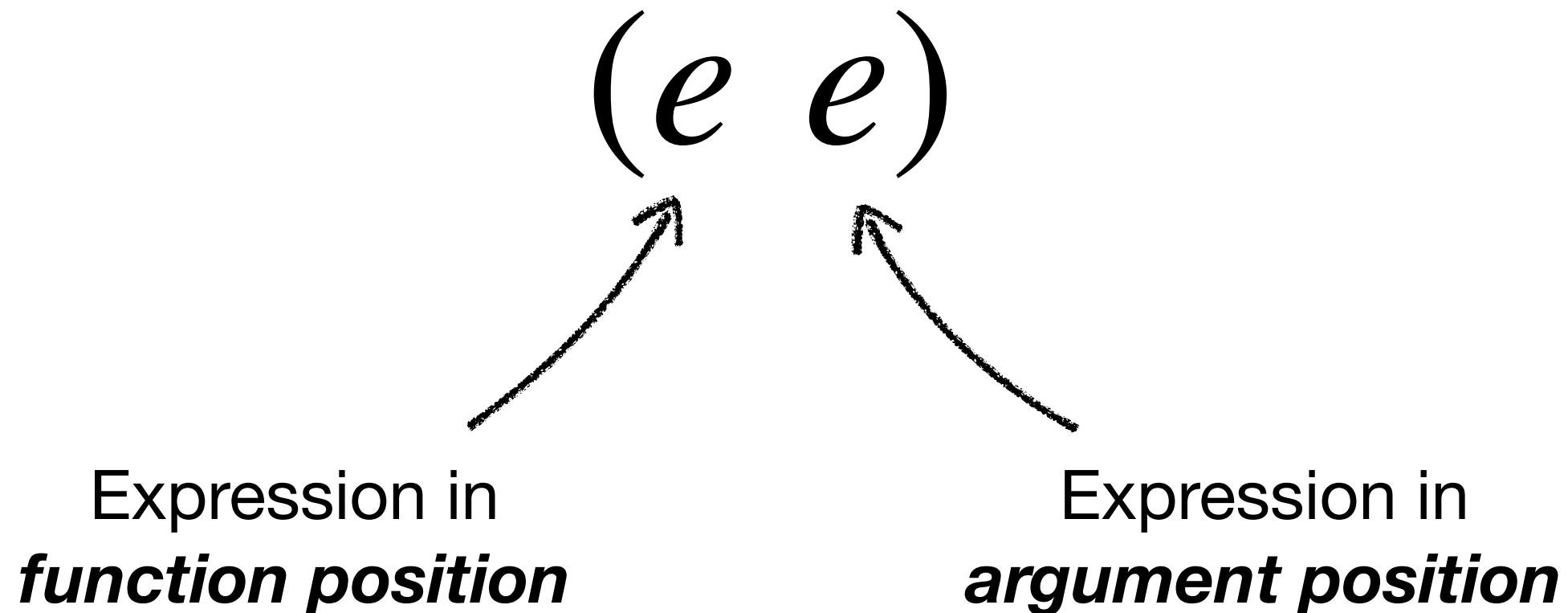
$$(\lambda \ (x) \ e)$$

Formal parameter   Function body

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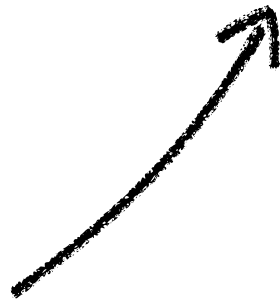
When the first expression is evaluated to a value (in this language, all values are functions!) it may be invoked / applied on its argument.

$$(e \ e)$$

Expression in
function position

Expression in
argument position

Variable

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.

$$((\lambda\ (x)\ E_0)\ E_1) \quad \rightarrow_\beta \quad E_0[x \leftarrow E_1]$$

redex

(**red**ucible **ex**pression)

Note: this is a definition you will want to remember! The term "redex" is frequently used to mean reducible expression in many different types of semantics.

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

$$\downarrow \ \beta$$

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

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

$$\downarrow \ \beta$$

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

Try an example. Can you beta-reduce this term?
Can you beta-reduce it more than once?

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

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

β reduction may continue
indefinitely (i.e., in non-
terminating programs)

$\downarrow$ β

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

$\downarrow$ β

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

$\downarrow$ β

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

$\downarrow$ β

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

$\Big\downarrow\ \beta$

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

This specific program is
known as Ω (Omega)

$\Big\downarrow\ \beta$

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

$\Big\downarrow\ \beta$

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

$\Big\downarrow\ \beta$

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

β

Ω is the smallest non-terminating program!

Note how it reduces to itself in a single step!

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

β

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

β

Evaluation with β reduction is nondeterministic!

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

↓ β

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

Evaluation with β reduction is nondeterministic!

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

or!

β   β

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

β

$$(((\lambda\ (w)\ w)\ (\lambda\ (x)\ x))\ (\lambda\ (z)\ z))$$
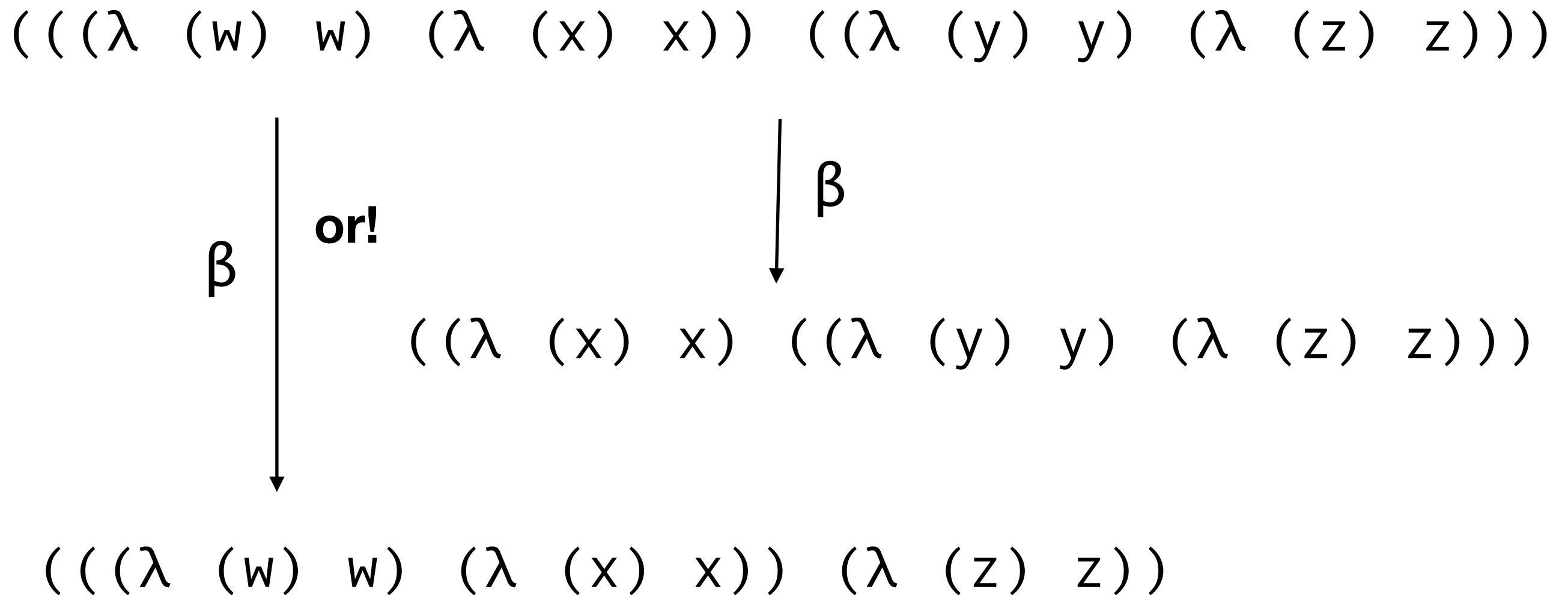
Try an example. Perform each possible β-reduction

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

How many different β-reductions are possible from the above?

