Lambda Calculus: Reduction / Substitution

CIS352 — Fall 2022

Kris Micinski

Last lecture: β−reduction, informally

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

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

replace every x in E_0 with E_1.

If you watch the **history of the lambda calculus discussion by Dana Scott**, I will award two participation points (min 5-30):

https://www.youtube.com/watch?v=uS9InrmPIoc

How can we define beta reduction as a Racket function…?

```
(define (beta-reduce e)
  (match e
    [`((lambda (,x) ,e-body) ,e-arg) (subst x e-arg e-body)]
    [_ (error "beta-reduction cannot apply...")]))
```

Today: how do we define the **subst** function?

Variables are **challenging**

Semantics of the Lambda Calculus

Typical presentations of the lambda calculus define a **textual-reduction semantics**.

You can envision a "machine" where the machine's **state** is the *text* of the program as it evolves

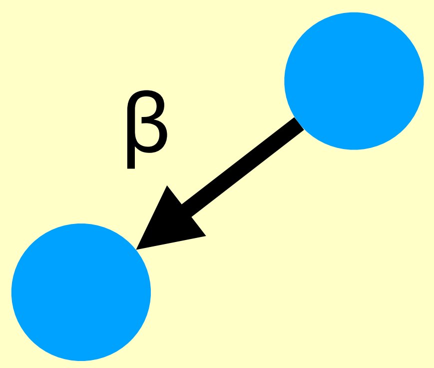
```
((lambda (x) x) ((lambda (z) z) y))
```

Semantics of the Lambda Calculus

Typical presentations of the lambda calculus define a **textual-reduction semantics**.

You can envision a "machine" where the machine's **state** is the *text* of the program as it evolves

```
((lambda (x) x) ((lambda (z) z) y))
```



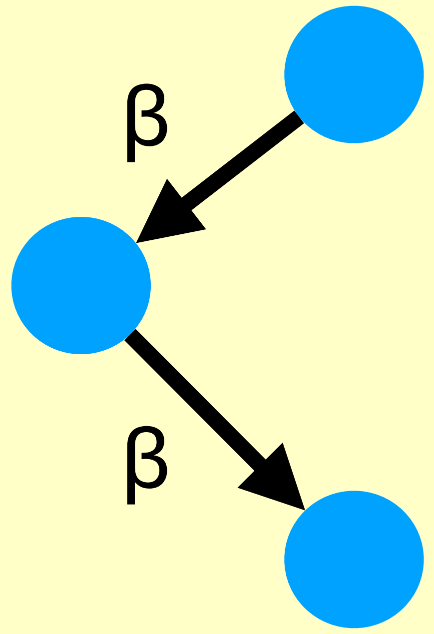

```
((lambda (x) x) y)
```

Semantics of the Lambda Calculus

Typical presentations of the lambda calculus define a **textual-reduction semantics**.

You can envision a "machine" where the machine's **state** is the *text* of the program as it evolves

```
((lambda (x) x) ((lambda (z) z) y))
```



