### CSCI 275: Programming Abstractions Lecture 32: Learning a Language Fall 2024

Stephen Checkoway Slides from Molly Q Feldman



# Goal for the next few days (lambda (x y) (+ x y)))

from?

3. A bunch of other cool things

The content in these lectures is adapted from "Types and Programming Languages" by Pierce, Cornell's CS 4110/6110 Notes and Phil Wadler's "Propositions as Types"

#### 1. Where does the lambda keyword actually come

## 2. Why does Racket's syntax look the way it does?

### MiniScheme

In the MiniScheme project, we wrote an **interpreter** for a language called MiniScheme

- We made parse trees to represent an intermediate version of the language
- We then interpreted those parse trees to evaluate **MiniScheme expressions**

MiniScheme has a formal grammar that we wrote down

### Learning a Language & Practical Concerns

What I want you to take away from this class is a practiced, defined notion of

#### Language design and implementation fundamentals

What's a good way to learn a language?

Know the most *fundamental* underlying structure!

## To Spoil the Punchline....

The rest of this week we are programming language

It's called the *lambda calculus* 

#### The rest of this week we are going to talk about the first

# Invented in 1935 by Alonzo Church

http://dspace.mit.edu/bitstream/handle/1721.1/5794/AIM-349.pdf

#### MASSACHUSETTS INSTITUTE OF TECHNOLOGY ARTIFICIAL INTELLIGENCE LABORATORY

#### AI Memo No. 349

#### SCHEME

#### AN INTERPRETER FOR EXTENDED LAMBDA CALCULUS

Gerald Jay Sussman and Guy Lewis Steele Jr.

Abstract: Inspired by ACTORS [Greif and Hewitt] [Smith and Hewitt], we have implemented an interpreter for a LISP-like language, SCHEME, based on the lambda calculus [Church], but extended for side effects, multiprocessing, and process synchronization. The purpose of this implementation is tutorial. We wish to:

#### December 1975

by





# Introduction to the Lambda Calculus

### The Lambda Calculus

a semantics. Here is its syntax:

variable e :: = x**λx.e** *function abstraction*  $e_1 e_2$ function application

Use parentheses for grouping terms together ( $\lambda x$ .  $\lambda y$ . x) a b Function application is left associative: f x y is the same as (f x) y

Much like other languages, the lambda calculus has a syntax and

### How do we compute with this?

calculus is apply functions to arguments.

#### **Examples:** $(\lambda x. x)$ a gives a $(\lambda x. x. (\lambda x. x))$ b gives us b $(\lambda x. x)$

# It is very simple: all we can do in the base lambda

### How do we compute with this?

calculus is apply functions to arguments.

### **Examples:** $(\lambda x. x)$ a gives a $(\lambda x. x. (\lambda x. x))$ b gives us b $(\lambda x. x)$

These terms are called reducible expressions

# It is very simple: all we can do in the base lambda

Substituting arguments into functions is called betareduction



# How do we compute with the lambda calculus?

# We can actually write *many more meaningful* programs than you might expect!

Church Booleans Church Numerals

## **Reminder: Currying**

Currying is the approach of returning a function from another function:

(define equal-x-checker (lambda (x) (lambda (y) (equal? y x)))

Then (equal-x-checker 3) whether any input is equal to 3

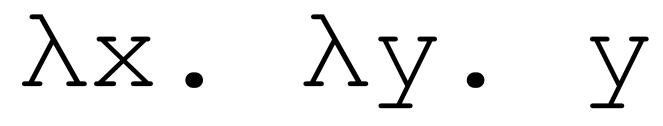
((equal-x-checker 3) 4) is #f

#### will be a procedure that checks

# Currying is default in the lambda calculus Curried functions are actually the only multi-argument functions in the lambda calculus:



#### We could add something like below, but we choose not to:



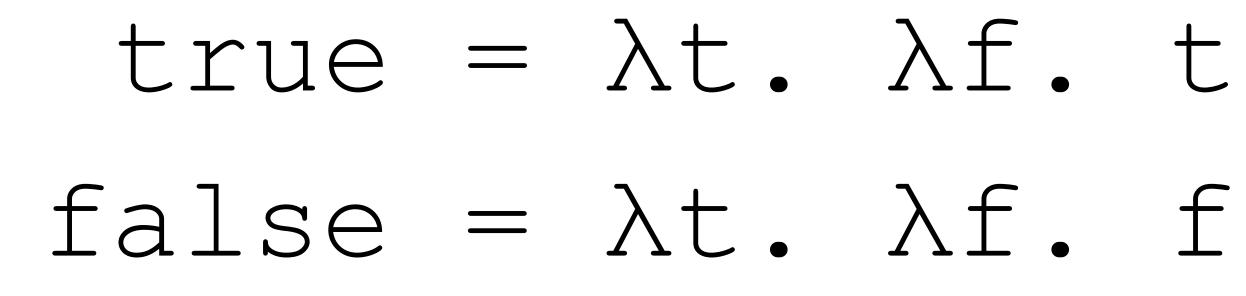
### **Church Booleans**

#t t f) and (if #f t f) in Scheme

true t f = t false t f = f

- We can encode values for true and false. We call these "Church Booleans"
- Intuition: true and false are two argument functions; they act like (i f

# **Church Booleans** Rewriting these in lambda calculus





# **Encoding And**

## Let's walk through the fact this works on the board !

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

#### and = $\lambda b$ . $\lambda c$ . b c false



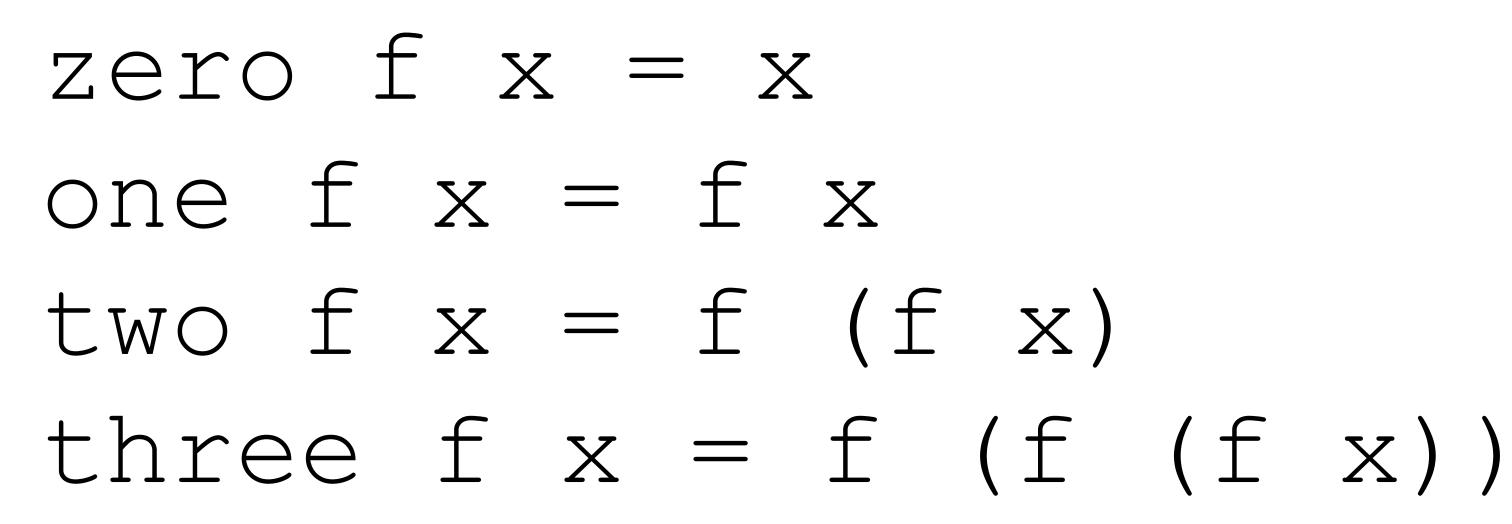
Tf Remember we defined previously as true =  $\lambda t$ .  $\lambda f$ . t and =  $\lambda b$ .  $\lambda c$ . b c false false =  $\lambda t$ .  $\lambda f$ . f Is there another way to encode and? A.  $\lambda b$ .  $\lambda c$ . b c cB.λb. λc. b c b C. $\lambda b$ .  $\lambda c$ . b c true D.Something else E. Nope, only one and!



