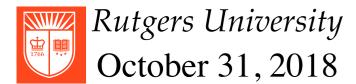
# CS 314 Principles of Programming Languages

Lecture 17: Lambda Calculus

Prof. Zheng Zhang



## **Class Information**

- Midterm exam 11/7 Wednesday 10:20am 11:40am
- Extended hours are Posted
- No classes on 11/2 this Friday

## Review: Lambda Calculus - Historical Origin

### • Church's model of computing is called the *lambda calculus*

It is based on the notion of parameterized expressions (with each parameter introduced by an occurrence of the letter  $\lambda$  — hence the notation's name). Lambda calculus was the inspiration for functional programming: one uses it to compute by *substituting* parameters into expressions, just as one computes in a high level functional program by passing arguments to functions.

## **Review: Functional Programming**

• Functional languages such as Lisp, Scheme, FP, ML, Miranda, and Haskell are an attempt to realize Church's lambda calculus in practical form as a programming language

### • The key idea: do everything by composing functions

- No mutable state
- No side effects
- Function as first-class values

#### **Review: Lambda Calculus**

 $\lambda$ -terms are inductively defined.

#### A $\lambda$ -term is:

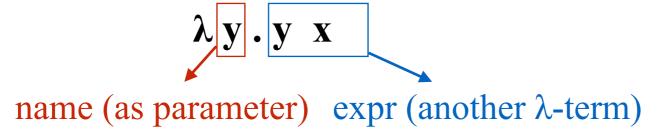
- a variable x
- $(\lambda x. M)$   $\Rightarrow$  where x is a variable and  $\lambda$  is a  $\lambda$ -term (abstraction)
- (M N)  $\Rightarrow$  where M and N are both  $\lambda$ -terms (application)

**Review:** λ-terms

The context-free grammar for  $\lambda$ -terms:

```
\begin{array}{lll} \lambda\text{-term} \to expr \\ expr & \to name \mid number \mid \lambda \ name \ . \ expr \mid func \ arg \\ func & \to name \mid (\lambda \ name \ . \ expr ) \mid func \ arg \\ arg & \to name \mid number \mid (\lambda \ name \ . \ expr ) \mid (\ func \ arg \ ) \end{array}
```

#### Example 1:

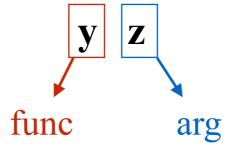


**Review:** λ-terms

The context-free grammar for  $\lambda$ -terms:

```
\begin{array}{lll} \lambda\text{-term} \to expr \\ expr & \to name \mid number \mid \lambda \ name \ . \ expr \mid func \ arg \\ func & \to name \mid (\lambda \ name \ . \ expr ) \mid func \ arg \\ arg & \to name \mid number \mid (\lambda \ name \ . \ expr ) \mid (\ func \ arg \ ) \end{array}
```

### Example 2:



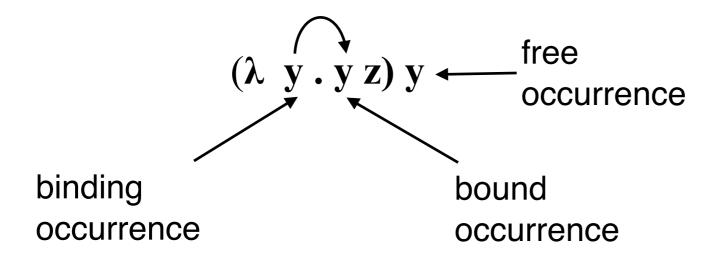
#### **Review: Lambda Calculus**

### Associativity and Precedence

- Function application is left associative: (f g z) is ((f g) z)
- Function application has precedence over function abstraction. "function body" extends as far to the right as possible:  $(\lambda x.yz)$  is  $(\lambda x.yz)$
- Multiple arguments:  $(\lambda xy.z)$  is  $(\lambda x(\lambda y.z))$

#### Review: Free and Bound Variables

Abstraction ( $\lambda x$ . M) "binds" variable x in "body" M. You can think of this as a declaration of variable x with scope M.

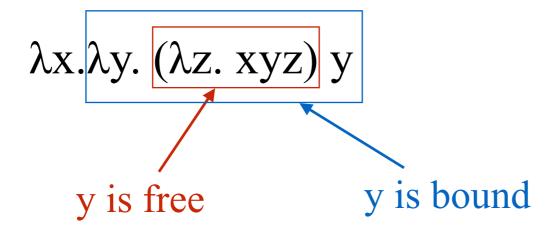


#### **Review: Free and Bound Variables**

#### Note:

A variable can occur **free** and **bound** in a  $\lambda$ -term.

### Example:



"free" is relative to a  $\lambda$ -sub-term.

