CS 162 Programming languages

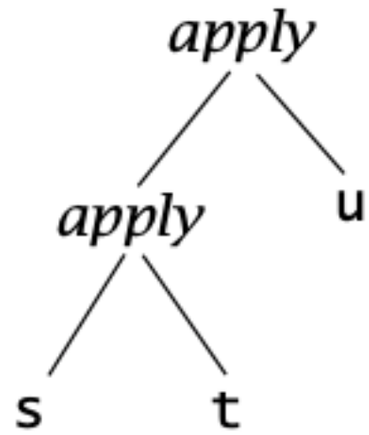
Lecture 6: λ-calculus II

Yu Feng

Winter 2026

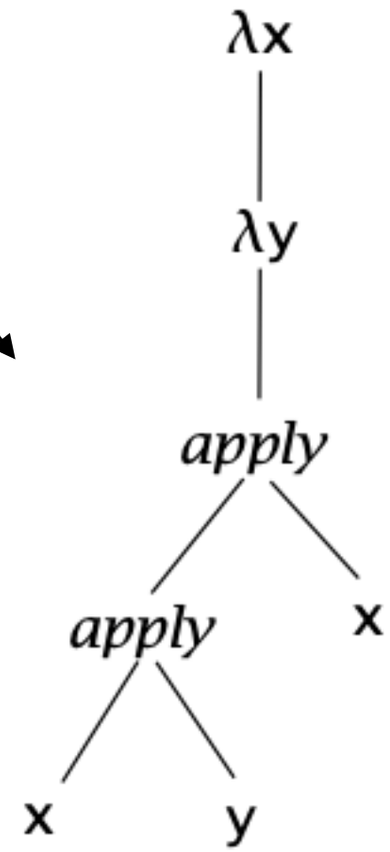
Precedence

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



Application associates to the **left**

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



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

—Types and programming languages

Semantics: what programs mean

- How do I execute a λ-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
| λx. e
| e_1 e_2

What are the rewrite rules for λ-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 "t_1 with all ***free* occurrences** of x replaced with t_2"

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

inc1(2);
```

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

$$[x \mapsto y]\lambda x\,.\,x = \lambda x\,.\,y$$ ✖

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:

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

$$[x \mapsto s]y = y (x \neq y)$$

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

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

`(λx. t1) t2`

> “Apply the function `λx. t1` to argument `t2`”

`(λx. t1) t2 → [x ↦ t2] t1`

> Replace the parameter `x` in `t1` with the argument `t2`

What is `[x ↦ t2] t1`?

> Take t1 and replace **all free occurrences** of x with t2

Free vs Bound Variables

- **Bound variable**: introduced by λ
- **Free variable**: not bound by any λ

λx. x y // x: bound, y: free

Substitution Rule 1: Matching Variable

```
[x ↦ s] x = s

[x ↦ 5] x = 5
```

Substitution Rule 2: Different Variable

```
[x ↦ s] y = y    (x ≠ y)
[x ↦ 5] y = y
```

➡️ If it's not the variable we're replacing, do nothing

Substitution Rule 3: Application

```
[x ↦ s] (t1 t2)
  = ([x ↦ s] t1) ([x ↦ s] t2)

[x ↦ 3] (x y) = (3 y)
```

➡️ Substitute in **both parts**

Substitution Rule 4: Lambda (Careful!)

```
[x ↦ s] (λy. t)
  = λy. [x ↦ s] t
```

➡️ But only if substitution is **safe**.

Why Safety Matters (Variable Capture)

Consider:

[x ↦ y] (λy. x)

Naive substitution gives:

λy. y ❌

What went wrong?

- The free y became **accidentally bound**
- This changes the meaning of the program

This is why we require: $y \neq x$ and $y \notin FV(s)$

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

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

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

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

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

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

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

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

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**
$(x\ (\lambda x.\ y))[y \mapsto x]$		
$((\lambda y.\ x)\ (\lambda z.\ y))[y \mapsto x]$		
$(\lambda z.\ ((\lambda y.\ y)\ y))[y \mapsto \lambda x.(y\ z)]$		

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

$$[x \mapsto s]y = y (x \neq y)$$

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

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

What about the others?

- ~~Assignment~~
- ~~Booleans, integers, characters, strings, …~~
- ~~Conditionals~~
- ~~Loops~~
- Functions
- ~~Recursion~~
- ~~References / pointers~~
- ~~Objects and classes~~
- ~~Inheritance~~

λ-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 = λb 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 = λx y. x -- Returns its first argument
- let FALSE = λx y. y -- Returns its second argument
- let ITE = λb x y. b x y -- Applies condition to branches

Why they are correct?

Booleans: examples

eval ite_true:

ITE TRUE e_1 e_2

= (λb x y. b x y) TRUE e_1 e_2 -- expand def ITE

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

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

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

= (λx y. x) e_1 e_2 -- beta-step

$=_\beta$ (λy. e_1) e_2 -- beta-step

$=_\beta$ e_1

Other boolean API:

let NOT = λb. ITE b FALSE TRUE

let AND = λb_1 b_2. ITE b_1 b_2 FALSE

let OR = λb_1 b_2. ITE b_1 TRUE b_2

λ-calculus:Numbers

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

let ONE = λf λx. f x

let TWO = λf λx. f (f x)

let THREE = λf λx. f (f (f x))

let FOUR = λf λx. f (f (f (f x)))

let FIVE = λf λx. f (f (f (f (f x))))

let SIX = λf λx. f (f (f (f (f (f x)))))

let ZERO = λf λx. x

https://en.wikipedia.org/wiki/Church_encoding

λ-calculus:Numbers API

- Numbers API
  - let INC = (λn λf λx. f (n f x)) -- Call `f` on `x` one more time than `n` does

- let ADD = λn λm. n INC m. -- Call `f` on `x` exactly `n + m` times

eval inc_zero :
INC ZERO
= (λn λf λx. f (n f x)) ZERO
$=_\beta$ λf λx. f (ZERO f x)
= λf λx. f x
= ONE

eval add_one_zero :
ADD ONE ZERO = ONE

https://en.wikipedia.org/wiki/Church_encoding

TODOs by next lecture

- Install λ+
- Start to work on HW2
- Prepare for the 1st midterm next week
- Come to the discussion session if you have questions