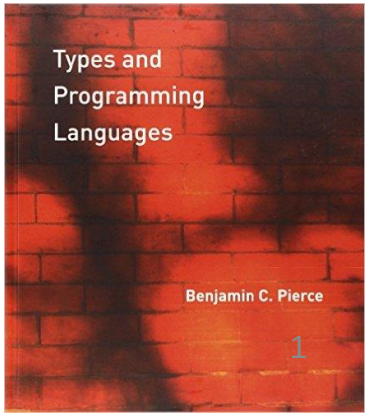
Concepts in Programming Languages
Recitation 5:

Untyped Lambda Calculus

Yotam Feldman

(original slides by Kathleen Fisher, John Mitchell, Shachar Itzhaky, Oded Padon, Mooly Sagiv, S. Tanimoto)

Reference:
Types and Programming Languages
by Benjamin C. Pierce, Chapter 5



Untyped Lambda Calculus - Syntax

t ::=	terms
x	variable
λ x. t	abstraction
t t	application

- Terms can be represented as abstract syntax trees
- Syntactic Conventions:
  - Applications associates to left :
    $e_1\ e_2\ e_3 \equiv (e_1\ e_2)\ e_3$
  - The body of abstraction extends as far as possible:
    $\lambda x.\ \lambda y.\ x\ y\ x \equiv \lambda x.\ (\lambda y.\ (x\ y)\ x)$
- Examples (taken from 2015 exams):
  - (λx. λx. (λx.x) x) ((λx. x x) λx.x)
  - (λt. λf. t) (λx.x) ((λx.x) (λs. λz. s z))

Free vs. Bound Variables

- An occurrence of x in t is bound in λx. t
  - otherwise it is free
  - λx is a binder
- Examples
  - λx. x
  - λy. x (y z)
  - λz. λx. λy. x (y z)
  - (λx. x) x

FV: t $\rightarrow$ P(Var) is the set free variables of t

FV(x) = {x}

FV(λ x. t) = FV(t) – {x}

FV (t_1 t_2) = FV(t_1) $\cup$ FV(t_2)

Semantics: β-reduction, Substitution

- Substitution

  $[x \mapsto s]\ x = s$

  $[x \mapsto s]\ y = y$ if $y \neq x$

  $[x \mapsto s]\ (\lambda y.\ t_1) = \lambda y.\ [x \mapsto s]\ t_1$ if $y \neq x$ and $y \notin FV(s)$

  $[x \mapsto s]\ (t_1\ t_2) = ([x \mapsto s]\ t_1)\ ([x \mapsto s]\ t_2)$

- β-reduction

  $(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$

Beta-Reduction: Examples

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

redex

$(\lambda\ x.\ x)\ y \Rightarrow_\beta\ y$

$(\lambda\ x.\ x\ (\lambda\ x.\ x)\ )\ (u\ r) \Rightarrow_\beta\ u\ r\ (\lambda\ x.\ x)$

$(\lambda\ x\ (\lambda w.\ x\ w))\ (y\ z) \Rightarrow_\beta\ \lambda w.\ y\ z\ w$

Substitution Subtleties

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

$[x \mapsto s]\ x = s$

$[x \mapsto s]\ y = y$ if $y \neq x$

$[x \mapsto s]\ (\lambda y.\ t_1) = \lambda y.\ [x \mapsto s]\ t_1$ **if $y \neq x$ and $y \notin FV(s)$**

$[x \mapsto s]\ (t_1\ t_2) = ([x \mapsto s]\ t_1)\ ([x \mapsto s]\ t_2)$

$(\lambda x.\ (\lambda x.\ x))\ y \Rightarrow_\beta [x \mapsto y]\ (\lambda x.\ x) = \lambda x.\ y$?

$(\lambda x.\ (\lambda y.\ x))\ y \Rightarrow_\beta [x \mapsto y]\ (\lambda y.\ x) = \lambda y.\ y$?

