# Lecture 6: λ-calculus II

Yu Feng
Winter 2025

# Precedence

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

$$s \ t \ u = (s \ t) \ u$$

apply
apply u
s t

λx
λy
apply
apply x
x y
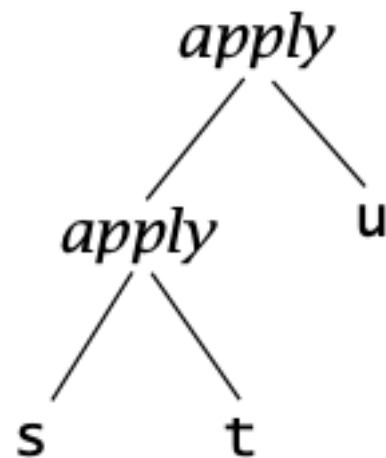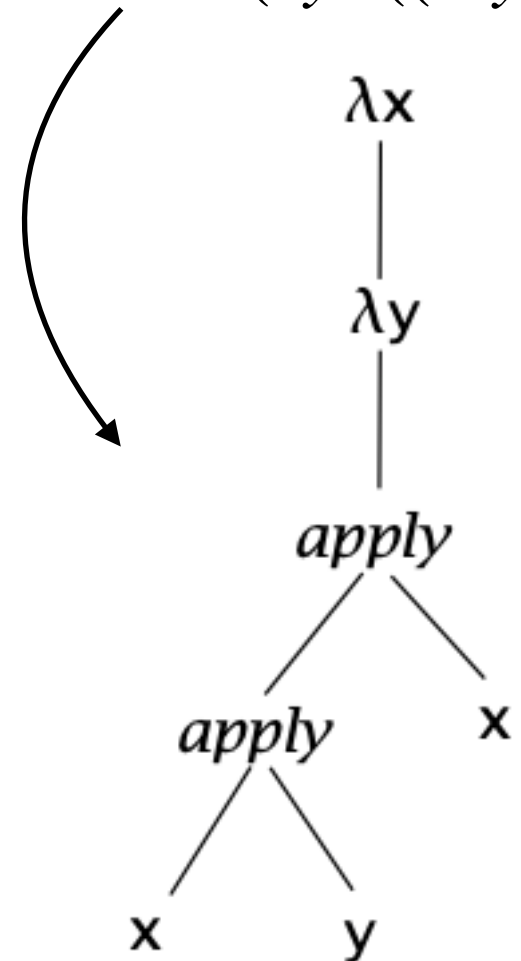
Application associates to the **left**

bodies of abstractions are as far to the **right** as possible

*—Types and programming languages*

# Semantics: what programs mean

- How do I execute a $\lambda$-term?

- "Execute": rewrite step-by-step following simple rules, until no more rules apply

Similar to simplifying $(x+1) * (2x - 2)$ using middle-school algebra

$e ::= x$
$\quad | \ \lambda x.\, e$
$\quad | \ e_1 \ e_2$

**What are the rewrite rules for $\lambda$-calculus?**

# Operational semantics

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

β-reduction (function call)

$[x \mapsto t_2]t_1$ means "t₁ with all **free occurrences** of x replaced with t₂"

```
inc1(int x ) {
  return x+1
}

inc1(2);
```

$$(\lambda x \, . \, x + 1) \; 2 \rightarrow [x \mapsto 2]x + 1 = 3$$

$$[x \mapsto y]\lambda x \, . \, x = \lambda x \, . \, y \quad \otimes$$

**What does free occurrences mean?**

# Semantics: β-reduction

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

β-reduction (function call)

$[x \mapsto t_2]t_1$ means "$t_1$ with all ***free occurrences*** of x replaced with $t_2$"

The core of β-reduction reduces to substitution:

$$[x \mapsto s]x = s$$
$$[x \mapsto s]y = y(x \neq y)$$
$$[x \mapsto s]\lambda y \,.\, t_1 = \lambda y \,.\, [x \mapsto s]t_1(y \neq x \wedge y \notin FV(s))$$
$$[x \mapsto s]t_1\ t_2 = [x \mapsto s]t_1\ [x \mapsto s]t_2$$

# Semantics: α-renaming

$$\lambda x \,.\, e =_\alpha \lambda y \,.\, [x \mapsto y]e$$

- Rename a formal parameter and replace all its occurrences in the body

$$\lambda x \,.\, x =_\alpha \lambda y \,.\, y =_\alpha \lambda z \,.\, z$$

$$[x \mapsto y]\lambda x \,.\, x = \lambda x \,.\, y \quad ❌$$

$$[x \mapsto y]\lambda x \,.\, x =_\alpha [x \mapsto y]\lambda z \,.\, z = \lambda z \,.\, z \quad ✅$$

# Call-by-name v.s. Call-by-value

$$(\lambda x . e_1) \; e_2 =_{\text{name}} [x \mapsto e_2]e_1$$

Call-by-Name: From leftmost/outermost, allowing **no reductions** inside abstractions.

