

# CMSC 330: Organization of Programming Languages

---

## Lambda Calculus Encodings

# The Power of Lambdas

---

- ▶ Despite its simplicity, the lambda calculus is quite expressive: it is **Turing complete!**
- ▶ Means we can **encode** any computation we want\*
  - If we're sufficiently clever...
- ▶ Examples
  - Let bindings
  - Booleans
  - Pairs
  - Natural numbers & arithmetic
  - Looping

\*To show Turing completeness we have to map every Turing machine to lambda calculus term. We are not doing that here. Rather, we are showing how typical PL constructs can be represented in lambda calculus, to show what it can do

# Let bindings

---

- ▶ Local variable declarations are like defining a function and applying it immediately (once):
  - $\text{let } x = e1 \text{ in } e2 = (\lambda x. e2) e1$

- ▶ Example

- $\text{let } x = (\lambda y. y) \text{ in } x x = (\lambda x. x x) (\lambda y. y)$

where

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

# Booleans

---

- ▶ Church's encoding of mathematical logic

- $\text{true} = \lambda x. \lambda y. x$
- $\text{false} = \lambda x. \lambda y. y$
- if  $a$  then  $b$  else  $c$ 
  - Defined to be the expression:  $a b c$

- ▶ Examples

- if true then  $b$  else  $c = (\lambda x. \lambda y. x) b c \rightarrow (\lambda y. b) c \rightarrow b$
- if false then  $b$  else  $c = (\lambda x. \lambda y. y) b c \rightarrow (\lambda y. y) c \rightarrow c$

# Booleans (cont.)

---

## ▶ Other Boolean operations

- **not** =  $\lambda x.x \text{ false true}$

- ▶  $\text{not } x = x \text{ false true} = \text{if } x \text{ then false else true}$

- ▶  $\text{not true} \rightarrow (\lambda x.x \text{ false true}) \text{ true} \rightarrow (\text{true false true}) \rightarrow \text{false}$

- **and** =  $\lambda x.\lambda y.x y \text{ false}$

- ▶  $\text{and } x y = \text{if } x \text{ then } y \text{ else false}$

- **or** =  $\lambda x.\lambda y.x \text{ true } y$

- ▶  $\text{or } x y = \text{if } x \text{ then true else } y$

## ▶ Given these operations

- Can build up a logical inference system

# Quiz #1

---

What is the lambda calculus encoding of `xor x y`?

- `xor true true = xor false false = false`
- `xor true false = xor false true = true`

- A. `x x y`
- B. `x (y true false) y`
- C. `x (y false true) y`
- D. `y x y`

`true =  $\lambda x.\lambda y.x$`

`false =  $\lambda x.\lambda y.y$`

`if a then b else c =  $a b c$`

`not =  $\lambda x.x$  false true`

# Quiz #1

---

What is the lambda calculus encoding of **xor x y**?

- xor true true = xor false false = false
- xor true false = xor false true = true

- A.  $x x y$
- B.  $x (y \text{ true false}) y$
- C.  $x (y \text{ false true}) y$**
- D.  $y x y$

true =  $\lambda x.\lambda y.x$

false =  $\lambda x.\lambda y.y$

if a then b else c =  $a b c$

not =  $\lambda x.x \text{ false true}$

# Pairs

---

- ▶ Encoding of a pair  $a, b$ 
  - $(a,b) = \lambda x. \text{if } x \text{ then } a \text{ else } b$
  - $\text{fst} = \lambda f. f \text{ true}$
  - $\text{snd} = \lambda f. f \text{ false}$
- ▶ Examples
  - $\text{fst } (a,b) = (\lambda f. f \text{ true}) (\lambda x. \text{if } x \text{ then } a \text{ else } b) \rightarrow$   
 $(\lambda x. \text{if } x \text{ then } a \text{ else } b) \text{ true} \rightarrow$   
 $\text{if true then } a \text{ else } b \rightarrow a$
  - $\text{snd } (a,b) = (\lambda f. f \text{ false}) (\lambda x. \text{if } x \text{ then } a \text{ else } b) \rightarrow$   
 $(\lambda x. \text{if } x \text{ then } a \text{ else } b) \text{ false} \rightarrow$   
 $\text{if false then } a \text{ else } b \rightarrow b$



