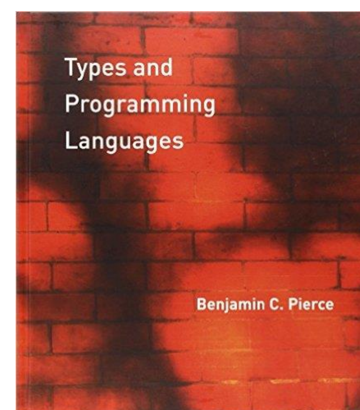
Concepts in Programming Languages – Recitation 4:

Untyped Lambda Calculus

Oded Padon & Mooly Sagiv

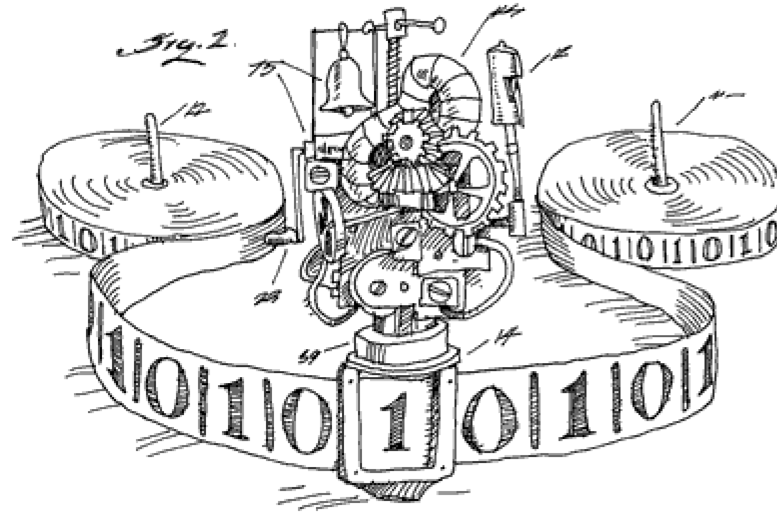
(original slides by Kathleen Fisher, John Mitchell, Shachar Itzhaky, S. Tanimoto)

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



Computation Models

- Turing Machines
- Wang Machines
- Counter Programs
- Lambda Calculus



Historical Context

Like Alan Turing, another mathematician, Alonzo Church, was very interested, during the 1930s, in the question “What is a computable function?”

He developed a formal system known as the pure lambda calculus, in order to describe programs in a simple and precise way.

Today the Lambda Calculus serves as a mathematical foundation for the study of functional programming languages, and especially for the study of “denotational semantics.”

Reference: http://en.wikipedia.org/wiki/Lambda_calculus

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):
  - $(\lambda x.\ \lambda x.\ (\lambda x.x)\ x)\ ((\lambda x.\ x\ x)\ \lambda x.x)$
  - $(\lambda t.\ \lambda f.\ t)\ (\lambda x.x)\ ((\lambda x.x)\ (\lambda s.\ \lambda z.\ s\ z))$

Free vs. Bound Variables

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

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

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

$FV(\lambda\ x.\ t) = FV(t) - \{x\}$

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

Semantics: Substitution, β-reduction, α-conversion

- 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

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

- α-conversion

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

Beta-Reduction: Examples

$$\underbrace{(\lambda\ x.\ t_1)\ t_2}_{\text{redex}} \Rightarrow_\beta [x \mapsto t_2]\ t_1 \qquad (\beta\text{-reduction})$$

$$\underline{(\lambda\ x.\ x)\ y} \Rightarrow_\beta \quad y$$

$$\underline{(\lambda\ x.\ x\ (\lambda\ x.\ x)\ )\ (u\ r)} \Rightarrow_\beta \quad u\ r\ (\lambda\ x.\ x)$$

$$\underline{(\lambda\ x\ (\lambda w.\ x\ w))\ (y\ z)} \Rightarrow_\beta \quad \lambda w.\ y\ z\ w$$

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 $y \notin FV(s)$**

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

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

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

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

Alpha – Conversion

Alpha conversion:
Renaming of a bound variable and its bound occurrences

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

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

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

Non-Deterministic Operational Semantics

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

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

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

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

Why is this semantics non-deterministic?