Answer

$$((\lambda \ (x) \ \boxed{((\lambda \ (y) \ (x \ y)) \ x)}) \ (\lambda \ (z) \ (z \ z)))$$

β

$$((\lambda \ (x) \ \boxed{(x \ x)}) \ (\lambda \ (z) \ (z \ z)))$$

Can reduce inner redex…

Answer

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

$$\Big\downarrow \beta$$

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

Or the outer redex.

Answer

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

β

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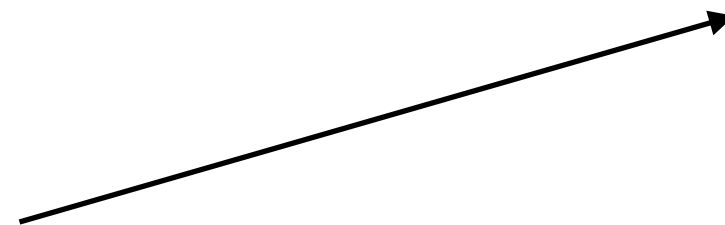
Can't reduce this since we don't (yet) know about
the particular value (function) z in call position.

Free variables

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

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

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

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

Free variables

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

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

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

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

Try an example. What are the free
variables of each of the following terms?

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

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

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

Try an example. 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}

The problem with (naive) textual substitution

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

$$\Big\downarrow \beta$$

The problem with (naive) textual substitution

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

$$\downarrow \beta$$

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

The problem with (naive) textual substitution

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

$$\Big\downarrow \beta$$

$$(\lambda\ (a)\ (\lambda\ (b)\ b))\quad \textcolor{red}{\times}$$

Capture-avoiding substitution

$$E_0[x \leftarrow E_1]$$

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

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

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

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

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

$$\text{where} \quad y \neq x \quad \text{and} \quad y \notin FV(E)$$

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

Capture-avoiding substitution

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

$$\downarrow \beta$$

$$(\lambda \; (a) \; a) \quad \checkmark$$

Try an example. How can you beta-
reduce the following expression using
capture-avoiding substitution?

```
((λ (y)
      ((λ (z) (λ (y) (z y))) y))
 (λ (x) x))
```

Try an example. How can you beta-reduce the following expression using capture-avoiding substitution?

```
((λ (y)
     ((λ (z) (λ (y) (z y))) y))
 (λ (x) x))
```

β

```
((λ (z) (λ (y) (z y))) (λ (x) x))
```

Try an example. How can you beta-
reduce the following expression using
capture-avoiding substitution?

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

Try an example. How can you beta-reduce the following expression using capture-avoiding substitution?

(λ (y) ((λ (z) (λ (y) z)) (λ (x) y)))

You cannot! This redex would require:

(λ (y) z)[z ← (λ (x) y)]

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

Try an example. How can you beta-reduce the following expression using capture-avoiding substitution?
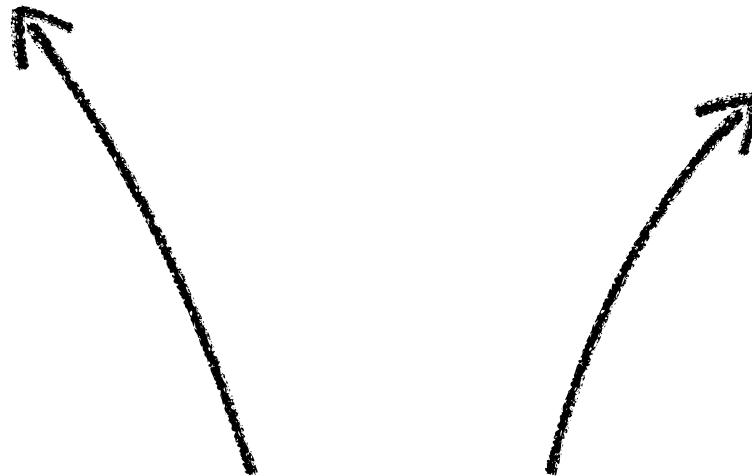
(λ (y) ((λ (z) (λ (y) z)) (λ (x) y)))

→α (λ (y) ((λ (z) (λ (w) z)) (λ (x) y)))

→β (λ (y) (λ (w) (λ (x) y)))

Instead we alpha-convert first.

α - renaming

(λ (x) (λ (y) x))   (λ (a) (λ (b) a))

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

α-renaming

$$(\lambda\ (x)\ E_0) \qquad \rightarrow_\alpha \qquad (\lambda\ (y)\ E_0[x \leftarrow y])$$

$=_\alpha$

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

α - renaming

α renaming/conversions can be used to implement capture-avoiding substitution

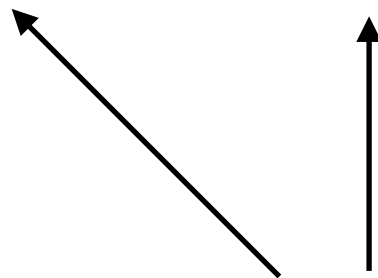Rename variables that would break naive substitution!

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

α - renaming

α 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.

α - renaming

α 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 α renaming to z

α - renaming

α 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 α renaming to z

α - renaming

α 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

η-reduction

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

η-expansion

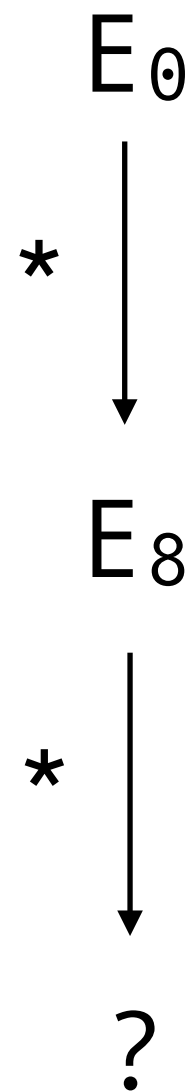$$E_0 \quad \longrightarrow_\eta \quad (\lambda\ (x)\ (E_0\ x))\ \text{ where } x \notin FV(E_0)$$