$$(\lambda x . e_1) \; e_2 =_{\text{value}} [x \mapsto [e_2]]e_1$$

Call-by-Value: only when its right-hand side has already been reduced to a value—a term that **cannot be reduced any further**

# Currying: multiple arguments

$$\lambda(x, y) \,.\, e = \lambda x \,.\, \lambda y \,.\, e$$

$(\lambda(x, y) \,.\, x + y)\ 2\ 3 =$

$(\lambda x \,.\, \lambda y \,.\, x + y)\ 2\ 3 = (\lambda y.2 + y)\ 3 = [y \mapsto 3]2 + y = 5$

Transformation of multi-arguments functions to higher-order functions is called currying (in the honor of Haskell Curry)

# Putting all together

| Substitution $c[v \mapsto e]$ | Alpha-Renaming Necessary? | Output |
|---|---|---|
| $(x\ (\lambda x.\ y))[y \mapsto x]$ | | |
| $((\lambda y.\ x)\ (\lambda z.\ y))[y \mapsto x]$ | | |
| $(\lambda z.\ ((\lambda y.\ y)\ y))[y \mapsto \lambda x.(y\ z)]$ | | |

# What about the others?

- ~~Assignment~~

- ~~Booleans, integers, characters, strings, …~~

- ~~Conditionals~~

- ~~Loops~~

- Functions

- ~~Recursion~~

- ~~References / pointers~~

- ~~Objects and classes~~

- ~~Inheritance~~

# $\lambda$-calculus:Booleans

- How do we encode Boolean values (TRUE and FALSE) as functions?

- What do we do with Boolean?

- Make a binary choice

  - if b then e1 else e2

# Booleans: API

We need to define three functions

- let TRUE  = ???

- let FALSE = ???

- let ITE   = $\lambda$b x y -> ???  -- if b then x else y

such that

- ITE TRUE apple banana = apple

- ITE FALSE apple banana = banana

# Booleans: implementation

Boolean implementation

- let TRUE = $\lambda x\, y.\, x$    -- Returns its first argument

- let FALSE = $\lambda x\, y.\, y$    -- Returns its second argument

- let ITE = $\lambda b\, x\, y.\, b\, x\, y$   -- Applies condition to branches

Why they are correct?

# Booleans: examples

eval ite_true:

ITE  TRUE  $e_1$ $e_2$

$= (\lambda b\ x\ y.\ b\ \ x\ \ y)$  TRUE  $e_1$ $e_2$    -- expand def ITE

$=_\beta$   $(\lambda x\ y.\ TRUE\ x\ \ y)$        $e_1$ $e_2$    -- beta-step

$=_\beta$     $(\lambda y.\ TRUE\ e_1\ y)$        $e_2$    -- beta-step

$=_\beta$           TRUE $e_1$ $e_2$              -- expand def TRUE

$=$    $(\lambda x\ y.\ \ x)$ $e_1$ $e_2$            -- beta-step

$=_\beta$      $(\lambda y.\ e_1)$   $e_2$          -- beta-step

$=_\beta$    $e_1$

Other boolean API:

let NOT = $\lambda b.$ ITE b FALSE TRUE

let AND = $\lambda b_1\ b_2.$ ITE $b_1$ $b_2$ FALSE

let OR  = $\lambda b_1\ b_2.$ ITE $b_1$ TRUE $b_2$

# $\lambda$-calculus:Numbers

- Church numerals: a number N is encoded as a combinator that calls a function on an argument N times

let ONE   = $\lambda$f $\lambda$x. f x

let TWO   = $\lambda$f $\lambda$x. f (f x)

let THREE = $\lambda$f $\lambda$x. f (f (f x))          let ZERO   = $\lambda$f $\lambda$x. x

let FOUR  = $\lambda$f $\lambda$x. f (f (f (f x)))

let FIVE  = $\lambda$f $\lambda$x. f (f (f (f (f x))))

let SIX   = $\lambda$f $\lambda$x. f (f (f (f (f (f x)))))

# $\lambda$-calculus:Numbers API

- Numbers API

- let INC = $(\lambda n \; \lambda f \; \lambda x . \; f \; (n \; f \; x))$    -- Call `f` on `x` one more time than `n` does

- let ADD = $\lambda n \; \lambda m . \; n$ INC m. -- Call `f` on `x` exactly `n + m` times

eval inc_zero :
 INC ZERO
 $= (\lambda n \; \lambda f \; \lambda x . \; f \; (n \; f \; x))$ ZERO
 $=_\beta \lambda f \; \lambda x . \; f \; (ZERO \; f \; x)$
 $= \lambda f \; \lambda x . \; f \; x$
 = ONE

eval add_one_zero :
ADD ONE ZERO = ONE

# TODOs by next lecture

- Install $\lambda$+

- Start to work on HW2

- Come to the discussion session if you have questions