Different Evaluation Orders

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

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

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

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

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

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

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

This example: same final result but lazy performs more computations

Different Evaluation Orders

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

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

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

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

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

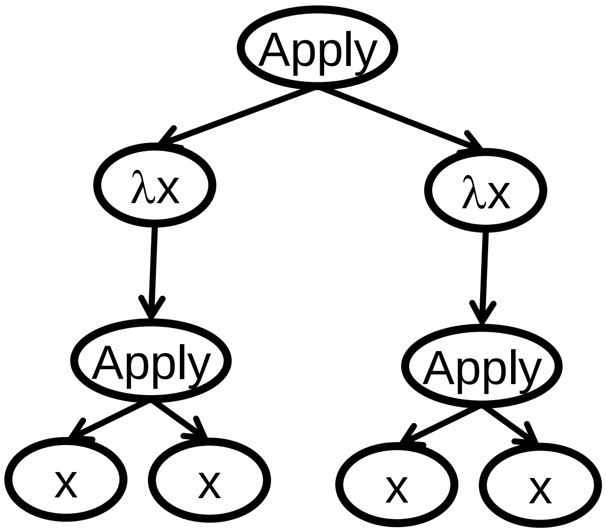
Many times we want this, for example:

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

Divergence

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

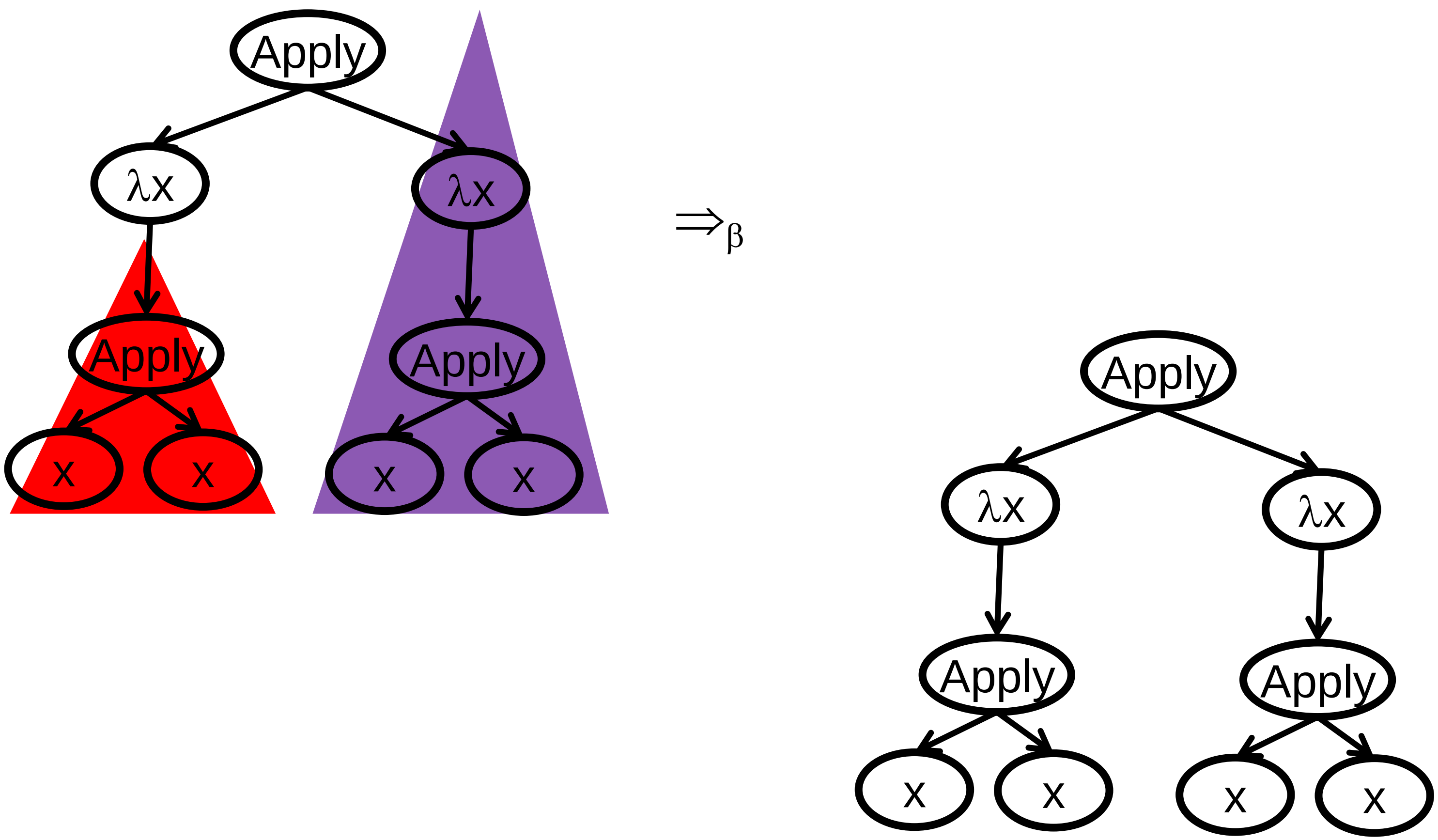
$(\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x))$