Reduction

$$(\rightarrow) = (\rightarrow_\beta) \cup (\rightarrow_\alpha) \cup (\rightarrow_\eta)$$
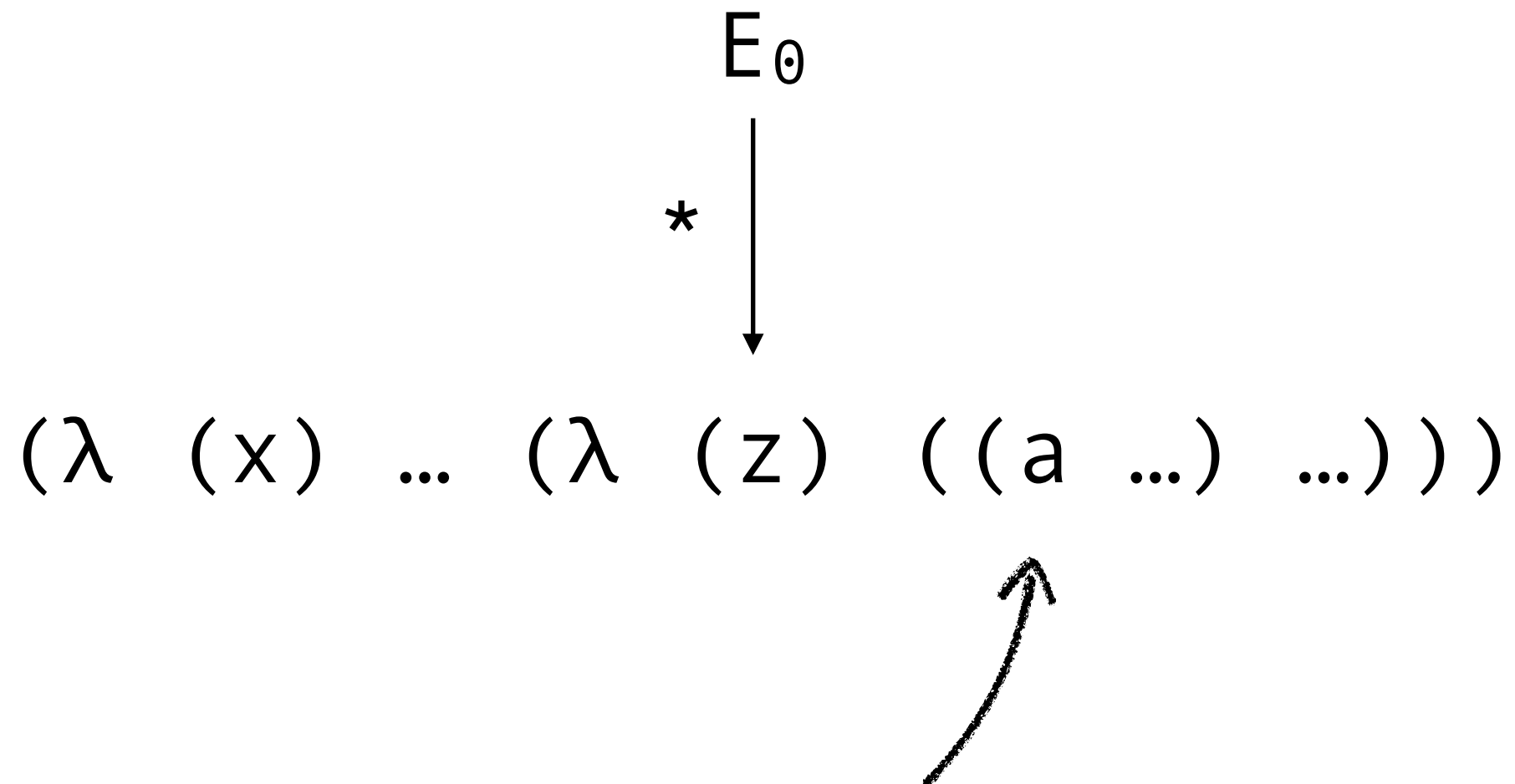
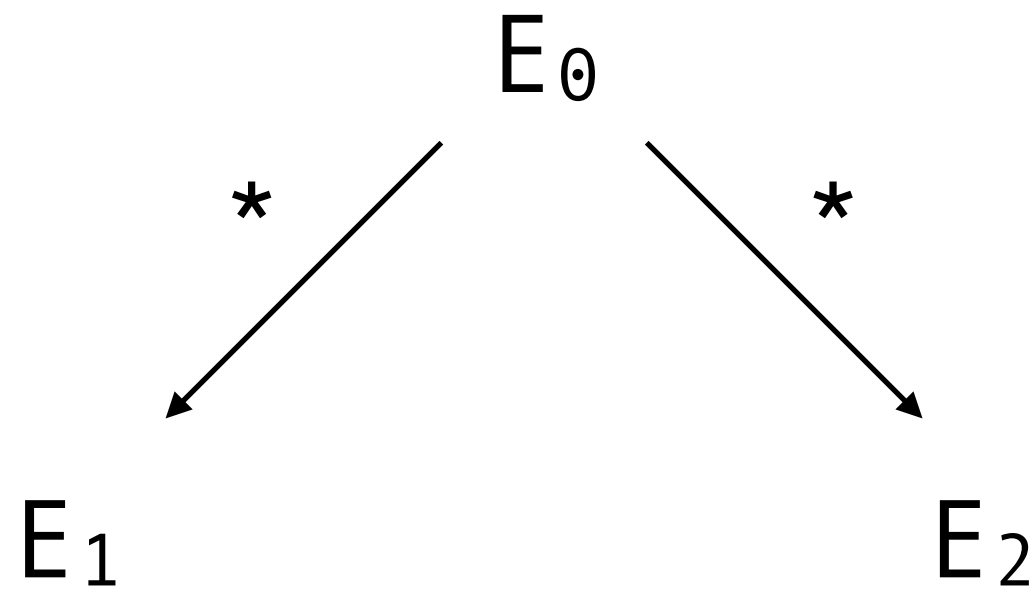$$(\rightarrow^*)$$

reflexive/transitive closure

Evaluation

$$E_0$$

$$\Big\downarrow *$$

$$E_8$$

$$\Big\downarrow *$$

$$?$$

Evaluation to *normal form*

$$E_0$$

*

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

Evaluation to *normal form*

$$E_0$$

$$*$$

$$(\lambda \ (x) \ ... \ (\lambda \ (z) \ ((a \ ...) \ ...)))$$

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

Evaluation Strategy

$$E_0$$

$*$ $*$

$$E_1 \qquad\qquad E_2$$

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

$((\lambda\ (x)\ \boxed{((\lambda\ (y)\ y)\ x)})\ (\lambda\ (z)\ z))$
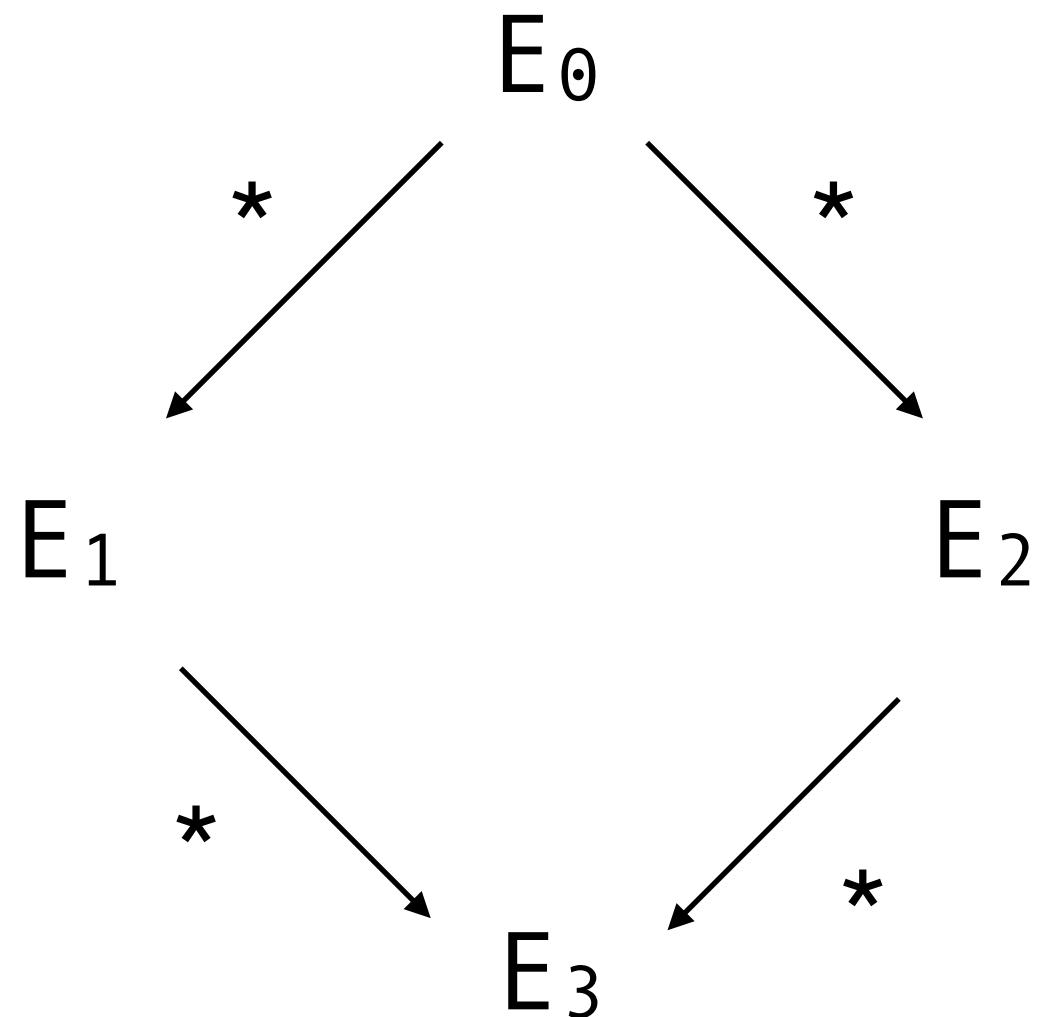
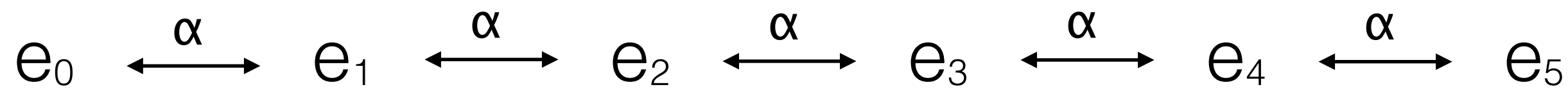$\to_\beta\ ((\lambda\ (x)\ x)\ (\lambda\ (z)\ z))$

