

# EECS 490 – Lecture 12

## Lambda Calculus II

1

# Announcements

- HW3 due Fri 10/20
- Project 3 due Fri 10/27

# Review: $\lambda$ -Calculus

- Context-free grammar:

$Expression \rightarrow Variable$

|  $\lambda Variable . Expression$       **(function abstraction)**  
|  $Expression Expression$       **(function application)**  
|  $( Expression )$

- Variables denoted by single letters

- Examples:

$\lambda x. x$       (identity function)

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

# Review: $\alpha$ -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  $E$

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

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

# Review: $\beta$ -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 equivalent 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

# 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 b) 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
```

- We'll start again in five minutes.



# Church Numerals

- A natural number  $n$  is represented as a function that takes in another function  $f$  and an item  $x$  and applies  $f$  to the item a total of  $n$  times:

zero =  $\lambda f. \lambda x. x$

one =  $\lambda f. \lambda x. f\ x$

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

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

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

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

...

- Mathematically speaking, a number  $n$  takes  $f$  and produces the self-composition  $f^n$

$$\text{three } f \rightarrow f^3 = f \circ f \circ f$$

# Increment

$\begin{aligned}\text{zero} &= \lambda f. \lambda x. x \\ \text{one} &= \lambda f. \lambda x. f\ x \\ \text{two} &= \lambda f. \lambda x. f\ (f\ x)\end{aligned}$
---

- We can increment a number as follows:

$$\text{incr} = \lambda n. \lambda f. \lambda y. f\ (n\ f\ y)$$
$$\begin{aligned}\text{incr zero} &= (\lambda n. \lambda f. \lambda y. f\ (n\ f\ y))\ \text{zero} \\ &\rightarrow \lambda f. \lambda y. f\ (\text{zero}\ f\ y) \\ &= \lambda f. \lambda y. f\ ((\lambda x. x)\ y) \\ &\rightarrow \lambda f. \lambda y. f\ y \\ &= \text{one}\end{aligned}$$
$$\begin{aligned}\text{incr one} &= (\lambda n. \lambda f. \lambda y. f\ (n\ f\ y))\ \text{one} \\ &\rightarrow \lambda f. \lambda y. f\ (\text{one}\ f\ y) \\ &= \lambda f. \lambda y. f\ ((\lambda x. f\ x)\ y) \\ &\rightarrow \lambda f. \lambda y. f\ (f\ y) \\ &= \text{two}\end{aligned}$$

```
two = λf. λx. f (f x)
incr = λn. λf. λy. f (n y)
```

# Addition and Multiplication

- We can define addition as follows:

```
plus = λm. λn. m incr n
```

Apply the *incr* function  
*m* times to *n*

```
plus two three = (λm. λn. m incr n) two three
                → (λn. two incr n) three
                = (λn. (λf. λx. f (f x)) incr n) three
                → (λn. (λx. incr (incr x)) n) three
                → (λx. incr (incr x)) three
                → incr (incr three)
                → incr four
                → five
```

- We can similarly define multiplication:

```
times = λm. λn. m (plus n) zero
```

Apply the *(plus n)*  
function *m* times  
to zero

We can define exponentiation using the same strategy.

10/12/17

# Zero Predicate

```
zero = λf. λx. x
one  = λf. λx. f x
two  = λf. λx. f (f x)
```

- Predicate to check for zero:

```
iszero = λn. n (λy. false) true
```

Only results in true if function never applied

```
iszero zero = (λn. n (λy. false) true) zero
→ zero (λy. false) true
= (λf. λx. x) (λy. false) true
→ (λx. x) true
→ true
```

```
iszero one = (λn. n (λy. false) true) one
→ one (λy. false) true
= (λf. λx. f x) (λy. false) true
→ (λx. (λy. false) x) true
→ (λx. false) true
→ false
```

# Recursion

- Functions are anonymous, so need to arrange to pass in function as an argument to itself

```
fact = λf. λn. if (iszero n)
               one
               (times n (f f (decr n)))
```

Decrement  
function



Pass along function to  
itself in recursive call



- We also need an auxiliary apply function