Divergence

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

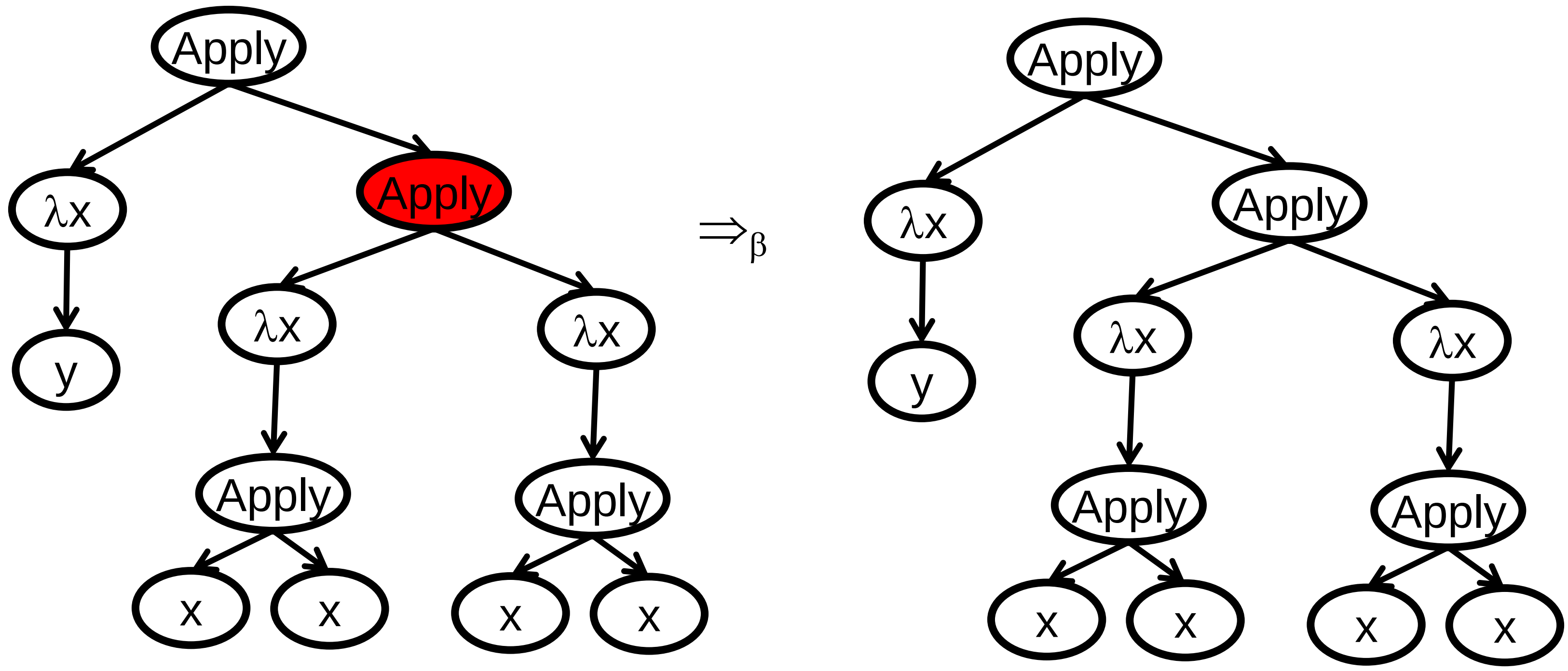
$(\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x))$