$\to_\beta\ (\lambda\ (z)\ z)$

Confluence

Diverging paths of evaluation must eventually join back together.

E_0

$*$ $*$

Church-Rosser Theorem
If—starting from E_0—we can reach *both* E_1 and E_2, then we **must** be able to get to some E_3 starting from *either* E_2 or E_1.

E_1 E_2

$*$ $*$

E_3

$$e_0 \xleftrightarrow{\alpha} e_1 \xleftrightarrow{\alpha} e_2 \xleftrightarrow{\alpha} e_3 \xleftrightarrow{\alpha} e_4 \xleftrightarrow{\alpha} e_5$$

$$e_0 \xleftrightarrow{\alpha} e_1 \xleftrightarrow{\alpha} e_2 \xleftrightarrow{\alpha} e_3 \xleftrightarrow{\alpha} e_4 \xleftrightarrow{\alpha} e_5$$

$$e_1 \xrightarrow{\beta} e_6 \xleftarrow{\eta} e_2$$

$e_0 \xleftrightarrow{\alpha} e_1 \xleftrightarrow{\alpha} e_2 \xleftrightarrow{\alpha} e_3 \xleftrightarrow{\alpha} e_4 \xleftrightarrow{\alpha} e_5$

$e_1 \xrightarrow{\beta} e_6$

$e_2 \xrightarrow{\eta} e_6$

$e_6 \xrightarrow{\beta}$

$e_6 \xrightarrow{\beta}$

$e_6 \xrightarrow{\beta}$

$$e_0 \xleftrightarrow{\alpha} e_1 \xleftrightarrow{\alpha} e_2 \xleftrightarrow{\alpha} e_3 \xleftrightarrow{\alpha} e_4 \xleftrightarrow{\alpha} e_5$$

$$e_0 \xrightarrow{\beta} e_7$$

$$e_7 \xrightarrow{\beta} e_8$$

$$e_8 \xrightarrow{\beta} e_9$$

$$e_2 \xrightarrow{\beta} e_{10}$$

$$e_{10} \xrightarrow{\beta} e_{11}$$

$$e_0 \xleftrightarrow{\alpha} e_1 \xleftrightarrow{\alpha} e_2 \xleftrightarrow{\alpha} e_3 \xleftrightarrow{\alpha} e_4 \xleftrightarrow{\alpha} e_5$$