$(\lambda x.\ (\lambda x.\ x))\ y$ and $(\lambda x.\ (\lambda y.\ x))\ y$ are stuck! They have no β-reduction

Alpha – Conversion

α-conversion:
Renaming of a bound variable and its bound occurrences

$$(\lambda x.\ t) \Rightarrow_\alpha \lambda y.\ [x \mapsto y]\ t \qquad \text{if } y \notin FV(t)$$

$$(\lambda x.\ (\lambda x.\ x))\ y \Rightarrow_\alpha (\lambda x.\ (\lambda z.\ z))\ y \Rightarrow_\beta [x \mapsto y]\ (\lambda z.\ z) = \begin{array}{c} \lambda x.\ x \\ \Uparrow \alpha \\ \lambda z.\ z \end{array} \neq \lambda x.\ y$$

$$(\lambda x.\ (\lambda y.\ x))\ y \Rightarrow_\alpha (\lambda x.\ (\lambda z.\ x))\ y \Rightarrow_\beta [x \mapsto y]\ (\lambda z.\ x) = \lambda z.\ y \neq \lambda y.\ y$$

Currying – Multiple Arguments

- Say we want to define a function with two arguments:
  - "f = λ(x, y). s"
- We do this by Currying:
  - f = λx. λy. s
  - f is now "a function of x that returns a function of y"
- Currying and β-reduction:

$$f\ v\ w = (f\ v)\ w = ((\lambda x.\ \lambda y.\ s)\ v)\ w$$

$$\Rightarrow (\ \lambda y.[x \mapsto v]s\ )\ w \Rightarrow [x \mapsto v]\ [y \mapsto w]\ s$$

- Conclusion:
  - "f = λ(x, y). s" → f = λx. λy. s
  - "f (v,w)" → f v w

Church Booleans

- Define: tru = λt. λf. t fls = λt. λf. f test = λl. λm. λn. l m n
- test tru then else = (λl. λm. λn. l m n) (λt. λf. t) then else
  ⇒(λm. λn. (λt. λf. t) m n) then else
  ⇒(λn. (λt. λf. t) then n) else
  ⇒(λt. λf. t) then else
  ⇒(λf. then) else
  ⇒then
- test fls then else = (λl. λm. λn. l m n) (λt. λf. f) then else
  ⇒(λm. λn. (λt. λf. f) m n) then else
  ⇒(λn. (λt. λf. f) then n) else
  ⇒(λt. λf. f) then else
  ⇒(λf. f) else
  ⇒else
- and = λb. λc. b c fls
- or =
- not =

Church Numerals

- $c_0 = \lambda s.\ \lambda z.\ z$
- $c_1 = \lambda s.\ \lambda z.\ s\ z$
- $c_2 = \lambda s.\ \lambda z.\ s\ (s\ z)$
- $c_3 = \lambda s.\ \lambda z.\ s\ (s\ (s\ z))$
- …
- $scc = \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)$
- $plus = \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z)$
- $times = \lambda m.\ \lambda n.\ m\ (plus\ n)\ c_0$
- iszero =

Non-Deterministic Operational Semantics

(E-AppAbs) $(\lambda\, x.\, t_1)\; t_2 \Rightarrow [x \mapsto t_2]\; t_1$

$$\frac{t \Rightarrow t'}{\lambda\, x.\, t \Rightarrow \lambda\, x.\, t'}$$ (E-Abs)

(E-App$_1$) $$\frac{t_1 \Rightarrow t'_1}{t_1\;\; t_2 \Rightarrow t'_1\; t_2}$$

$$\frac{t_2 \Rightarrow t'_2}{t_1\;\; t_2 \Rightarrow t_1\; t'_2}$$ (E-App$_2$)

Why is this semantics non-deterministic?