```
apply = λg. g g
```

# Example

- Computing factorial:

```

apply fact m = (λg. g g) fact m
              → fact fact m
              = (λf. λn. if (iszero n) one
                           (times n (f f (decr n))))
                fact m
              → (λn. if (iszero n) one
                           (times n (fact fact
                                       (decr n))))
                m
              → if (iszero m) one
                 (times m (fact fact (decr m)))
              =β if (iszero m) one
                 (times m (apply fact (decr m)))
  
```

# Y Combinator

- Also known as a *fixed-point combinator*


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

- Applying  $Y$  to a function  $F$  results in

$$\begin{aligned} Y F &= (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) F \\ &\rightarrow (\lambda x. F (x x)) (\lambda x. F (x x)) \\ &\rightarrow (\lambda x. F (x x)) (\lambda y. F (y y)) \\ &\rightarrow F ((\lambda y. F (y y)) (\lambda y. F (y y))) \\ &= F (Y F) \end{aligned}$$

# Simpler Factorial

No longer have  
to pass function  
to itself



- First, define the concrete function  $F$ :

$$F = \lambda f. \lambda n. \text{if } (\text{iszero } n) \text{ one } (\text{times } n \text{ (f (decr } n)))$$

- Now apply  $Y$  to  $F$ , and apply result to a number:

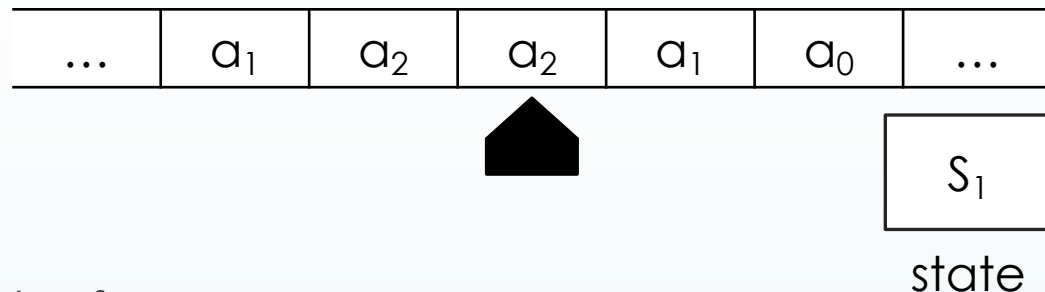
$$\begin{aligned} Y \ F \ m &\rightarrow F \ (Y \ F) \ m \\ &= (\lambda f. \lambda n. \text{if } (\text{iszero } n) \text{ one } (\text{times } n \text{ (f (decr } n)))) \\ &\quad (Y \ F) \ m \\ &\rightarrow (\lambda n. \text{if } (\text{iszero } n) \text{ one } (\text{times } n \text{ (Y } F \text{ (decr } n)))) \ m \\ &\rightarrow \text{if } (\text{iszero } m) \text{ one } (\text{times } m \text{ (Y } F \text{ (decr } m))) \end{aligned}$$

- Letting  $\text{fact} = Y \ F$ , we get

$$\text{fact } m \rightarrow \text{if } (\text{iszero } m) \text{ one } (\text{times } m \text{ (fact (decr } m)))$$



# Turing Machine



- Consists of:
  - An infinite *tape* divided into cells, each containing a symbol from a finite alphabet
  - A *head* positioned at a cell and that can move left or right
  - A *state register* that keeps track of the state of the machine, one of finitely many
  - A *table* of instructions that specifies what to do given a state and the symbol currently under the head
    - Write a specific symbol to the current cell
    - Move the head one step to the left or right
    - Go to a new state

# Church-Turing Thesis

- Alan Turing proved that Turing machines solve the same set of problems as  $\lambda$ -calculus
- The Church-Turing thesis states that all problems that can be solved by a human using an algorithm can also be solved on a Turing machine
- All known computational models are weaker than or equivalent to Turing machines
  - Equivalent models are *Turing complete*
- A programming language defines a computational model
  - All general-purpose programming languages are Turing complete