$e_0 \xrightarrow{\beta} e_7$

$e_7 \xrightarrow{\beta} e_8$

$e_8 \xrightarrow{\beta} e_9$

$e_2 \xrightarrow{\beta} e_{10}$

$e_{10} \xrightarrow{\beta} e_{11}$

$e_{11} \xrightarrow{*} e_{12}$

$e_9 \xrightarrow{*} e_{12}$
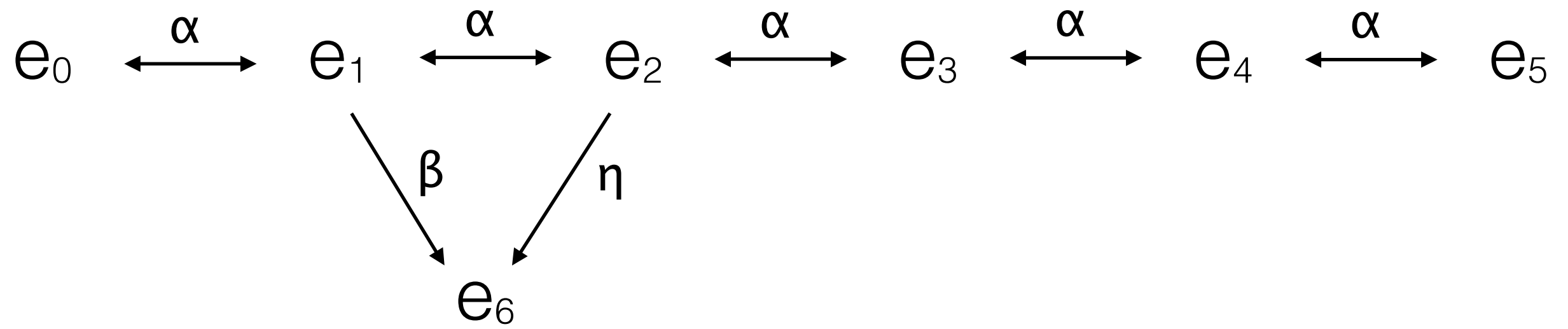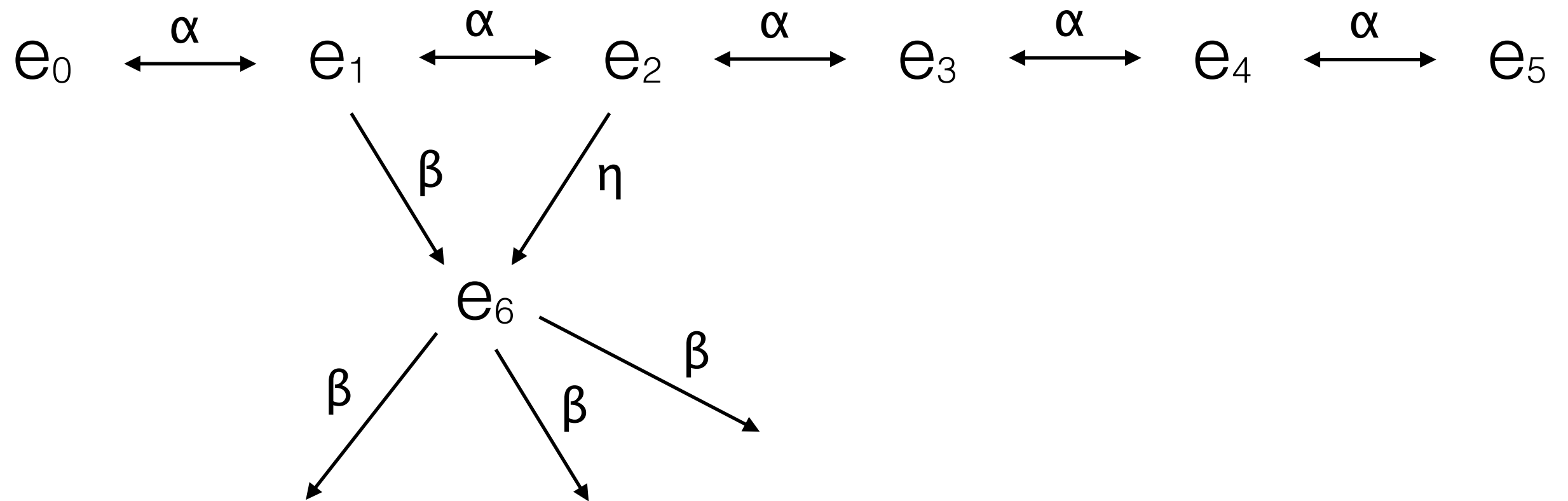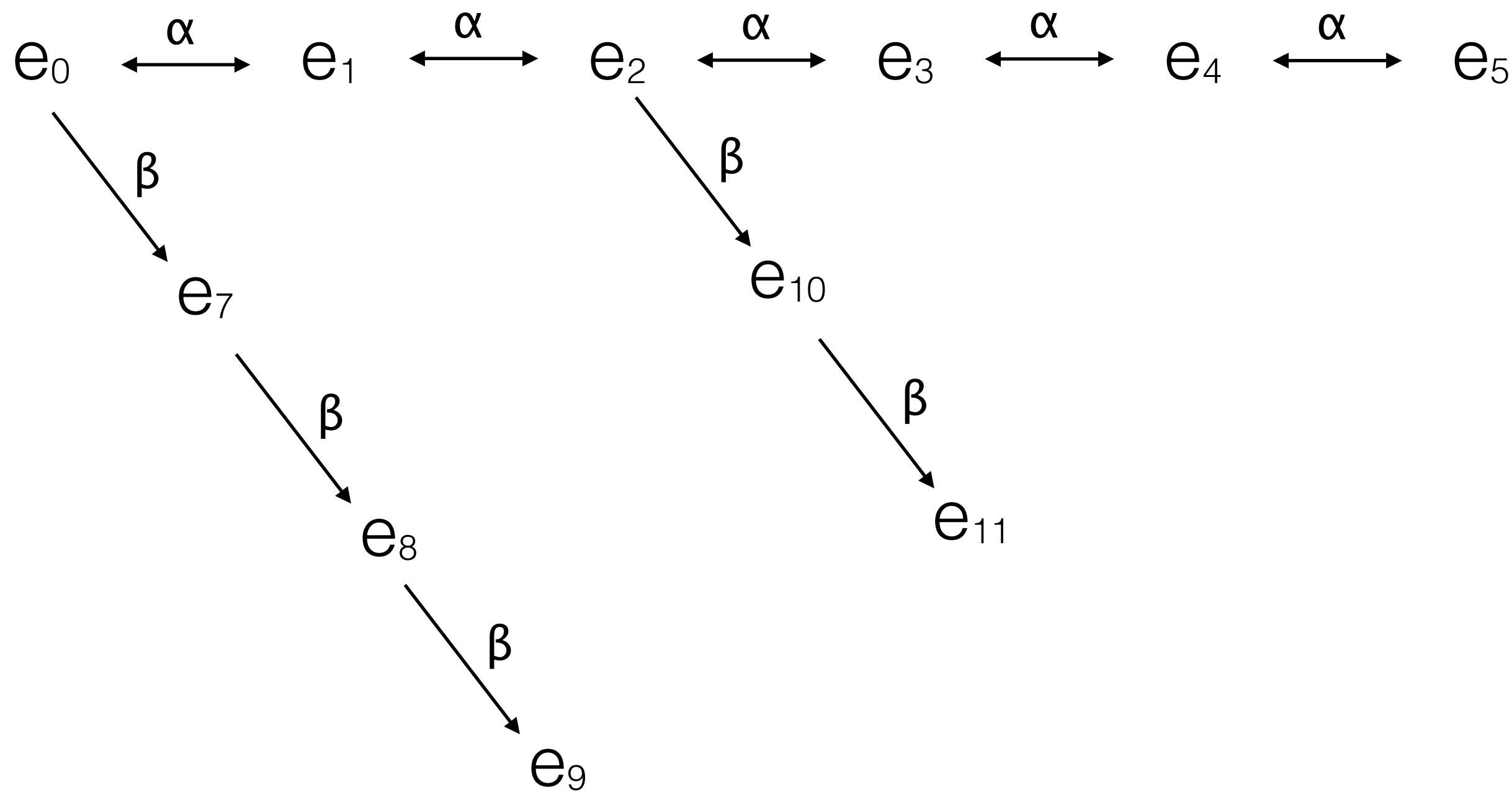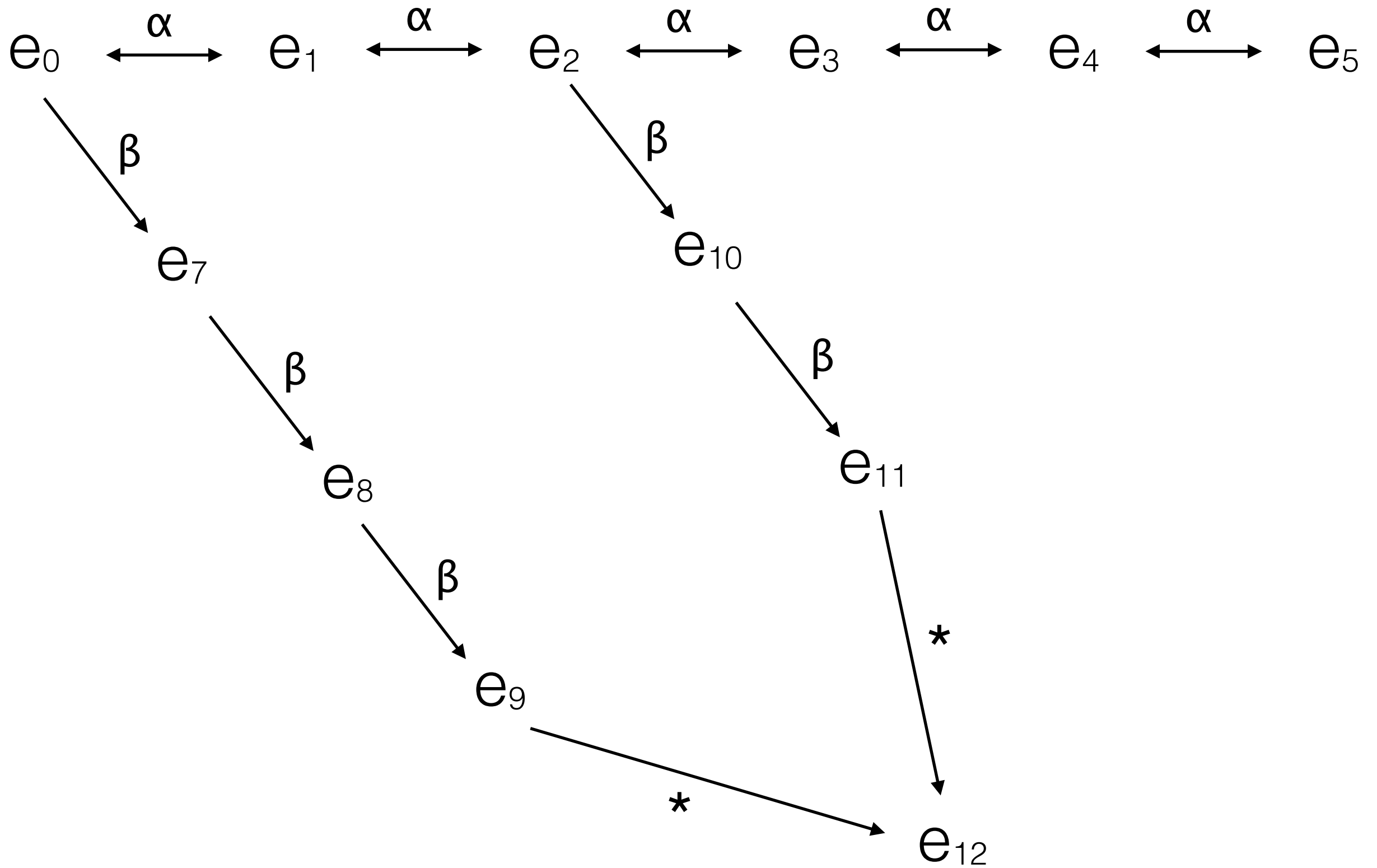
Applicative evaluation order

