QuasiPatterns

We can also use quasiquoting in a match pattern
We call this a quasi*pattern*

It turns out this lets us build an
implementation of a **little language**!

```
(define (interpret-binary-arith e)
  (match e
    [`(+ ,e1 ,e2) (+ (interpret-binary-arith e1)
                     (interpret-binary-arith e2))]
    [`(- ,e1 ,e2) (- (interpret-binary-arith e1)
                     (interpret-binary-arith e2))]
    [(? number? n) n]
    [else (error “bad expression…”)]))
```

Exercise: call `interpret-binary-arith` on the following…

```
3            (+ 2 3)

(+ (- (+ 2 3) 5) (+ 1 (- 2 3)))
```

Quiz

What's the difference between the following two expressions?

```
(interpret-binary-arith
  (+ (- (+ 2 3) 5) (+ 1 (- 2 3))))

(interpret-binary-arith
  '(+ (- (+ 2 3) 5) (+ 1 (- 2 3))))
```

Answer: in one we're cheating. We're not really using our interpreter, we're just using Racket

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!

$e \in \textbf{Exp} ::= (\lambda\ (x)\ e)$	λ-abstraction
$\mid\ (e\ e)$	function application
$\mid\ x$	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

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

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

- 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.:

$$\frac{}{a_0 \times a_1 \Rightarrow n_0 * n_1}$$ **where** a_0 is n_0 and a_1 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!

$$
\begin{aligned}
e \in \textbf{Exp} &::= (\lambda\ (x)\ e) && \text{λ-abstraction} \\
&\ \mid\ (e\ e) && \text{function application} \\
&\ \mid\ x && \text{variable reference}
\end{aligned}
$$

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

The Lambda Calculus

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

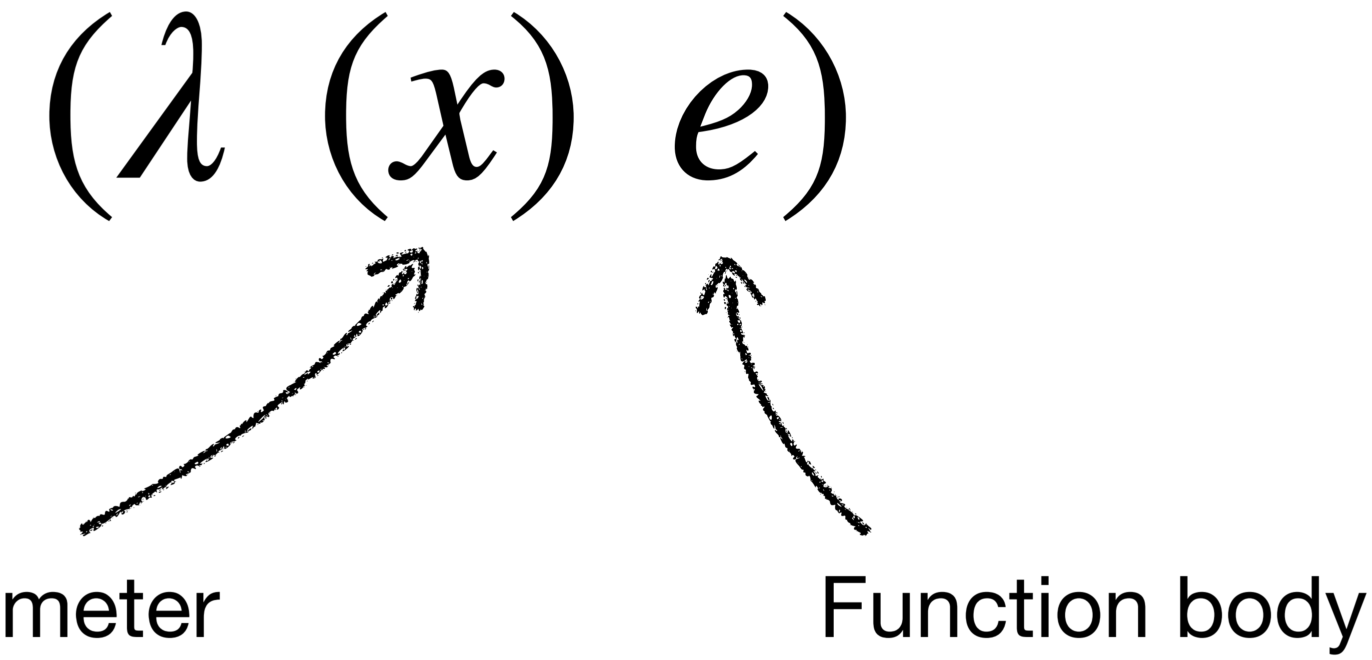
Just the following subset of Racket is Turing-equivalent!