#### **Review: Free and Bound Variables**

Let M, N be  $\lambda$ -terms and x is a variable. The set of *free variable* of M, free(M), is defined inductively as follows:

- free(x) =  $\{x\}$
- free(M N) = free(M)  $\cup$  free(N)
- free  $(\lambda x.M)$  = free(M) free(x)

## **Review: Function Application**

Computation in lambda calculus is based on the concept or reduction. Simplify an expression until it can no longer be simplified.

### **β**–reduction:

$$(\lambda x.\mathbf{E})y \rightarrow_{\beta} \mathbf{E}[y/x]$$

- 1. Return function body E
- 2. Replace every free occurrence of x in E with y

## **Review: Function Application**

Computation in lambda calculus is based on the concept or reduction. Simplify an expression until it can no longer be simplified.

### **β**–reduction:

$$(\lambda x.\mathbf{E})y \rightarrow_{\beta} \mathbf{E}[y/x]$$

- 1. Return function body E
- 2. Replace every free occurrence of x in E with y

### Example:

$$(\lambda a.\lambda b.a+b) 2 x \rightarrow_{\beta} (\lambda b.2+b) x$$

$$\rightarrow_{\beta} 2+x$$

## **Function Application**

Computation in lambda calculus is based on the concept or reduction. Simplify an expression until it can no longer be simplified.

#### **β**–reduction:

$$(\lambda x.\mathbf{E})y \rightarrow_{\beta} \mathbf{E}[y/x]$$

### We should not perform β-reduction if y is a bound variable within E

Example:

$$(\lambda a.\lambda b.a+b) b 2 \rightarrow_{\beta} (\lambda b.b+b) 2 \rightarrow Incorrect$$

$$\rightarrow_{\beta} 2+2$$

b is a bound variable within λa.λb.a+b

This is called capturing

## **Review: Function Application**

Computation in lambda calculus is based on the concept or reduction. Simplify an expression until it can no longer be simplified.

#### α-reduction:

$$(\lambda x.E) \rightarrow_{\alpha} \lambda y.E[y/x]$$

## **Review: Function Application**

Computation in lambda calculus is based on the concept or reduction. Simplify an expression until it can no longer be simplified.

#### α-reduction:

$$(\lambda x.\mathbf{E}) \rightarrow_{\alpha} \lambda y.\mathbf{E}[\mathbf{y}/\mathbf{x}]$$
Perform  $\alpha$ —reduction first
$$(\lambda a.\underline{\lambda b.a+b}) \ b \ 2 \rightarrow_{\alpha} (\lambda a.\underline{\lambda x.a+x}) \ b \ 2$$

$$\rightarrow_{\beta} \lambda x.b+x \ 2$$

$$\rightarrow_{\beta} b+2$$

## Review: Programming in Lambda Calculus

Remember: Computation in the lambda calculus is a sequence of applications of reduction rules (mostly  $\beta$ –reductions).

Logical constants and operations (incomplete list):

true 
$$\equiv \lambda a$$
.  $\lambda b$ . a

select-first

**false** 
$$\equiv \lambda a. \lambda b. b$$

select-second

**cond** 
$$\equiv \lambda p. \lambda m. \lambda n.(p m n)$$

**not** 
$$\equiv \lambda x$$
. (x false true)

and 
$$\equiv \lambda x. \lambda y. (x y false)$$

$$or \equiv homework$$

## Review: Programming in Lambda Calculus

What about data structures?

Data structures:

pairs can be represented as:

$$[M.N] \equiv \lambda z. (z M N)$$

first 
$$\equiv \lambda x$$
. (x true)(car)second  $\equiv \lambda x$ . (x false)(cdr)build  $\equiv \lambda x.\lambda y.\lambda z.$  (z x y)(cons)

## Programming in Lambda Calculus

What about the encoding of arithmetic constants?

#### Church Numerals:

```
0 \equiv \lambda f x. x
1 \equiv \lambda f x. (f x)
2 \equiv \lambda f x. (f (f x))
...
n \equiv \lambda f x. (f (f (... (f x) ...)) \equiv \lambda f x. (f ^n x)
```

The natural number n is represented as a function that applies a function f n-times to x.

```
succ \equiv \lambda m. (\lambda fx.(f(m f x)))
add \equiv \lambda mn. (\lambda fx.((m f) (n f x)))
mult \equiv \lambda mn. (\lambda fx.((m (n f)) x))
isZero? \equiv \lambda m. (m \lambda x.false true)
```

### Recursion in Lambda Calculus

Does this make sense?

$$f \equiv \dots f \dots$$

In lambda calculus, ≡ is "abbreviated as", but not an assignment.

