**CS 162 Programming languages**

# Lecture 3: λ-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 λ-calculus look like

$$
\begin{aligned}
e ::= &\ x \\
 &\mid \backslash x \to e \\
 &\mid e_1\ e_2
\end{aligned}
$$

\x → e (Haskell)

**fun** x → e (OCaml)

- Programs are expressions e (also called λ-terms) of one of three kinds:
  - Variable x, y, z
  - Abstraction (i.e. nameless function definition)
    - \x → 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 λ-calculus 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
    | \x → e
    | e1 e2
```

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

# Rewrite rules of $\lambda$-calculus

- α-renaming — renaming formals
- β-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 \x → e

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

\x → x

\x →(\y → x)

x is bounded

x y

\y → x y

(\x → \y → 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:

$FV(x) = x$
$FV(\backslash x \rightarrow e) = vars(e) - x$
$FV(e_1\ e_2) = FV(e_1) + FV(e_2)$

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

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

# Semantics: β-Reduction

β-Reduction: $(\backslash x \to e_1)\ e_2 =_\beta e_1[x := e_2]$

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*

$(\backslash x \to x)$ apple $=_\beta$ 

$(\backslash x \to (\backslash y \to y))$ apple $=_\beta$ ???

```
y[x := e]             = y              -- assuming x /= y
(e1 e2)[x := e]       = (e1[x := e]) (e2[x := e])
(\x → e1)[x := e]    = \x → e1
(\y → e1)[x := e]
  | not (y in FV(e)) = \y → e1[x := e]
  | otherwise          = undefined
```

Operational semantics of β-Reduction

https://crypto.stanford.edu/~blynn/lambda/

# Semantics: α-renaming

\x → e $=_\alpha$ \y → e[x := y]

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

\x → x $=_\alpha$ \y → y $=_\alpha$ \z → z

\f → f x $=_\alpha$ \x → x x

(\x -> \y → y) y $=_\alpha$ (\x → \z → z) z

\x → \y → x y $=_\alpha$ \apple → \orange → apple orange

# Semantics: evaluation

A λ-term e evaluates to e' if there is a sequence of steps

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

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

(\x → x) apple $=_\beta$ apple

(\f → f (\x → x)) (\x → x) $=_?$ ???

(\x → x x) (\x → x) $=_?$ ???

# 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  = \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
= (\b x y → b    x  y)  TRUE  e1 e2     -- expand def ITE
=β   (\x y → TRUE x  y)        e1 e2     -- beta-step
=β     (\y → TRUE e1 y)        e2     -- beta-step
=β           TRUE e1 e2                 -- expand def TRUE
=     (\x y → x) e1 e2                  -- beta-step
=β       (\y → e1)   e2                 -- beta-step
=β    e1
```

Other boolean API:

```
let NOT = \b     → ITE b FALSE TRUE
let AND = \b1 b2 → ITE b1 b2 FALSE
let OR  = \b1 b2 → ITE b1 TRUE b2
```

# λ-calculus:Numbers

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

```
let ONE   = \f x → f x

let TWO   = \f x → f (f x)

let THREE = \f x → f (f (f x))

let FOUR  = \f x → f (f (f (f x)))

let FIVE  = \f x → f (f (f (f (f x))))

let SIX   = \f x → f (f (f (f (f (f x)))))
```

```
let ZERO  = \f x → x
```

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

# λ-calculus:Numbers API

- Numbers API
  - let INC  = (\n f x -> f (n f x))     -- Call `f` on `x` one more time than `n` does
  - let ADD = \n m -> n INC m. -- Call `f` on `x` exactly `n + m` times

eval inc_zero :
INC ZERO
= (\n f x -> f (n f x)) ZERO
$=_\beta$ \f x -> f (ZERO f x)
= \f x -> f x
= ONE

eval add_one_zero :
ADD ONE ZERO = ONE

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

# 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