Different Evaluation Orders

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

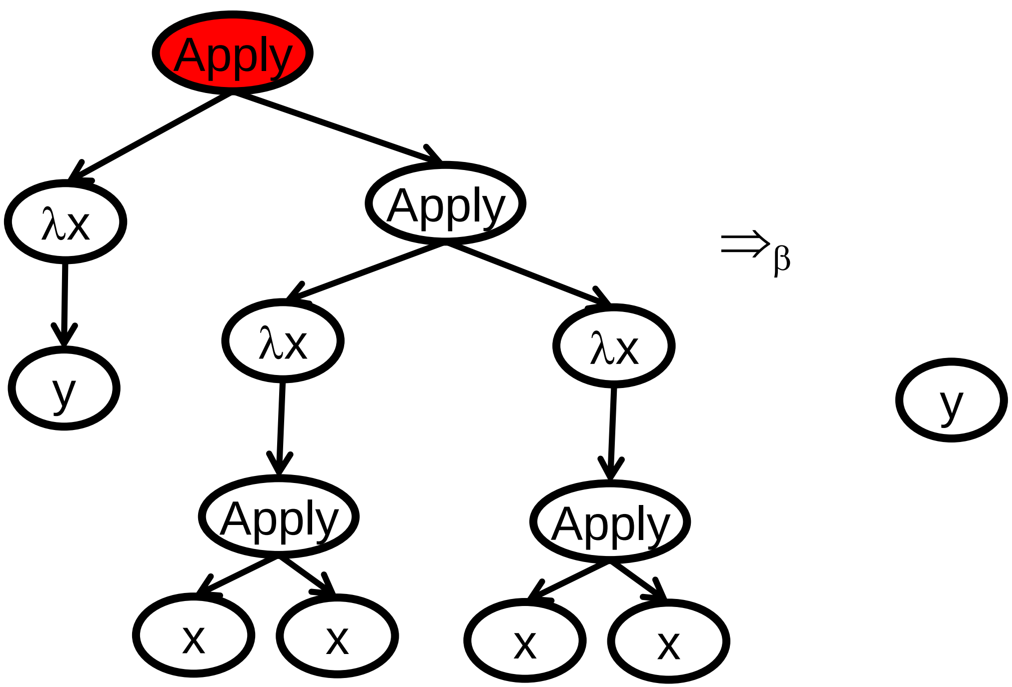
$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$



