Lecture 16
Programming in the $\lambda$-calculus
We can encode TRUE, FALSE, and IF, as:

\[
\text{TRUE } \triangleq \lambda x. \lambda y. x \\
\text{FALSE } \triangleq \lambda x. \lambda y. y \\
\text{IF } \triangleq \lambda b. \lambda t. \lambda f. b \, t \, f
\]

This way, IF behaves how it ought to:

\[
\text{IF TRUE } v_t \, v_f \rightarrow^* v_t \\
\text{IF FALSE } v_t \, v_f \rightarrow^* v_f
\]
Review: Church Numerals

Church numerals encode a number $n$ as a function that takes $f$ and $x$, and applies $f$ to $x$ $n$ times.

\[
\begin{align*}
0 & \equiv \lambda f. \lambda x. x \\
1 & \equiv \lambda f. \lambda x. f x \\
2 & \equiv \lambda f. \lambda x. f (f x)
\end{align*}
\]

We can define other functions on integers:

\[
\text{SUCC} \equiv \lambda n. \lambda f. \lambda x. f (n f x)
\]
Review: Church Numerals

Church numerals encode a number $n$ as a function that takes $f$ and $x$, and applies $f$ to $x$ $n$ times.

$$
\begin{align*}
0 & \triangleq \lambda f. \lambda x. x \\
1 & \triangleq \lambda f. \lambda x. f x \\
2 & \triangleq \lambda f. \lambda x. f (f x)
\end{align*}
$$

We can define other functions on integers:

$$
\begin{align*}
\text{SUCC} & \triangleq \lambda n. \lambda f. \lambda x. f (n f x) \\
\text{PLUS} & \triangleq \lambda n_1. \lambda n_2. n_1 \text{SUCC} n_2
\end{align*}
$$
Review: Church Numerals

Church numerals encode a number $n$ as a function that takes $f$ and $x$, and applies $f$ to $x$ $n$ times.

$$
\bar{0} \triangleq \lambda f. \lambda x. x
$$

$$
\bar{1} \triangleq \lambda f. \lambda x. f x
$$

$$
\bar{2} \triangleq \lambda f. \lambda x. f (f x)
$$

We can define other functions on integers:

$$
\text{SUCC} \triangleq \lambda n. \lambda f. \lambda x. f (n f x)
$$

$$
\text{PLUS} \triangleq \lambda n_1. \lambda n_2. n_1 \text{SUCC} n_2
$$

$$
\text{TIMES} \triangleq \lambda n_1. \lambda n_2. n_1 (\text{PLUS} n_2) \bar{0}
$$
Review: Church Numerals

Church numerals encode a number \( n \) as a function that takes \( f \) and \( x \), and applies \( f \) to \( x \) \( n \) times.

\[
\begin{align*}
\overline{0} & \triangleq \lambda f. \lambda x. x \\
\overline{1} & \triangleq \lambda f. \lambda x. f x \\
\overline{2} & \triangleq \lambda f. \lambda x. f (f x)
\end{align*}
\]

We can define other functions on integers:

\[
\begin{align*}
\text{SUCC} & \triangleq \lambda n. \lambda f. \lambda x. f (n f x) \\
\text{PLUS} & \triangleq \lambda n_1. \lambda n_2. n_1 \text{SUCC} n_2 \\
\text{TIMES} & \triangleq \lambda n_1. \lambda n_2. n_1 (\text{PLUS} n_2) \overline{0} \\
\text{ISZERO} & \triangleq \lambda n. n (\lambda z. \text{FALSE}) \text{TRUE}
\end{align*}
\]
Recursive Functions

How would we write recursive functions like factorial?
Recursive Functions

How would we write recursive functions like factorial?

We’d like to write it like this...

\[
\text{FACT} \triangleq \lambda n. \, \text{IF} \ (\text{ISZERO} \ n) \ 1 \ (\text{TIMES} \ n \ (\text{FACT} \ (\text{PRED} \ n)))
\]
Recursive Functions

How would we write recursive functions like factorial?