Always evaluates the *innermost* leftmost redex first.

Normal evaluation order

Always evaluates the *outermost* leftmost redex first.

Applicative evaluation order

$$((\lambda\ (x)\ \boxed{((\lambda\ (y)\ y)\ x)})\ (\lambda\ (z)\ z))$$

Normal evaluation order

$$\boxed{(((\lambda\ (x)\ ((\lambda\ (y)\ y)\ x))\ (\lambda\ (z)\ z))\ (\lambda\ (w)\ w))}$$

Call-by-value (CBV) semantics

Applicative evaluation order, *but not under lambdas.*

Call-by-name (CBN) semantics

Normal evaluation order, *but not under lambdas.*

Church encoding

Church encoding is the process of encoding all values as lambda abstractions. E.g., Church numerals are an encoding of numbers, 0, 1, 2, …, as first-class functions. Church booleans are an encoding of #t and #f as functions. Church lists are an encoding of lists (pairs and null) as functions.

Church encoding

Church encoding is the process of encoding all values as lambda abstractions. E.g., Church numerals are an encoding of numbers, 0, 1, 2, …, as first-class functions. Church booleans are an encoding of #t and #f as functions. Church lists are an encoding of lists (pairs and null) as functions.

"If I only let you use the lambda calculus, can you still write normal programs (e.g., ones that use recursion/+/if/etc…)?"

YES!

Church encoding

Church encoding is the process of encoding all values as lambda abstractions. E.g., Church numerals are an encoding of numbers, 0, 1, 2, …, as first-class functions. Church booleans are an encoding of #t and #f as functions. Church lists are an encoding of lists (pairs and null) as functions.
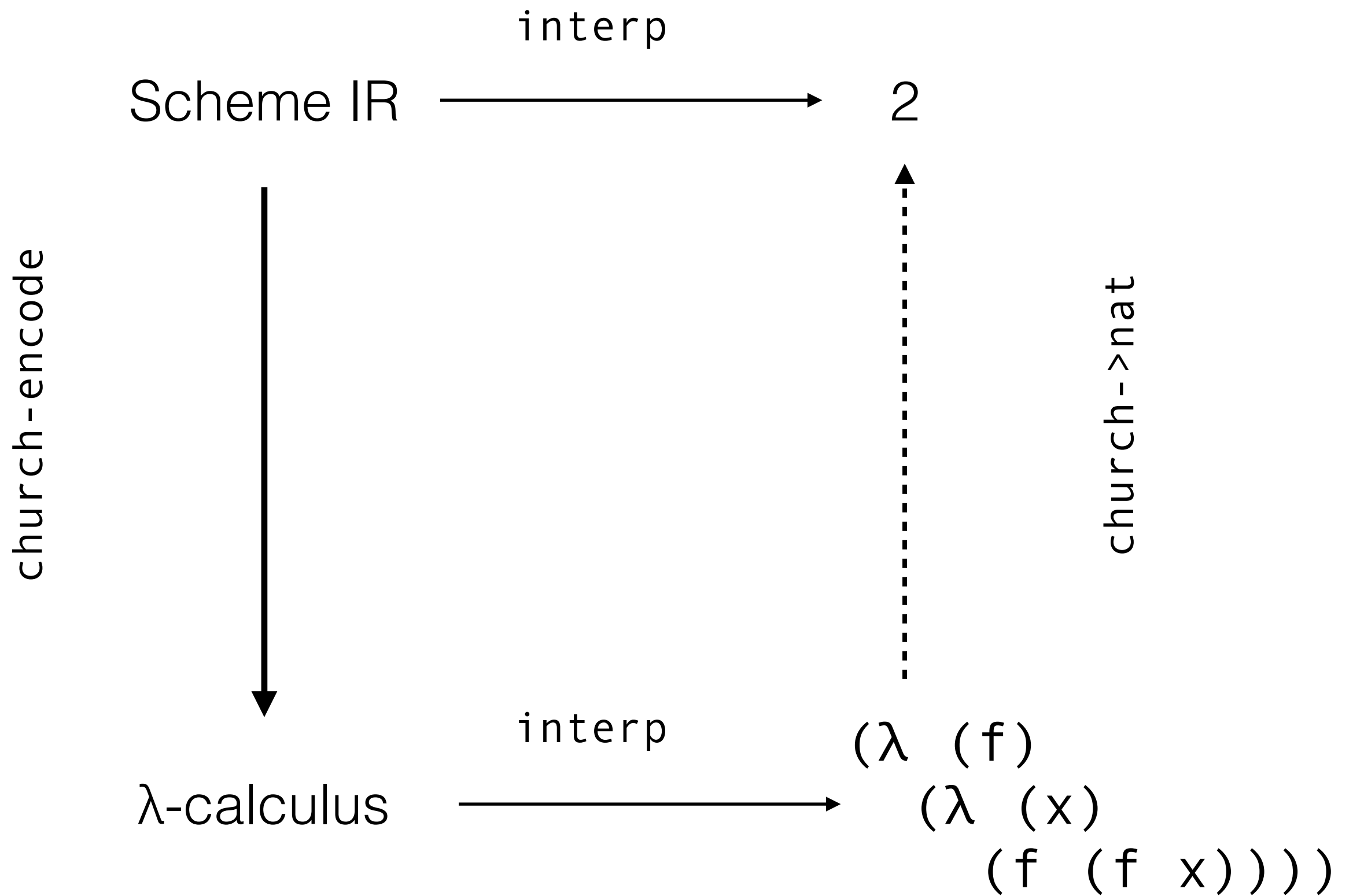
We will start from core scheme…

```
e ::= (letrec ([x (lambda (x …) e)]))
    | (let ([x e] …) e)
    | (lambda (x …) e)
    | (e e …)
    | x
    | (if e e e)
    | (prim e e) | (prim e)
    | d
d ::= ℕ | #t | #f | '()
x ::= <vars>
prim ::= + | - | * | not | cons | …
```

```
e ::= (lambda (x) e)
    | (e e)
    | x
```

Scheme IR $\xrightarrow{\texttt{interp}}$ 2

church-encode

church->nat

λ-calculus $\xrightarrow{\texttt{interp}}$ (λ (f)
(λ (x)
(f (f x))))