Different Evaluation Orders

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

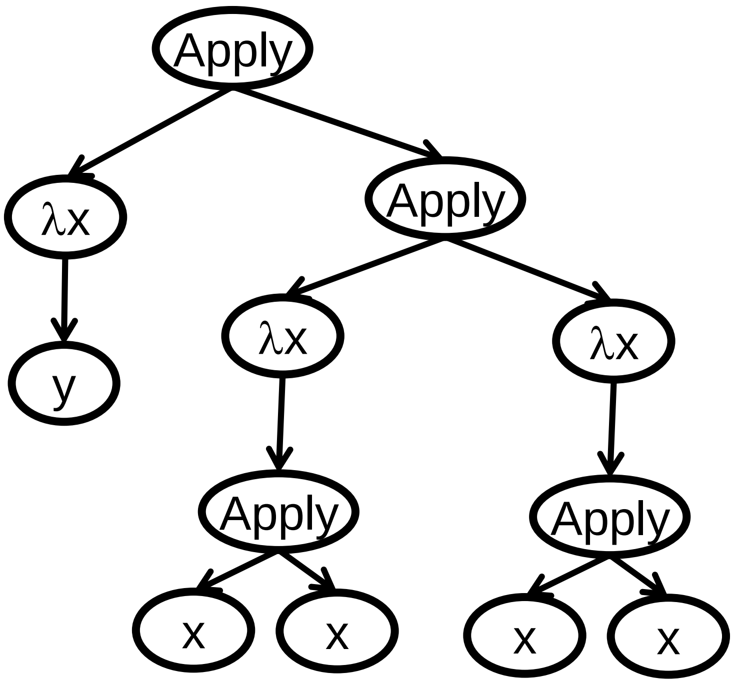
$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$



Different Evaluation Orders

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

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



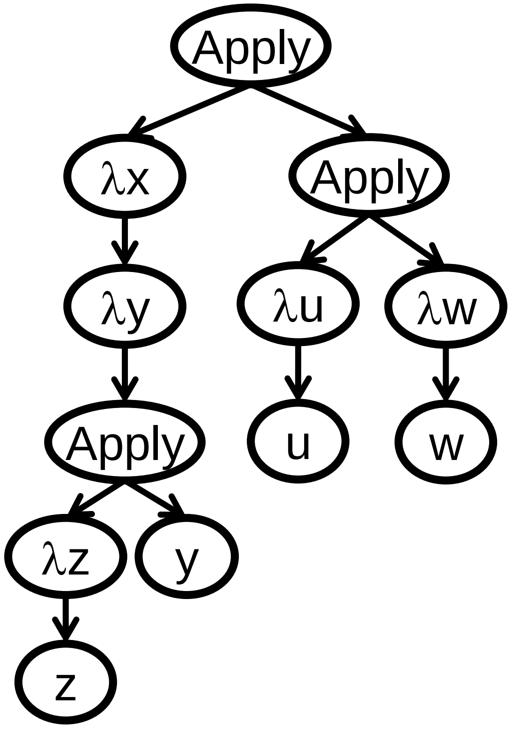

```
def f():
    while True: pass

def g(x):
    return 2

print g(f())
```

Summary Order of Evaluation

- Full-beta-reduction
  - All possible orders
- Applicative order call by value (strict)
  - 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