We’d like to write it like this...

\[
\text{FACT} \triangleq \lambda n. \text{IF (ISZERO } n\text{) 1 (TIMES } n\text{ (FACT (PRED } n\text{)))}
\]

In slightly more readable notation this is...

\[
\text{FACT} \triangleq \lambda n. \text{if } n = 0 \text{ then 1 else } n \times \text{FACT } (n - 1)
\]

...but this is an equation, not a definition!
Recursion removal trick

We can perform a “trick” to define a function FACT that satisfies the recursive equation on the previous slide.
Recursion removal trick

We can perform a “trick” to define a function FACT that satisfies the recursive equation on the previous slide.

Define a new function FACT’ that takes a function f as an argument. Then, for “recursive” calls, it uses f f:

\[
\text{FACT’} \triangleq \lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times ((ff)(n - 1))
\]
Recursion removal trick

We can perform a “trick” to define a function FACT that satisfies the recursive equation on the previous slide.

Define a new function FACT’ that takes a function \( f \) as an argument. Then, for “recursive” calls, it uses \( f f \):

\[
\text{FACT'} \triangleq \lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times ((ff) (n - 1))
\]

Then define FACT as FACT’ applied to itself:

\[
\text{FACT} \triangleq \text{FACT'} \text{ FACT'}
\]
Example

Let’s try evaluating FACT on 3...

FACT 3
Example

Let’s try evaluating FACT on 3...

FACT 3 = (FACT’ FACT’) 3
Example

Let’s try evaluating FACT on 3...

FACT 3 = (FACT’ FACT’) 3
    = ((λf. λn. if n = 0 then 1 else n × ((ff) (n − 1))) FACT’) 3
Let’s try evaluating FACT on 3...

FACT 3 = (FACT’ FACT’) 3

= ((λf. λn. if n = 0 then 1 else n × ((ff) (n − 1))) FACT’) 3
→ (λn. if n = 0 then 1 else n × ((FACT’ FACT’) (n − 1))) 3
Let’s try evaluating FACT on 3...

FACT 3 = (FACT’ FACT’) 3

= ((λf. λn. if n = 0 then 1 else n × ((ff) (n – 1))) FACT’) 3
→ (λn. if n = 0 then 1 else n × ((FACT’ FACT’) (n – 1))) 3
→ if 3 = 0 then 1 else 3 × ((FACT’ FACT’) (3 – 1))
Example

Let’s try evaluating FACT on 3...

FACT 3 = (FACT’ FACT’) 3

= ((\lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times ((f f) (n - 1))) \text{ FACT’}) 3
→ (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times ((\text{FACT’ FACT’}) (n - 1))) 3
→ \text{if } 3 = 0 \text{ then } 1 \text{ else } 3 \times ((\text{FACT’ FACT’}) (3 - 1))
→ 3 \times ((\text{FACT’ FACT’}) (3 - 1))
Example

Let’s try evaluating FACT on 3...

FACT 3 = (FACT’ FACT’) 3

= ((λf. λn. if n = 0 then 1 else n × ((f f) (n − 1))) FACT’) 3
→ (λn. if n = 0 then 1 else n × ((FACT’ FACT’) (n − 1))) 3
→ if 3 = 0 then 1 else 3 × ((FACT’ FACT’) (3 − 1))
→ 3 × ((FACT’ FACT’) (3 − 1))
= 3 × (FACT (3 − 1))
Example

Let's try evaluating FACT on 3...

FACT 3 = (FACT' FACT') 3

= ((λf. λn. if n = 0 then 1 else n × ((ff) (n - 1))) FACT') 3
→ (λn. if n = 0 then 1 else n × ((FACT' FACT') (n - 1))) 3
→ if 3 = 0 then 1 else 3 × ((FACT' FACT') (3 - 1))
→ 3 × ((FACT' FACT') (3 - 1))
= 3 × (FACT (3 - 1))
→ ...
→ 3 × 2 × 1 × 1
Example

Let’s try evaluating FACT on 3...