Different Evaluation Orders

(E-AppAbs) $(\lambda\, x.\, t_1)\; t_2 \Rightarrow [x \mapsto t_2]\; t_1$

$$\frac{t \Rightarrow t'}{\lambda\, x.\, t \Rightarrow \lambda\, x.\, t'}$$ (E-Abs)

(E-App$_1$) $$\frac{t_1 \Rightarrow t'_1}{t_1\;\; t_2 \Rightarrow t'_1\; t_2}$$

$$\frac{t_2 \Rightarrow t'_2}{t_1\;\; t_2 \Rightarrow t_1\; t'_2}$$ (E-App$_2$)

$(\lambda x.\ (\text{add } x\ x))\ (\text{add } 2\ 3) \Rightarrow (\lambda x.\ (\text{add } x\ x))\ (5) \Rightarrow \text{add } 5\ 5 \Rightarrow 10$

$(\lambda x.\ (\text{add } x\ x))\ (\text{add } 2\ 3) \Rightarrow (\text{add } (\text{add } 2\ 3)\ (\text{add } 2\ 3)) \Rightarrow$

$(\text{add } 5\ (\text{add } 2\ 3)) \Rightarrow \quad (\text{add } 5\ 5) \Rightarrow \ 10$

This example: same final result but lazy performs more computations

Different Evaluation Orders

(E-AppAbs) $(\lambda\, x.\, t_1)\; t_2 \Rightarrow [x \mapsto t_2]\; t_1$

$$\frac{t \Rightarrow t'}{\lambda\, x.\, t \Rightarrow \lambda\, x.\, t'}\ \text{(E-Abs)}$$

$$\text{(E-App}_1\text{)}\quad \frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2} \qquad \frac{t_2 \Rightarrow t'_2}{t_1\ t_2 \Rightarrow t_1\ t'_2}\ \text{(E-App}_2\text{)}$$

$(\lambda x.\, \lambda y.\, x)\ 3\ (\text{div}\ 5\ 0) \Rightarrow (\lambda y.\, 3)\ (\text{div}\ 5\ 0) \Rightarrow$ Exception: Division by zero

$(\lambda x.\, \lambda y.\, x)\ 3\ (\text{div}\ 5\ 0) \Rightarrow (\lambda y.\, 3)\ (\text{div}\ 5\ 0) \Rightarrow 3$

This example: lazy suppresses erroneous division and reduces to final result

Can also suppress non-terminating computation.

Many times we want this, for example:

```
if i < len(a) and a[i]==0: print “found zero”
```

Strict (Applicative)

(E-App$_1$)

$$\frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$$

precedence

(E-App$_2$)

$$\frac{t_2 \Rightarrow t'_2}{t_1\ t_2 \Rightarrow t_1\ t'_2}$$

precedence

(E-AppAbs)

$$(\lambda\, x.\, t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$$

Lazy

(E-AppAbs)

$$(\lambda\, x.\, t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$$

(E-App$_1$)

$$\frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$$

Normal Order

(E-AppAbs)

$$(\lambda\, x.\, t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$$

precedence

(E-App$_1$)

$$\frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$$

precedence

(E-App$_2$)

$$\frac{t_2 \Rightarrow t'_2}{t_1\ t_2 \Rightarrow t_1\ t'_2}$$

(E-Abs)

$$\frac{t \Rightarrow t'}{\lambda\, x.\, t \Rightarrow \lambda\, x.\, t'}$$

Call-by-value Operations Semantics via Inductive Definition (no precedence)

t ::=	terms	v ::= λ x. t	abstraction values
x	variable		
λ x. t	abstraction		
t t	application		