$e \in \textbf{Exp} ::= (\lambda\ (x)\ e)$	`(lambda (x) e)`
$\mid (e\ e)$	`(e e)`
$\mid x$	`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.



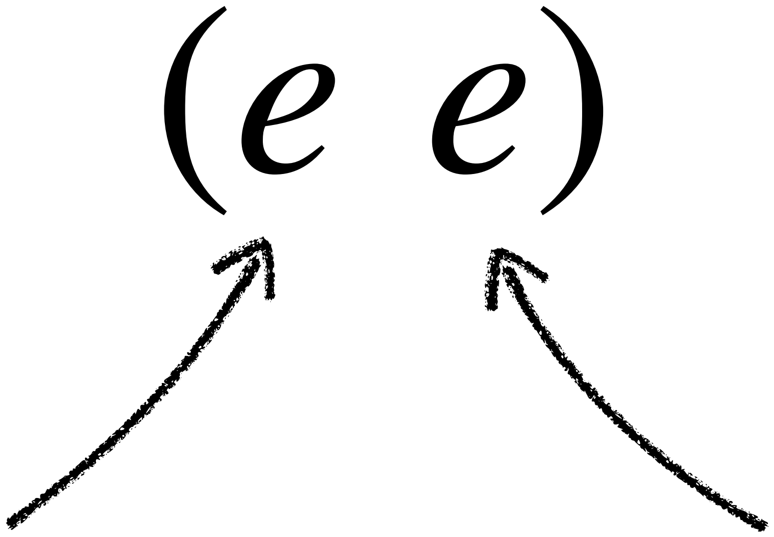
$$(\lambda\ (x)\ e)$$

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

```
((λ (f) (f (f (λ (x) x)))) (λ (x) x))
```

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

$$\Big\downarrow \beta$$

```
((λ (x) x) ((λ (x) x) (λ (x) x)))
```

$$\Big\downarrow \beta$$

```
((λ (x) x) (λ (x) x))
```

$$\Big\downarrow \beta$$

```
(λ (x) x)
```

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

$$\underbrace{((\lambda\ (x)\ E_0)\ E_1)}_{\text{redex}} \quad \to_\beta \quad E_0[x \leftarrow E_1]$$

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

$$\begin{tikzcd}
\texttt{((}\lambda\texttt{ (x) x) (}\lambda\texttt{ (x) x))} \arrow[d, "\beta"] \\
\texttt{x[x} \leftarrow \texttt{(}\lambda\texttt{ (x) x)]}
\end{tikzcd}$$

$$\begin{tikzcd}
\texttt{((}\lambda\texttt{ (x) x) (}\lambda\texttt{ (x) x))} \arrow[d, "\beta"] \\
\texttt{(}\lambda\texttt{ (x) x)}
\end{tikzcd}$$

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

`((λ (x) (x x)) (λ (x) (x x)))`

`((λ (x) (x x)) (λ (x) (x x)))`

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

↓ β

`((λ (x) (x x)) (λ (x) (x x)))`

↓ β

`((λ (x) (x x)) (λ (x) (x x)))`

↓ β

`((λ (x) (x x)) (λ (x) (x x)))`

↓ β

```
((λ (x) (x x)) (λ (x) (x x)))
```

$$\Big\downarrow \beta$$

```
((λ (x) (x x)) (λ (x) (x x)))
```

$$\Big\downarrow \beta$$

This specific program is known as Ω (Omega)

```
((λ (x) (x x)) (λ (x) (x x)))
```

$$\Big\downarrow \beta$$

```
((λ (x) (x x)) (λ (x) (x x)))
```

$$\Big| \beta$$

```
((λ (x) (x x)) (λ (x) (x x)))
```

β

Ω is the smallest non-terminating program!

```
((λ (x) (x x)) (λ (x) (x x)))
```