# Natural Numbers (Church\* Numerals)

---

► Encoding of non-negative integers

- $0 = \lambda f. \lambda y. y$

- $1 = \lambda f. \lambda y. f y$

- $2 = \lambda f. \lambda y. f (f y)$

- $3 = \lambda f. \lambda y. f (f (f y))$

i.e.,  $n = \lambda f. \lambda y. \langle \text{apply } f \text{ } n \text{ times to } y \rangle$

- Formally:  $n+1 = \lambda f. \lambda y. f (n f y)$

\*(Alonzo Church, of course)

## Quiz #2

$n = \lambda f.\lambda y.<apply\ f\ n\ times\ to\ y>$

What OCaml type could you give to a Church-encoded numeral?

- A.  $('a \rightarrow 'b) \rightarrow 'a \rightarrow 'b$
- B.  $('a \rightarrow 'a) \rightarrow 'a \rightarrow 'a$
- C.  $('a \rightarrow 'a) \rightarrow 'b \rightarrow int$
- D.  $(int \rightarrow int) \rightarrow int \rightarrow int$

## Quiz #2

$n = \lambda f.\lambda y.<apply\ f\ n\ times\ to\ y>$

What OCaml type could you give to a Church-encoded numeral?

- A.  $('a \rightarrow 'b) \rightarrow 'a \rightarrow 'b$
- B.  $('a \rightarrow 'a) \rightarrow 'a \rightarrow 'a$**
- C.  $('a \rightarrow 'a) \rightarrow 'b \rightarrow int$
- D.  $(int \rightarrow int) \rightarrow int \rightarrow int$

# Operations On Church Numerals

---

## ▶ Successor

- $\text{succ} = \lambda z.\lambda f.\lambda y.f (z f y)$

- $0 = \lambda f.\lambda y.y$

- $1 = \lambda f.\lambda y.f y$

## ▶ Example

- $\text{succ } 0 =$

$$(\lambda z.\lambda f.\lambda y.f (z f y)) (\lambda f.\lambda y.y) \rightarrow$$

$$\lambda f.\lambda y.f ((\lambda f.\lambda y.y) f y) \rightarrow$$

$$\lambda f.\lambda y.f ((\lambda y.y) y) \rightarrow$$

$$\lambda f.\lambda y.f y$$

$$= 1$$

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

# Operations On Church Numerals (cont.)

---

## ► IsZero?

- $\text{iszero} = \lambda z.z (\lambda y.\text{false}) \text{true}$

This is equivalent to  $\lambda z.((z (\lambda y.\text{false})) \text{true})$

## ► Example

- $\text{iszero } 0 =$

$(\lambda z.z (\lambda y.\text{false}) \text{true}) (\lambda f.\lambda y.y) \rightarrow$

$(\lambda f.\lambda y.y) (\lambda y.\text{false}) \text{true} \rightarrow$

$(\lambda y.y) \text{true} \rightarrow$       Since  $(\lambda x.y) z \rightarrow y$

$\text{true}$

- $0 = \lambda f.\lambda y.y$

# Arithmetic Using Church Numerals

---

- ▶ If M and N are numbers (as  $\lambda$  expressions)
  - Can also encode various arithmetic operations
- ▶ Addition
  - $M + N = \lambda f. \lambda y. M f (N f y)$   
Equivalently:  $+ = \lambda M. \lambda N. \lambda f. \lambda y. M f (N f y)$ 
    - In prefix notation (+ M N)
- ▶ Multiplication
  - $M * N = \lambda f. M (N f)$   
Equivalently:  $* = \lambda M. \lambda N. \lambda f. \lambda y. M (N f) y$ 
    - In prefix notation (\* M N)

# Arithmetic (cont.)

---

► Prove  $1+1 = 2$

- $1+1 = \lambda x.\lambda y.(1\ x)\ (1\ x\ y) =$
- $\lambda x.\lambda y.((\lambda f.\lambda y.f\ y)\ x)\ (1\ x\ y) \rightarrow$
- $\lambda x.\lambda y.(\lambda y.x\ y)\ (1\ x\ y) \rightarrow$
- $\lambda x.\lambda y.x\ (1\ x\ y) \rightarrow$
- $\lambda x.\lambda y.x\ ((\lambda f.\lambda y.f\ y)\ x\ y) \rightarrow$
- $\lambda x.\lambda y.x\ ((\lambda y.x\ y)\ y) \rightarrow$
- $\lambda x.\lambda y.x\ (x\ y) = 2$