FACT 3 = (FACT’ FACT’) 3
= ((λf. λn. if n = 0 then 1 else n × ((ff) (n − 1))) FACT’) 3
→ (λn. if n = 0 then 1 else n × ((FACT’ FACT’) (n − 1))) 3
→ if 3 = 0 then 1 else 3 × ((FACT’ FACT’) (3 − 1))
→ 3 × ((FACT’ FACT’) (3 − 1))
= 3 × (FACT (3 − 1))
→ ... 
→ 3 × 2 × 1 × 1
→ * 6
Fixed point combinators

Our “trick” requires following human-readable instructions. Write a different function \( f' \) that takes itself as an argument and uses self-application for recursive calls, and then define \( f \) as \( f' \circ f' \).
Fixed point combinators

Our “trick” requires following human-readable instructions. Write a different function $f'$ that takes itself as an argument and uses self-application for recursive calls, and then define $f$ as $f' \ f'$. There is another way: fixed points!
Fixed point combinators

Our “trick” requires following human-readable instructions. Write a different function $f'$ that takes itself as an argument and uses self-application for recursive calls, and then define $f$ as $f' \ f'$.

There is another way: fixed points!

Consider factorial again. It is a fixed point of the following:

$$G \triangleq \lambda f. \lambda n. \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (f (n - 1))$$
Fixed point combinators

Our “trick” requires following human-readable instructions. Write a different function $f'$ that takes itself as an argument and uses self-application for recursive calls, and then define $f$ as $f' \ f'$.

There is another way: fixed points!

Consider factorial again. It is a fixed point of the following:

$$G \triangleq \lambda f. \lambda n. \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (f (n - 1))$$

Recall that if $g$ if a fixed point of $G$, then $G \ g = g$. To see that any fixed point $g$ is a real factorial function, try evaluating it:

$$g \ 5 = (G \ g) \ 5$$
Fixed point combinators

Our “trick” requires following human-readable instructions. Write a different function $f'$ that takes itself as an argument and uses self-application for recursive calls, and then define $f$ as $f' f'$.

There is another way: fixed points!

Consider factorial again. It is a fixed point of the following:

$$ G \triangleq \lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (f(n - 1)) $$

Recall that if $g$ if a fixed point of $G$, then $G g = g$. To see that any fixed point $g$ is a real factorial function, try evaluating it:

$$ g \ 5 = (G \ g) \ 5 $$

$$ \rightarrow^* 5 \times (g \ 4) $$
Fixed point combinators

Our “trick” requires following human-readable instructions. Write a different function $f'$ that takes itself as an argument and uses self-application for recursive calls, and then define $f$ as $f' \ f'$. 

There is another way: fixed points!

Consider factorial again. It is a fixed point of the following:

$$G \triangleq \lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (f(n - 1))$$

Recall that if $g$ if a fixed point of $G$, then $G \ g = g$. To see that any fixed point $g$ is a real factorial function, try evaluating it:

$$g\ 5 = (G \ g)\ 5$$

$$\rightarrow^* 5 \times (g\ 4)$$

$$= 5 \times ((G \ g)\ 4)$$
Fixed point combinators

How can we generate the fixed point of $G$?

In denotational semantics, finding fixed points took a lot of math. In the $\lambda$-calculus, we just need a suitable combinator...
Y Combinator

The (infamous) Y combinator is defined as

\[ Y \triangleq \lambda f. (\lambda x. f(x x)) (\lambda x. f(x x)) \]

We say that Y is a **fixed point combinator** because Yf is a fixed point of f (for any lambda term f).
Y Combinator

The (infamous) Y combinator is defined as

\[ Y \triangleq \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \]

We say that Y is a \textit{fixed point combinator} because Y f is a fixed point of f (for any lambda term f).