β

Note how it reduces to itself in a single step!

```
((λ (x) (x x)) (λ (x) (x x)))
```

β

```
((λ (x) (x x)) (λ (x) (x x)))
```

β

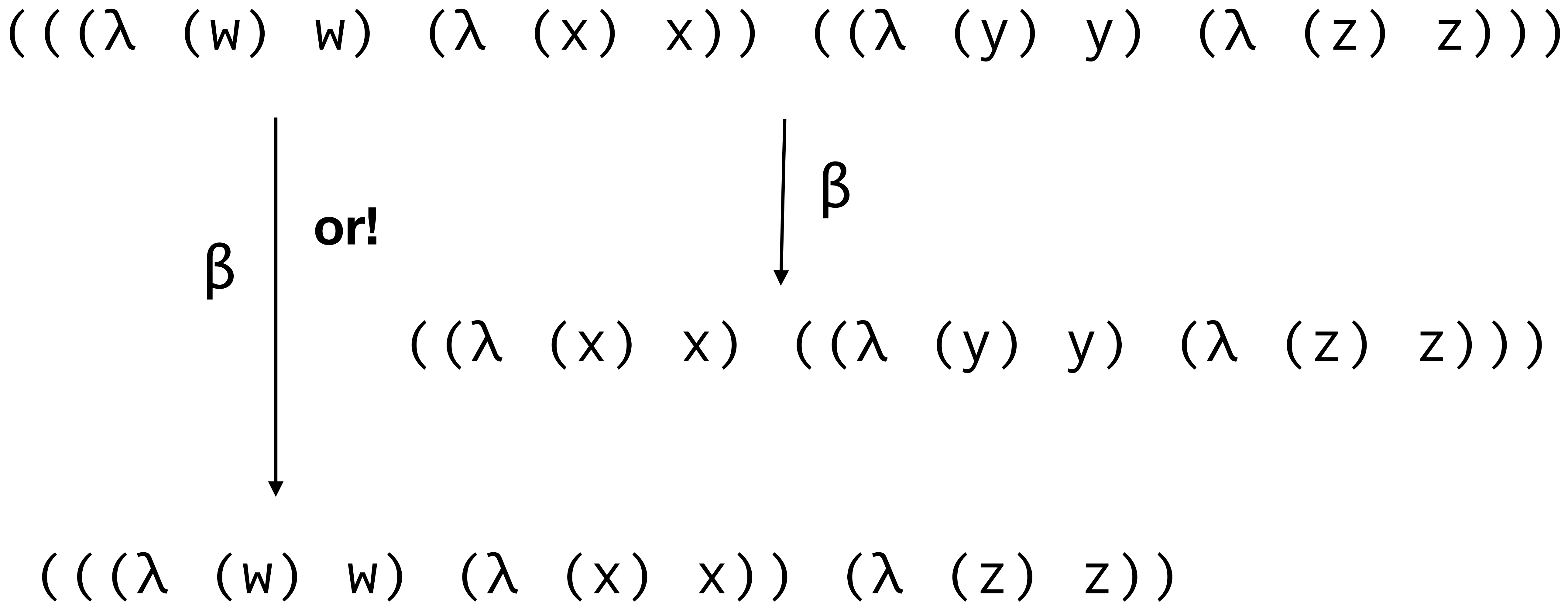
Evaluation with β reduction is nondeterministic!

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

$\big\downarrow \beta$

`((λ (x) x) ((λ (y) y) (λ (z) z)))`

Evaluation with β reduction is nondeterministic!



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

β

or!