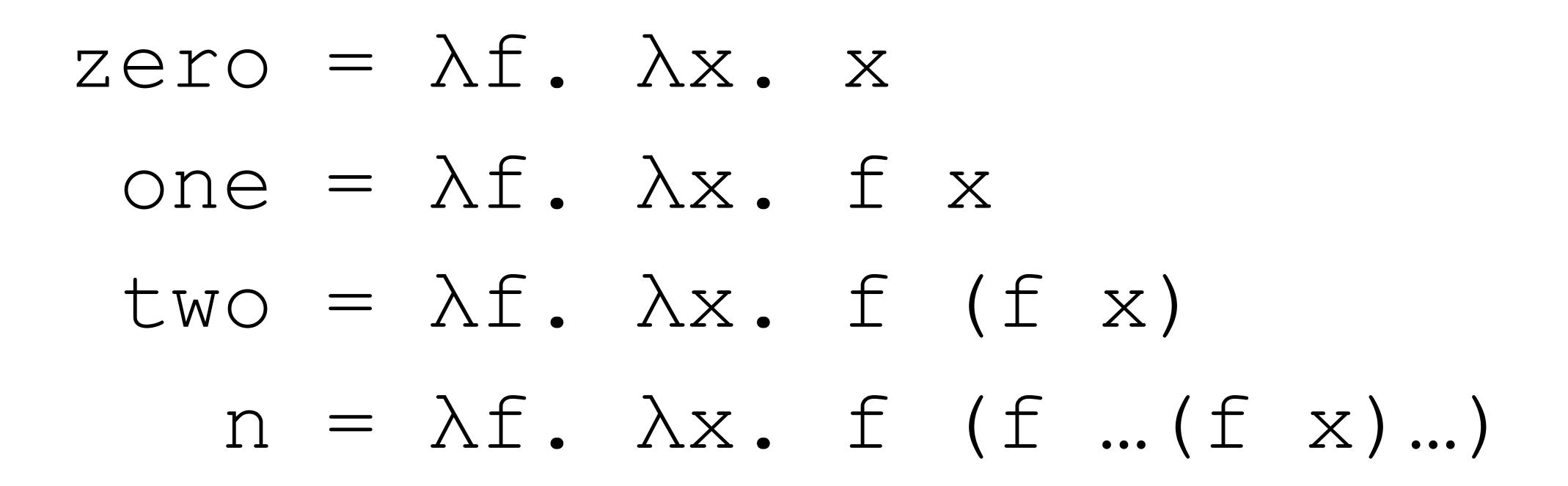
## **Church Numerals**

We can also encode numbers in the lambda calculus

- Intuition: We'll encode numbers as repeated applications of a function f to a value Χ
- Think of each number as a two argument function that applies its first argument to its second argument that number of times



# **Church Numerals** Rewriting this in lambda calculus gives



Wait. If false =  $\lambda t$ .  $\lambda f$ . f and  $zero = \lambda f \cdot \lambda x \cdot x$ 

### Is this a problem?

- A. Yes
- B. No because they have different types (false is a Boolean and zero is a number)
- C. No because they have different parameters
- different contexts to do different things

D. No because we can use the same function in

# Given one, how can we get two? We can define a successor function: one = $\lambda f$ . $\lambda x$ . f xsucc = $\lambda n$ . $\lambda f$ . $\lambda x$ . f (n f x)

To get:  $two = \lambda f \cdot \lambda x \cdot f (f x)$ 

# Let's try it out: https://capra.cs.cornell.edu/lambdalab/

### How can we add two numbers together?

Given two numbers m and n, discuss in your small groups how you might intuitively compute m + n with just the successor function.