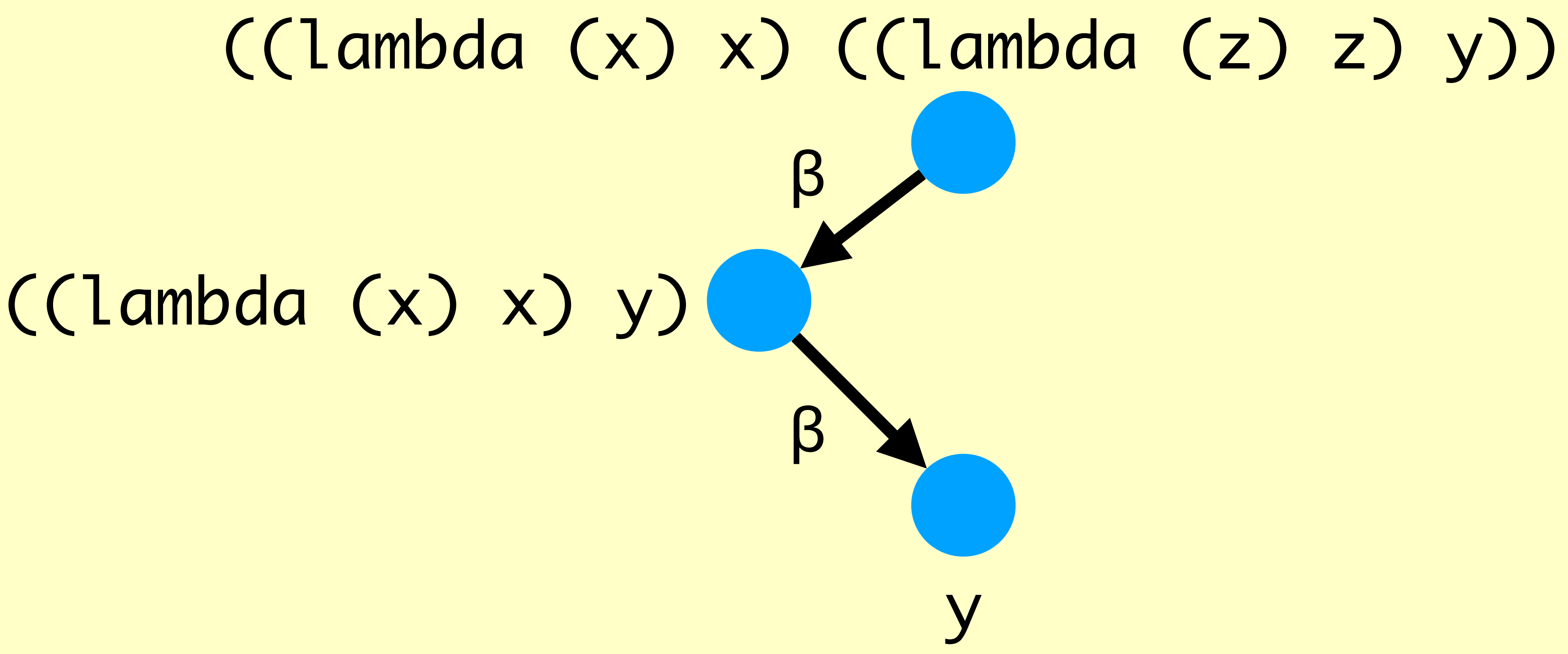

```
((lambda (x) x) y)
```

```
y
```

Semantics of the Lambda Calculus

Observe! Β-Reduction is nondeterministic

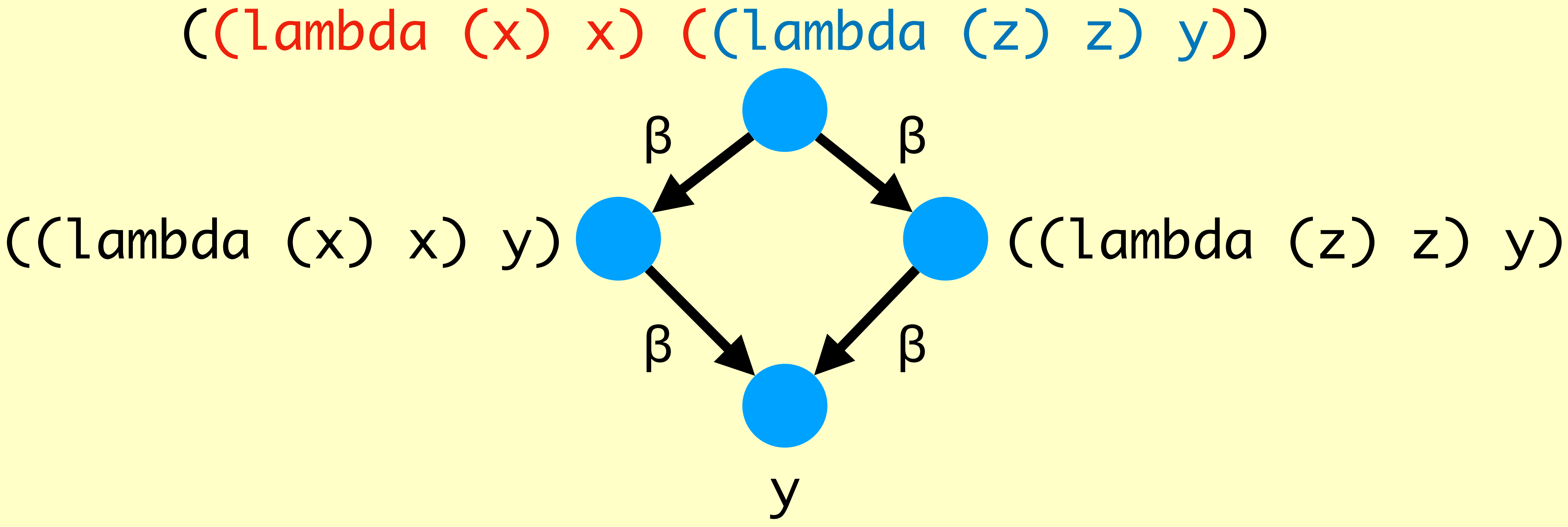
In general, a term may have **multiple** β redexes, and thus multiple β reductions