β

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

```
(((λ (w) w) (λ (x) x)) (λ (z) z))
```

Try an example. Perform each possible β-reduction

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

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

Answer

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

$$\Big\downarrow \beta$$

`((λ (x) (x x)) (λ (z) (z z)))`

Can reduce inner redex…

Answer

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

$$\Big\downarrow \beta$$

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

Or the outer redex.

Answer

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

$$\Big\downarrow \beta$$

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

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

Free variables

$$\textbf{FV} : \textbf{Exp} \to \mathscr{P}(\textbf{Var})$$

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

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

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

Free variables

FV(`(x y)`) = {x, y}

FV(`((λ (x) x) y)`) = {y}

FV(`((λ (x) x) x)`) = {x}

FV(`((λ (y) ((λ (x) (z x)) x)))`) = {z, x}

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

```
((λ (x) x) y)
```

```
((λ (x) (x x)) (λ (x) (x x)))
```

```
((λ (x) (z y)) x)
```

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

`((λ (x) x) y)`

{y}

`((λ (x) (x x)) (λ (x) (x x)))`

{}

`((λ (x) (z y)) x)`

{x, y, z}

The problem with (naive) textual substitution

```
((λ (a) (λ (a) a)) (λ (b) b))
```

$$\Big\downarrow \beta$$

The problem with (naive) textual substitution

$$\begin{tikzcd}
\texttt{((}\lambda\texttt{ (a) (}\lambda\texttt{ (a) a)) (}\lambda\texttt{ (b) b))} \arrow[d, "\beta"] \\
\texttt{(}\lambda\texttt{ (a) a)[a } \leftarrow \texttt{ (}\lambda\texttt{ (b) b)]}
\end{tikzcd}$$

The problem with (naive) textual substitution

```
((λ (a) (λ (a) a)) (λ (b) b))
```

$$\Big\downarrow \beta$$

```
(λ (a) (λ (b) b))
```

✗

Capture-avoiding substitution

$E_0[x \leftarrow E_1]$

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

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

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

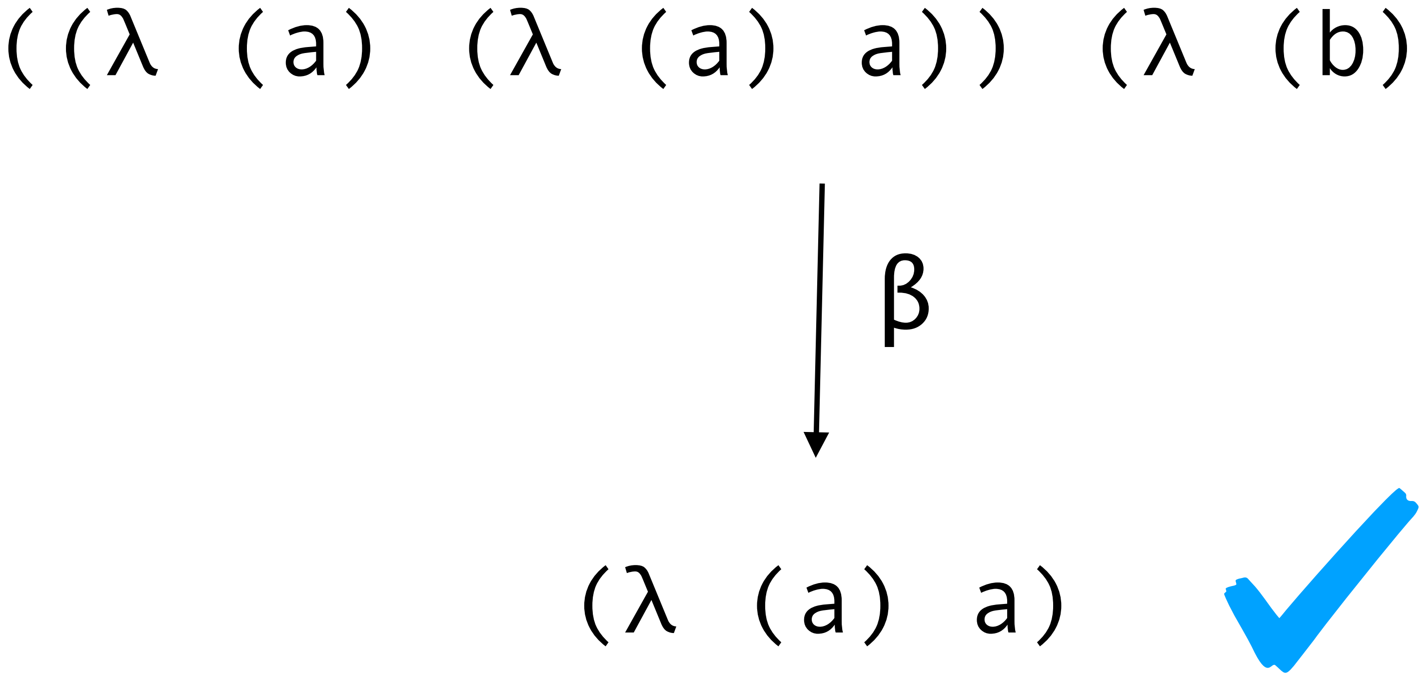
$$(\lambda \;\; (x) \;\; E_0)[x \leftarrow E] = (\lambda \;\; (x) \;\; E_0)$$