Today, we'll start with:

```
e ::= (letrec ([x (lambda (x …) e)]))
    | (let ([x e] …) e)
    | (lambda (x …) e)
    | (e e …)
    | x
    | (if e e e)
    | (+ e e) | (* e e)
    | (cons e e) | (car e) | (cdr e)
    | d
d ::= ℕ | #t | #f | '()
x ::= <vars>
```

Desugaring Let

```
(let ([x e] …) ebody)
```

```
(let ([x e] …) ebody)
```

↓

```
((λ (x …) ebody) e …)
```

Currying

$$(\lambda\ (x\ y\ z)\ e) \longrightarrow \begin{array}{l} (\lambda\ (x) \\ \quad (\lambda\ (y) \\ \quad\quad (\lambda\ (z)\ e))) \end{array}$$

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

$$(\lambda\ ()\ e) \longrightarrow (\lambda\ (\_)\ e)$$

$$(f\ a\ b\ c\ d) \longrightarrow ((((f\ a)\ b)\ c)\ d)$$

$$(f\ a) \longrightarrow (f\ a)$$

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

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
    | (if e e e)
    | ((+ e) e) | ((* e) e)
    | ((cons e) e) | (car e) | (cdr e)
    | d
d ::= ℕ | #t | #f | ‘()
x ::= <vars>
```

Conditionals & Booleans

$$(\text{if } \#t \ e_T \ e_F)$$

$$\downarrow$$

$$e_T$$

$$(\text{if } \#f \ e_T \ e_F)$$

$$\downarrow$$

$$e_F$$

$$((\lambda\ (t\ f)\ t)\ e_T\ e_F) \qquad ((\lambda\ (t\ f)\ f)\ e_T\ e_F)$$

$$\downarrow \qquad\qquad\qquad\qquad\qquad \downarrow$$

$$((\lambda\ (t\ f)\ t)\ v_T\ v_F) \qquad ((\lambda\ (t\ f)\ f)\ v_T\ v_F)$$

$$\downarrow \qquad\qquad\qquad\qquad\qquad \downarrow$$

$$v_T \qquad\qquad\qquad\qquad\qquad\qquad v_F$$

What issues arise with this encoding?

$$((\lambda \ (t \ f) \ t) \ e_T \ \Omega)$$

$$\downarrow$$

$$\ldots\ldots\ldots$$

$$((\lambda\ (t\ f)\ (t))\ (\lambda\ ()\ e_T)\ (\lambda\ ()\ \Omega))$$

$$\downarrow$$

$$((\lambda\ ()\ e_T))$$

$$\downarrow$$

$$e_T$$

$$\downarrow$$

$$v_T$$

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
    | ((+ e) e) | ((* e) e)
    | ((cons e) e) | (car e) | (cdr e)
    | d
d ::= ℕ | ‘()
x ::= <vars>
```

Natural Numbers

Hint: turn all nouns into verbs!

(Focus on the *behaviors* that are implicit in values.)

$$(\lambda\ (f)\ (\lambda\ (x)\ (f^N\ x)))$$

0: $(\lambda\ (f)\ (\lambda\ (x)\ x))$

1: $(\lambda\ (f)\ (\lambda\ (x)\ (f\ x)))$

2: $(\lambda\ (f)\ (\lambda\ (x)\ (f\ (f\ x))))$

3: $(\lambda\ (f)\ (\lambda\ (x)\ (f\ (f\ (f\ x)))))$

```
church+ = (λ (n) (λ (m)
            (λ (f) (λ (x)
             …))))
```

```
church+ = (λ (n) (λ (m)
           (λ (f) (λ (x)
             ((n f) ((m f) x)))))))
```

```
church* = (λ (n) (λ (m)
             (λ (f) (λ (x)
               …))))
```

```
church* = (λ (n) (λ (m)
           (λ (f) (λ (x)
             ((n (m f)) x)))))
```

$$f N^M = f N*M$$

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
    | ((cons e) e) | (car e) | (cdr e)
    | d
d ::= '()
x ::= <vars>
```

Lists

The fundamental problem:

We need to be able to case-split.

The solution:

We take two callbacks as with `#t, #f`!

```
'() = (λ (when-cons) (λ (when-null)
        (when-null)))



(cons a b) = (λ (when-cons) (λ (when-null)
              (when-cons a b)))
```

Try an Example. How can we define null?

Try an Example. How can we define null?

```
church:null? = (λ (p)
                  (p (λ (a b) #f)
                     (λ () #t)))
```

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
x ::= <vars>
```

Ω

((λ (x) (x x)) (λ (x) (x x)))

Key: U takes a function and calls it on itself

Ω

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

U

(define U (λ (f) (f f)))

```
(letrec ([fib (lambda (x) (if (= x 0) 1 (* x (fib (- x 1)))))])
  (fib 3))


(let ([fib (U (lambda (f)
               (lambda (x) (if (= x 0) 1 (* x (... (- x 1)))))))])
  (fib 3))
```

What can I type right here to make fib work?

(Hint: the answer can be written in 5 characters)

```
(define U (λ (f) (f f)))
```

```
(letrec ([fib (lambda (x) (if (= x 0) 1 (* x (fib (- x 1)))))])
  (fib 3))
```

```
(let ([fib (U (lambda (f)
              (lambda (x) (if (= x 0) 1 (* x (... (- x 1))))))])
  (fib 3))
```

What can I type right here to make fib work?

(f f)

Y combinator

```
(letrec ([fact (λ (n)
                (if (= n 0)
                    1
                    (* n (fact (- n 1)))))])
  (fact 5))
```

