# EECS 490 – Lecture 11

Lambda Calculus

1

# Announcements

- Project 3 released, due Fri 10/27
- ► HW3 due Fri 10/20

# Yin-Yang Puzzle

Prints out unary representations of the natural numbers

#### Continuations and Goto

- First-class continuations are often criticized for the same reasons as goto, since they allow unstructured transfer of control
- As with goto, continuations should be used judiciously
  - Implementing more restricted forms of control transfer such as exceptions
  - Adhering to conventions as in continuation-passing style

# Theory

- In this unit, we will learn about theoretical foundations of programming languages and the meaning of code
- Topics
  - Lambda calculus helps us understand how functions work and how they can be used to model computation
  - Operational semantics formally specify the behavior of code fragments, and the rules map directly to implementation in an interpreter
  - Formal type systems allow us to reason about the types in a program, and the rules map directly to implementation in a compiler

#### Lambda Calculus



- Model of computation introduced by Alonzo Church in 1936
- Based entirely on function abstraction ( $\lambda$  expressions) and function application (β-reduction)
- All functions are anonymous

 Inspiration for functional programming and lambda expressions

# Elements of $\lambda$ -Calculus $f_{\chi}$

Context-free grammar:

Expression → Variable

| ∆ Variable . Expression
| Expression Expression (function abstraction)
| (Expression)
| (Expression)
| (Expression)

- Variables denoted by single letters
- **■** Examples:

$$\frac{\lambda x. x}{(\lambda x. x) y}$$

(identity function)
(identity function applied to variable V)

10/16/17



#### Precedence and Associativity

- Function application is left associative and has higher precedence than function abstraction
- Function abstraction extends as far to the right as possible
- Examples:

f g h = (f \ g) h   
 
$$\lambda x \cdot (x \ \lambda y \cdot (x \ y \ z)) = \lambda x \cdot (x \ (\lambda y \cdot ((x \ y) \ z)))$$

Functions are first-class values

#### α-Reduction

In  $(\lambda x, E)$ , replacing all occurrences of x with y does not change the meaning as long as y does not appear in E

$$\lambda x. \quad x \quad x \rightarrow_{\alpha} \lambda y. \quad y \quad y$$

XXXXXXX rename x

to y

- This renaming is called  $\alpha$ -reduction
- Two expressions are  $\alpha$ -equivalent if they only differ by  $\alpha$ -reductions

$$\frac{\lambda x. \times x}{} =_{\alpha} \lambda y. y y$$

# β-Reduction

ightharpoonup In function application, we apply α-reduction to ensure that the function and its argument have distinct names

β-equivalent to identity function

Accomplishes the same thing as frames and environments

$$(\lambda x. x) (\lambda x. x) \rightarrow_{\alpha} (\lambda x. x) (\lambda y. y)$$

- We then substitute the argument expression for the parameter in the scope of the parameter
  - This is  $\beta$ -reduction and is similar to a call-by-name parameter-passing strategy

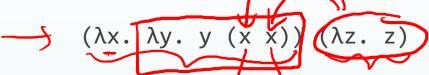
$$(\lambda x. x) (\lambda y. y) \rightarrow_{\beta} \lambda y. y$$

Two expressions are β-equivalent if they β-reduce to the same thing

$$(\lambda x. x) (\lambda x. x) =_{\beta} \lambda y. y$$

## Example





No α-reduction is necessary, so we can do β-reduction  $^1$ 

$$\rightarrow_{\beta} \lambda y. y ((\lambda z. z) (\lambda z. z))$$

■ We need an a reduction green

$$\rightarrow_{\alpha} \lambda y. y ((\lambda z. z)(\lambda w. w))$$

Applying β-reduction

$$\rightarrow_{\beta} \lambda y. y (\lambda w. w)$$

<sup>1</sup>Technically, a β-reduction is not allowed at this point by the normal-order evaluation rules for  $\lambda$ -calculus, as we'll see momentarily.

#### Normal Forms and Termination

- An expression is evaluated by applying β-reduction as long as possible
- An expression that can no longer be β-reduced is in normal form
- Not all evaluations terminate

$$(\lambda x. x. x) (\lambda x. x. x)$$

$$\rightarrow_{\alpha} (\lambda x. x. x) (\lambda y. y. y)$$

$$\rightarrow_{\beta} (\lambda y. y. y) (\lambda y. y. y)$$

$$\rightarrow_{\alpha} (\lambda y. y. y) (\lambda z. z. z)$$

$$\rightarrow_{\beta} (\lambda z. z. z) (\lambda z. z. z)$$

#### Applicative Order (Call by Value)

- In call-by-value languages, arguments are evaluated before the are bound to the parameter of a function
- Function evaluation process in Scheme
  - Evaluate the arguments
  - Create a new frame with its parent as the function's definition environment
  - Bind the parameter names to the argument values in this new frame
  - Run the body in the context of the new frame
- Example that does not terminate:

```
((lambda (y) Function that returns the identity function (lambda (x) (x x)) (lambda (x) (x x))

Non-terminating
```

computation

# Call by Name

- In call by name, arguments are not evaluated until they have been substituted into the body
- Example using Scheme syntax:

