



EECS 490 – Lecture 11

Lambda Calculus

1

Announcements

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

- ```
(let* ((yin
 ((lambda (cc) (display "@") cc)
 (call/cc (lambda (c) c))))
 (yang
 ((lambda (cc) (display "*") cc)
 (call/cc (lambda (c) c)))))
 (yin yang))
```

[illegible]

# 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* ( $\beta$ -reduction)
- All functions are anonymous
- Inspiration for functional programming and lambda expressions

# Elements of $\lambda$ -Calculus

- Context-free grammar:

*Expression*  $\rightarrow$  *Variable*

|                                               |                               |
|-----------------------------------------------|-------------------------------|
| $\lambda$ <i>Variable</i> . <i>Expression</i> | <b>(function abstraction)</b> |
| <i>Expression</i> <i>Expression</i>           | <b>(function application)</b> |
| ( <i>Expression</i> )                         |                               |

- Variables denoted by single letters

- Examples:

$\lambda x. x$                       (identity function)

$(\lambda x. x) y$                 (identity function applied to variable  $y$ )

# 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:

$$f\ g\ h = (f\ g)\ h$$

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

- ▶ Functions are first-class values

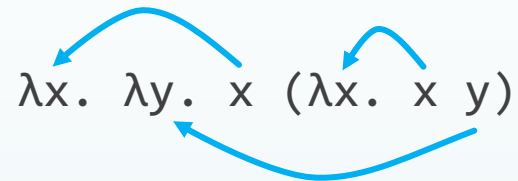
$$(\lambda y. (\lambda x. x))\ (\lambda z. z\ z)$$



# Scope

- Functions are statically scoped

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



- A variable that is not bound is called a *free variable*

$\lambda x. x y$

Bound  
variable

Free  
variable

# $\alpha$ -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  $E$

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

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

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

# $\beta$ -Reduction

$\beta$ -equivalent  
to identity  
function

- In function application, we apply  $\alpha$ -reduction to ensure that the function and its argument have distinct names
  - 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  $\beta$ -equivalent if they  $\beta$ -reduce to the same thing

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

# Example

- Consider:

$$(\lambda x. \lambda y. y (x x)) (\lambda z. z)$$

- No  $\alpha$ -reduction is necessary, so we can do  $\beta$ -reduction<sup>1</sup>

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

- We need an  $\alpha$ -reduction

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

- Applying  $\beta$ -reduction

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

<sup>1</sup>Technically, a  $\beta$ -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  $\beta$ -reduction as long as possible
- An expression that can no longer be  $\beta$ -reduced is in *normal form*
- Not all evaluations terminate

$$\begin{array}{l} (\lambda x. x x) (\lambda x. x x) \\ \rightarrow_{\alpha} (\lambda x. x x) (\lambda y. y y) \\ \rightarrow_{\beta} (\lambda y. y y) (\lambda y. y y) \\ \rightarrow_{\alpha} (\lambda y. y y) (\lambda z. z z) \\ \rightarrow_{\beta} (\lambda z. z z) (\lambda z. z z) \\ \dots \end{array}$$