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

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

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

Capture-avoiding substitution



`((λ (a) (λ (a) a)) (λ (b) b))`

β

`(λ (a) a)` ✓

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))
```

$$\Big\downarrow \beta$$

```
((λ (z) (λ (y) (z 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)) (λ (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)))`

$\rightarrow_{\alpha}$ `(λ (y) ((λ (z) (λ (w) z)) (λ (x) y)))`

$\rightarrow_{\beta}$ `(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

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

$$=_\alpha$$

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!

```
((λ (x) (λ (x) x)) (λ (y) y))
```

α - renaming

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

Rename variables that would break naive substitution!

```
((λ (x) (λ (x) x)) (λ (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!

```
((λ (x) (λ (x) x)) (λ (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!

```
((λ (x) (λ (z) z)) (λ (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!

```
((λ (x) (λ (z) z)) (λ (y) y))
```

Could now perform beta-reduction with naive substitution

η - reduction

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

η - expansion

$E_0 \rightarrow_\eta$ `(λ (x) (E₀ x))` where $x \notin FV(E_0)$

Reduction

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

$$(\rightarrow^*)$$

reflexive/transitive closure

Evaluation

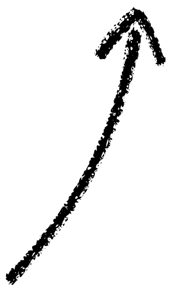
$$\begin{tikzcd}
E_0 \arrow[d, "*"'] \\
E_8 \arrow[d, "*"'] \\
?
\end{tikzcd}$$

Evaluation to *normal form*

$$\begin{tikzcd}
E_0 \arrow[d, "*"'] \\
(\lambda\ (x)\ \ldots)
\end{tikzcd}$$

Evaluation to *normal form*

$$\begin{tikzcd} E_0 \arrow[d, "*"'] \\ (\lambda\ (x)\ \ldots\ (\lambda\ (z)\ ((a\ \ldots)\ \ldots))) \end{tikzcd}$$



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

Evaluation Strategy

$$\begin{tikzcd}
 & \mathtt{E_0} \arrow[dl, "*"'] \arrow[dr, "*"] & \\
\mathtt{E_1} & & \mathtt{E_2}
\end{tikzcd}$$

Evaluation Strategy

`((λ (x) ((λ (y) y) x)) (λ (z) z))`

$\to_\eta$ `((λ (y) y) (λ (z) z))`

$\to_\beta$ `(λ (z) z)`

Evaluation Strategy

`((λ (x) ((λ (y) y) x)) (λ (z) z))`

$\rightarrow_\beta$ `((λ (y) y) (λ (z) z))`

$\rightarrow_\beta$ `(λ (z) z)`

Evaluation Strategy

`((λ (x) ((λ (y) y) x)) (λ (z) z))`

$\rightarrow_{\beta}$ `((λ (x) x) (λ (z) z))`

$\rightarrow_{\beta}$ `(λ (z) z)`

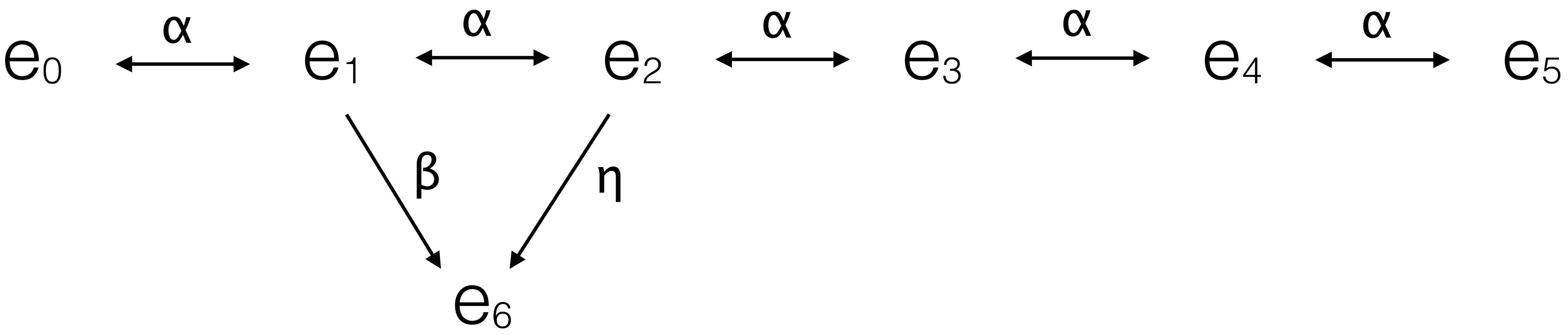
Confluence

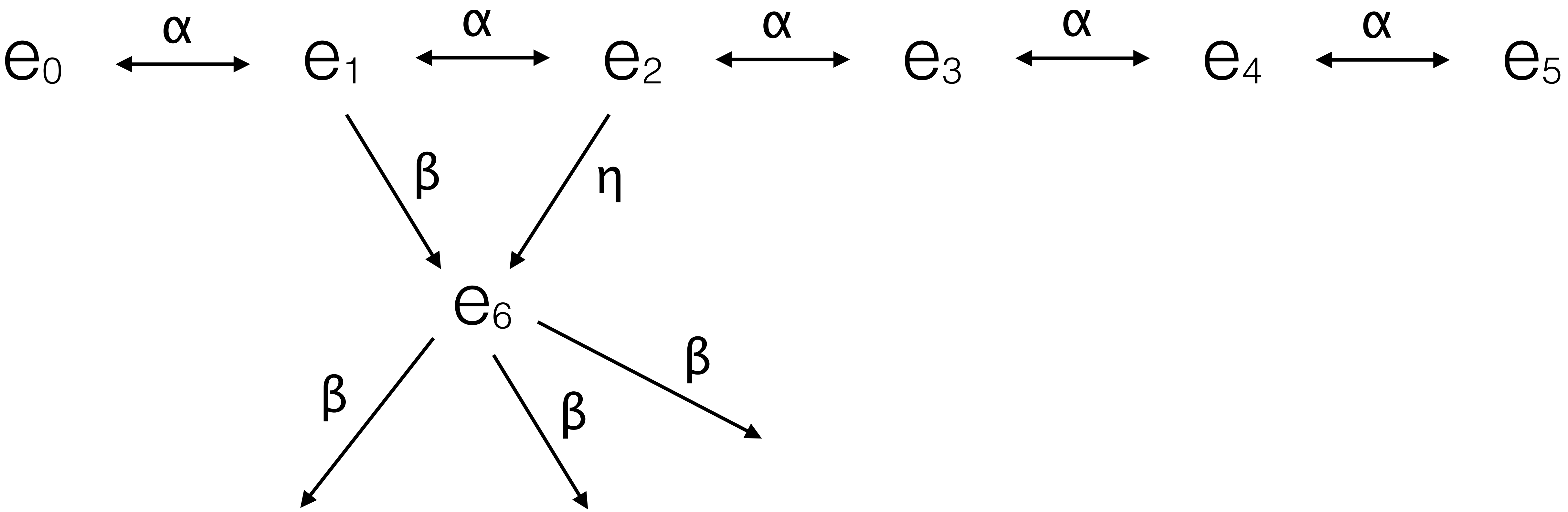
Diverging paths of evaluation must eventually join back together.

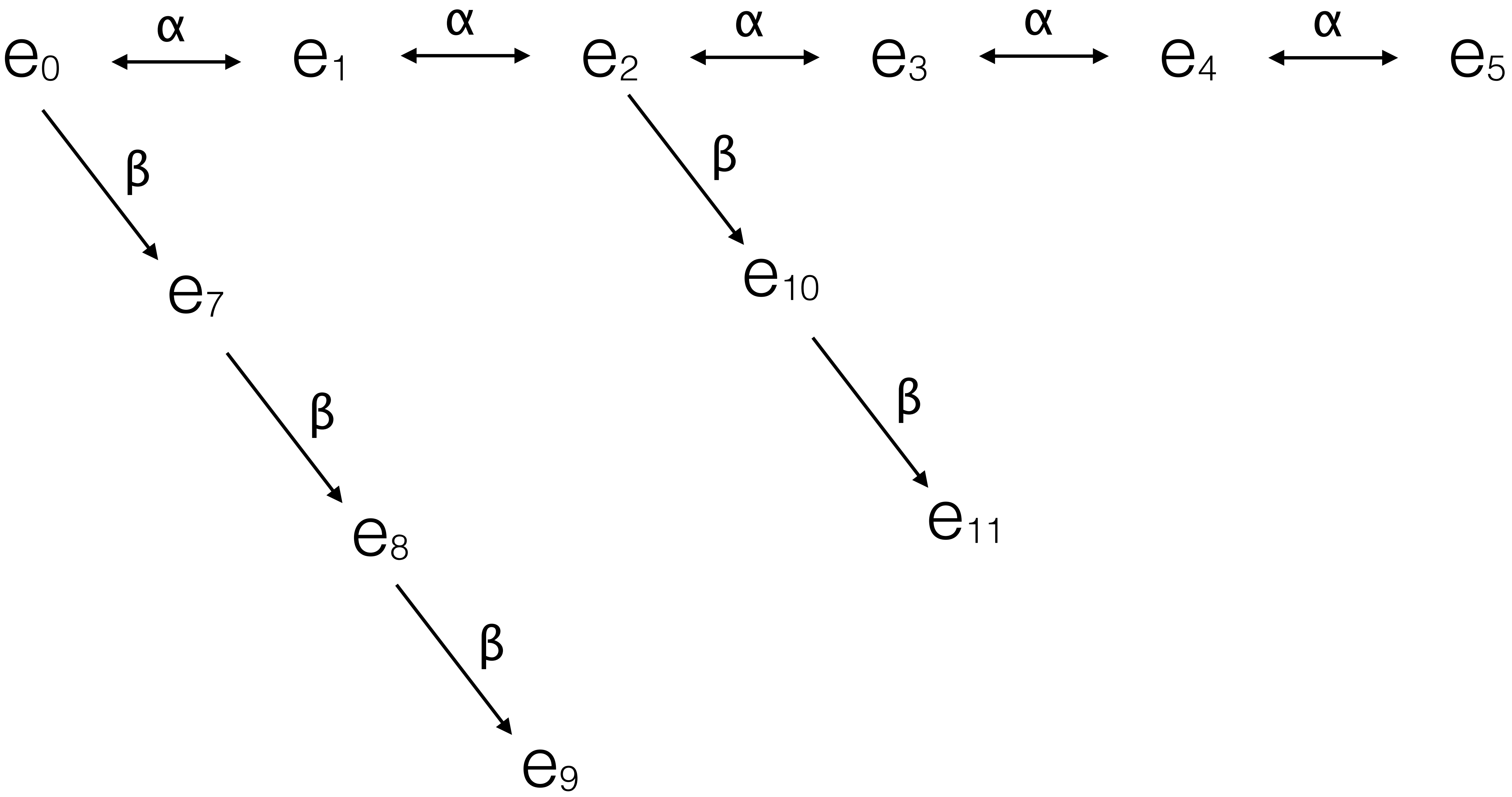
$$\begin{tikzcd}
