

**CS 162 Programming languages**

# Lecture 3: $\lambda$ -calculus II

Yu Feng  
Winter 2020

# Design a programming language

- Syntax: what do programs look like?
  - Grammar: what programs are we allowed to write?
- Semantics: what do programs mean?
  - Operational semantics: how do programs execute step-by-step?

# Syntax: what $\lambda$ -calculus look like

$$e ::= x$$
$$| \lambda x \rightarrow e$$
$$| e_1 e_2$$

$\lambda x \rightarrow e$  (Haskell)

**fun**  $x \rightarrow e$  (OCaml)

- Programs are expressions  $e$  (also called  $\lambda$ -terms) of one of three kinds:
  - Variable  $x, y, z$
  - Abstraction (i.e. nameless function definition)
    - $\lambda x \rightarrow e$
    - $x$  is the formal parameter,  $e$  is the function body
  - Application (i.e. function call)
    - $e_1 e_2$
    - $e_1$  is the function,  $e_2$  is the argument

# Semantics: what $\lambda$ -calculus mean



- How do I execute a  $\lambda$ -term?
- “Execute”: rewrite step-by-step following simple rules, until no more rules apply

$e ::= x$   
|  $\lambda x \rightarrow e$   
|  $e_1 e_2$

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

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

# Rewrite rules of $\lambda$ -calculus

- $\alpha$ -renaming renaming formals
- $\beta$ -reduction function call

**Let us take a detour and talk about scope**

# Semantics: variable scope

The part of a program where a variable is visible

In the expression  $\lambda x \rightarrow e$

- $x$  is the newly introduced variable
- $e$  is the scope of  $x$
- any occurrence of  $x$  in  $\lambda x \rightarrow e$  is bound (by the binder  $\lambda x$ )

$\lambda x \rightarrow x$

$\lambda x \rightarrow (\lambda y \rightarrow x)$

$x$  is bounded

$x y$

$\lambda y \rightarrow x y$

$(\lambda x \rightarrow \lambda y \rightarrow y) x$

$x$  is free

An occurrence of  $x$  in  $e$  is free if it's *not bound* by an enclosing abstraction

# Semantics: free variables

An variable  $x$  is free in  $e$  if there exists a free occurrence of  $x$  in  $e$

We use “FV” to represent the set of all free variables in a term:

$$FV(x) = x$$

$$FV(\lambda x \rightarrow e) = \text{vars}(e) - x$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$$

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

$$FV((\lambda y \rightarrow x y)) = \{x\}$$

$$FV((\lambda x \rightarrow \lambda y \rightarrow y) x) = \{x\}$$

If  $e$  has no free variables it is said to be closed, or combinators

# Semantics: $\beta$ -Reduction

$\beta$ -Reduction:  $(\lambda x \rightarrow e_1) e_2 =_{\beta} e_1[x := e_2]$

where  $e_1[x := e_2]$  means “ $e_1$  with all **free occurrences** of  $x$  replaced with  $e_2$ ”  
In other words, If you see an *abstraction* applied to an *argument*, take the *body* of the abstraction and replace all free occurrences of the *formal* by that *argument*

$(\lambda x \rightarrow x) \text{ apple} =_{\beta}$  

$(\lambda x \rightarrow (\lambda y \rightarrow y)) \text{ apple} =_{\beta} ???$

$y[x := e] = y$  -- assuming  $x \neq y$   
 $(e_1 e_2)[x := e] = (e_1[x := e]) (e_2[x := e])$   
 $(\lambda x \rightarrow e_1)[x := e] = \lambda x \rightarrow e_1$   
 $(\lambda y \rightarrow e_1)[x := e]$   
  | not ( $y$  in  $FV(e)$ ) =  $\lambda y \rightarrow e_1[x := e]$   
  | otherwise = undefined

Operational semantics of  $\beta$ -Reduction



# Semantics: $\alpha$ -renaming

$$\lambda x \rightarrow e =_{\alpha} \lambda y \rightarrow e[x := y]$$

- Rename a formal parameter and replace all its occurrences in the body
- $\lambda x \rightarrow e$   **$\alpha$ -equivalent** to  $\lambda y \rightarrow e[x := y]$

$$\lambda x \rightarrow x =_{\alpha} \lambda y \rightarrow y =_{\alpha} \lambda z \rightarrow z$$

$$\lambda f \rightarrow f\ x =_{\alpha} \lambda x \rightarrow x\ x$$



$$(\lambda x \rightarrow \lambda y \rightarrow y)\ y =_{\alpha} (\lambda x \rightarrow \lambda z \rightarrow z)\ z$$



$$\lambda x \rightarrow \lambda y \rightarrow x\ y =_{\alpha} \lambda \text{apple} \rightarrow \lambda \text{orange} \rightarrow \text{apple}\ \text{orange}$$



# Semantics: evaluation

A  $\lambda$ -term  $e$  evaluates to  $e'$  if there is a sequence of steps

$$e =_? e_1 =_? \dots =_? e_n =_? e'$$

where each  $=_?$  is either  $=_\beta$  or  $=_\alpha$

$$(\lambda x \rightarrow x) \text{ apple} =_\beta \text{ apple}$$

$$(\lambda f \rightarrow f (\lambda x \rightarrow x)) (\lambda x \rightarrow x) =_? ???$$

$$(\lambda x \rightarrow x x) (\lambda x \rightarrow x) =_? ???$$



# 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 = \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 = \x y → x` -- Returns its first argument
- `let FALSE = \x y → y` -- Returns its second argument
- `let ITE = \b x y → b x y` -- Applies condition to branches

Why they are correct?

# Booleans: examples

eval ite\_true:

```
ITE TRUE e1 e2
= (λb x y → b x y) TRUE e1 e2  -- expand def ITE
=β (λx y → TRUE x y) e1 e2  -- beta-step
=β (λy → TRUE e1 y) e2  -- beta-step
=β TRUE e1 e2  -- expand def TRUE
= (λx y → x) e1 e2  -- beta-step
=β (λy → e1) e2  -- beta-step
=β e1
```

Other boolean API:

```
let NOT = λb → ITE b FALSE TRUE
```

```
let AND = λb1 b2 → ITE b1 b2 FALSE
```

```
let OR = λb1 b2 → ITE b1 TRUE b2
```

# $\lambda$ -calculus:Numbers

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

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

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

let THREE =  $\lambda f x \rightarrow f (f (f x))$

let ZERO =  $\lambda f x \rightarrow x$

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

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

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



# $\lambda$ -calculus:Numbers API

- Numbers API
- **let** **INC** =  $(\lambda n f x \rightarrow f (n f x))$  -- Call `f` on `x` one more time than `n` does
- **let** **ADD** =  $\lambda n m \rightarrow n \text{ INC } m$ . -- Call `f` on `x` exactly `n + m` times

eval inc\_zero :

INC ZERO

=  $(\lambda n f x \rightarrow f (n f x)) \text{ ZERO}$

= $_{\beta}$   $\lambda f x \rightarrow f (\text{ZERO } f x)$

=  $\lambda f x \rightarrow f x$

= ONE

eval add\_one\_zero :

ADD ONE ZERO = ONE

# TODOs by next lecture

- Install OCaml on your laptop
- Come to the discussion session if you have questions
- Check out the research seminars in CS: <https://github.com/fredfeng/CS595N>