- $1 = \lambda f.\lambda y.f\ y$
- $2 = \lambda f.\lambda y.f\ (f\ y)$

► With these definitions

- Can build a theory of arithmetic

# Looping & Recursion

---

- ▶ Define  $D = \lambda x.x x$ , then
  - $D D = (\lambda x.x x) (\lambda x.x x) \rightarrow (\lambda x.x x) (\lambda x.x x) = D D$
- ▶ So  $D D$  is an infinite loop
  - In general, **self application** is how we get looping



# The Fixpoint Combinator

---

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

▶ Then

$$Y F =$$

$$(\lambda f.(\lambda x.f (x x)) (\lambda x.f (x x))) F \rightarrow$$

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

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

$$= F (Y F)$$



▶  $Y F$  is a *fixed point* (aka *fixpoint*) of  $F$

▶ Thus  $Y F = F (Y F) = F (F (Y F)) = \dots$

- We can use  $Y$  to achieve recursion for  $F$

# Example

---

$\text{fact} = \lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * (f (n-1))$

- The second argument to `fact` is the integer
- The first argument is the function to call in the body
  - We'll use `Y` to make this recursively call `fact`

$(Y \text{ fact}) 1 = (\text{fact } (Y \text{ fact})) 1$

→  $\text{if } 1 = 0 \text{ then } 1 \text{ else } 1 * ((Y \text{ fact}) 0)$

→  $1 * ((Y \text{ fact}) 0)$

=  $1 * (\text{fact } (Y \text{ fact}) 0)$

→  $1 * (\text{if } 0 = 0 \text{ then } 1 \text{ else } 0 * ((Y \text{ fact}) (-1)))$

→  $1 * 1 \rightarrow 1$

# Factorial 4=?

---

```
(Y G) 4
G (Y G) 4
(λr.λn.(if n = 0 then 1 else n × (r (n-1)))) (Y G) 4
(λn.(if n = 0 then 1 else n × ((Y G) (n-1)))) 4
if 4 = 0 then 1 else 4 × ((Y G) (4-1))
4 × (G (Y G) (4-1))
4 × ((λn.(1, if n = 0; else n × ((Y G) (n-1)))) (4-1))
4 × (1, if 3 = 0; else 3 × ((Y G) (3-1)))
4 × (3 × (G (Y G) (3-1)))
4 × (3 × ((λn.(1, if n = 0; else n × ((Y G) (n-1)))) (3-1)))
4 × (3 × (1, if 2 = 0; else 2 × ((Y G) (2-1))))
4 × (3 × (2 × (G (Y G) (2-1))))
4 × (3 × (2 × ((λn.(1, if n = 0; else n × ((Y G) (n-1)))) (2-1))))
4 × (3 × (2 × (1, if 1 = 0; else 1 × ((Y G) (1-1)))))
4 × (3 × (2 × (1 × (G (Y G) (1-1)))))
4 × (3 × (2 × (1 × ((λn.(1, if n = 0; else n × ((Y G) (n-1)))) (1-1)))))
4 × (3 × (2 × (1 × (1, if 0 = 0; else 0 × ((Y G) (0-1)))))
4 × (3 × (2 × (1 × (1))))
24
```

# Call-by-name vs. Call-by-value

---

- ▶ Sometimes we have a choice about where to apply beta reduction. Before call (i.e., argument):
  - $(\lambda z.z) ((\lambda y.y) x) \rightarrow (\lambda z.z) x \rightarrow x$
- ▶ Or after the call:
  - $(\lambda z.z) ((\lambda y.y) x) \rightarrow (\lambda y.y) x \rightarrow x$
- ▶ The former strategy is called **call-by-value (CBV)**
  - Evaluate any arguments before calling the function
- ▶ The latter is called **call-by-name (CBN)**
  - Delay evaluating arguments as long as possible

# Partial Evaluation

---

- ▶ It is also possible to evaluate within a function (without calling it):
  - $(\lambda y. (\lambda z. z) y x) \rightarrow (\lambda y. y x)$
- ▶ Called **partial evaluation**
  - Can combine with CBN or CBV
  - In practical languages, this evaluation strategy is employed in a limited way, as **compiler optimization**

```
int foo(int x) {  
    return 0+x;  
}
```



```
int foo(int x) {  
    return x;  
}
```