Semantics of the Lambda Calculus

This term has **two** beta redexes!

The outer one in **red**
The inner one in **blue**



The two challenges for this lecture:

- How do we implement substitution
- How do we deal with nondeterminism in the semantics

Substitution seems conceptually simple, but it is surprisingly tricky. But consider this: substitution is fundamentally **where computation happens**!

```
(define (beta-reduce e)
  (match e
    [`((lambda (,x) ,e-body) ,e-arg) (subst x e-arg e-body)]
    [_ (error "beta-reduction cannot apply...")]))
```

If we have **subst**, we can easily define **beta-reduce**.

Free Variables

We define the free variables of a lambda expression via the function FV:

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

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

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

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

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

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

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

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

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

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)
```

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}

Closed Terms

A term is **closed** when it has no free variables:

- ((lambda (x) x) (lambda (y) y))
- (lambda (z) (lambda (x) (z (lambda (z) z)))

Sometimes we call these (closed terms) **combinators**

Some **open** terms…

- (lambda (x) ((lambda (z) z) **z**))
- ((lambda (x) x) (lambda (z) **x**))

Alpha-Renaming

α-renaming allows us to rename variables:

$$\frac{y \notin FV(e)}{(\lambda\,(x)\;\, e) \stackrel{\alpha}{\rightarrow} (\lambda\,(y)\;\, e[x \mapsto y])}$$

Still need to define substitution…

Important consequence: terms are unique **up to α equivalence**

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

`(lambda (😋) 😋)`

`(lambda (x) x)`

Every term has infinitely-many terms to which it is α equivalent

What breaks if the antecedent isn't enforced..?

$$\frac{y \notin FV(e)}{(\lambda\,(x)\ e) \xrightarrow{\alpha} (\lambda\,(y)\ e[x \mapsto y])}$$

Meaning of term changes! Someone might have an intention to **use** that free variable y

`(lambda (x) add1)` very different from `(lambda (x) x)`

```
(((Lambda (x) add1) (lambda (y) y)) 2)
!=
(((Lambda (x) x) (lambda (y) y)) 2)
```

Can we define lambda calculi without explicit variables? (**Yes**!)

- De-Bruin Indices (variables are numbers indicating to which binder they belong)
- Combinatory logic uses bases of fully-closed terms. Always possible to rewrite any LC term to use only several closed combinators

We won't study either of these

$$\lambda\ (\lambda\ 1\ (\lambda\ 1))\ (\lambda\ 2\ 1)$$

We define **capture-avoiding substitution**, in which we are careful to avoid places where variables would become **captured** by a substitution.

The problem with (naive) textual substitution