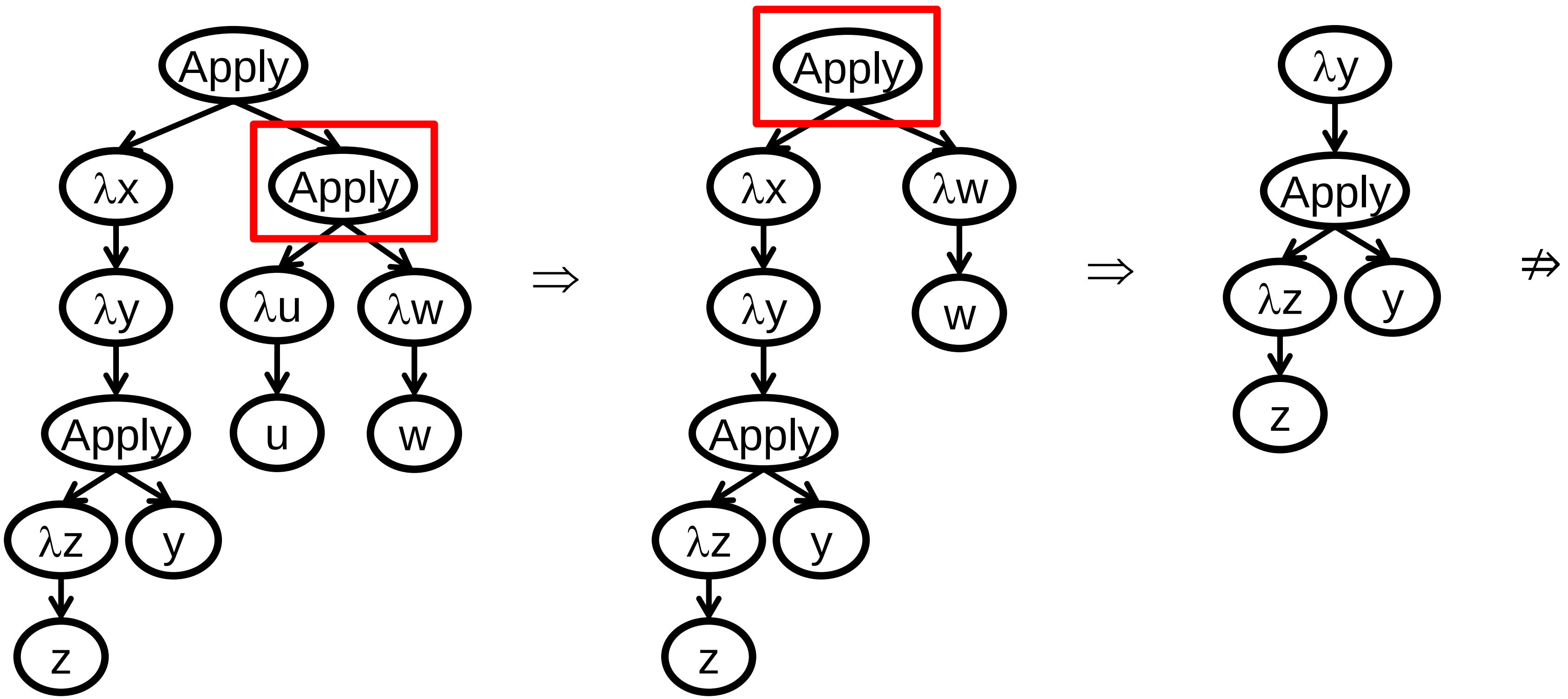
Different Evaluation Orders

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



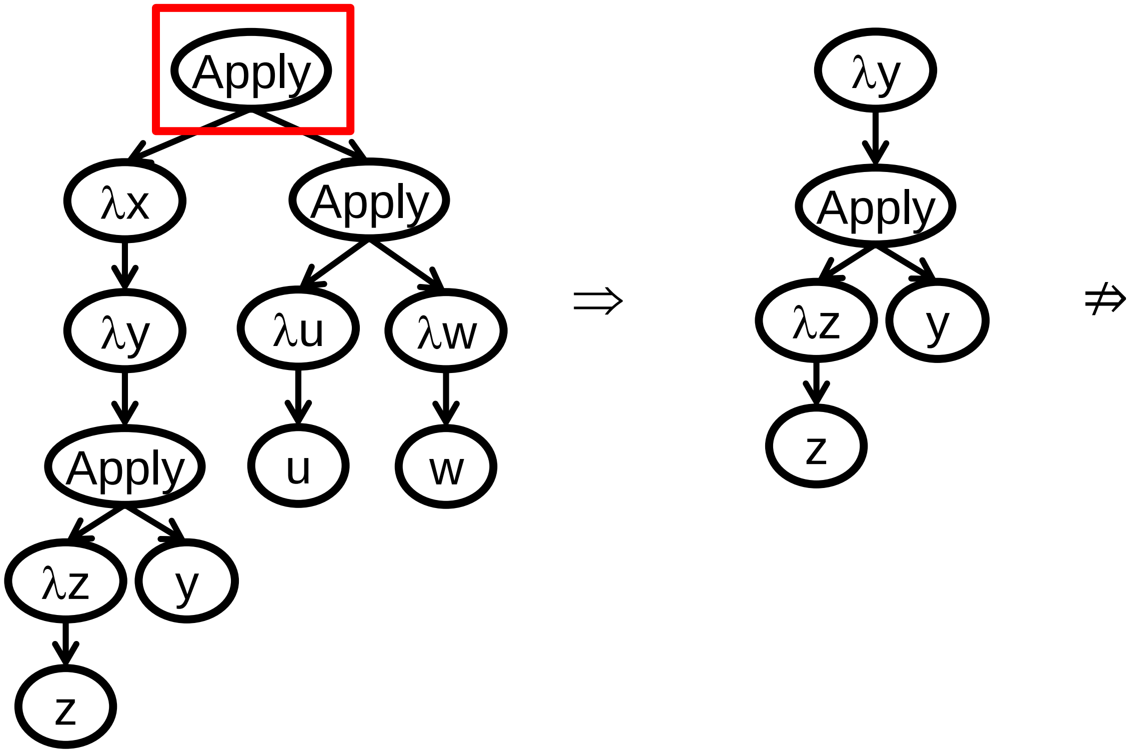