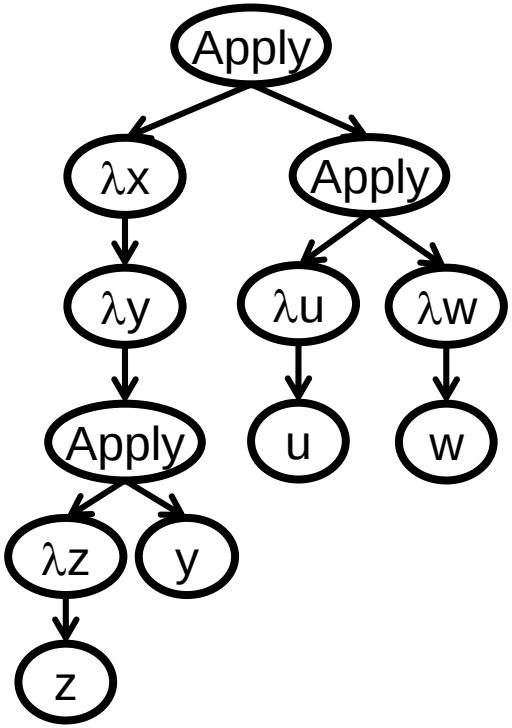
$$(\lambda\ x.\ t_1)\ v_2 \Rightarrow [x \mapsto v_2]\ t_1 \qquad \text{(E-AppAbs)}$$

$$\frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2} \qquad \text{(E-APPL1)}$$

$$\frac{t_2 \Rightarrow t'_2}{v_1\ t_2 \Rightarrow v_1\ t'_2} \qquad \text{(E-APPL2)}$$