```
((λ (a) (λ (a) a)) (λ (b) b))
```

$$\Big\downarrow \beta$$

```
(λ (a) a)[a ← (λ (b) b)]
```

The problem with (naive) textual substitution

```
((λ (a) (λ (a) a)) (λ (b) b))
```

$\downarrow \beta$

```
(λ (a) (λ (b) b))
```

Capture-avoiding substitution

$$E_0[x \leftarrow E_1]$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\downarrow \beta$

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

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

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

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)

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.

To formally define the semantics of the lambda calculus via reduction, we also need rules that will let us apply reductions **inside of** rules:

$$\alpha\ \frac{y \notin FV(e)}{(\lambda\,(x)\ e) \overset{\alpha}{\to} (\lambda\,(y)\ e[x \mapsto y])} \qquad \beta\ \frac{e' = e_b\,[x \mapsto e_1]}{\big((\lambda\,(x)\ e_b)\ e_1\big) \overset{\beta}{\to} e'}$$

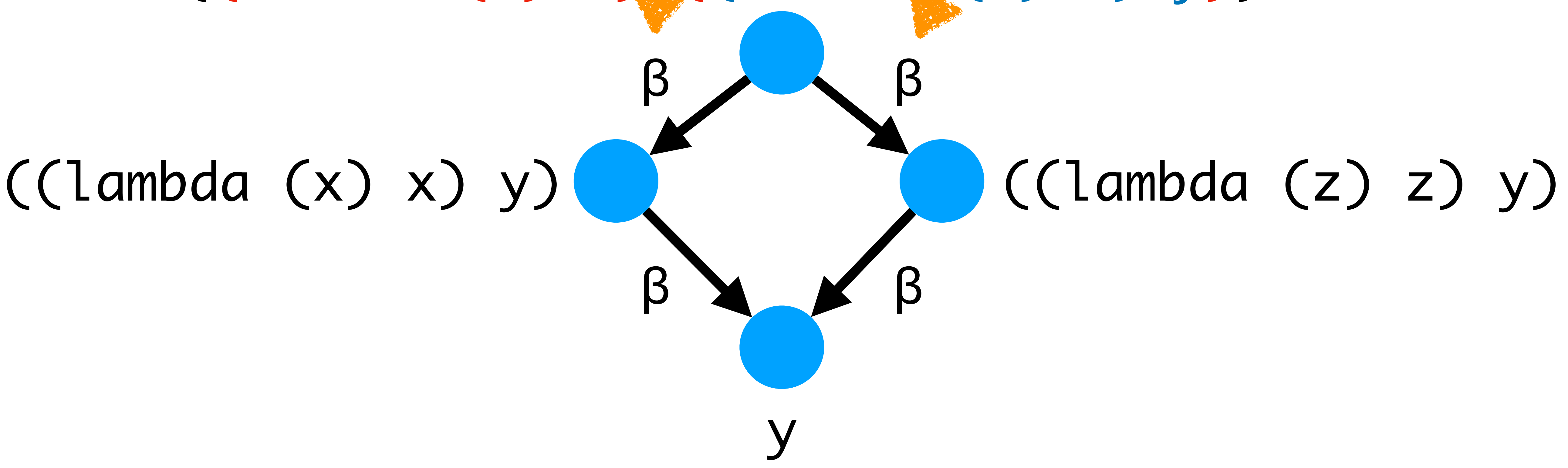
$$\beta_{\mathbf{0}}\ \frac{e_0 \overset{\beta\alpha}{\to} e'}{(e_0\ e_1) \to (e'\ e_1)} \qquad \beta_{\mathbf{1}}\ \frac{e_1 \overset{\beta\alpha}{\to} e'}{(e_0\ e_1) \to (e_0\ e')}$$

$$\alpha\ \frac{y \notin FV(e)}{(\lambda\,(x)\ e) \xrightarrow{\alpha} (\lambda\,(y)\ e[x \mapsto y])} \qquad \beta\ \frac{e' = e_b\,[x \mapsto e_1]}{\big((\lambda\,(x)\ e_b)\ e_1\big) \xrightarrow{\beta} e'}$$

$$\beta_0\ \frac{e_0 \xrightarrow{\beta\alpha} e'}{(e_0\ e_1) \to (e'\ e_1)} \qquad \beta_1\ \frac{e_1 \xrightarrow{\beta\alpha} e'}{(e_0\ e_1) \to (e_0\ e')}$$

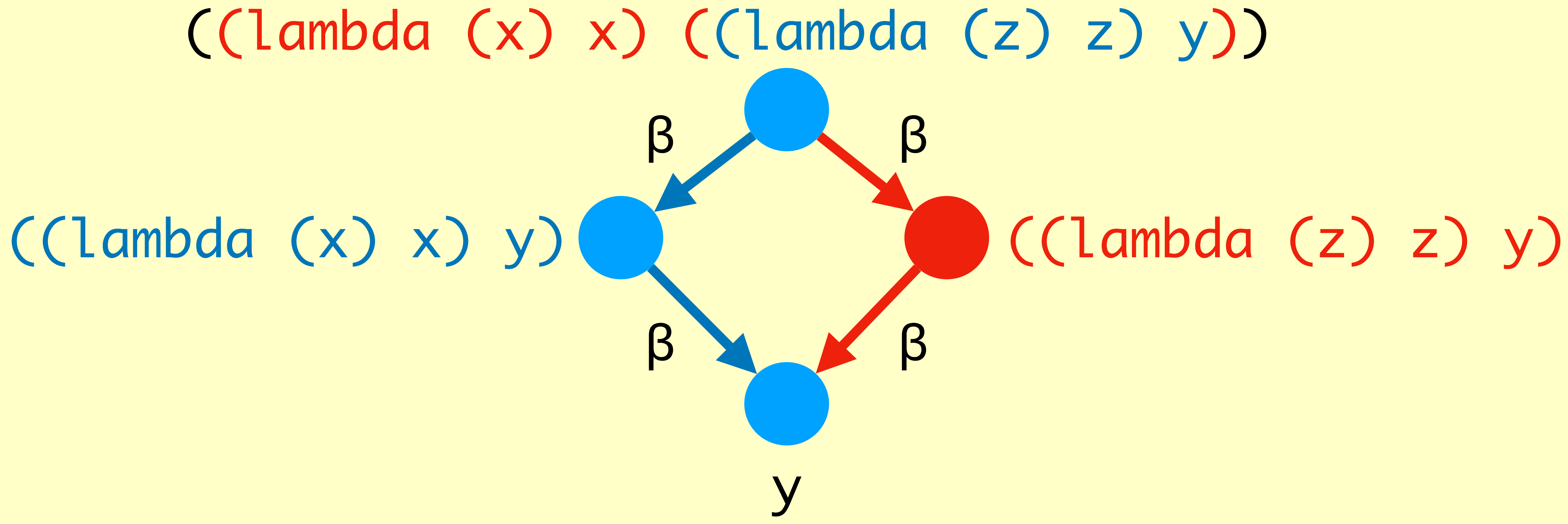
Recall: a term may have **multiple redexes**!