& E_0 \arrow[dl, "*"'] \arrow[dr, "*"] & \\
E_1 \arrow[dr, "*"'] & & E_2 \arrow[dl, "*"] \\
& E_3 &
\end{tikzcd}$$

Church-Rosser Theorem

$$\begin{tikzcd} e_0 \arrow[r, leftrightarrow, "\alpha"] & e_1 \arrow[r, leftrightarrow, "\alpha"] & e_2 \arrow[r, leftrightarrow, "\alpha"] & e_3 \arrow[r, leftrightarrow, "\alpha"] & e_4 \arrow[r, leftrightarrow, "\alpha"] & e_5 \end{tikzcd}$$


e0
α
e1
α
e2
α
e3
α
e4
α
e5
β
η
e6


e₀
α
e₁
α
e₂
α
e₃
α
e₄
α
e₅
β
η
e₆
β
β
β


e_0
α
e_1
α
e_2
α
e_3
α
e_4
α
e_5
β
β
e_7
e_{10}
β
β
e_8
e_{11}
β
e_9

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

$$\begin{tikzcd}
e_0 \arrow[rr, leftrightarrow, "\alpha"] \arrow[dr, "\beta"] & & e_1 \arrow[rr, leftrightarrow, "\alpha"] & & e_2 \arrow[rr, leftrightarrow, "\alpha"] \arrow[dr, "\beta"] & & e_3 \arrow[rr, leftrightarrow, "\alpha"] & & e_4 \arrow[rr, leftrightarrow, "\alpha"] & & e_5 \\
& e_7 \arrow[dr, "\beta"] & & & & e_{10} \arrow[dr, "\beta"] & & & & & \\
& & e_8 \arrow[dr, "\beta"] & & & & e_{11} \arrow[dd, "*"] & & & & \\
& & & e_9 \arrow[drrr, "*"'] & & & & & & & \\
& & & & & & e_{12} & & & &
\end{tikzcd}$$

Applicative evaluation order

Always evaluates the *innermost* leftmost redex first.

Normal evaluation order

Always evaluates the *outermost* leftmost redex first.

Applicative evaluation order

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

Normal evaluation order