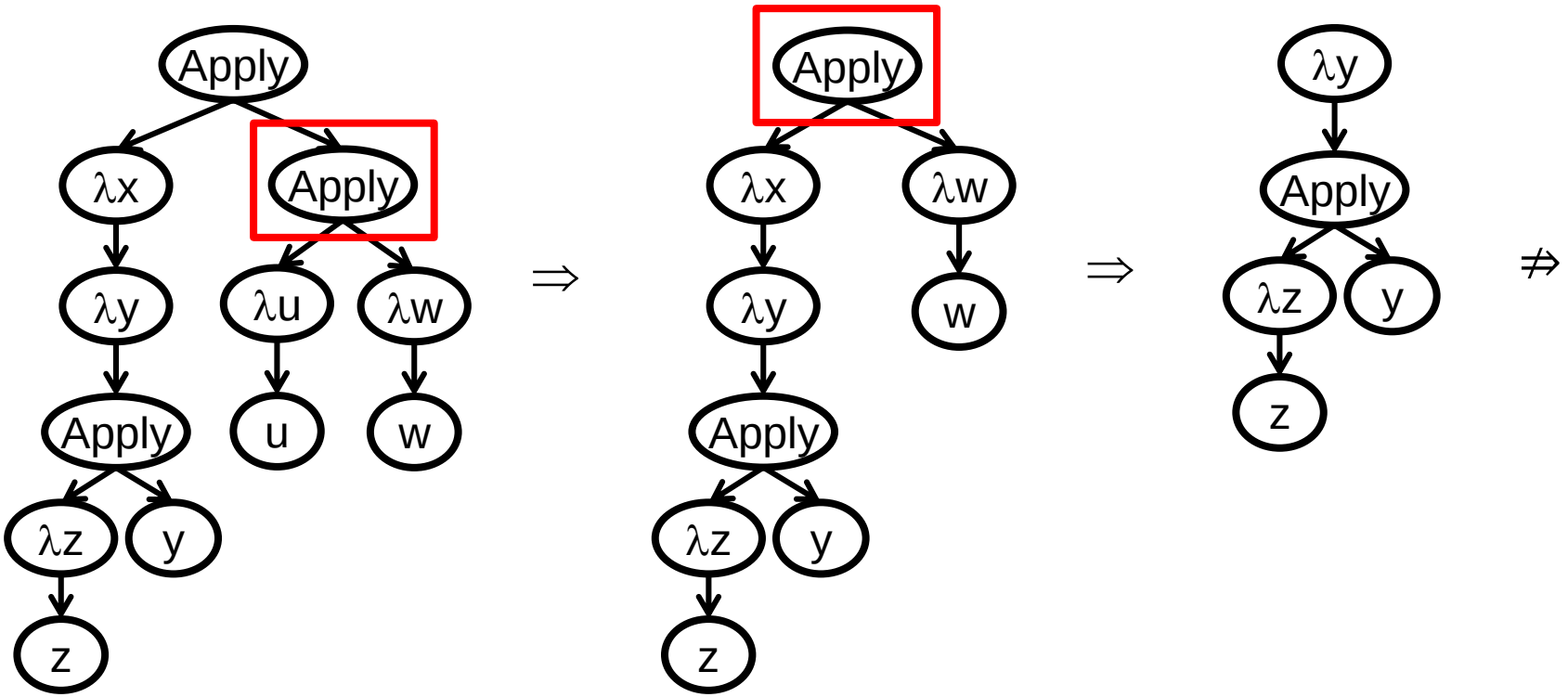
Different Evaluation Orders - Example

(λ x. λ y. (λ z. z) y) ((λ u. u) (λ w. w))



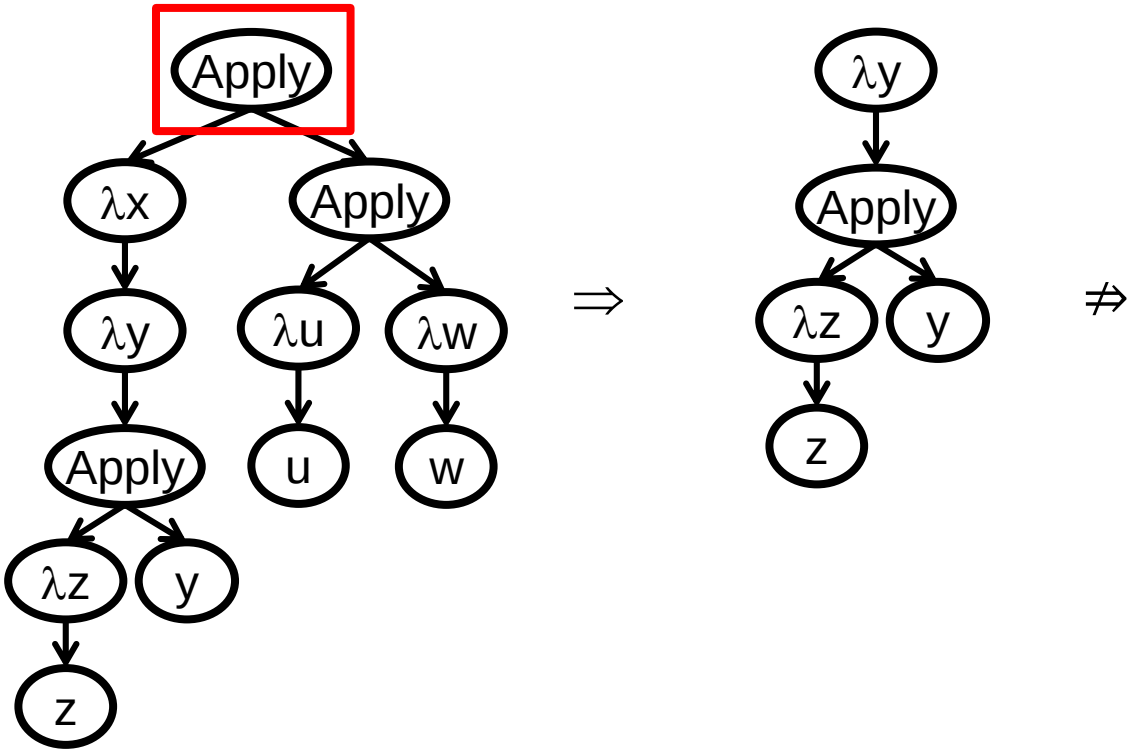
Call By Value

(λ x. λ y. (λ z. z) y) ((λ u. u) (λ w. w))