## Applicative Order (Call by Value)

- In call-by-value languages, arguments are evaluated before they 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)
 (lambda (x) x))
)
((lambda (x) (x x)) (lambda (x) (x x)))
```

Function that returns the identity function

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

Substitute argument expression for y in the body

↓

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

Result is identity function

# Normal Order ( $\lambda$ -Calculus)

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

Now  
argument  
can be  
substituted

$$\begin{aligned}
 & (\lambda x. (\lambda y. y y) x) (\lambda z. z) \\
 \rightarrow_{\beta} & (\lambda x. x x) (\lambda z. z) \\
 \rightarrow_{\beta} & (\lambda z. z) (\lambda z. z) \\
 \rightarrow_{\alpha} & (\lambda z. z) (\lambda w. w) \\
 \rightarrow_{\beta} & \lambda w. w
 \end{aligned}$$

Reduce  
body first



# 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  $f_{\text{normal}}$  and  $x$ , apply  $\alpha$ -reduction to  $x$ , obtaining  $x_\alpha$
  3. Perform  $\beta$ -reduction by substituting  $x_\alpha$  for the parameter of  $f_{\text{normal}}$  in the body of  $f_{\text{normal}}$ , resulting in just the substituted body
  4. Reduce the substituted body until it is in normal form

## More Examples

►  $(\lambda x. \lambda y. x\ y\ y)\ (\lambda y. y)\ y$

►  $(\lambda x. y)\ ((\lambda y. y\ y\ y)\ (\lambda x. x\ x\ x))$

- ▶ We'll start again in five minutes.

# Encoding Data

- ▶ Lambda calculus consists only of variables and functions
  - ▶ We can apply  $\beta$ -reduction to substitute functions for variables
- ▶ None of the familiar values, such as integers or booleans, exist directly in  $\lambda$ -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

$\text{true} = \lambda t. \lambda f. t$   
 $\text{false} = \lambda t. \lambda f. f$

Picks the first value

Picks the second value

Mathematical definition,  
not assignment

- Logical operators are defined as follows:

$\text{and} = \lambda a. \lambda b. a \ b \ a$   
 $\text{or} = \lambda a. \lambda b. a \ a \ b$   
 $\text{not} = \lambda b. b \ \text{false} \ \text{true}$

# Conjunction

```
true = λt. λf. t
false = λt. λf. f
and = λa. λb. a b a
```

- Applying *and* to *true* and another boolean results in:

```
and true bool = ((λa. λb. a b a) true) bool
 → (λb. true b true) bool
 → (λb. b) bool
 → bool
```

- Applying *and* to *false* and another boolean results in:

```
and false bool = ((λa. λb. a b a) false) bool
 → (λb. false b false) bool
 → (λb. false) bool
 → false
```

# Disjunction

```
true = λt. λf. t
false = λt. λf. f
or = λa. λb. a a b
```

- Applying *or* to *true* and another boolean results in:

```
or true bool = ((λa. λb. a a b) true) bool
 → (λb. true true bool) bool
 → (λb. true) bool
 → 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 = λt. λf. t
false = λt. λf. f
not = λb. b false true
```

- Applying *not* to a boolean results in:

```
not true = (λb. b false true) true
 → true false true
 → false
```

```
not false = (λb. b false true) false
 → false false true
 → true
```



# Conditional

```
true = λt. λf. t
false = λt. λf. f
```

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

```
if = λp. λa. λb. p a b
```

- ▶ Applying *if* to *true* and *false* results in:

```
if true x y = (λp. λa. λb. p a b) true x y
 → (λa. λb. true a b) x y
 → (λa. λb. a) x y
 → x
```

```
if false x y = (λp. λa. λb. p a b) false x y
 → (λa. λb. false a b) x y
 → (λa. λb. b) x y
 → 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 = λx. λy. λf. f x y
pair a b = (λx. λy. λf. f x y) a b
 → λf. f a b
```

- We can define selectors:

```
first = λp. p true
second = λp. p false
```

- We can define nil and a null predicate:

```
nil = λx. true
null = λp. p (λx. λy. false)
```

# Selectors

```
pair a b → λf. f a b
first = λp. p true
second = λp. p false
```

- Selectors work as follows:

```
first (pair a b) = (λp. p true) (pair a b)
→ (pair a b) true
= (λf. f a b) true
→ true a b
→ a
```

```
second (pair a b) = (λp. p false) (pair a b)
→ (pair a b) false
= (λf. f a b) false
→ false a b
→ b
```

# Null Predicate

```
pair a b → λf. f a b
nil = λx. true
null = λp. p (λx. λy. false)
```

- The null predicate works as follows:

```
null nil = (λp. p (λx. λy. false)) λx. true
 → (λx. true) (λx. λy. false)
 → true
```

```
null (pair a b) = (λp. p (λx. λy. false)) (pair a b)
 → (pair a b) (λx. λy. false)
 = (λf. f a b) (λx. λy. false)
 → (λx. λy. false) a b
 → (λy. false) b
 → false
```

# Trees

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

```
tree = λd. λl. λr. pair d (pair l r)
datum = λt. first t
left = λt. first (second t)
right = λt. second (second t)
empty = nil
isempty = null
```