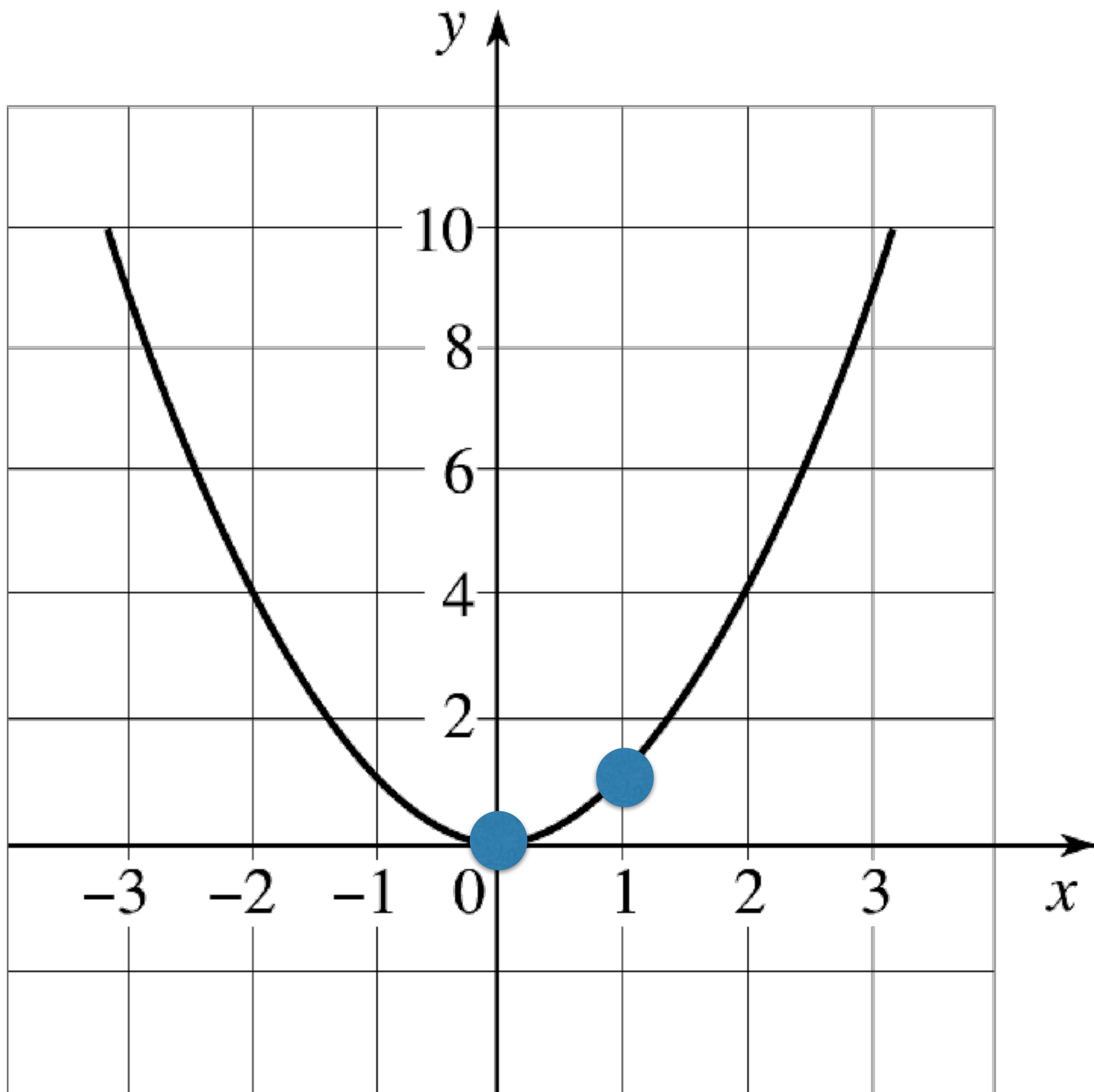**Key idea: instead of**

```
(let ([mk (λ (mk) (λ (n)
                    (if (= n 0)
                        1
                        (* n ((mk mk) (- n 1)))))))])
  ((mk mk) 5))
```

Y

$$(Y\ f) = f\ (Y\ f)$$

(It's a fixed-point combinator!)

Three step process for deriving Y

```
(Y f) = f (Y f)


Y = (λ (f) (f (Y f)))        1. Treat as definition


mY = (λ (mY)
       (λ (f)                    2. Lift to mk-Y,
         (f ((mY mY) f)))))   use self-application


mY = (λ (mY)
       (λ (f)                   3. Eta-expand
         (f (λ (x) (((mY mY) f) x)))))
```

U-combinator: (U U) **is Omega**

$\downarrow$

```
Y  =   (U  (λ  (y)  (λ  (f)
           (f  (λ  (x)  (((y  y)  f)  x)))))) 
```

```
          ↓
(let ([fact (Y (λ (fact) (λ (n)
                (if (= n 0)
                    1
                    (* n (fact (- n 1))))))])
  (fact 5))
```

Try an example!!!

```
(define Y ((λ (x) (x x)) (λ (y) (λ (f)
                              (f (λ (x) (((y y) f) x)))))))
```

```
(define (fib x)
  (if (or (= x 0) (= x 1))
      1
      (+ (fib (- x 1)) (fib (- x 2))))))
```

Rewrite this to use the Y combinator instead

```
e ::= (lambda (x) e)
    | (e e)
    | x
```

De-churching

```
(define (church->nat cv)
                              )



(define (church->list cv)




                              )




(define (church->bool cv)


                              )
```

De-churching

```
(define (church->nat cv)
        ((cv add1) 0))


(define (church->list cv)




                              )




(define (church->bool cv)



                    )
```

De-churching

```scheme
(define (church->nat cv)
        ((cv add1) 0))


(define (church->list cv)
        ((cv (λ (car)
                (λ (cdr)
                    (cons car
                          (church->list cdr)))))
         (λ (na) ‘())))


(define (church->bool cv)


        )
```

De-churching

```
(define (church->nat cv)
        ((cv add1) 0))


(define (church->list cv)
        ((cv (λ (car)
                (λ (cdr)
                   (cons car
                         (church->list cdr)))))
         (λ (na) '())))


(define (church->bool cv)
        ((cv (λ () #t))
         (λ () #f)))
```

```
(letrec ([map (λ (f lst)
                (if (null? lst)
                    '()
                    (cons (f (car lst))
                          (map f (cdr lst)))))])

  (map (λ (x) (+ 1 x))
       '(0 5 3)))
```

```
(define lst
  (((((((((λ (Y-comb)
           (λ (church:null?)
             (λ (church:cons)
               (λ (church:car)
                 (λ (church:cdr)
                   (λ (church:+)
                     (λ (church:*)
                       (λ (church:not)
                         ((λ (map)
                            ((map
                              (λ (x)
                                ((church:+ (λ (f) (λ (x) (f x))))
                                 x)))
                             ((church:cons (λ (f) (λ (x) x)))
                              ((church:cons
                                (λ (f)
                                  (λ (x) (f (f (f (f (f x))))))))
                               ((church:cons
                                 (λ (f) (λ (x) (f (f (f x)))))
                                 (λ (when-cons)
                                   (λ (when-null)
                                     (when-null (λ (x) x)))))))))))
                          (Y-comb
                           (λ (map)
```

> (map church->nat (church->list lst))
'(1 6 4)

Try an example.

Write a lambda term other than Ω which also does not terminate

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

Write a lambda term other than Ω which also does not terminate

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

$$((\lambda \ (u) \ ((u \ u) \ u))$$
$$(\lambda \ (u) \ ((u \ u) \ u)))$$

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

Evaluation contexts

Restrict the order in which we may simplify a program's redexes

$$\mathscr{E} ::= (\mathscr{E}\ e)$$
$$\mid\ (v\ \mathscr{E})$$
$$\mid\ \Box$$

(left-to-right) CBV evaluation

$$v ::= (\lambda\ (x)\ e)$$

$$e ::= (\lambda\ (x)\ e)$$
$$\mid\ (e\ e)$$
$$\mid\ x$$

$$\mathscr{E} ::= (\mathscr{E}\ e)$$
$$\mid\ \Box$$

(left-to-right) CBN evaluation

Context and redex

For CBV a redex must be (v v)
For CVN, a redex must be (v e)

r

$\mathscr{E}$ [(v v)] =

(((λ (x) ((λ (y) y) x)) (λ (z) z)) (λ (w) w))

$\mathscr{E}$ = (□ (λ (w) w))

r = ((λ (x) ((λ (y) y) x)) (λ (z) z))

Context and redex

$$\mathcal{E}[\ r\ ]\ =$$

(((λ (x) ((λ (y) y) x)) (λ (z) z)) (λ (w) w))

$$\mathcal{E}\ =\ (\square\ (λ\ (w)\ w))$$

r = ((λ (x) ((λ (y) y) x)) (λ (z) z))

$$\longrightarrow_β\ ((λ\ (y)\ y)\ (λ\ (z)\ z))$$

Put the reduced redex back in its evaluation context…

$\mathscr{E}$ = (□ (λ (w) w))

r = ((λ (x) ((λ (y) y) x)) (λ (z) z))

→β ((λ (y) y) (λ (z) z))

$\mathscr{E}$[r]

(((λ (y) y) (λ (z) z)) (λ (w) w))

Exercises—can you evaluate…

1) (((λ (y) y) (λ (z) z)) (λ (w) w))

2) ((λ (u) (u u)) (λ (x) (λ (x) x)))

3) (((λ (x) x) (λ (y) y))
   ((λ (u) (u u)) (λ (z) (z z))))