# Confluence

---

- ▶ No matter what evaluation order (or combination) you choose, you get the **same answer**
  - Assuming the evaluation always terminates
- ▶ However, termination behavior differs between call-by-value and call-by-name
  - $\text{if true then true else (D D)} \rightarrow \text{true}$  under call-by-name
    - ▶  $\text{true true (D D)} = (\lambda x. \lambda y. x) \text{ true (D D)} \rightarrow (\lambda y. \text{true}) \text{ (D D)} \rightarrow \text{true}$
  - $\text{if true then true else (D D)} \rightarrow \dots$  under call-by-value
    - ▶  $(\lambda x. \lambda y. x) \text{ true (D D)} \rightarrow (\lambda y. \text{true}) \text{ (D D)} \rightarrow (\lambda y. \text{true}) \text{ (D D)} \rightarrow \dots$   
never terminates

# Quiz #3

---

**Y** is a fixed point combinator under which evaluation order?

- A. Call-by-value
- B. Call-by-name
- C. Both
- D. Neither

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

$$Y F =$$

$$(\lambda f.(\lambda x.f (x x)) (\lambda x.f (x x))) F \rightarrow$$

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

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

$$= F (Y F)$$

# Quiz #3

---

**Y** is a fixed point combinator under which evaluation order?

- A. Call-by-value
- B. Call-by-name**
- C. Both
- D. Neither

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

$$Y F =$$

$$(\lambda f.(\lambda x.f (x x)) (\lambda x.f (x x))) F \rightarrow$$

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

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

$$= F (Y F)$$

**In CBV, we expand**

**$Y F = F (Y F) = F (F (Y F)) \dots$  indefinitely, for any  $F$**



# Discussion

---

- ▶ Lambda calculus is Turing-complete
  - Most powerful language possible
  - Can represent pretty much anything in “real” language
    - Using clever encodings
- ▶ But programs would be
  - Pretty slow ( $10000 + 1 \rightarrow$  thousands of function calls)
  - Pretty large ( $10000 + 1 \rightarrow$  hundreds of lines of code)
  - Pretty hard to understand (recognize 10000 vs. 9999)
- ▶ In practice
  - We use richer, more **expressive** languages
  - That include built-in primitives

# The Need For Types

---

- ▶ Consider the **untyped** lambda calculus
  - $\text{false} = \lambda x.\lambda y.y$
  - $0 = \lambda x.\lambda y.y$
- ▶ Since everything is encoded as a function...
  - We can easily misuse terms...
    - $\text{false } 0 \rightarrow \lambda y.y$
    - if 0 then ...
  - ...because everything evaluates to some function
- ▶ The same thing happens in assembly language
  - Everything is a machine word (a bunch of bits)
  - All operations take machine words to machine words

# Simply-Typed Lambda Calculus (STLC)

---

- ▶  $e ::= n \mid x \mid \lambda x:t.e \mid e e$ 
  - Added integers  $n$  as primitives
    - Need at least two distinct types (integer & function)...
    - ...to have type errors
  - Functions now include the type  $t$  of their argument
- ▶  $t ::= \text{int} \mid t \rightarrow t$ 
  - $\text{int}$  is the type of integers
  - $t_1 \rightarrow t_2$  is the type of a function
    - That takes arguments of type  $t_1$  and returns result of type  $t_2$

# Types are limiting

---

- ▶ STLC will reject some terms as ill-typed, even if they will not produce a run-time error
  - Cannot type check  $Y$  in STLC
    - Or in OCaml, for that matter, at least not as written earlier.
- ▶ Surprising theorem: All (well typed) simply-typed lambda calculus terms are **strongly normalizing**
  - A normal form is one that cannot be reduced further
    - A **value** is a kind of normal form
  - Strong normalization means STLC terms **always** terminate
    - Proof is *not* by straightforward induction: Applications “increase” term size

# Summary

---

- ▶ Lambda calculus is a core model of computation
  - We can encode familiar language constructs using only functions
    - These encodings are enlightening – make you a better (functional) programmer
- ▶ Useful for understanding how languages work
  - Ideas of types, evaluation order, termination, proof systems, etc. can be developed in lambda calculus,
    - then scaled to full languages