Call By Value

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



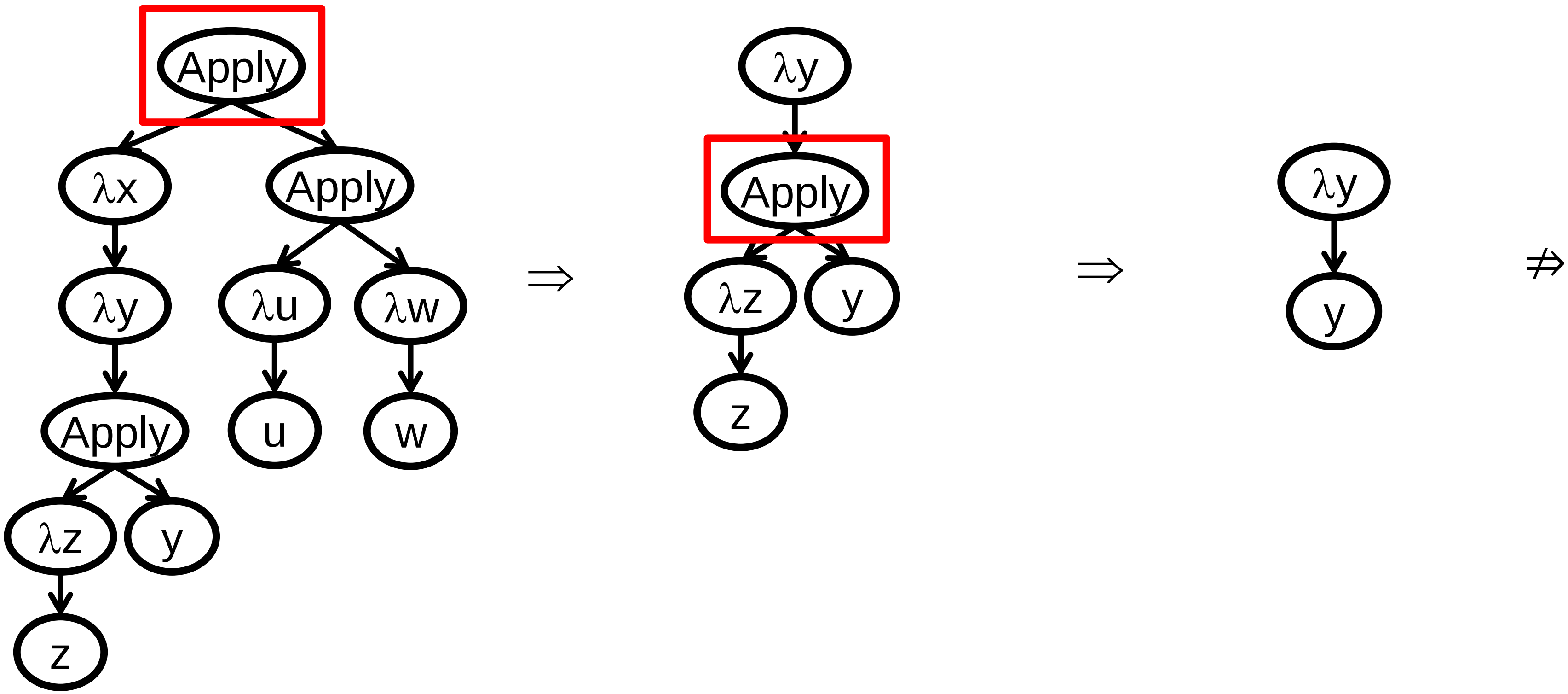
Call By Name (Lazy)