What happens when we evaluate Y G under CBV?

\[
Y G = \left( \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \right) G
\]

\[
\rightarrow (\lambda x. G (x x)) (\lambda x. G (x x))
\]

\[
\rightarrow G (\lambda x. G (x x)) (\lambda x. G (x x))
\]

\[
\rightarrow G (\lambda x. G (x x)) (\lambda x. G (x x))
\]
Z Combinator

To avoid this issue, we’ll use a slight variant of the Y combinator, called Z, which is easier to use under CBV.
Z Combinator

To avoid this issue, we’ll use a slight variant of the Y combinator, called Z, which is easier to use under CBV.

\[ Z \triangleq \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) \]

\[ \lambda y. e \ y \]
Example

Let’s see Z in action, on our function G.

FACT
Example

Let’s see $Z$ in action, on our function $G$.

\[
\text{FACT} = Z^{} G
\]
Example

Let’s see Z in action, on our function G.

\[
\text{FACT} = Z \ G = (\lambda f. (\lambda x. f (\lambda y. x \ x \ y)) (\lambda x. f (\lambda y. x \ x \ y))) \ G
\]
Example

Let’s see Z in action, on our function G.

\[
\text{FACT} \\
= Z G \\
= (\lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))) G \\
\rightarrow (\lambda x. G (\lambda y. x x y)) (\lambda x. G (\lambda y. x x y))
\]
Example

Let’s see $Z$ in action, on our function $G$.

\[
\begin{align*}
\text{FACT} \\
\quad = & \quad Z \ G \\
\quad = & \quad (\lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))) \ G \\
\quad \rightarrow & \quad (\lambda x. G (\lambda y. x x y)) (\lambda x. G (\lambda y. x x y)) \\
\quad \rightarrow & \quad G (\lambda y. (\lambda x. G (\lambda y. x x y)) (\lambda x. G (\lambda y. x x y))) \ y)
\end{align*}
\]
Example

Let's see Z in action, on our function G.

FACT

\[ Z \ G \]

\[ \Rightarrow (\lambda x. \ G (\lambda y. \ x \ x \ y)) (\lambda x. \ G (\lambda y. \ x \ x \ y)) \]

\[ \Rightarrow G (\lambda y. \ (\lambda x. \ G (\lambda y. \ x \ x \ y)) (\lambda x. \ G (\lambda y. \ x \ x \ y)) \ y) \]

\[ \Rightarrow (\lambda f. \ \lambda n. \ \text{if } n = 0 \ \text{then } 1 \ \text{else } n \times (f(n - 1))) \]

\[ \quad \quad (\lambda y. \ (\lambda x. \ G (\lambda y. \ x \ x \ y)) (\lambda x. \ G (\lambda y. \ x \ x \ y)) \ y) \]
Example

Let’s see Z in action, on our function G.

FACT

\[
\begin{align*}
\ &= Z \ G \\
\ &= (\lambda f. (\lambda x. f (\lambda y. x \ x \ y)) (\lambda x. f (\lambda y. x \ x \ y))) \ G \\
\rightarrow \ &= (\lambda x. G (\lambda y. x \ x \ y)) (\lambda x. G (\lambda y. x \ x \ y)) \\
\rightarrow \ &= G (\lambda y. (\lambda x. G (\lambda y. x \ x \ y)) (\lambda x. G (\lambda y. x \ x \ y)) \ y) \\
\ &= (\lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (f(n-1))) \\
\ &\quad \quad (\lambda y. (\lambda x. G (\lambda y. x \ x \ y)) (\lambda x. G (\lambda y. x \ x \ y)) \ y) \\
\rightarrow \ &= \lambda n. \text{if } n = 0 \text{ then } 1 \\
\ &\quad \quad \text{else } n \times ((\lambda y. (\lambda x. G (\lambda y. x \ x \ y)) (\lambda x. G (\lambda y. x \ x \ y)) \ y) (n - 1))
\end{align*}
\]
Let’s see $Z$ in action, on our function $G$.