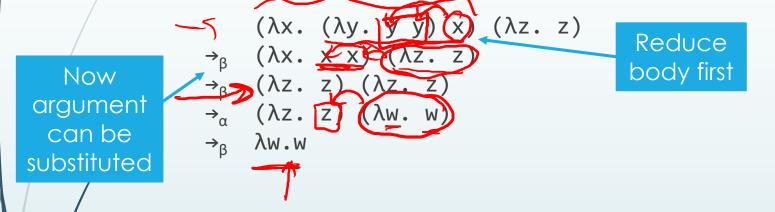
```
((lambda (y) Substitute argument expression for y in the body)

((lambda (x) (x x)) (lambda (x) (x x))

((lambda (x) x) Result is identity function
```

# Normal Order (λ-Calculus)

 λ-Calculus differs from call by name in that function bodies are reduced to normal form before an argument is substituted into the body



#### **Evaluation Rules**

- Full evaluation rules for function application f x
  - 1. Reduce the body of the function f to normal form, resulting in  $f_{\text{normal}}$
  - 2. If a bound variable name appears in both  $f_{normal}$  and x, apply  $\alpha$ -reduction to x, obtaining  $x_{\alpha}$
  - 3. Perform  $\beta$ -reduction by substituting  $x_{\alpha}$  for the parameter of  $f_{normal}$  in the body of  $f_{normal}$ , resulting in just the substituted body
  - 4. Reduce the substituted body until it is in normal form

# More Examples

**•** (λx. λy. x y y) (λy. y) y

**→** (λx. y) ((λy. y y y) (λx. x x x))

■ We'll start again in five minutes.

# **Encoding Data**

- Lambda calculus consists only of variables and functions
  - We can apply β-reduction to substitute functions for variables
- None of the familiar values, such as integers or booleans, exist directly in λ-calculus
- However, we can encode values as functions

#### Booleans

 True and false are represented as functions that take in a true and a false value and return the appropriate value

true = 
$$\lambda t$$
.  $\lambda f$ .  $t$  false =  $\lambda t$ .  $\lambda f$ .  $f$ 

Picks the first value

Picks the second value

Mathematical definition, not assignment

Logical operators are defined as follows:

```
and = \lambda a. \lambda b. a b a or = \lambda a. \lambda b. a a b not = \lambda b. b false true
```

# Conjunction

```
true = \lambda t. \lambda f. t
false = \lambda t. \lambda f. f
and = \lambda a. \lambda b. a b a
```

Applying and to true and another boolean results in:

```
and true bool = ((\lambda a. \lambda b. a b a) \text{ true}) bool

\rightarrow (\lambda b. \text{ true } b \text{ true}) bool

\rightarrow (\lambda b. b) bool

\rightarrow \text{ bool}
```

Applying and to false and another boolean results in:

# Disjunction

```
true = \lambda t. \lambda f. t
false = \lambda t. \lambda f. f
or = \lambda a. \lambda b. a a b
```

Applying or to true and another boolean results in:

```
or true bool = ((\lambda a. \lambda b. a a b) \text{ true}) bool

\rightarrow (\lambda b. \text{ true true bool}) bool

\rightarrow (\lambda b. \text{ true}) bool

\rightarrow \text{ true}
```

Applying or to false and another boolean results in:

```
or false bool = ((λa. λb. a a b) false) bool

→ (λb. false false b) bool

→ (λb. b) bool

→ bool
```

# Negation

true =  $\lambda t$ .  $\lambda f$ . t false =  $\lambda t$ .  $\lambda f$ . f not =  $\lambda b$ . b false true

Applying not to a boolean results in:

#### Conditional

true =  $\lambda t$ .  $\lambda f$ . t false =  $\lambda t$ .  $\lambda f$ . f

 A conditional takes in a boolean, a "then" value, and an "else" value

```
if = \lambda p. \lambda a. \lambda b. p a b
```

Applying if to true and false results in:

```
if true x y = (\lambda p. \lambda a. \lambda b. p a b) true x y

\rightarrow (\lambda a. \lambda b. true a b) x y

\rightarrow (\lambda a. \lambda b. a) x y

\rightarrow x

if false x y = (\lambda p. \lambda a. \lambda b. p a b) false x y

\rightarrow (\lambda a. \lambda b. false a b) x y

\rightarrow (\lambda a. \lambda b. b) x y

\rightarrow y
```

#### **Pairs**

 A pair is represented as a higher-order function that takes in two items and a function, then applies its function argument to the two items

```
pair = \lambda x. \lambda y. \lambda f. f x y
pair a b = (\lambda x. \lambda y. \lambda f. f x y) a b
\rightarrow \lambda f. f a b
```

■ We can define selectors:

```
first = \lambda p. p true
second = \lambda p. p false
```

We can define nil and a null predicate:

```
nil = \lambda x. true
null = \lambda p. p (\lambda x. \lambda y. false)
```

#### Selectors

pair a b  $\rightarrow$   $\lambda f$ . f a b first =  $\lambda p$ . p true second =  $\lambda p$ . p false

Selectors work as follows:

### Null Predicate

```
pair a b \rightarrow \lambda f. f a b
nil = \lambda x. true
null = \lambda p. p (\lambda x. \lambda y. false)
```

The null predicate works as follows:

#### Trees

Now that we have pairs, we can represent arbitrary data structures, including trees:

```
tree = \lambda d. \lambda l. \lambda r. pair d (pair l r) datum = \lambda t. first t left = \lambda t. first (second t) right = \lambda t. second (second t) empty = nil isempty = null
```