(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))



Call-by-value Operational Semantics

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

v ::=	values
λ x. t	abstraction values
other values	

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

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

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

Currying – Multiple arguments

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

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

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

- Conclusion:
  - "$f = \lambda(x, y).\ s$" $\rightarrow$ $f = \lambda x.\ \lambda y.\ s$
  - "$f\ (v,w)$" $\rightarrow$ $f\ v\ w$

Church Booleans

- Define: $tru = \lambda t.\ \lambda f.\ t$ $\quad fls = \lambda t.\ \lambda f.\ f$ $\quad test = \lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n$
- $test\ tru$ then else $= (\lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n)\ (\lambda t.\ \lambda f.\ t)$ then else

  $\Rightarrow(\lambda m.\ \lambda n.\ (\lambda t.\ \lambda f.\ t)\ m\ n)$ then else

  $\Rightarrow(\lambda n.\ (\lambda t.\ \lambda f.\ t)$ then $n)$ else

  $\Rightarrow(\lambda t.\ \lambda f.\ t)$ then else

  $\Rightarrow(\lambda f.$ then$)$ else

  $\Rightarrow$then
- $test\ fls$ then else $= (\lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n)\ (\lambda t.\ \lambda f.\ f)$ then else

  $\Rightarrow(\lambda m.\ \lambda n.\ (\lambda t.\ \lambda f.\ f)\ m\ n)$ then else

  $\Rightarrow(\lambda n.\ (\lambda t.\ \lambda f.\ f)$ then $n)$ else

  $\Rightarrow(\lambda t.\ \lambda f.\ f)$ then else

  $\Rightarrow(\lambda f.\ f)$ else

  $\Rightarrow$else
- $and = \lambda b.\ \lambda c.\ b\ c\ fls$
- 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 =

Combinators

- A combinator is a function in the Lambda Calculus having no free variables
- Examples
  - $\lambda x.\ x$ is a combinator
  - $\lambda x.\ \lambda y.\ (x\ y)$ is a combinator
  - $\lambda x.\ \lambda y.\ (x\ z)$ is not a combinator
- Combinators can serve nicely as modular building blocks for more complex expressions
- The Church numerals and simulated Booleans are examples of useful combinators

Iteration in Lambda Calculus

- omega = $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$
  - $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \Rightarrow (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

- Y = $\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$
- Z = $\lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))$
- Recursion can be simulated
  - Y only works with call-by-name semantics
  - Z works with call-by-value semantics
- Defining factorial:
  - g = $\lambda f.\ \lambda n.$ if n==0 then 1 else (n * (f (n - 1)))
  - fact = Y g (for call-by-name)
  - fact = Z g (for call-by-value)