# How can we add two numbers together? One way: given m, apply the successor function m times to n!

#### plus = $\lambda m$ . $\lambda n$ . n succ m

### Let's try it out!

# How can we write a recognizer? Let's write a recognizer (something that returns a Boolean): iszero

### This should return (our definition) of true if the argument is zero, and false otherwise

**Idea:** zero f x = x for any f and x n f x = f (f ... (f x)...)

### iszero continued

We want: iszero zero = true First attempt: iszero n = n f true (for some f to be determined shortly) iszero zero = zero f true =  $(\lambda f. \lambda x. x)$  f true  $\rightarrow$  true

### iszero continued

#### We want: iszero one = false

**Need:** a function f such that f x = false so how about that one iszero n = n ( $\lambda x$ . false) true

 $iszero two = two (\lambda x. false) true$ 

=  $(\lambda f. \lambda x. f (f x))$  ( $\lambda x. false$ ) true

- $\rightarrow$  ( $\lambda x$ . false) (( $\lambda x$ . false) true)
- $\rightarrow$  ( $\lambda x$ . false) false

 $\rightarrow$  false

### Bonus stuff: Lists

Let's implement lists in the lambda calculus We need:

- cons creating a pair
- fst car in Scheme
- snd cdr in Scheme
- null the empty list
- isnull null? in Scheme

### The "easy" stuff: Pairs

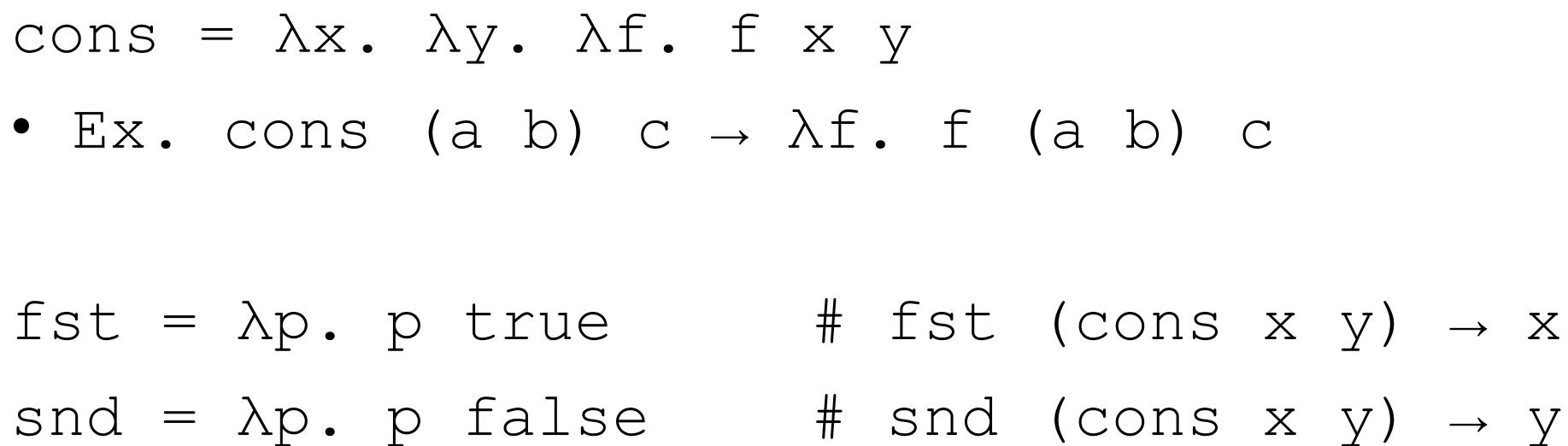
returned their first (for true) or second (for false) arguments

the second element

- For Church Booleans, we decided to use two-argument functions that
- We have a similar situation where there are two parts to the pair and we want fst to return the first element of the pair and snd to return

For Church pairs, let's define the pair as a function that takes a twoargument function and applies that to the two parts of the pair  $\rightarrow$ 