```
(((λ (x) ((λ (y) y) x)) (λ (z) z)) (λ (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*.

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

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

```
((λ (u) ((u u) u))
 (λ (u) ((u u) u)))
```

```
((λ (x) x)
 ((λ (u) (u u))
  (λ (u) (u u))))
```

Evaluation contexts

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

```
ℰ ::= (ℰ e)
    | (v ℰ)
    | □
```

(left-to-right) CBV evaluation

```
ℰ ::= (ℰ e)
    | □
```

(left-to-right) CBN evaluation

```
v  ::= (λ (x) e)
```

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

Context and redex

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

$$\mathcal{E}[\,\overbrace{\texttt{(v v)}}^{r}\,] \;=$$

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

$\mathcal{E}$ = `(□ (λ (w) w))`

r = `((λ (x) ((λ (y) y) x)) (λ (z) z))`

Context and redex

```
ℰ[r] =
(((λ (x) ((λ (y) y) x)) (λ (z) z)) (λ (w) w))
```

```
ℰ = (□ (λ (w) w))
```

```
r = ((λ (x) ((λ (y) y) x)) (λ (z) z))
           →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))

$\to_{\beta}$ ((λ (y) y) (λ (z) z))

$\Big\downarrow$ $\mathscr{E}$[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))))
```