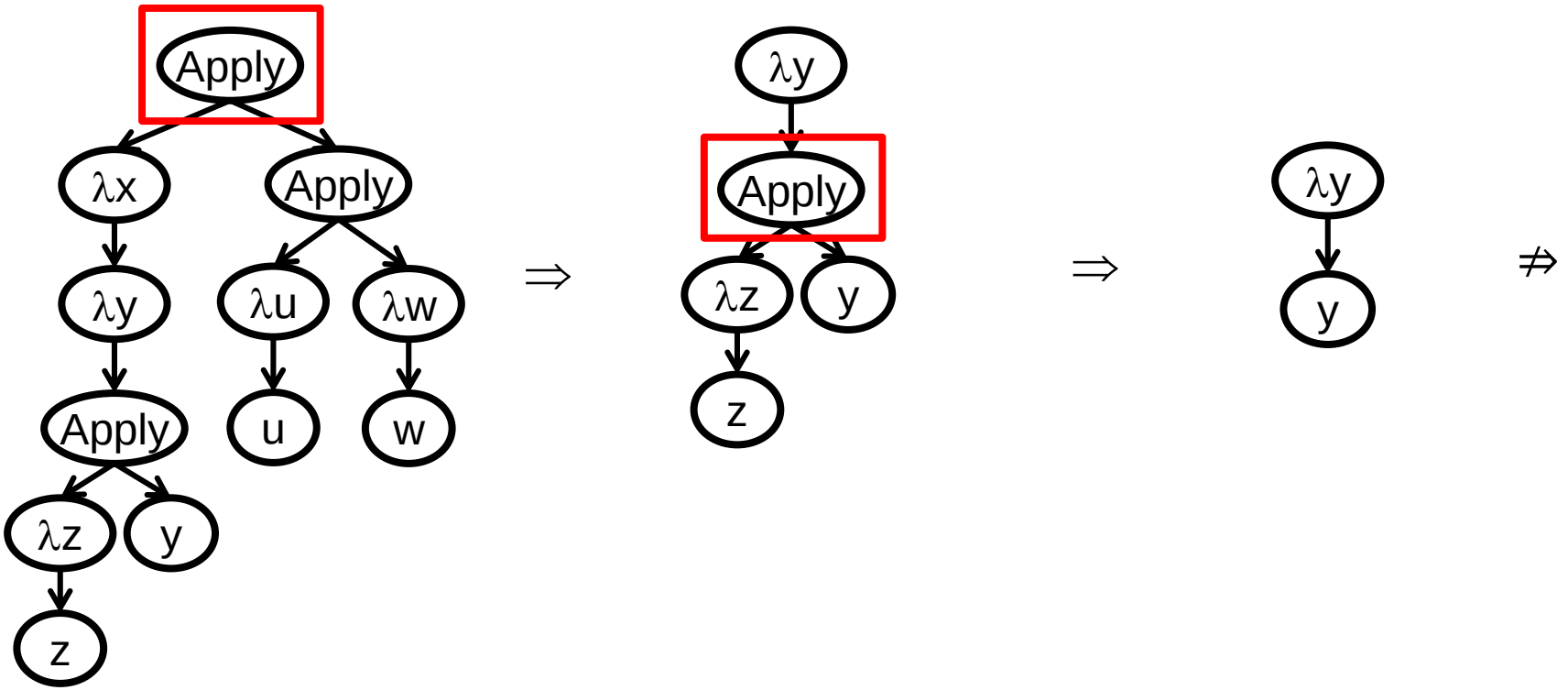
Call By Name (Lazy)

(λ x. λ y. (λ z. z) y) ((λ u. u) (λ w. w))



Normal Order

(λ x. λ y. (λ z. z) y) ((λ u. u) (λ w. w))



Summary: Orders of Evaluation

- Full-beta-reduction, Non-deterministic semantics
  - All possible orders
- Call by value, Eager, Strict, Applicative order
  - Left to right
  - Fully evaluate arguments before function
- Normal order
  - The leftmost, outermost redex is always reduced first
- Call by name, Lazy
  - Evaluate arguments as needed
- Call by need
  - Evaluate arguments as needed and store for subsequent usages
  - Implemented in Haskell