#### Pairs



# fst (cons x y)  $\rightarrow$  x

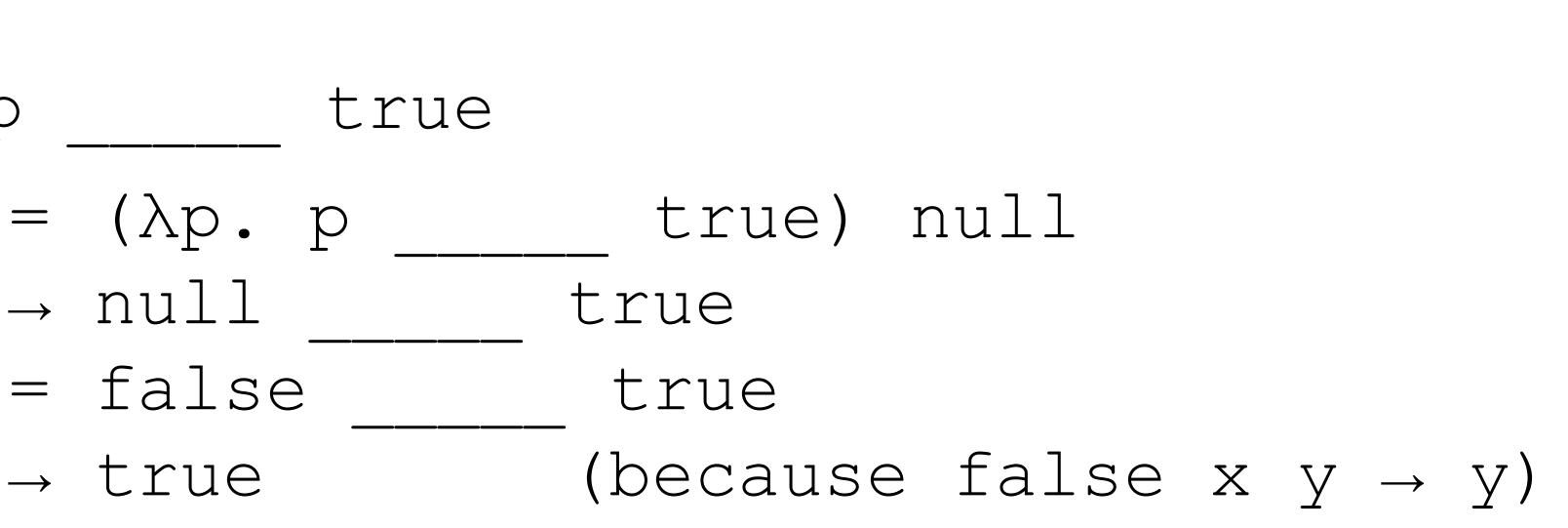
# From pairs to lists (Tricky!)

#### A list is either a pair that we get from cons x y or is null

#### Tricky definition:

- null = false
- $isnull = \lambda p. p$  true
- isnull null =  $(\lambda p. p true)$  null

  - = false
  - $\rightarrow$  true



### isnull $isnull = \lambda p. p$ true

What if p is not null? What if it's cons x y? cons x y  $\rightarrow \lambda f$ . f x y isnull (λf. f x y) = (λp. p \_\_\_\_ true) (λf. f x y)  $\rightarrow$  ( $\lambda$ f. f x y) true  $\rightarrow$  \_\_\_\_ x y true  $\rightarrow$  false

isnull (λf. f x y) =  $(\lambda p. p _ true) (\lambda f. f x y)$  $\rightarrow$  ( $\lambda$ f. f x y) true  $\rightarrow$  \_\_\_\_ x y true  $\rightarrow$  false

What can we replace the with such that the final reduction is correct? Work on this in groups and when you have a solution, select any answer



### Lists

cons =  $\lambda x$ .  $\lambda y$ .  $\lambda f$ . f x y fst =  $\lambda p$ . p true snd =  $\lambda p$ . p false null = falseisnull =  $\lambda p$ . p ( $\lambda x$ .  $\lambda y$ .  $\lambda z$ . false) true