\[
\begin{align*}
\text{FACT} & = Z \; G \\
& = (\lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))) \; G \\
& \rightarrow (\lambda x. G (\lambda y. x x y)) (\lambda x. G (\lambda y. x x y)) \\
& \rightarrow G (\lambda y. (\lambda x. G (\lambda y. x x y)) (\lambda x. G (\lambda y. x x y)) \; y) \\
& = (\lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (f (n - 1))) \; (\lambda y. (\lambda x. G (\lambda y. x x y)) (\lambda x. G (\lambda y. x x y)) \; y) \\
& \rightarrow \lambda n. \text{if } n = 0 \text{ then } 1 \\
& \quad \text{else } n \times ((\lambda y. (\lambda x. G (\lambda y. x x y)) (\lambda x. G (\lambda y. x x y)) \; y) \; (n - 1)) \\
& = \beta \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (\lambda y. (Z \; G) \; y) \; (n - 1)
\end{align*}
\]
Example

Let’s see Z in action, on our function G.

FACT

\[
Z \ G = (\lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))) \ G \\
\rightarrow (\lambda x. G (\lambda y. x x y)) (\lambda x. G (\lambda y. x x y)) \\
\rightarrow G (\lambda y. (\lambda x. G (\lambda y. x x y)) (\lambda x. G (\lambda y. x x y)) y) \\
= (\lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (f (n - 1))) \\
\quad (\lambda y. (\lambda x. G (\lambda y. x x y)) (\lambda x. G (\lambda y. x x y)) y) \\
\rightarrow \lambda n. \text{if } n = 0 \text{ then } 1 \\
\quad \text{else } n \times ((\lambda y. (\lambda x. G (\lambda y. x x y)) (\lambda x. G (\lambda y. x x y)) y) (n - 1)) \\
= \beta \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (\lambda y. (Z \ G) y) (n - 1) \\
= \beta \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (Z \ G)(n - 1))
Example

Let’s see Z in action, on our function G.

FACT

\[ Z \quad G \]

\[ = (\lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))) \quad G \]

\[ \rightarrow (\lambda x. G (\lambda y. x x y)) (\lambda x. G (\lambda y. x x y)) \]

\[ \rightarrow G (\lambda y. (\lambda x. G (\lambda y. x x y)) (\lambda x. G (\lambda y. x x y)) y) \]

\[ = (\lambda f. \lambda n. \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times f(n - 1)) \]

\[ (\lambda y. (\lambda x. G (\lambda y. x x y)) (\lambda x. G (\lambda y. x x y)) y) \]

\[ \rightarrow \lambda n. \textbf{if } n = 0 \textbf{ then } 1 \]

\[ \textbf{else } n \times \left((\lambda y. (\lambda x. G (\lambda y. x x y)) (\lambda x. G (\lambda y. x x y)) y) (n - 1)\right) \]

\[ = \beta \lambda n. \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (\lambda y. (Z \quad G) y) (n - 1) \]

\[ = \beta \lambda n. \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (Z \quad G (n - 1)) \]

\[ = \lambda n. \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (F\text{FACT} \quad (n - 1)) \]
Other fixed point combinators

There are many (indeed infinitely many) fixed-point combinators. Here’s a cute one:

\[ Y_k \triangleq (L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L) \]

where

\[ L \triangleq \lambda abcdefghijklmnopqrstuvwxyz. \]
\[ (r \text{this is a fixed point combinator}) \]
Turing’s Fixed Point Combinator

To gain some more intuition for fixed point combinators, let’s derive a combinator $\Theta$ originally discovered by Turing.
Turing’s Fixed Point Combinator

To gain some more intuition for fixed point combinators, let’s derive a combinator $\Theta$ originally discovered by Turing.