```
((lambda (x) x) ((lambda (z) z) y))
```



Because β and α reduction are inherently nondeterministic, we use a **reduction strategy**, which is system that tells us which reduction to apply:

- **Normal Order — Leftmost (outermost) application**
- **Applicative Order — Innermost application**



We'll talk more about these **next time**. They relate to the computational notions of **call-by-name (normal)** and **call-by-value (applicative)** η-reduction / expansion capture a property akin to extensionality

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

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

We do not use η-reduction/expansion in computation (unlike β), but it helps us establish certain equalities in lambda theories

When unambiguous, we refer to **reduction** in the lambda calculus as the application of a beta, alpha, or eta reduction:

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

$$(\rightarrow^*)$$

(When necessary for exams, we will clarify…)

It is often helpful to think of applying a sequence of reductions to arrive at some final "result."

In the lambda calculus, we call these results / values "normal forms."

A **normal form** is a form that has no more possible applications of some kind of reduction…

$$E_0 \xrightarrow{*} E_8 \xrightarrow{*} ?$$

$$E_0 \;\big\downarrow{*}$$

```
(λ (x) … (λ (z) ((a …) …)))
```

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

We covered a lot of material!

- Free variables
- Alpha renaming
- Beta reduction
- Eta reduction / expansion
- Capture-avoiding substitution
- Applicative / normal order

Next time: **reduction strategies and more normal forms…**