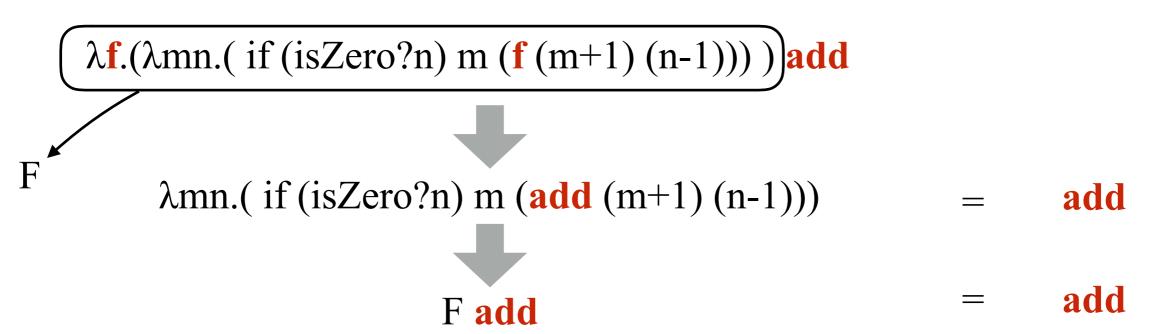
#### Recursion in Lambda Calculus

Does this make sense?

$$f \equiv \dots f \dots$$

In lambda calculus, ≡ is "abbreviated as", not an assignment.

How about



"add" is a fixed point to function F

## The fixed point of a function g is the set of values

$$\{ x \mid x = g(x) \}$$

## Examples:

function <b>g</b>	fix(g)
λx.6	{6}
$\lambda x.(6 - x)$	{3}
$\lambda x.((x * x) + (x - 4))$	{-2, 2}
$\lambda x.x$	entire domain of function f
$\lambda x.(x+1)$	{ }

## Is there a way to "compute" the fixed point of any function F

$$x = F(x)$$

$$Y \equiv \lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))$$

$$YF = ((\lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))) F)$$

### Is there a way to "compute" the fixed point of any function F

$$x = F(x)$$

$$Y \equiv \lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))$$

$$YF = ((\lambda f.((\lambda x.\underline{f}(x x)) (\lambda x.\underline{f}(x x)))) \underline{F})$$

### Is there a way to "compute" the fixed point of any function F

$$x = F(x)$$

$$Y \equiv \lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))$$

$$YF = ((\lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))) F)$$
$$= (\lambda x.F(x x)) (\lambda x.F(x x))$$

## Is there a way to "compute" the fixed point of any function F

$$x = F(x)$$

$$Y \equiv \lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))$$

$$YF = ((\lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))) F)$$
$$= (\lambda x.F(\underline{x} \underline{x})) (\underline{\lambda x.F(x x)})$$

## Is there a way to "compute" the fixed point of any function F

$$x = F(x)$$

$$Y \equiv \lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))$$

$$YF = ((\lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))) F)$$

$$= (\lambda x.F(\underline{x} \underline{x})) (\lambda x.F(\underline{x} \underline{x}))$$

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

## Is there a way to "compute" the fixed point of any function F

$$x = F(x)$$

$$Y \equiv \lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))$$

$$YF = ((\lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))) F)$$

$$YF = (\lambda x.F(x x)) (\lambda x.F(x x))$$

$$YF = F((\lambda x.F(x x)) (\lambda x.F(x x)))$$

### Is there a way to "compute" the fixed point of any function F

$$x = F(x)$$

$$Y \equiv \lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))$$

$$YF = ((\lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))) F)$$

$$YF = (\lambda x.F(x x)) (\lambda x.F(x x))$$

$$YF = F((\lambda x.F(x x)) (\lambda x.F(x x)))$$

$$YF = F(YF)$$

### The Y - Combinator Example (Cont.)

• Informally, the Y-Combinator allows us to get as many copies of the recursive procedure body as we need. The computation "unrolls" recursive procedure calls one at a time

$$Y \equiv \lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))$$

### The Y - Combinator

### Example:

```
\mathbf{F} = \lambda \mathbf{f}. (\lambdamn. if (isZero? n) then m else (\mathbf{f} (succ m) (pred n)))
(YF 3 2) = (((\lambda f.((\lambda x.f(x x))(\lambda x.f(x x)))) F) 3 2)
            = ((F(YF))32)
            = ((\lambda mn.if (isZero? n) then m else YF (succ m) (pred n))) 3 2)
            = if (isZero? 2) then 3 else YF (succ 3) (pred 2))
            = (YF 4 1)
            = ((F (YF)) 4 1)
            = if (isZero? 1) then 4 else YF (succ 4) (pred 1))
            = (YF 5 0)
            = (F(YF) 5 0)
            = if (isZero? 0) then 5 else (YF (succ 5) (pred 0))
            = 5
```

#### Lambda Calculus - Final Remarks

- We can express all computable functions in our  $\lambda$ -calculus.
- All computable functions can be expressed by the following two combinators, referred to as **S** and **K**.
  - $K \equiv \lambda xy.x$
  - $S = \lambda xyz.xz(yz)$

Combinatoric logic is as powerful as Turing Machines.

### **Next Lecture**

## Reading:

- Scott, Chapter 11.1 11.3
- Scott, Chapter 11.7
- ALSU, Chapter 11.1 11.3