We know that $\Theta \, f$ is a fixed point of $f$, so we have

$$\Theta \, f = f \, (\Theta \, f).$$
Turing’s Fixed Point Combinator

To gain some more intuition for fixed point combinators, let’s derive a combinator \( \Theta \) originally discovered by Turing.

We know that \( \Theta f \) is a fixed point of \( f \), so we have

\[
\Theta f = f (\Theta f).
\]

We can write the following recursive equation:

\[
\Theta = \lambda f. f (\Theta f)
\]
Turing’s Fixed Point Combinator

To gain some more intuition for fixed point combinators, let’s derive a combinator $\Theta$ originally discovered by Turing.

We know that $\Theta f$ is a fixed point of $f$, so we have

$$\Theta f = f (\Theta f).$$

We can write the following recursive equation:

$$\Theta = \lambda f. f (\Theta f).$$

Now use the recursion removal trick:

$$\Theta' \triangleq \lambda t. \lambda f. f (t t f)$$

$$\Theta \triangleq (\Theta \Theta')$$
$\theta$ Example

\[ \text{FACT} = \Theta \ G \]
\[ \text{FACT} = \Theta G \\
= ((\lambda t. \lambda f. f \, (t \, t \, f)) \, (\lambda t. \lambda f. f \, (t \, t \, f))) \, G \]
Example

\[
\text{FACT} = \Theta G \\
= ((\lambda t. \lambda f. f(t t f)) (\lambda t. \lambda f. f(t t f))) G \\
\rightarrow (\lambda f. f((\lambda t. \lambda f. f(t t f)) (\lambda t. \lambda f. f(t t f)) f)) G
\]
Example

\[
\text{FACT} = \Theta G \\
= ((\lambda t. \lambda f. f (t t f)) (\lambda t. \lambda f. f (t t f))) G \\
\rightarrow (\lambda f. f ((\lambda t. \lambda f. f (t t f)) (\lambda t. \lambda f. f (t t f)) f)) G \\
\rightarrow G ((\lambda t. \lambda f. f (t t f)) (\lambda t. \lambda f. f (t t f)) G)
\]
\( \theta \text{ Example} \)

\[
\text{FACT} = \Theta G \\
= ((\lambda t. \lambda f. f(t \ t \ f)) \ (\lambda t. \lambda f. f(t \ t \ f))) \ G \\
\rightarrow (\lambda f. f((\lambda t. \lambda f. f(t \ t \ f)) \ (\lambda t. \lambda f. f(t \ t \ f)) \ f)) \ G \\
\rightarrow G ((\lambda t. \lambda f. f(t \ t \ f)) \ (\lambda t. \lambda f. f(t \ t \ f)) \ G) \\
= G (\Theta G)
\]
FACT $\equiv \Theta G$

$\equiv ((\lambda t. \lambda f. f (t t f)) (\lambda t. \lambda f. f (t t f))) G$

$\rightarrow (\lambda f. f ((\lambda t. \lambda f. f (t t f)) (\lambda t. \lambda f. f (t t f)) f)) G$

$\rightarrow G ((\lambda t. \lambda f. f (t t f)) (\lambda t. \lambda f. f (t t f)) G)$

$\equiv G (\Theta G)$

$\equiv (\lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (f (n - 1))) (\Theta G)$

$\rightarrow \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times ((\Theta G) (n - 1))$

$\equiv \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (\text{FACT} (n - 1))$
Review: Call-by-Value

Here are the syntax and CBV semantics of $\lambda$-calculus:

\[
e ::= x \mid \lambda x. \, e \mid e_1 \, e_2 \\
\nu ::= \lambda x. \, e
\]

\[
\begin{align*}
e_1 & \rightarrow e_1' \\
e_1 \, e_2 & \rightarrow e_1' \, e_2
\end{align*}
\]

\[
\begin{align*}
e & \rightarrow e' \\
\nu \, e & \rightarrow \nu \, e'
\end{align*}
\]

\[
(\lambda x. \, e) \, \nu \rightarrow e\{\nu/x\} \, ^{\beta}
\]

There are two kinds of rules: *congruence rules* that specify evaluation order and *computation rules* that specify the “interesting” reductions.
Evaluation Contexts

**Evaluation contexts** let us separate out these two kinds of rules.
Evaluation Contexts

**Evaluation contexts** let us separate out these two kinds of rules.

An evaluation context $E$ is an expression with a “hole” in it: a single occurrence of the special symbol `[·]` in place of a subexpression.

$$E ::= [·] \mid E\ e \mid \nu\ E$$
Evaluation Contexts

Evaluation contexts let us separate out these two kinds of rules.

An evaluation context $E$ is an expression with a “hole” in it: a single occurrence of the special symbol $[\cdot]$ in place of a subexpression.

$$E ::= [\cdot] \mid E \ e \mid \nu \ E$$

We write $E[e]$ to mean the evaluation context $E$ where the hole has been replaced with the expression $e$. 
Examples

\[ E_1 = [\cdot] (\lambda x. x) \]
\[ E_1[\lambda y. y y] = (\lambda y. y y) \lambda x. x \]
Examples

\[ E_1 = [\cdot] (\lambda x. x) \]
\[ E_1[\lambda y. y y] = (\lambda y. y y) \lambda x. x \]

\[ E_2 = (\lambda z. z z) [\cdot] \]
\[ E_2[\lambda x. \lambda y. x] = (\lambda z. z z) (\lambda x. \lambda y. x) \]
Examples

\[ E_1 = [\cdot] (\lambda x. x) \]
\[ E_1[\lambda y. y y] = (\lambda y. y y) \lambda x. x \]

\[ E_2 = (\lambda z. z z) [\cdot] \]
\[ E_2[\lambda x. \lambda y. x] = (\lambda z. z z) (\lambda x. \lambda y. x) \]

\[ E_3 = ([\cdot] \lambda x. x x) ((\lambda y. y) (\lambda y. y)) \]
\[ E_3[\lambda f. \lambda g. f g] = ((\lambda f. \lambda g. f g) \lambda x. x x) ((\lambda y. y) (\lambda y. y)) \]
CBV With Evaluation Contexts

With evaluation contexts, we can define the evaluation semantics for the CBV $\lambda$-calculus with just two rules: one for evaluation contexts, and one for $\beta$-reduction.
CBV With Evaluation Contexts

With evaluation contexts, we can define the evaluation semantics for the CBV \( \lambda \)-calculus with just two rules: one for evaluation contexts, and one for \( \beta \)-reduction.

With this syntax:

\[
E ::= \cdot | E \, e | \nu \, E
\]

The small-step rules are:

\[
e \rightarrow e' \\
\frac{}{E[e] \rightarrow E[e']} \\
\frac{}{(\lambda x. \, e) \, \nu \rightarrow e\{\nu/x\}}^\beta
\]
We can also define the semantics of CBN $\lambda$-calculus with evaluation contexts.
CBN With Evaluation Contexts

We can also define the semantics of CBN $\lambda$-calculus with evaluation contexts.

For call-by-name, the syntax for evaluation contexts is different:

$$E ::= [\cdot] | E e$$
CBN With Evaluation Contexts

We can also define the semantics of CBN \( \lambda \)-calculus with evaluation contexts.

For call-by-name, the syntax for evaluation contexts is different:

\[
E ::= [\cdot] \mid E \, e
\]

But the small-step rules are the same:

\[
e \rightarrow e' \\
\frac{}{E[e] \rightarrow E[e']}
\]

\[
(\lambda x. \, e) \, e' \rightarrow e\{e'/x\}^\beta
\]